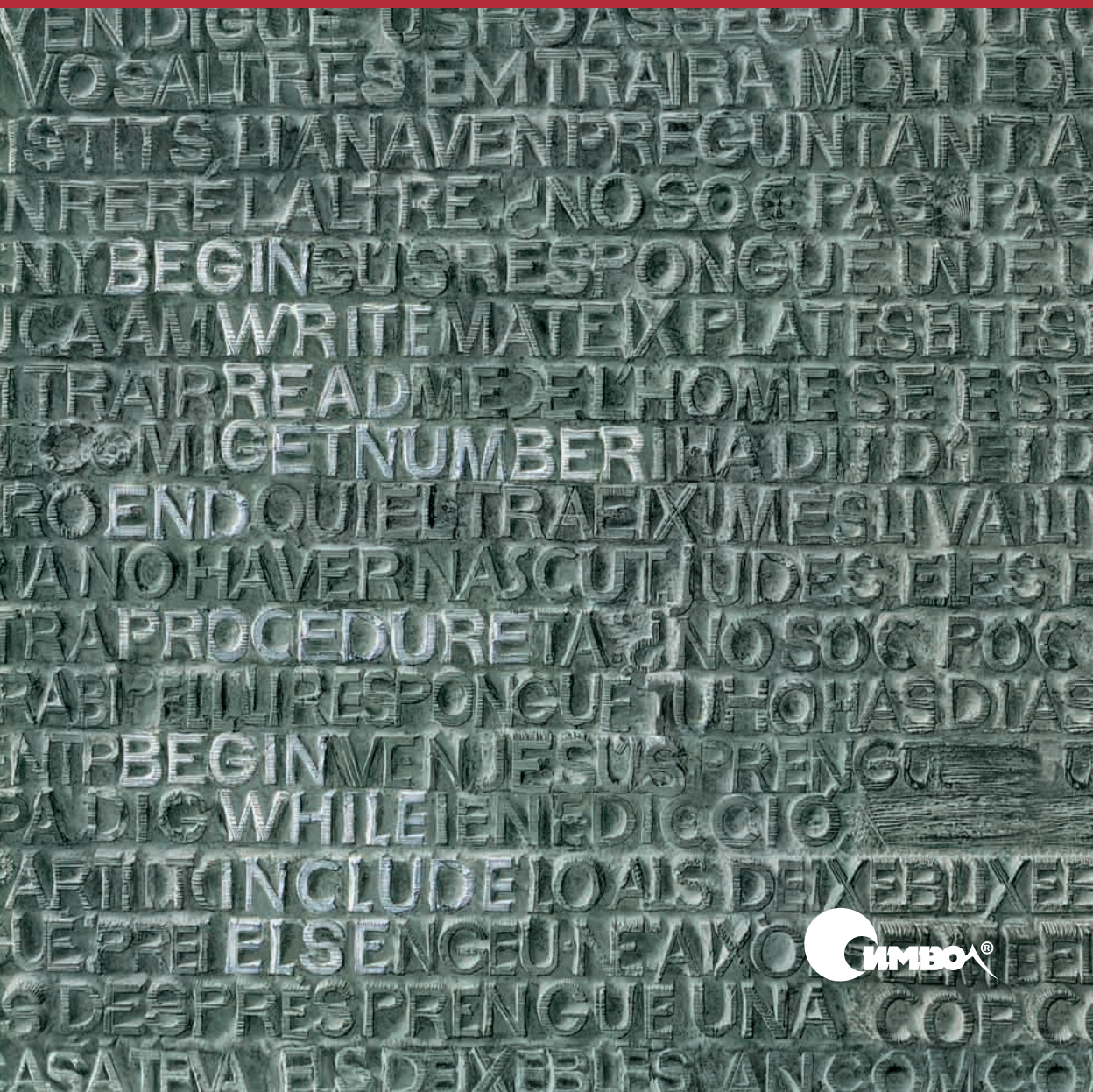
ПРОФЕССИОНАЛЬНО

Питер Гудлиф

РЕМЕСЛО ПРОГРАММИСТА

//практика написания хорошего кода



СИМВОЛ

Code Craft

The Practice of Writing Excellent Code

Pete Goodliffe

ПРОФЕССИОНАЛЬНО

Ремесло программиста

Практика написания хорошего кода

Питер Гудлиф

СИМВОЛ®

Санкт-Петербург — Москва
2009

Серия «Профессионально»

Питер Гудлиф

Ремесло программиста.
Практика написания хорошего кода

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*Б. Попов*
Редактор	*А. Петухов*
Корректор	*О. Макарова*
Верстка	*Д. Орлова*
Художник	*О. Макарова*

Гудлиф П.

Ремесло программиста. Практика написания хорошего кода. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 704 с., ил.

ISBN 978-5-93286-127-1

Ничто не сравнится по ценности с советами настоящего программиста-профессионала. Книга Питера Гудлифа «Ремесло программиста» написана ясно, практично и занимательно. Она поможет вам перейти на более высокий уровень мастерства программирования и покажет, как писать код, который больше чем «просто работает». Да, вы умеете писать работающий код, но как написать понятный код? Как добиться его надежности и отсутствия ошибок? Смогут ли другие программисты выяснить логику и цель вашего кода? Выдающиеся программисты не просто обладают техническими знаниями – у них есть правильный подход и отношение к программированию.

Перед вами руководство по выживанию в условиях промышленного производства ПО. Эта книга посвящена тому, чему вас никто не учил: как *правильно* программировать в *реальной* жизни. Здесь вы найдете не связанные с конкретными языками рекомендации, полезные всем разработчикам и касающиеся таких проблем, как стиль представления, выбор имен переменных, обработка ошибок, безопасность, эффективность групповой работы, технологии разработки и составление документации.

Читатель должен обладать опытом программирования, ибо книга не учит программированию – она учит *правильно* программировать. Издание будет полезно и студентам старших курсов, знакомым с принципами программирования.

ISBN 978-5-93286-127-1
ISBN 978- 1-59327-119-0 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 26.12.2008. Формат 70x100 1/16 . Печать офсетная.
Объем 44 печ. л. Тираж 1500 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Посвящается Брайони, моей чудесной жене.
Элис, рисовавшей овалы со словами.
Милли, попробовавшей книгу на зуб.
И Джессике, еще не способной на это.

Псалом 150

Оглавление

Отзывы на книгу «Ремесло программиста»

«Овладеть ремеслом – это не только освоить приемы и инструменты; здесь нужны позиция и мастерство. Те программисты, которые неравнодушны к своему делу, именно это и узнают из книги. При содействии большого числа обезьянок эта книга приглашает читателя задуматься над тем, чем они занимаются».

Кевлин Хенни (Kevlin Henney),
независимый консультант

«Легко читается, занимательно, даже забавно... Книга полна мудрости, накопленной за годы реальной работы, мучений и побед в сфере программных разработок... Жаль, что у меня не было такой книги, когда я начинал работать программистом».

Стив Лав (Steve Love), старший разработчик

«Эта книга – кладезь информации, необходимой каждому профессиональному разработчику программного обеспечения».

Тим Пенхи (Tim Penhey), редактор C VU

«Здравость суждений приходит с опытом. А опыт – вы получаете его в результате своих «нездравых» суждений! Здесь у вас есть возможность поучиться на чужом, дорого доставшемся опыте, с бóльшей пользой и меньшими муками».

Луи Гольдтвейт (Lois Goldthwaite),
член комитетов по стандартизации C++ и POSIX BSI

«Именно та книга, которая нужна необстрелянным новобранцам. Рассказывает всю правду, легко читается и охватывает широкий круг тем, которые должен знать новичок».

Джон Джеггер (Jon Jagger),
наставник, консультант, преподаватель, программист

«Уникальное и практическое руководство для становления профессионального программиста в современных условиях».

Эндрю Берроуз (Andrew Burrows), разработчик ПО

«Пит обладает редким талантом. Он может не только определить технику, которой пользуются лучшие профессиональные разработчики программ (часто не догадываясь об этом), но и описать ее четким и сжатым образом».

Грег Лоу (Greg Law), CEO, UNDO Ltd.

«Жаль, что этой книги не существовало в начале моей карьеры, когда учили меня. Но хотя бы теперь я могу воспользоваться ею, когда учу других».

Доктор Эндрю Беннет (Andrew Bennett),
старший программист, B.ENG., Ph.D., MIET, MIEEE

«Те, кому посчастливилось присутствовать на лекциях Пита Гудлифа, сразу узнают его манеру рассказывать о предмете понятно и с юмором. В учебной среде это оборачивается направленным структурированным преподаванием, которое позволяет учиться и развиваться как новичку, так и опытному профессионалу».

Роберт Д. Шофилд (Robert D. Schofield), M.SC.,
основатель MIET, SCIENTIFIC SOFTWARE SERVICES Ltd.

«Пит хочет, чтобы программисты писали не просто код, а хороший код, пользуясь правильными инструментами и методами. Книга исследует широкий круг проблем программирования и излагает нормы и принципы, с которыми должен быть знаком каждый разработчик, которому небезразлично его дело».

Крис Рид (Chris Reed), программист

«Увлеченность Пита Гудлифа идеей повышения профессионализма в области разработки ПО давно известна. Обладая солидными знаниями вкупе с даром занимательно и понятно излагать их, Пит проявил себя как прекрасный наставник и для новичков, и для опытных разработчиков».

Роб Войси (Rob Voisey),
технический директор, AKAI DIGITAL Ltd.

«Мне больше всего понравились обезьянки».

Алиса Гудлиф, $4\,{}^{1}/_{2}$ года

Об авторе

Пит Гудлиф (Pete Goodliffe) – опытный разработчик программного обеспечения, постоянно меняющий свою роль в цепи программных разработок; он занимался разработками на многих языках в различных проектах. Кроме того, у него большой опыт обучения и повышения квалификации программистов. Пит ведет регулярную колонку «Professionalism in Programming» в журнале *C Vu*, издаваемом ACCU (*www.accu.org*). Он любит писать превосходный код, в котором нет ошибок, благодаря чему он может больше времени проводить со своими детьми.

Благодарности

Всегда найдется, за что поблагодарить.

Чарльз Диккенс

Эта книга создавалась в течение ряда лет. Говорят, что *терпение вознаграждается*. Множество людей оказывало мне поддержку на протяжении этого пути...

Больше всего благодарности и сочувствия заслуживает моя жена Бриони, которая терпела меня вместе с этим проектом на протяжении столь долгого времени. *Послание к Филиппийцам 1,3*.

Мой друг, отличный программист и выдающийся иллюстратор Дэйвид Брукс (David Brookes), превратил мои ужасные комиксы с обезьянками и дурацкими шутками в прелестные вещицы. Спасибо, Дэйв! Неуклюжие шутки остаются на моей совести.

Ряд людей прочел первые наброски этой книги в том или ином виде. Особая благодарность ACCU (*www.accu.org*), которая успешно помогла мне проверить свои писательские навыки. Спасибо специалистам cthree.org Энди Берроузу (Andy Burrows), Эндрю Беннету (Andrew Bennett) и Крису Риду (Chris Reed), давшим ценные отзывы, а также Стиву Лаву (Steve Love) и ребятам с #ant.org. Джон Джеггер (Jon Jagger) написал взвешенную техническую рецензию и поделился рассказами о собственных сражениях и полученных в них шрамах, благодаря чему книга стала намного лучше.

По большей части эта книга отражает мой собственный опыт и огорчение жалким состоянием разработки программного обеспечения в реальном мире, а также желание помочь людям совершенствоваться в своей специальности. Поэтому «благодарности» заслуживают также различные проблемные компании, в которых я когда-то работал, и ужасные программисты, с которыми я там встречался. Они дали мне столько материала для возмущения, что его хватит едва ли не на всю жизнь! Я *не сразу* понял, насколько мне повезло.

Наконец, спасибо всем сотрудникам No Starch Press, принявшим мою рукопись в малоприятном формате XML и превратившим ее в отличную книгу. Спасибо за вашу веру в проект и за ваш труд.

Предисловие

Есть много вещей, о которых умный человек
предпочел бы не знать.

Ральф Уолдо Эмерсон

В основе этой книги лежит полученный боевой опыт. На самом деле, она отражает глубинные процессы, идущие там, где разрабатывают программы, но часто между тем и другим мало различий. Книга написана для программистов, которым *небезразлично* дело, которым они занимаются. Если вы не из их числа, можете сразу закрыть ее и аккуратно поставить обратно на полку.

Какая мне от этого может быть польза?

Программирование – ваша страсть. Печально, но это так. И как закоренелый технарь вы программируете чуть ли не во время ночного сна. И вот вы попали в центр реального мира, в саму эту отрасль, и занимаетесь тем, о чем и не мечтали: забавляетесь с компьютером, а вам за это еще и деньги платят. Ведь *вы сами* готовы были заплатить за то, чтобы иметь такую возможность!

Но все не так просто и не похоже на то, чего вы ожидали. Огорошенные назначением вам нереальных сроков выполнения задач и неумелым руководством (если его можно назвать этим словом), непрерывным изменением технического задания и необходимостью разбираться в дрянном коде, доставшемся вам от предшественников, вы начинаете сомневаться в том, что выбрали для себя *правильный путь*. Все вокруг мешает вам писать тот код, о котором вы мечтали. Что ж, таковы условия существования в организациях, где пишут программы. Вы попали на передний край упорной битвы за создание шедевров художественного мастерства и научного гения. Удачи вам!

Вот тут вам и может пригодиться «Ремесло программиста». Эта книга посвящена тому, чему вас никто не учил: как правильно программировать *в реальной жизни*. Конечно, в ней рассказывается о технических приемах и хитростях, позволяющих писать хороший код. Но в ней

говорится и кое о чем еще: о том, как писать *правильный* код *правильным* образом.

Что это значит? Есть много аспектов написания хорошего кода в реальном мире:

- Разработка технически элегантного кода
- Создание кода, доступного для сопровождения, т. е. понятного другим
- Способность разобраться в чужом запутанном коде и переделать его
- Умение работать вместе с другими программистами

Все эти навыки (и многие другие) необходимы, чтобы стать настоящим кодером. Вы должны знать скрытую жизнь своего кода: что происходит с ним после того, как вы его набрали. У вас должно быть развито эстетическое чувство: красивый код отличается от уродливого. И нужно обладать практицизмом: решать, когда оправданы упрощения, когда требуется поработать над архитектурой кода, а когда нужно все бросить и двигаться дальше (прагматический принцип «*не трогай то, что уже работает*»). Эта книга поможет вам решать такие задачи. Вы узнаете, как выжить в условиях промышленного производства программ, как вести разведку и выяснять замыслы противника, какой тактики придерживаться, чтобы не угодить в расставленные противником ловушки, и как, несмотря на все препятствия, все-таки создавать *прекрасные* программы.

Разработка программ – интересная профессия. Она динамична, в ней множество преходящих модных поветрий, схем быстрого обогащения и проповедников новых идеологий. Она еще не достигла зрелости. Я не претендую на изобретение чудодейственных средств, но у меня есть некоторые практичные и полезные рекомендации, которыми я хочу поделиться. Это не теория башни из слоновой кости, а реальный опыт и добросовестная практика.

К тому моменту, когда вы переварите этот материал, вы не просто научитесь лучше программировать. Вы сможете успешнее выживать в условиях этой отрасли. Станете настоящим бойцом. Вы освоите ремесло кодировщика. Если такая перспектива вас не вдохновляет, вам, возможно, стоит подумать о военной карьере.

Стремление к совершенству

Так чем же *хорошие* программисты отличаются от *плохих*? Или лучше – чем *отличные* программисты разнятся от *удовлетворительных*? Секрет заключается не только в технической компетенции – я встречал толковых программистов, способных энергично и впечатляюще писать на C++, знающих этот язык на зубок, но код их просто ужасал. Знавал я и более скромных программистов, которые старались писать очень простой код, но их программы были чрезвычайно элегантны и хорошо продуманы.

В чем реальная разница? В основе хорошего программирования лежит ваша *позиция*. Она состоит в умении профессионально решать задачи и в стремлении всегда как можно лучше писать код вопреки давлению, оказываемому на вас условиями работы. Позиция – это очки, через которые мы смотрим на вещи. Через нее мы воспринимаем свою работу и свое поведение. Хороший код появляется в результате тщательного труда мастера, а не бездумного хакерства неряшливого программиста.

Позиция – угол сближения

Чем дольше я изучал и систематизировал область разработки программного обеспечения, тем более убеждался, что незаурядные программисты отличаются именно особой позицией. В словаре значение слова «позиция» (attitude) объясняется примерно так:

attitude (at.ti.tude)

1. **Состояние ума или чувств; настрой к чему-либо.**
2. **Положение самолета в воздухе относительно некоторого эталонного.**

Первое определение не содержит ничего неожиданного, но второе... Оно оказывается интереснее первого.

Сквозь самолет проводят три воображаемые осевые линии: одну через крылья, другую от носа до хвоста и третью вертикально через пересечение первых двух. Летчик поворачивает самолет вокруг этих осей: они определяют угол наклона траектории. Это и есть угловое положение самолета. Если самолет, находясь в неверном угловом положении, слегка прибавит мощности, то он существенно отклонится от цели. Пилот должен постоянно контролировать угловое положение летательного аппарата, особенно в такие критические периоды, как взлет или посадка.

Рискуя уподобиться дрянному мотивационному фильму, я все же отмечу большое сходство этой ситуации с разработкой программного обеспечения. Позиция самолета в пространстве определяет его угловое положение, а наша позиция определяет наше угловое положение относительно задачи кодирования. Неважно, насколько грамотен программист технически – если его способности не сдерживаются разумной позицией, от этого пострадает результат.

Неверная позиция может привести к краху программного проекта, поэтому в программировании очень важно сохранять правильное угловое положение. Ваша позиция окажется либо тормозом, либо ускорителем вашего личного роста. Чтобы совершенствоваться как программисты, мы должны обеспечить себе правильную позицию.

Путь к гибельному коду вымощен благими намерениями. Чтобы стать отличным программистом, нужно научиться быть выше своих намерений, искать положительные перспективы и развивать в себе такую здравую позицию.

В этой книге будет показано, как это делается. Она охватывает широкий спектр – от практических приемов написания кода до крупных организационных проблем. И во всех этих темах я подчеркиваю, какие позиции и подходы будут правильными.

Кому адресована эта книга?

Очевидно, что читать эту книгу следует тем, кто заинтересован в улучшении качества своего кода. Мы все должны стремиться к совершенствованию своего программистского мастерства. Если у вас нет такого стремления, эта книга вам не нужна. Она написана для профессиональных программистов, занимающихся этой работой несколько лет. Книга будет полезна и студентам старших курсов, которые знакомы с принципами программирования, но не уверены в том, как лучше их применять.

Читатель должен обладать опытом программирования. Эта книга не учит программированию; она учит *правильно* программировать. Я старался не навязывать определенных языков и не быть категоричным, но в книгу необходимо было включить примеры кода. Большинство из них написано на популярных в данное время языках: C, C++ и Java. Для понимания этих примеров не требуется глубокого знания языка, поэтому не стоит пугаться, если вы не являетесь классным специалистом по C++.

Предполагается, что читатель активно занимается разработкой кода в условиях некоего программного производства или планирует заняться ею в будущем. Это может означать работу в коммерческой организации, либо участие в каком-нибудь хаотичном проекте свободно распространяемого программного обеспечения, либо разработку программ по контракту.

Какие темы освещаются

В книге рассматриваются вопросы позиции и отношения программиста, но это не учебник психологии. В число обсуждаемых тем входят:

- Представление исходного кода
- Технологии защитного кодирования
- Эффективная отладка программ
- Особенности работы в группе
- Управление исходным кодом

Взглянув на оглавление, вы сможете точно выяснить, какие темы освещены. Чем я руководствовался при выборе тем? Я многие годы занимаюсь обучением программистов, поэтому выбрал вопросы, которые периодически возникают в этом процессе. Я также достаточно долго участвовал в разработке программ и знаю, какие проблемы при этом постоянно возникают – к ним я также обращаюсь.

Если вы сможете победить всех этих злых духов программирования, то превратитесь из подмастерья в настоящего мастера программирования.

Структура книги

Я постарался максимально облегчить чтение книги. Здравый смысл подсказывает, что книгу нужно читать последовательно с начала до конца. К данной книге это не относится. Можете открыть любую главу, которая вас заинтересовала, и начать чтение с нее. Каждая глава самостоятельна, но содержит полезные перекрестные ссылки, благодаря которым видна общая взаимосвязь. Конечно, если вы предпочитаете традиционное чтение, ничто не мешает начать с самого начала.

Структура всех глав одинакова и не сулит неприятных сюрпризов. Главы состоят из следующих разделов:

В этой главе

Сначала перечисляются основные темы главы. В нескольких строках дается обзор содержания. Можете прочесть их и выяснить, о чем будет рассказано.

Глава

Тот захватывающий материал, за возможность прочесть который вы заплатили приличные деньги.

По всей главе встречаются «золотые правила». Таким образом я выделяю важные советы, проблемы и позиции, поэтому обратите на них внимание. Выглядят они так:

Это важно. Обратите внимание!

Резюме

Этот маленький раздел в конце каждой главы подытоживает изложение. В нем дан общий обзор материала. Если вас действительно поджимает время, можете прочесть только «золотые правила» и этот завершающий раздел. Только никому не говорите, что я вам это посоветовал.

После этого я сравниваю подходы, применяемые хорошими и плохими программистами, чтобы вывести из них правильные установки, которые вам необходимо в себе выработать. При достаточной

смелости можете оценить себя по этим примерам; будем надеяться, что правда окажется не слишком болезненной!

См. также

Перечисляются родственные главы и объясняется, каким образом они связаны с рассматриваемой темой.

Контрольные вопросы

В конце задается несколько вопросов. Они включены не для того, чтобы «раздуть» книгу, а входят неотъемлемой частью в каждую главу. Вопросы рассчитаны не на простое перелистывание прочитанного материала, а имеют целью *заставить читателя задуматься*, не ограничиваясь при этом содержанием конкретной главы. Вопросы делятся на две группы:

- **Вопросы для размышления.** В них углубленно разбирается тема главы и поднимаются некоторые важные проблемы.
- **Вопросы личного характера.** Эти вопросы анализируют практику работы и качество кодирования, характерные для вас и группы разработчиков, в которую вы входите.

Не проходите мимо этих вопросов! Даже если вам лень сесть и серьезно поискать ответ на каждый вопрос (поверьте, это принесло бы вам большую пользу), хотя бы прочтите их и слегка задумайтесь.

В последней части книги есть *ответы* на эти вопросы и их *обсуждение*. Это не просто список ответов, потому что многие вопросы не позволяют четко ответить *да* или *нет*. После обдумывания вопросов сравните свои ответы с моими. Мои «ответы» часто содержат дополнительную информацию, которой нет в основной главе.

Содержание глав

Каждая глава посвящена одной теме – одной из современных проблем разработки программного обеспечения. Это обычные причины, по которым программисты пишут плохой код или плохо пишут код. В каждой главе описаны правильные подходы и установки, делающие более сносной жизнь бойца на передовой.

Главы распределены по шести частям. В начале каждой части приводится оглавление и краткое описание содержимого каждой главы. Части организованы так, что сначала рассматривается, *какой* код мы пишем, а в конце – *как* мы его пишем.

Наше исследование начинается с внешнего вида кода, сосредоточиваясь на микроуровне написания исходного кода. Я умышленно начинаю с этих вопросов – программисты *редко* обращают на них внимание:

Часть I. Перед лицом кода

В этой части рассматриваются основные элементы разработки исходного кода. Мы изучим методы защитного программирования

и способы форматирования кода. Затем займемся способами выбора имен и документирования кода. Обсуждаются также стандарты написания комментариев и методики обработки ошибок.

Часть II. Тайная жизнь кода

Здесь мы переходим к *процессу* написания кода: как мы создаем его и как с ним работаем. Мы рассмотрим строительный инструмент, методы тестирования, технику отладки, правильную процедуру сборки выполняемых модулей и оптимизацию. В конце остановимся на том, как писать безопасные программы.

Часть III. Проектирование кода

Эта часть посвящена более общим проблемам построения исходного кода. Мы обсудим проектирование кода, архитектуру программного обеспечения и развитие (или распад) кода с течением времени.

Затем мы перейдем на *макроуровень* – оторвем взгляд от земли и посмотрим, что делается вокруг – как идет жизнь в организации, разрабатывающей программы. Большие программы создаются только группами разработчиков, и в следующих трех частях мы узнаем о приемах и методах повышения эффективности работы таких групп.

Часть IV. Стадо программистов?

Программисты редко живут в вакууме. (Для этого им нужно специальное дыхательное оборудование.) В этой части мы расширим свой кругозор и рассмотрим правильные приемы разработки и их место в повседневной жизни профессионального программиста. Здесь будет рассказано о хорошей практике программирования отдельного программиста и группы, а также об использовании системы контроля версий.

Часть V. Часть процесса

Рассматриваются некоторые процедуры и ритуалы процесса разработки программного обеспечения: составление спецификаций, ревизии кода, черная магия составления графиков работ.

Часть VI. Вид сверху

В заключительной части процесс разработки рассматривается на более высоком уровне и исследуются методологии разработки программного обеспечения, а также различные дисциплины, входящие в программирование.

Как пользоваться этой книгой

Начнете ли вы читать эту книгу с первой страницы или выберете те места, которые вам интересны, значения не имеет.

Важно, чтобы вы читали «Ремесло программиста» непредвзято и размышляли о том, как применить прочитанное в своей работе. *Умный учится на своих ошибках, а тот, кто еще умнее, – на чужих.* Всегда

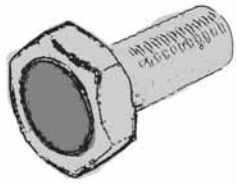
полезно узнать что-то из опыта других людей, поэтому изучая материал этой книги, обсудите его с программистом, чье мнение вы цените. Рассмотрите вместе с ним предлагаемые вопросы.

Я надеюсь, что процесс освоения мастерства кодировщика доставит вам удовольствие. Завершив чтение, оцените, насколько лучше вы стали разбираться в этом ремесле, насколько выросло ваше мастерство и как улучшились ваши установки. Если не произошло никаких перемен, значит, книга не достигла своей цели. Уверен, что такого не случится.

Замечание для тех, кто занимается обучением

Эта книга очень полезна при обучении менее опытных программистов. Она специально задумывалась для этих целей, и ее содействие росту их мастерства и понимания несомненно.

Не советую методически прорабатывать на занятиях каждый раздел. Пусть лучше ученики прочтут главу самостоятельно, а потом ее содержание можно обсудить вместе. Включенные в главу вопросы послужат трамплином для обсуждения, поэтому полезно начать с них.



1

Перед лицом кода

Программисты пишут программы. Не нужно быть гением, чтобы это понять. Но есть тонкое отличие: только хорошие программисты обычно пишут хороший код. Плохие программисты... *не умеют* этого делать. Чтобы навести порядок в том хаосе, который они создают, требуется больше труда, чем они потратили на его написание.

К какому типу программистов вы хотели бы себя отнести?

Ремесло программирования начинается с ввода кода. Мы, программисты, совершенно счастливы, когда углубляемся в редактор и вводим одну за другой идеально отформатированные строки безупречно выполняющегося кода. Мы ничуть не огорчились бы исчезновению окружающего нас мира по мановению булевой логики. К несчастью, окружающая действительность никуда не собирается исчезать – и не хочет оставить нас в покое.

Вокруг вашего тщательно сработанного кода беснуется мир в цепи непрерывных изменений. Практически каждый программный проект находится в непрерывном движении: изменяются технические требования, бюджет, срок завершения, приоритеты и состав участников. Все вокруг нацелено на то, чтобы сделать написание хорошего кода очень трудной задачей. Добро пожаловать в реальный мир.

Естественно, что хорошие программисты пишут отличный код, когда им никто не мешает. Но у них есть тактические приемы, позволяющие писать надежный код, находясь *на передовой*. Они знают, как бороться с суровой действительностью фабрики программ и писать код, который выдержит вихрь перемен.

Этим мы здесь и займемся. В первой части мы углубимся в сугубо практические детали создания кода, технические приемы написания операторов исходного кода. Вы узнаете о стратегиях, которые позволят остаться наплаву в бурных морях разработки программного обеспечения, и о путях совершенствования своего умения писать код.

Главы посвящены следующим вопросам:

Глава 1. Держим оборону

Защитное программирование: как писать надежный код, когда весь мир в заговоре против вас.

Глава 2. Тонкий расчет

Хорошее представление: в чем важность представления кода и какое расположение кода считать хорошим.

Глава 3. Что в имени тебе моем?

Выбор хороших имен для элементов программы.

Глава 4. Литературоведение

Самодокументирующийся код. Практические способы рассказать о коде, когда нельзя написать целый роман.

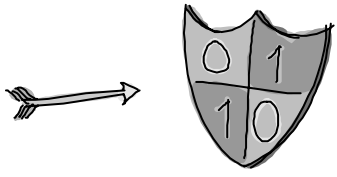
Глава 5. Заметки на полях

Эффективные приемы написания наиболее удачных комментариев к коду.

Глава 6. Людям свойственно ошибаться

Обработка ошибок: как организовать операции, в которых могут произойти сбои, и что с ними потом делать.

Это рассказ о пути к надежному коду в ненадежном мире; это крепкие приемы написания кода, которые должны быть доведены до автоматизма. Если вы не научитесь писать четкий, понятный, защитный, легко тестируемый, легко сопровождаемый код, то вам придется решать скучные проблемы, связанные с кодом, вместо того чтобы готовиться к очередным испытаниям, которым вас подвергнет фабрика программ.



1

Держим оборону

Технологии защитного кодирования для создания надежного кода

В этой главе:

- Что такое защитное программирование?
- Стратегии создания безопасного кода
- Ограничения и операторы контроля

Верить нельзя никому. Это наша единственная защита от предательства.

Теннесси Вильямс

Когда моей дочке было 10 месяцев, ей нравилось играть с деревянными кубиками. Точнее, с кубиками и *со мной*. Я старался построить башню повыше, и тогда она легким толчком по нижнему кубику обрушивала все сооружение и повизгивала от удовольствия. Я не стремился построить крепкую башню – в этом не было смысла. Но если бы я задался целью построить нечто прочное, я действовал бы совсем иначе. Я бы расчистил место для фундамента и сначала положил широкое основание, а не стал бы поспешно громоздить кубики ввысь.

Очень многие программисты пишут код, напоминающий непрочные башни из кубиков; одного легкого толчка в основание достаточно, чтобы обрушить всю конструкцию. Код строится последовательными слоями,

и нужно применять технологии, гарантирующие прочность каждого уровня, чтобы на нем можно было возводить следующий.

На пути к хорошему коду

Есть огромная разница между кодом, который *на первый взгляд* работает, *правильным* кодом и *хорошим* кодом. М. Э. Джексон (M. A. Jackson) писал: «Всякий мудрый программист должен понимать, что есть разница между тем, чтобы заставить программу работать, и тем, чтобы заставить ее делать это *правильно*». (Jackson 75) И вот в чем эта разница:

- Легко написать код, который почти всегда *работает*. Вводишь в программу обычные данные и получаешь обычные результаты. Но стоит подать на вход нечто необычное, и все может рухнуть.
- Правильный код не рухнет. Для любого набора входных данных результат будет корректен. Однако обычно количество всевозможных комбинаций входных данных оказывается невероятно большим, и все их трудно протестировать.
- Однако не всякий правильный код оказывается *хорошим* – например, его логику трудно проследить, код непонятен, его практически невозможно сопровождать.

Следуя этим определениям, мы должны стремиться к созданию хорошего кода. Он надежен, достаточно эффективен и, конечно, правилен. Получив необычные входные данные, код промышленного качества не завершится аварийно и не выдаст неправильный результат. Он также будет удовлетворять другим требованиям, например защищенности потоков, синхронизации и реентерабельности.

Одно дело писать такой хороший код, находясь в удобных домашних условиях, которыми вы вполне управляете. Совершенно иная перспектива – заниматься этим в бурных условиях производства, когда обстановка непрерывно меняется, объем кода быстро растет и постоянно приходится сталкиваться с фантастическим *старым* кодом – архаичными программами, авторов которых найти невозможно. Попробуйте написать хороший код, когда все направлено против вас!

Как в таких условиях обеспечить профессиональное качество кода? На помощь приходит *защитное программирование*.

При всем многообразии методов разработки кода (объектно-ориентированное программирование, модели, основанные на компонентах, структурное проектирование, экстремальное программирование и т. д.) защитное программирование оказывается универсальным подходом. Оно представляет собой не столько формальную технологию, сколько неформальный набор основных принципов. Защитное программирование – не волшебное средство от всех болезней, а практический способ предотвратить множество проблем кодирования.

Готовьтесь к худшему

Во время написания кода невольно делаются какие-то предположения о том, как он будет выполняться, как будет происходить его вызов, какие входные данные допустимы и т. д. Можно даже не заметить сделанные допущения, потому что они кажутся очевидными. Проходят месяцы беззаботного кодирования, и прежние предположения все более стираются из памяти.

Иногда вы берете старый код и делаете в нем очень важное исправление за пять минут до отправки готового продукта. Времени остается только на то, чтобы бегло взглянуть на структуру кода и прикинуть, как он будет работать. Полный критический разбор проводить некогда, и вам остается только надеяться, что код *действительно* делает то, что следует, пока не представится возможность проверить это на практике.

Предположения служат причиной появления кода с ошибками. Очень легко предположить следующее:

- Функцию *никогда* не станут вызывать таким способом. Ей всегда будут передаваться только допустимые параметры.
- Этот фрагмент кода *всегда* будет работать, он никогда не сгенерирует ошибку.
- *Никто* не станет пытаться обратиться к этой переменной, если я напишу в документации, что она предназначена *только для внутреннего употребления*.

В защитном программировании нельзя делать *никаких* допущений. Нельзя предполагать, что *некое событие никогда не случится*. Никогда нельзя предполагать, что все в мире будет происходить так, как нам бы того хотелось.

Опыт показывает, что уверенным *можно* быть только в одном: в один прекрасный день ваш код по каким-то причинам даст сбой. Кто-то *обязательно* сделает глупость. Закон Мерфи гласит: «Если какая-нибудь неприятность может произойти, она обязательно случится». Прислушайтесь к автору – он говорит о том, что познал на собственном опыте.[1] Защитное программирование ограждает от таких несчастных случаев, поскольку предвидит их заранее или хотя бы пытается предугадать путем выяснения неприятностей, способных произойти на каждом этапе выполнения кода, и принятия защитных мер.

Вы считаете это паранойей? Возможно, вы правы. Но быть *немного* параноиком не вредно. На самом деле, это весьма разумно. По мере

[1] Эдвард Мерфи, служивший инженером в ВВС США, сформулировал свой знаменитый закон, когда обнаружил, что некий техник регулярно подключает целый ряд устройств наизнанку. Неправильное подключение было возможно из-за симметричности разъемов. Чтобы избежать этого, пришлось изменить их конструкцию.

развития вашего кода забывается, какие предположения были сделаны вначале (а настоящий код должен развиваться – см. главу 15). Другие программисты не будут иметь никакого понятия о сделанных вами предположениях или, того хуже, сделают собственные ошибочные предположения о возможностях вашего кода. Развитие программного обеспечения выявляет его слабые места, а увеличение объема кода скрывает простые исходные предположения. Немного паранойи вначале может в конечном счете сделать код гораздо устойчивее.

Не делайте никаких допущений. Не зафиксированные формально допущения часто служат причиной отказов, особенно с ростом объема кода.

Учтите еще неприятные обстоятельства, которые нельзя предвидеть ни вам, ни вашим пользователям: недостаток памяти на диске, отказ сети или сбой компьютера. Всегда могут случиться неприятности. И помните, что ошибку совершает не программа – она всегда лишь выполняет то, что вы от нее потребовали. Аварии в системе возникают из-за ошибок в алгоритмах или клиентском коде.

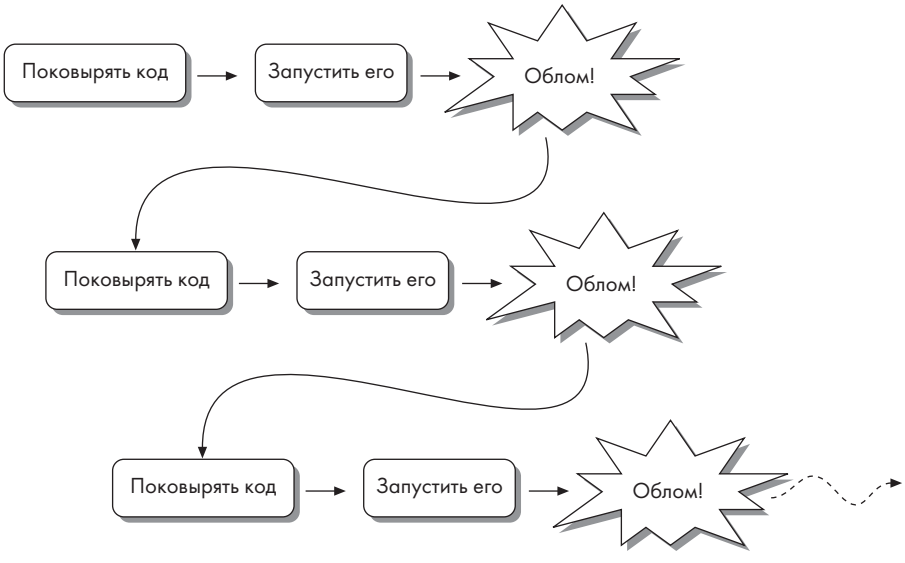
Чем больше объем написанного вами кода и чем быстрее вы его проглядываете, тем выше вероятность допустить ошибку. Нельзя написать устойчивый код, не потратив время, необходимое для проверки каждого предположения. К сожалению, в спешке, в которой создаются программы, редко есть возможность притормозить, критически оценить сделанное и поразмыслить над фрагментом кода. Слишком быстро происходят события в мире, и программистам нельзя отставать. Следовательно, нужно использовать любые возможности для сокращения числа ошибок, и защитные меры оказываются одним из главных видов нашего оружия.

Что такое защитное программирование?

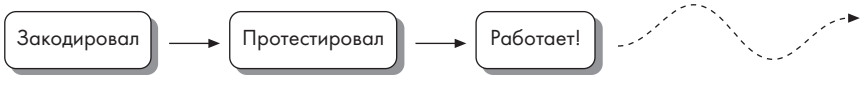
Как следует из названия, защитное программирование – это тщательное, осторожное программирование. Чтобы построить надежную программу, мы должны спроектировать каждую компоненту системы так, чтобы она была как можно лучше *защищена*. Мы расправимся с неписаными допущениями, сделав для них явные проверки в коде. Таким способом мы попытаемся предотвратить или хотя бы обнаружить такое обращение к нашему коду, которое вызывает некорректное поведение.

Защитное программирование позволяет обнаружить мелкие проблемы на ранней стадии, не дожидаясь момента, когда они приведут к серьезным катастрофам. Сплошь и рядом можно столкнуться с тем, как «профессиональные» разработчики спешат написать код, не дав себе труда задуматься – примерно так, как на верхнем рисунке на стр. 33.

Они постоянно спотыкаются на некорректных допущениях, проверить которые им недосуг. Не к чести современного программирования, но это происходит постоянно. Защитное программирование помогает



писать грамотные программы с самого начала и позволяет избавиться от цикла *написать код*, *проверить*, *написать код*, *проверить*... При защитном программировании картинка примерно такая:



Действительно, защитное программирование не устраняет отказы программ полностью, но возникающие проблемы вызывают меньше хлопот, и их проще устранять. Те, кто придерживается этой методики, ловят падающие снежинки, а не ждут, пока их накроет лавина ошибок.

Защитное программирование – это метод профилактики, а не способ лечения. Этим оно отличается от отладки, т. е. устранения ошибок после того, как они дали о себе знать. Отладка – это именно поиск способа лечения.

Действительно ли защитное программирование стоит этих хлопот? Приведем аргументы за и против:

Против

Защитное программирование требует дополнительных ресурсов – ваших и компьютера.

- Оно снижает эффективность кода: даже небольшой дополнительный код требует выполнения лишних операций. Снижение эффективности может быть незаметным, когда касается одной

С чем не нужно путать защитное программирование

Существует ряд распространенных заблуждений, касающихся защитного программирования. К защитному программированию не относятся:

Контроль ошибок

Если в вашем коде может возникнуть состояние ошибки, его всегда нужно проверять. Это не защитный код. Это просто обычная правильная практика – часть написания правильного кода.

Тестирование

Тестирование кода – это не защитное программирование. Это обычный элемент разработки программ. Наборы тестов не дают защиты; с их помощью можно убедиться, что в данный момент код корректен, но это не значит, что код перенесет последующие модификации. Имея в наличии лучший в мире набор тестов, можно сделать такую модификацию, которая проскочит непротестированной.

Отладка

Защитный код можно добавить в период отладки, но отладкой занимаются уже тогда, когда произошел отказ программы. Защитное программирование нужно в первую очередь для того, чтобы предотвратить отказ (или обнаружить ошибки на раннем этапе, пока они не проявили себя недопустимым образом, который потребует от вас отлаживать программу всю ночь).

функции или класса, но если в системе 100 000 функций, могут возникнуть проблемы.

- Каждая защитная мера требует дополнительной работы. Зачем она вам нужна? Вам что – нечем заняться? Заставьте людей правильно пользоваться вашим кодом. Если они этого не могут, то это их проблемы, а не ваши.

За

Аргументы другой стороны убедительны.

- Защитное программирование позволяет намного сократить отладку и заняться чем-то более интересным в освободившееся время. Вспомните Мерфи: если вашим кодом *можно* воспользоваться некорректно, так оно и случится.
- Код, который выполняется правильно, но немного медленнее, *значительно* предпочтительнее кода, который почти всегда работает правильно, но временами с большим треском падает.

- Иногда защитный код проектируется так, что его можно удалить из окончательной версии, и таким образом проблема быстродействия снимается. Однако большинство приемов, которые будут здесь рассмотрены, не влечет сколько-нибудь заметных накладных расходов.
- Защитное программирование избавляет от множества проблем безопасности, что весьма существенно в современных разработках программного обеспечения. Ниже будет сказано об этом подробнее.

Поскольку рынок требует, чтобы программы писались быстро и были дешевы, мы должны сосредоточить свое внимание на технологиях, которые дают нужные результаты. Небольшой дополнительный труд вначале (если им не пренебречь) избавит в дальнейшем от массы неприятностей и задержек.

Этот страшный, ужасный мир

Кто-то однажды сказал, что не следует приписывать злонамеренности то, что объяснимо обычной глупостью.[1] Чаще всего мы строим защиту от глупости, от ошибочных и непроверенных предположений. Однако злоумышленники *существуют*, и они постараются воспользоваться вашим кодом так, чтобы достичь своих преступных целей.

Защитное программирование повышает безопасность программ, предохраняя от такого рода умышленного злоупотребления. Взломщики и создатели вирусов не упускают случая воспользоваться неаккуратно написанным кодом, чтобы получить контроль над приложением и реализовать свои зловредные планы. Это серьезная угроза в современном мире программных разработок, которая приводит к огромному ущербу, наносимому продуктивности, финансам и конфиденциальности.

Злоумышленником может оказаться как беспринципный пользователь, эксплуатирующий мелкую недоработку программы, так и закоренелый взломщик, только и занятый тем, чтобы найти способ незаконно получить доступ к вашей системе. Очень многие программисты, сами не зная того, оставляют бреши, через которые проникают злоумышленники. По мере объединения компьютеров в сети последствия неряшливости в написании кода становятся все более тяжелыми.

Многие крупные компании-разработчики наконец-то начали осознавать эту угрозу и серьезно относиться к данной проблеме, вкладывая время и средства в разработку защитного кода. На практике бывает трудно дополнить программы средствами защиты после того, как атака уже случилась. Более подробно безопасность программного обеспечения рассматривается в главе 12.

---

[1] Некоторые историки приписывают это высказывание Наполеону. Что ж, этот человек кое-что смыслил в том, как нужно защищаться.

Технологии защитного программирования

Но хватит общих слов. Что все это значит для программистов на практике?

Защитное программирование предполагает соблюдение ряда разумных правил. Обычно, когда заходит речь о защитном программировании, людям приходит в голову использовать *операторы контроля*, и это правильно. Мы поговорим о них позже. Но существует также масса простых приемов программирования, которые неизмеримо повысят надежность вашего кода.

Несмотря на свою очевидность, они часто игнорируются, откуда и проистекает низкое качество большей части существующего программного обеспечения. Большей безопасности и надежной разработки можно достичь на удивление легко, если только программисты проявят бдительность и компетентность.

Несколько последующих страниц посвящены перечислению правил защитного программирования. Сначала мы нарисуем общую картину, рассмотрев защитные приемы, процессы и процедуры верхнего уровня. Затем займемся деталями и более пристально рассмотрим отдельные операторы кода. Некоторые из этих защитных приемов специфичны для применяемых языков программирования. И это естественно: если язык допускает какие-то вредоносные действия, то именно от них и нужно защищаться.

Читая этот список, проверьте себя. Многие ли из этих правил вы соблюдаете в своей практике? Какие из них вы теперь готовы принять на вооружение?

Выберите хороший стиль кодирования и пользуйтесь крепкой архитектурой

Значительной части ошибок можно избежать, придерживаясь добротного стиля кодирования. Это правило согласуется с остальными главами данной части. Такие простые вещи, как выбор осмысленных имен переменных и разумная расстановка скобок, делают код понятнее и уменьшают шансы пропустить ошибку.

Точно так же очень важно изучить проект в целом, прежде чем погружаться в написание кода. «Лучшая документация компьютерной программы – это ее четкая структура». (Kernighan Plaugher 78) Если вы начнете с того, что реализуете понятные API, определите логичную структуру системы и четкие роли и задачи компонент, то избавитесь от возможной головной боли в будущем.

Пишите код без спешки

Сплошь и рядом программы пишут сломя голову. Программист быстренько ляпает функцию, пропускает ее через компилятор для провер-

ки синтаксиса, запускает, чтобы убедиться в ее работоспособности, и переходит к очередной задаче. Такой подход чреват опасными последствиями.

Правило: необходимо обдумывать каждую новую строку. Какие ошибки могут в ней возникнуть? Все ли возможные повороты логики вы рассмотрели? Медленное и методичное программирование может показаться скучным, но оно действительно сокращает количество ошибок в программе.

Чем больше спешки, тем меньше скорость. Всегда думайте, что вы собираетесь ввести с клавиатуры.

Пример ловушки, которая поджидает спешащих программистов, пишущих на языках семейства C, – это ошибочный ввод = вместо ==. Первый оператор присваивает значение переменной, а второй проверяет равенство. Недружественный компилятор (или с отключенным выводом предупреждений) не покажет вам, что программа будет выполняться не так, как вам хотелось.

Не бросайтесь вперед, пока не выполните *все* задачи, необходимые для завершения работы с каким-то разделом кода. Например, если вы собираетесь сначала написать основной код, а потом заняться проверкой и обработкой ошибок, вы должны твердо решить, что сделаете и то, и другое. Никогда не откладывайте на будущее проверку ошибок ради того, чтобы написать основной код еще нескольких разделов. Ваше намерение вернуться к обработке ошибок позже может быть вполне искренним, но «позже» может превратиться в «гораздо позже», когда вы в значительной мере забудете контекст, что в результате потребует еще большего труда и времени. (И разумеется, к тому моменту вам будет поставлен какой-нибудь нереальный срок завершения работы.)

Дисциплинированность – это привычка, которую надо усвоить и укреплять. Всякий раз, когда вы откладываете выполнение необходимого дела, вы увеличиваете вероятность того, что не выполните его и в будущем. Сделайте это сейчас, не дожидаясь того дня, когда в Сахаре пройдет дождь. На самом деле, для того чтобы сделать это потом, а не сейчас, требуется обладать еще *большей* дисциплинированностью!

Не верьте никому

Мама говорила вам, чтобы вы не вступали в разговор с незнакомыми людьми? К сожалению, разработка хороших программ требует еще больше цинизма и еще меньше доверия к человеческой натуре. Даже добропорядочные пользователи могут вызвать проблемы у вашей программы; защищаясь, нельзя доверять никому.

У вас могут возникнуть проблемы по следующим причинам:

- **Обычный пользователь** случайно введет в программу неверные данные или воспользуется ею некорректно.

- **Злоумышленник** сознательно попытается заставить программу вести себя некорректно.
- **Клиентский код** вызовет вашу функцию, неправильно передав ей параметры или задав им недопустимые значения.
- **Операционная среда** не сможет предоставить программе необходимый сервер.
- **Внешние библиотеки** окажутся некорректными и не выполнят те контракты по интерфейсам, на которые вы полагались.

А может быть, вы сами сделаете глупую ошибку в какой-то функции или забудете, как работает код, написанный вами три года назад, и неправильно воспользуетесь им. Не рассчитывайте, что все будет хорошо и весь код будет работать корректно. Ставьте проверки везде, где только можно. Все время ищите слабые точки и помещайте в них дополнительные средства защиты.

Не верьте никому. Кто угодно, включая вас самих, может сделать ошибки в логике вашей программы. Ко всем входным и выходным данным относитесь с подозрением, пока не проверите, что они допустимы.

Стремитесь к ясности, а не к краткости

Когда встает выбор между кратким (но непонятным) и ясным (но скучным) кодом, делайте его в пользу того кода, смысл которого *понятен*, даже если он менее элегантен. Например, сложные арифметические выражения разбивайте на последовательность отдельных операторов, логика которых понятнее.

Подумайте о тех, кто будет читать ваш код. Возможно, его будет сопровождать менее опытный кодировщик, и если он не разберется в логике, то может сделать ошибки. Сложные конструкции или оригинальные фокусы с языком могут свидетельствовать о ваших энциклопедических знаниях в области приоритетов операторов, но сильно ослабят возможность сопровождения кода. *Будьте проще*.

Код, который нельзя сопровождать, не может быть безопасным. В отдельных случаях чрезмерно сложные выражения приводят к генерации компилятором некорректного кода – так часто обнаруживаются ошибки в оптимизации, проводимой компилятором.

Простота – это достоинство. Не усложняйте код сверх необходимости.

Не позволяйте никому лезть туда, где ему нечего делать

То, что является внутренним делом, должно оставаться внутри. Ваши личные вещи должны храниться под замком. Не нужно демонстрировать свое грязное белье на публике. Несмотря на любые ваши просьбы, стоит только отвернуться, и люди начнут ковыряться в ваших дан-

Когда приступать?

Когда начинать программировать в защитном стиле? После того как выявятся неполадки? Или когда попадется непонятный код, написанный кем-то другим?

Нет, технологии защитного программирования нужно применять постоянно. Они должны войти в привычку. Опытные программисты знают это по своему опыту; они достаточно часто обжигались, и теперь размещают в программах разумные средства защиты.

Защитные средства гораздо проще применять с самого начала написания кода, нежели пытаться вмонтировать в уже существующий. Если пытаться втиснуть их в код на поздних стадиях работы с ним, трудно ничего не упустить и быть точным. Если вы начинаете добавлять защитный код уже после того, как возникли неполадки, то фактически занимаетесь отладкой, т. е. реагируете на события, а не предупреждаете их.

Тем не менее в процессе отладки или добавления новых функций обнаруживаются условия, которые желательно проверять. Вот тогда и полезно добавить защитный код.

ных, если вы оставите для этого малейшую возможность, и станут вызывать «внутренние» функции по своему разумению. Не позволяйте им этого делать.

- В объектно-ориентированных языках доступ к внутренним данным класса запрещается путем объявления его закрытым. В C++ можно воспользоваться идиомой Чеширского кота (или pimpl) – стандартным приемом, употребляемым для выведения внутренней структуры класса из его открытого файла заголовка. (Meyers 97)
- В процедурных языках также можно воспользоваться идеями объектно-ориентированной упаковки, заключив закрытые данные внутри непрозрачных типов и обеспечив корректные операции над ними.
- Дайте всем переменным минимально необходимую область видимости; не делайте их глобальными, если в этом нет необходимости. Если их можно сделать локальными для функции, не объявляйте их на уровне файла. Если их можно сделать локальными для цикла, не объявляйте их на уровне функции.

Включайте вывод всех предупреждений при компиляции

Большинство компиляторов выдает массу сообщений об ошибках, когда их что-то не устраивает в коде. Они также выводят различные *предупреждения*, если им кажется, что в коде может быть ошибка, например в C или C++ использование переменной до того, как ей присваивается

значение.[1] Обычно вывод таких предупреждений можно избирательно включать или отключать.

Если ваш код изобилует опасными конструкциями, такие предупреждения могут занять многие страницы. К сожалению, по этой причине очень часто отключают вывод предупреждений компилятором или просто не обращают на них внимания. Так поступать не следует.

Всегда включайте вывод предупреждений компилятором, и если ваш код генерирует предупреждения, немедленно исправьте его, чтобы таких сообщений больше не было. Нельзя успокаиваться, пока компиляция не станет проходить гладко при включенном выводе предупреждений. Они существуют не зря. Даже когда вам кажется, что какое-то предупреждение несущественно, добейтесь его исчезновения, потому что в один прекрасный день из-за него вы не заметите того предупреждения, которое окажется действительно важным.

Предупреждения компилятора помогают выявить массу глупых ошибок. Всегда включайте их вывод. Добейтесь того, чтобы ваш код компилировался молча.

Пользуйтесь средствами статического анализа

Предупреждения компилятора представляют собой результат частичного *статического анализа* кода – контроля кода, проводимого *перед* запуском программы.

Существует ряд самостоятельных инструментов статического анализа, например `lint` или его более современные потомки для C и FxCop для сборок .NET. Вашей постоянной практикой должна стать проверка своего кода с помощью этих средств. Они обнаруживают гораздо больше ошибок, чем компилятор.

Применяйте безопасные структуры данных

Или, если это невозможно, пользуйтесь опасными структурами, но с осторожностью.

Вероятно, самым распространенным видом уязвимостей в программах является *переполнение буфера*. Оно возникает из-за небрежного применения структур данных фиксированного размера. Если ваш код записывает данные в буфер, не проверив предварительно его размер, то возникает опасность записи за концом буфера.

Это происходит с чрезвычайной легкостью, как демонстрирует следующий фрагмент кода C++:

```
char *unsafe_copy(const char *source)
{
    char *buffer = new char[10];
```

[1] Во многих языках (например, Java и C#) это считается ошибкой.

```
    strcpy(buffer, source);
    return buffer;
}
```

Если в источнике (`source`) больше 10 символов, то их копирование выйдет за пределы памяти, выделенной под буфер (`buffer`). Результат может оказаться самым неожиданным. В лучшем случае произойдет разрушение данных, если окажется измененным содержимое какой-то другой структуры данных. В худшем – злоумышленник может воспользоваться этой ошибкой в программе и поместить в стек программы исполняемый код, с помощью которого запустит собственную программу и фактически завладеет компьютером. Ошибками такого рода систематически пользуются взломщики; это серьезная проблема.

Предотвратить появление таких уязвимостей нетрудно. Не пишите скверный код! Пользуйтесь более надежными структурами данных, которые не позволяют искажать программу, например управляемыми буферами типа класса `string` в C++. Либо применяйте к небезопасным типам данных только безопасные операции. Приведенный выше код C++ можно защитить, заменив `strcpy` на `strncpy`, которая копирует строку ограниченной длины:

```
char *safer_copy(const char *source)
{
    char *buffer = new char[10];
    strncpy(buffer, source, 10);
    return buffer;
}
```

Проверяйте все возвращаемые значения

Если функция возвращает значение, то делает это не зря. Проверяйте возвращаемое значение. Если это код ошибки, *нужно* ее обработать. Не допускайте появления в программе незамеченных ошибок; в итоге поведение программы может стать непредсказуемым.

Это относится как к определенным пользователям, так и к библиотечным функциям. Чаще всего коварные ошибки возникают тогда, когда программист не проверяет возвращаемые значения. Не забывайте, что иногда функции сообщают об ошибках с помощью иного механизма (например, при помощи стандартной переменной `errno` библиотеки C). Всегда перехватывайте и обрабатывайте соответствующие исключения на соответствующем уровне.

Аккуратно обращайтесь с памятью (и другими ценными ресурсами)

Будьте скрупулезны и освобождайте все ресурсы, которые захватываете во время выполнения. Чаще всего имеется в виду оперативная память, но это не единственный ресурс. К другим видам ценных ресурсов,

которые нужно беречь, относятся файлы и блокировки потоков. Распоряжайтесь своим добром экономно.

Не пренебрегайте тем, чтобы самостоятельно закрыть файлы или освободить память, не рассчитывая, что это сделает ОС по завершении программы. Неизвестно, сколь долго еще будет выполняться ваш код, пожирая дескрипторы файлов и память. Нельзя даже быть уверенным, что ОС правильно освободит ваши ресурсы – не все системы делают это.

Существует мнение, что не стоит заниматься освобождением памяти, пока вы не убедитесь, что программа работает. Не верьте этому. Это крайне опасная практика. Она приводит к многочисленным ошибкам работы с памятью; вы *обязательно* пропустите какие-то места, где нужно освободить память.

Берегите все ограниченные ресурсы. Тщательно организуйте их захват и освобождение.

В Java и .NET есть сборщик мусора, который сделает вместо вас всю эту скучную приборку, чтобы вы могли «забыть» об освобождении ресурсов. Можно мусорить на пол, потому что на этапе исполнения за вами будут периодически подметать. Это замечательно удобно, но у вас не должно возникнуть ложного чувства безопасности. Думать самому все равно нужно. Вы должны явно уничтожить ссылки на объекты, которые вам больше не нужны, иначе они не будут убраны; не делайте ошибки, сохраняя ссылки на объекты. Кроме того, менее изощренные сборщики мусора могут быть введены в заблуждение циклическими ссылками (например, *A* ссылается на *B*, *B* ссылается на *A*, но тот и другой никому не нужны). В этом случае объекты могут никогда не попасть в мусор, что ведет к скрытой утечке памяти.

Инициализируйте все переменные там, где вы их объявили

Это проблема ясности. Смысл каждой переменной становится ясен, если вы инициализируете ее. Опасно полагаться на эмпирические правила типа: «*Раз я ее не инициализировал, значит, ее начальное значение мне неважно*». Код со временем развивается. Отсутствие начального значения на каком-то этапе может превратиться в проблему.

C и C++ усугубляют эту проблему. Пользуясь неинициализированной переменной, вы можете получать разные результаты при каждом запуске программы в зависимости от мусора, находящегося в памяти. Когда вы объявляете переменную в одном месте, присваиваете ей значение в другом, а пользуетесь в третьем, это открывает широкий путь для возникновения ошибок. Если не выполнить присваивание начального значения, то можно потратить уйму времени, пытаясь понять причину случайного поведения программы. Закройте этот источник ошибок путем инициализации каждой переменной при ее объявлении. Даже если присваивается неверное значение, программа все равно будет вести себя предсказуемым образом.

Более защищенные языки (такие как Java и C#) обходят эту опасность путем определения начальных значений для всех переменных. Тем не менее полезно инициализировать переменные при их объявлении, так как это облегчает понимание кода.

Объявляйте переменные как можно позже

Благодаря этому переменная будет располагаться ближе к месту своего использования и не станет мешаться в других частях кода. Кроме того, понятнее станет код, в котором участвует эта переменная. Вам не придется рыскать, выясняя тип и значение переменной: расположенное рядом объявление сделает их очевидными.

Не пользуйтесь повторно одной и той же временной переменной в разных местах, даже если они логически независимы. В противном случае вы чрезвычайно осложните дальнейшую модернизацию кода. Каждый раз создавайте новую переменную, а компилятор сам разберется, можно ли оптимизировать такой код.

Пользуйтесь стандартными средствами языка

C и C++ в этом отношении представляют собой кошмар. Их спецификации существуют во многих вариантах, а *поведение* в наименее ясных ситуациях *не определено* и оставлено на усмотрение конкретных реализаций. На сегодняшний день существует множество компиляторов, обладающих тонкими различиями. В целом они совместимы между собой, но оставляют вам достаточно возможностей свернуть себе шею.

Четко определите, какой версией языка вы пользуетесь. Если только ваш проект не оставляет иного выбора (и причины должны быть достаточно вескими), *не пользуйтесь* индивидуальными особенностями компилятора и нестандартными расширениями языка. Если в языке есть неопределенная область, не полагайтесь на то, как ведет себя в ней ваш конкретный компилятор (например, если ваш компилятор C рассматривает `char` как величину со знаком, то другие компиляторы могут не делать этого). В противном случае код становится очень неустойчивым. А если вы перейдете на новую версию компилятора? А если в команде появится новый программист, не знакомый с вашими расширениями? Полагаясь на необычные свойства конкретного компилятора, вы *создаете источник* труднообнаруживаемых ошибок, которые могут появиться в будущем.

Пользуйтесь хорошими средствами регистрации диагностических сообщений

Когда пишут новый код, то часто включают в него много операторов вывода диагностики, чтобы разобраться в происходящем в программе. Следует ли удалить их в конце работы? Сохранив их, вы облегчите

себе жизнь в будущем, если придется вернуться к этому коду, особенно если можно управлять выводом диагностики.

Существует ряд систем диагностирования, облегчающих такую работу. При этом часто наличие диагностики не влечет накладных расходов, когда она не требуется; ее можно удалить путем условной компиляции.

Выполняйте приведение типов с осторожностью

Большинство языков позволяет *приводить* (преобразовывать) данные из одного типа в другой. Эта операция не всегда проходит успешно. Если вы попытаетесь преобразовать 64-разрядное целое в меньший, 8-разрядный тип, то что произойдет с 56 оставшимися битами? Среда выполнения может неожиданно сгенерировать исключительную ситуацию, а может исказить ваши данные, ничего при этом не сообщив. Программисты часто не задумываются над такого рода вещами, а потому их программы ведут себя неожиданным образом.

Если вы действительно хотите выполнить преобразование типа, тщательно продумайте его. Фактически вы говорите своему компилятору: «Забудь про контроль типа – я лучше знаю, что это за переменная». Вы делаете пролом в системе типов и смело идете в него. Это зыбкая почва: если вы ошибетесь, компилятор не станет вам помогать, а тихо пробормочет про себя: «Я тебя предупреждал». В лучшем случае (например, если вы пишете на Java или C#) среда исполнения может оповестить вас путем генерации исключения, но это зависит от конкретного преобразования.

Для C и C++ особенно характерна неопределенность в точности разных типов данных, поэтому не рассчитывайте на взаимозаменяемость типов. Не следует предполагать, что `int` и `long` имеют один и тот же размер и могут присваиваться один другому, даже если на *вашей* платформе это проходит. Код может переноситься с одной платформы на другую, но плохой код переносится плохо.

Подробности

Есть много приемов создания защитных конструкций на нижнем уровне, основанных на разумной практике кодирования и здоровом недоверии к окружающим. Например:

Определяйте поведение по умолчанию

В большинстве языков есть оператор `switch` (переключатель), позволяющий любую неожиданность обработать в группе `default` (по умолчанию). Если попадание в `default` является ошибкой, явно отразите это в коде. Если ошибки нет, также отразите *это* в коде, и тогда сопровождающему код программисту станет все ясно.

Аналогично, если вы пишете оператор `if` без предложения `else`, задумайтесь, не следует ли обрабатывать логический случай по умолчанию.

Пользуйтесь идиомами языка

Следуя этому простому совету, вы гарантируете, что ваш код будет понятен читающим его. Они реже будут ошибаться.

Проверяйте числовые результаты

Даже самые простые вычисления могут приводить к переполнению или потере точности. Будьте настороже. Спецификации языков и базовые библиотеки предоставляют механизмы для определения допустимых значений стандартных типов данных – воспользуйтесь ими. Вы должны знать, какие числовые типы существуют и для каких целей их лучше применять.

Проверяйте допустимость каждого вычисления. Например, следите за тем, чтобы ваши величины не приводили к *делению на ноль*.

Соблюдайте защищенность констант

Особенно это облегчает жизнь при программировании на C/C++. Старайтесь объявлять как `const` все, что только можно. Этим достигаются две цели: квалификаторы `const` способствуют документированию кода и позволяют компилятору обнаруживать глупые ошибки. Он не позволит вам изменить данные, модификация которых запрещена.

Ограничения

Как уже говорилось, при составлении программы мы делаем ряд неявных предположений. Но как физически учесть эти предположения в коде, чтобы они не привели к возникновению неуловимых проблем? Нужно просто написать некоторый дополнительный код для проверки всех этих условий. Такой код выступает в роли документации каждого предположения, переводя его в явный вид.[1] В результате мы приводим в систему *ограничения*, налагаемые на функциональность и поведение программы.

Каких действий хотим мы от программы в случае нарушения ограничений? Эти ограничения не укладываются в рамки простых обнаружимых и исправимых ошибок времени исполнения (их обработка уже должна присутствовать в коде) и представляют собой, по-видимому, ошибку в логике программы. Есть несколько вариантов реакции программы:

- Сделать вид, что ничего не случилось, и надеяться на лучшее.
- Оштрафовать на месте и разрешить дальнейшее движение (например, напечатать диагностическое предупреждение или записать его в журнал).
- Сразу арестовать; запретить двигаться дальше (например, прервать выполнение программы контролируемым или неконтролируемым образом).

[1] Это не значит, что не нужно писать хорошую документацию.

Например, в C недопустимо вызывать функцию `strlen` с указателем на строку, равным нулю, поскольку указатель будет сразу разыменован. В этом случае подходят последние два варианта. Вероятно, лучше всего сразу прервать выполнение программы, потому что в незащищенных операционных системах разыменование нулевого указателя может привести к катастрофическим последствиям.

Существует несколько разных сценариев использования ограничений:

Входные условия

Эти условия должны быть выполнены *до* входа в раздел кода. Если входное условие не выполнено, это означает, что в коде клиента есть ошибка.

Выходные условия

Эти условия должны быть выполнены *после* выхода из блока кода. Если выходное условие не выполнено, это означает, что в коде поставщика есть ошибка.

Инварианты

Эти условия должны быть выполнены при достижении в ходе выполнения программы определенной точки, например между проходами цикла, при вызове методов и т. п. Невыполнение инварианта означает, что в логике программы есть ошибка.

Операторы контроля

Это любое другое утверждение относительно состояния программы в данный момент.

Первые два из перечисленных сценариев трудно реализовать при отсутствии необходимой поддержки в языке – если у функции несколько точек выхода,[1] то проверка условий выхода становится хлопотной. Eiffel поддерживает входные и выходные условия в базовом языке и может также гарантировать отсутствие побочных эффектов проверок ограничений.

При всей своей утомительности правильные ограничения, оформленные в виде кода, делают программу понятнее и облегчают ее сопровождение. Такую технику называют также *проектированием по контракту*, поскольку ограничения образуют неизменяемый контракт между разделами кода.

Какие ограничения налагать

Есть несколько проблем, которые можно решать с помощью ограничений. Например, можно делать следующее:

- Проверять, чтобы все обращения к массивам не выходили за их границы.

[1] Спор о том, можно ли разрешать функции иметь несколько точек выхода, является религиозным.

- Контролировать неравенство нулю указателей перед их разыменованием.
- Проверять допустимость параметров функций.
- Контролировать результаты функций, прежде чем их возвращать.
- Проверять состояние объекта перед операциями с ним.
- Защищать те участки кода, где должны быть комментарии. *Они не должны получать управление*.

Первые два случая особенно актуальны для C/C++. В Java, C# и некоторых других языках есть встроенные средства, помогающие избежать эти ловушки.

Каким должен быть объем проверок ограничений? Ставить проверки на каждой второй строке было бы излишеством. Как и в других случаях, чувство меры приходит с опытом. Что лучше – перебрать или недобрать? Избыток проверок может затуманить логику программы. «Читаемость – лучший из критериев качества программы: если ее легко читать, то, скорее всего, это хорошая программа; если ее трудно читать, то, скорее всего, это нехорошая программа». (Kernighan Plaugher 76)

На практике достаточно бывает разместить проверки входных и выходных условий в основных функциях плюс инварианты в основных циклах.

Снятие ограничений

Такого рода проверка ограничений обычно требуется только на этапах разработки и отладки программы. После того как мы воспользовались ограничениями, чтобы убедить себя (возможно, ошибочно) в правильности логики программы, следовало бы убрать их, чтобы готовая программа не делала лишней работы.

Это вполне достижимо благодаря чудесам современных технологий. В стандартных библиотеках C и C++ есть механизм для реализации ограничений – оператор контроля `assert`. Оператор `assert` действует как процедурный защитный экран, проверяя логику своего аргумента. Его назначение – уведомить разработчика о том, что программа ведет себя некорректно. При этом он не должен запускать никакого кода, взаимодействующего с клиентом. Если контрольное условие выполнено, исполнение кода продолжается. В противном случае программа аварийно завершается, генерируя сообщение об ошибке. Например:

```
bugged.cpp:10: int main(): Assertion "1 == 0" failed.
```

Оператор `assert` реализован в виде макроса препроцессора и поэтому больше подходит для C, чем для C++. Существует ряд библиотек операторов контроля, более ориентированных на C++.

Чтобы использовать `assert`, необходимо подключить `#include <assert.h>`. Затем можно вставлять в свои функции что-то типа `assert(ptr != 0);`. Препроцессор позволяет убрать контрольные проверки при окончатель-

ной сборке программы путем передачи компилятору флага NDEBUG. В результате все операторы assert будут удалены и их аргументы не будут вычисляться. Это значит, что на эффективность работы окончательной сборки программы операторы контроля не будут оказывать никакого влияния.

Следует ли убрать проверки вообще или просто деактивировать их – вопрос спорный. Одни считают, что если их удалить, то в тестировании будет участвовать уже *совсем другой* код.[1] Другие говорят, что наличие в окончательной версии продукта операторов контроля неприемлемо и их нужно полностью удалить. (Но кто проводил профилирование программ, чтобы доказать это?)

Во всяком случае, операторы контроля не должны иметь побочных эффектов. Что будет, например, если вы по ошибке напишете:

```
int i = взятьНекоеЧисло();
assert(i = 6); // да... аккуратнее нужно быть!
printf("i is %d\n", i);
```

Очевидно, что при отладке оператор контроля никогда не сработает: его значение = 6 (для C это все равно что *истина*). Однако при окончательной сборке строка assert будет полностью удалена, и printf выведет что-то другое. Таким образом могут возникать скрытые проблемы на поздних стадиях разработки. Не так просто защититься от ошибок в коде, проверяющем наличие ошибок!

Нетрудно представить себе ситуации, в которых операторы контроля могут иметь еще более скрытые побочные эффекты. Например, если контроль выглядит как assert(invariants());, где у функции invariants() есть побочный эффект, то его непросто обнаружить.

Поскольку операторы контроля могут быть удалены из окончательного кода, очень важно, чтобы при контроле выполнялась проверка ограничений и ничего больше. Действительные проверки состояний ошибки, таких как неудачное выделение памяти или проблемы в файловой системе, должны проводиться в обычном коде. Вы же не собираетесь изъять их из своей программы? Законные ошибки времени исполнения (сколь бы нежелательны они ни были) должны обнаруживаться защитным кодом, который нельзя удалить.

В Java реализован аналогичный механизм контроля.[2] Он активизируется средствами управления JVM и генерирует исключение (java.lang.As-

---

[1] На практике окончательная версия и версия разработки могут иметь еще больше отличий, например в отношении уровня оптимизации или включения символов отладки. В этом случае окончательная версия может иметь тонкие отличия, мешающие проявиться другим ошибкам. Даже на самых ранних этапах разработки тестирование нужно проводить в равной мере и для версии разработчика, и для окончательной сборки.

[2] Он появился в JDK 1.4 и отсутствует в более ранних версиях.

sertionError), не прерывая сразу выполнение программы. В .NET механизм контроля реализован посредством класса Debug.

Если вы обнаружили и исправили ошибку, очень полезно вставить операторы контроля в исправленное место. Это поможет вам не наступить на те же грабли вторично. По крайней мере, останется полезное предупреждение для тех, кто будет потом сопровождать ваш код.

В C++/Java есть стандартный прием задания ограничений классов путем добавления в каждый класс одной функции-члена `bool invariant()`. (Естественно, что эта функция не должна давать побочных эффектов.) После этого в начале и в конце каждой функции-члена можно поместить `assert`, вызывающий этот инвариант. (По очевидным причинам операторов контроля не должно быть в начале конструктора или конце деструктора.) Например, инвариант класса `circle` может проверять условие `radius != 0`; в противном случае объект находился бы в недопустимом состоянии, чреватом ошибками в последующих вычислениях (например, вызовом деления на нуль).

Агрессивное программирование?

Лучший способ защиты – нападение.

Поговорка

Работая над этой главой, я подумал: а какой стиль служит противоположностью *защитного программирования*? Разумеется, *агрессивное программирование*!

Я знаю нескольких программистов, которых можно было бы назвать «агрессивными». Но мне кажется, что дело тут не ограничивается руганью перед компьютером и нежеланием пользоваться душем.

Разумно предположить, что агрессивному программированию должно быть свойственно активное стремление взломать код, а не воздвигать защиту от возможных проблем. Иными словами, не защищать код, а активно атаковать его. Я бы назвал это тестированием. Как будет показано в разделе «Кто, что, когда, зачем?» на стр. 187, правильно организованное тестирование оказывает огромное положительное влияние на производство программного обеспечения. Оно заметно повышает качество кода и стабилизирует процесс разработки.

Всем программистам следовало бы быть агрессивными.

Резюме

Начерпай воды на время осады; укрепляй крепости твои;
пойди в грязь, топчи глину, исправь печь для обжигания кирпичей.

Наум 3:14

Необходимо писать не просто корректный, но хороший код. Для этого нужно документировать все сделанные предположения. В результате код станет легче сопровождать и в нем окажется меньше места для ошибок. Защитное программирование ориентировано на то, чтобы быть готовым к худшему, что может случиться. Данная технология препятствует превращению случайных огрехов в трудноуловимые ошибки.

Систематическое применение ограничений наряду с защитным кодом делает программное обеспечение гораздо более надежным. Как и многие другие полезные приемы в кодировании (например, тестирование компонентов, см. «Типы тестирования» на стр. 195), защитное программирование исходит из того, что стоит заранее потратить немного дополнительного времени, чтобы избежать больших расходов времени, сил и средств в будущем. Поверьте мне, от этого *может* зависеть судьба проекта в целом.

Хорошие программисты...

- Заботятся о надежности своего кода
- Стремятся учесть все сделанные допущения в защитном коде
- Требуют четкого поведения при вводе некорректных данных
- Тщательно продумывают код, который пишут
- Пишут код, устойчивый к глупостям, которые совершают другие люди или они сами

Плохие программисты...

- Стараются не думать о том, какие неприятности могут возникнуть при выполнении их кода
- Выдают код, интеграция которого чревата возникновением ошибок, и надеются, что кто-то другой с этим разберется
- Хранят в собственной памяти важные сведения об особенностях своего кода, которые легко забываются
- Не обдумывают создаваемый ими код, что приводит к появлению непредсказуемых и ненадежных программ

См. также

Глава 8. Время испытаний

Агрессивное программирование – и этим все сказано.

Глава 9. Поиск ошибок

Когда ваши средства защиты не в силах предотвратить отказы, требуется стратегия борьбы с ними.

Глава 12. Комплекс незащищенности

Защитное программирование – важнейшая технология создания надежных программных систем.

Глава 19. Спецификации

Нужно документировать предусловия и постусловия; как еще кто-то другой может узнать об их существовании? Когда указаны ограничения, можно вставлять защитный код, который проверяет их выполнение.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 581.

Вопросы для размышления

1. Можно ли *переусердствовать* с защитным кодом?

2. Нужно ли добавлять операторы контроля в каждую точку, где выявлена и исправлена ошибка?
3. Следует ли исключать из окончательных сборок операторы контроля посредством условной компиляции? Если нет, то какие контрольные операторы стоит оставить в окончательной сборке?
4. Что обеспечивает лучшую защиту – обработка исключений или операторы контроля в стиле C?
5. Следует ли *включать* защитную проверку пред- и постусловий в каждую функцию или только при *вызове* важных функций?
6. Насколько ограничения соответствуют идеалу защитного средства? Каковы их недостатки?
7. Можно ли обойтись *без* защитного программирования?
   a. Можно ли изобрести язык, который устранит необходимость в защитном программировании? Как это сделать?
   b. Следует ли из этого, что C и C++ порочны, поскольку они оставляют слишком много простора для возможных проблем?
8. Какого рода код не должен вызывать заботы о защитном программировании?

Вопросы личного характера

1. Насколько тщательно вы обдумываете каждый оператор, который вводите? Проверяете ли вы безжалостно код возврата для каждой функции, даже если *уверены*, что она не может возвратить ошибку?
2. Указываете ли вы входные и выходные условия в описаниях функций?
   a. Присутствуют ли они в неявном виде в описании работы функции?
   b. Если входные и выходные условия отсутствуют, отмечаете ли вы это явно в документации?
3. Многие компании на словах одобряют защитное программирование. Рекомендуется ли придерживаться этой стратегии в ваших условиях? Изучите исходные коды – соблюдаются ли рекомендации на практике? Насколько распространена проверка допущений с помощью операторов контроля? Насколько тщательно осуществляется проверка ошибок в каждой функции?
4. Насколько вы по своему характеру недоверчивы? Пересекая дорогу, смотрите ли вы в обоих направлениях? Можете ли вы заставить себя делать неприятную работу? Проверяете ли вы свой код на все потенциальные ошибки, какими бы маловероятными они ни казались?
   a. Насколько легко делать это систематически? Не забываете ли вы о необходимости проверки ошибок?
   b. Можно ли каким-то образом облегчить себе написание защитного кода, более скрупулезно проверяющего ошибки?

2

Тонкий расчет

Расположение и представление исходного кода

В этой главе:

- В чем важность представления кода и какое расположение считать хорошим
- Как выбрать стиль расположения кода
- Выработка стандарта кодирования
- Религиозные войны – почему следует их избегать

Не судите по наружности,
но судите судом праведным.

Иоанн 7:24

Стиль кодирования всегда был, есть и будет причиной священных войн между программистами (профессионалами, любителями и студентами), в которых, к сожалению, существенные разногласия часто выливаются в обычную брань: «Я вас, дураков, научу правильно расставлять скобки!».

Самая первая из компаний, в которых я работал, попыталась разработать свой внутренний стандарт кодирования. Предполагалось, что будет создан руководящий документ, охватывающий несколько языков программирования и определяющий стандартные соглашения и лучшие способы. Спустя несколько месяцев после начала работы группа разработчиков документа все еще не могла договориться о том, где лучше

ставить скобки в C. Сомневаюсь, что кто-либо следовал тому стандарту, который был выработан в итоге.

Почему это так *волнует* людей? Как будет показано, представление оказывает очень сильное влияние на читаемость кода, и никто не захочет работать с кодом, который тяжело читать. Кроме того, представление воспринимается очень субъективно и индивидуально: *вам* может не понравиться тот стиль, от которого *я* в восторге. Знакомый стиль вызывает ощущение комфортности, а непривычный раздражает.

Программисты неравнодушно относятся к коду, поэтому его представление служит источником сильных переживаний.

Да в чем проблема?

Расположение и представление кода оказываются предметом разногласий в большинстве современных языков программирования. Произвольное форматирование, позволяющее выразить свою художественную индивидуальность, вошло *в моду* в начале 1960-х годов с появлением Алгола: существовавшие к тому времени версии Fortran допускали меньше свободы в формате. В последующем лишь очень немногие языки отошли от такого подхода к свободе форматирования.

Стиль представления кода охватывает на удивление широкий круг проблем, наиболее очевидной и, пожалуй, самой спорной из которых оказывается расстановка скобок.[1] Более широкие вопросы стиля кодирования, такие как правила именования функций и переменных, в сочетании с другими проблемами, такими как структура программы (например, *не применять goto* или *писать функции только с одним входом и одним выходом*), определяют стиль, используемый при написании программы. Все это в совокупности образует ваш *стиль кодирования*.

Несмотря на многочисленность вариантов выбора формата представления кода, все они имеют лишь эстетическое значение. По определению представление не имеет никакого синтаксического или семантического значения и игнорируется компилятором.

Тем не менее представление оказывает реальное влияние на качество кода. Понимание программистом кода при чтении зависит от его расположения. Последнее может выявлять структуру кода и облегчать понимание его работы. Разумеется, может происходить и обратное: смысл кода затеняется и запутывается. Как бы хорошо ни была спроектирована ваша программа, с ней будет неприятно работать, если она выглядит, как помойка. Однако плохое форматирование может не только мешать пониманию кода, но и *скрывать* имеющиеся в нем ошибки. В качестве простого примера приведем следующий код на C:

---

[1] Имеются в виду фигурные скобки { и }, весьма распространенные в языках типа C.

```
int error = doSomeMagicOperation();
if (error)
    fprintf(stderr, "Error: exiting...\n");
    exit(error);
```

Из расположения кода понятно, чего добивался автор, но при запуске этой программы его ждет сюрприз.

Будучи сознательными мастерами и приверженцами высококачественного кода, мы стремимся к ясному представлению. При разработке программного обеспечения и без того есть на чем споткнуться, так не будем создавать дополнительные проблемы из-за представления кода.

Знайте своих клиентов

Чтобы написать эффективный исходный код, необходимо понимать, *кто* его будет читать. Если придется кого-то поставить в трудное положение, выясните, перед кем потом извиняться. В действительности, есть три круга адресатов вашего кода:

Вы сами

У меня такой плохой почерк, что иногда *мне самому* трудно его разобрать. Это практически невозможно, если я не ставлю себе задачу писать разборчиво. То же самое происходит и с кодом. Созданное вами должно быть понятно не только сразу после написания, но и по прошествии лет. Кто мог подумать, что придется вернуться к (относительно) архаичному коду на COBOL, чтобы исправить в нем ошибки, связанные с Y2K?

Компилятор

Компилятору все равно, как выглядит ваш код, – лишь бы в нем не было синтаксических ошибок. Ему абсолютно безразлично, *какую задачу* этот код решает. Можно подробно описать в комментариях, что *должна* делать функция, но компилятор никогда не сообщит вам, действительно ли она выполняет то, что сказано в комментариях. Если код корректен, среда разработки будет вполне удовлетворена.

Прочие лица

Это самая важная аудитория, интересы которой часто меньше всего учитываются.

Вам кажется, что хотя вы работаете в составе команды, никто никогда не станет смотреть ваш код. Это ошибочное мнение.

Вы пишете какой-то код для себя дома. Стоит ли заботиться о его красоте? Если «да», то в чем польза? В выработке качеств, которые делают вас профессионалом. У вас появляется прекрасная возможность продемонстрировать настоящую дисциплину в проекте, когда никто не оказывает на вас давления. Это возможность выработать хорошие манеры. Если вы не справитесь с задачей в таких условиях, то неудивительно, что у вас не хватит дисциплины в *реальных* проектах.

Ваш исходный код – это документ, который описывает создаваемую программу. Он должен быть понятен любому, кто захочет к ней снова вернуться. Это может быть аудитор (рецензент) написанного вами кода или тот, кто будет его сопровождать. Позаботьтесь о тех, кому придется читать ваш код – представьте себя на их месте.

Детали стиля представления формируются с учетом предполагаемой аудитории. Как может аудитория повлиять на расположение кода? Как ни удивительно, менее всего нас должен волновать компилятор. Его задача – игнорируя все лишние пробельные символы, заняться серьезным делом интерпретации синтаксиса. Представление не влияет на синтаксический смысл, и компилятор справится с самым уродливым расположением.

Расположение кода решает задачу демонстрации его *логической структуры* людям, читающим код. Это передача информации, и чем понятнее код, тем лучше.

Поймите, кто реально станет читать ваш исходный код: другие программисты. Пишите с расчетом на них.

Что такое хорошее представление?

Как можно понять, хорошее представление не ограничивается одной аккуратностью. Опрятно выглядящий код, несомненно, производит впечатление хорошего качества, но аккуратный код может в то же время вводить в заблуждение. Мы стремимся к *ясности* представления; стратегия выбора отступов должна *подчеркивать* структуру кода, а не затемнять ее. Если некий алгоритм сложен по своей природе, то расположение кода должно облегчать его чтение. (Если вы написали последовательность команд, сложность которой не вызывается необходимостью, следует немедленно ее изменить.)

Расположение кода должно передавать его смысл, а не скрывать. Я предлагаю следующие критерии качества стиля представления кода.

Единообразие

Принципы отступов в коде должны быть одинаковы во всех частях проекта. Не меняйте стиль в середине пути. Это не только выглядит непрофессионально, но и может ввести в заблуждение, создавая впечатление отсутствия связи между файлами с исходным кодом.

Отдельные правила представления должны быть внутренне непротиворечивыми. Расположение фигурных и квадратных скобок и т. п. в различных ситуациях должно соответствовать единым правилам. Количество пробелов в отступах должно быть всегда одинаковым.

Керниган и Ричи, основоположники C, подчеркивают важность правильных отступов и затем говорят: «Положение скобок менее важно, хотя люди склонны проявлять фанатизм в таких вопросах.

Мы выбрали один из нескольких популярных стилей. Выберите тот стиль, который вам больше подходит, и точно ему следуйте». (Kernighan Ritchie 88)

Стандартность

Разумно принять какой-нибудь из господствующих ныне стилей, а не изобретать собственные правила отступа. Так будет проще для тех, кто станет читать ваш код. И меньше шансов, что ваш стиль вызовет у них отвращение.

Краткость

Можете ли вы кратко описать свою стратегию отступов? Подумайте над этим. Если вы делаете *нечто*, пока не произойдет *то-то и то-то*, а тогда вы будете делать *это*, если выполняется *X*; в противном случае вы станете делать нечто другое в зависимости от...

Кому-то может потребоваться дополнить написанный вами код, и делать это ему следует, придерживаясь того же стиля. Если его трудно ухватить, можно ли считать такой стиль представления удачным?

Размещение скобок

Для иллюстрации влияния, которое представление оказывает на исходный код, и компромиссов, на которые приходится идти при выборе стиля, разберем конкретный случай, связанный с важной проблемой расположения кода C. Рассматривая варианты, возникающие в этой одной простой области, мы продемонстрируем важность представления и степень его влияния на код.

Размещение фигурных скобок представляет собой большую проблему в некоторых языках, хотя это лишь малая часть всей задачи правильного расположения кода. Поскольку оно сразу бросается в глаза, то вызывает процентов 80 всех споров. В других языках есть свои аналогичные проблемы размещения.

Существует несколько стандартных стилей размещения скобок. Ваш выбор среди них определяется вашими представлениями о прекрасном, о культуре, внутри которой вы пишете код, и имеющимися у вас привычками. Уместность того или иного стиля может определяться контекстом – сравните, например, журнальную статью и редактор исходных текстов (см. врезку «Правильное представление» на стр. 59). Возможно, вы предпочитаете стиль без отступов, но в журнале вы вынуждены прибегать к стилю K&R, чтобы максимально использовать площадь страницы.

Скобки в стиле K&R

Стиль *K&R* относится к старейшим, будучи предложенным основоположниками языка C, Керниганом и Ричи, в книге «Язык программирования C» (Kernighan Ritchie 88). По этой причине его часто считают

исходным и лучшим. На него повлияла необходимость отобразить как можно больше информации на маленьком экране. Пожалуй, это превалирующий стиль для кода Java.

```
int k_and_r() {
    int a = 0, b = 0;
    while (a != 10) {
        b++;
        a++;
    }
    return b;
}
```

За

- Занимает мало места, поэтому на экране можно отобразить больше кода.
- Замыкающая скобка выравнивается с соответствующим ей оператором, поэтому, поднявшись взглядом, легко найти конструкцию, которую она завершает.

Против

- Скобки располагаются не по одной линии, поэтому зрительно тяжело обнаружить парные.
- Можно не заметить открывающую скобку, вышедшую за правый край страницы.
- Операторы кода выглядят слишком тесно упакованными.

Расширенный стиль скобок

Более свободное расположение кода достигается в *расширенном* (*extended*) стиле, называемом также стилем *Олмана*. Лично мне он нравится больше прочих.

```
int exdented()
{
    int a = 0, b = 0;
    while (a != 10)
    {
        b++;
        a++;
    }
    return b;
}
```

За

- Понятный, упорядоченный формат.
- Легко обнаруживаются открывающие скобки, поскольку они располагаются отдельно. В результате каждый блок кода лучше выделяется.

Против

- Требует больше вертикального пространства.
- Расточителен, если многие блоки содержат всего по одному оператору.
- Некоторых раздражает сходство с Pascal.

Правильное представление

Представление для кода нужно выбирать в зависимости от контекста, в котором он будет рассматриваться. Различных контекстов существует больше, чем может показаться на первый взгляд. Когда вы читаете некий код, полезно определить, чем было обусловлено выбранное для него представление. Обычно средой обитания кода могут быть:

Редактор исходного кода

Это естественная среда обитания почти любого кода. Ей свойственны все проблемы представления, автоматически возникающие перед программистами. Код должен читаться с экрана компьютера, обычно в каком-то специальном редакторе или среде разработки. Путем прокрутки или перемещения по файлу нужно найти интересующее место. Среда интерактивна: как правило, код читают для того, чтобы внести в него изменения. Таким образом, код должен быть доступен для переработки.

Редактор должен предоставлять горизонтальные полосы прокрутки для просмотра длинных строк или переносить строки, если ширина страницы ограничена. Часто для облегчения восприятия синтаксис выделяется цветом. При наборе кода редактор автоматически выполняет некоторые виды форматирования. Например, пытается правильно установить курсор при переходе на новую строку.

Публикуемый код

Если только вы не живете в каком-то замкнутом пространстве, огороженном от остального мира, то должны периодически читать опубликованный код. Существует множество источников: листинги в книгах или журналах, фрагменты кода в библиотечной документации или строки кода в сообщениях телеконференций. Их формат нацелен как на облегчение понимания, так и на экономию занимаемого места, которое стоит денег. Поэтому строки сжаты по вертикали, чтобы вместить больше кода в ограниченное пространство, и сжаты по горизонтали, чтобы уместиться на узких страницах.

В таком коде часто опускают обработку ошибок и все, что не относится к главной идее приводимого примера. Нужно передать смысл, не вникая в детали.

Возможно, вам никогда не придется писать код для такого рода применений, но вы, несомненно, будете часто встречаться с ним (хотя бы читая фрагменты кода в этой книге). Следует учитывать, для каких условий пишется такой код и чем он отличается от обычного, чтобы не перенять ненароком дурной стиль.

Распечатки

При распечатке кода проекта возникают новые проблемы. Что лучше – предварительно отформатировать текст, масштабировать страницы и применить мелкие шрифты или разрешить беспорядочные переносы на новую строку? Возможность выделения синтаксиса цветом отсутствует (если только средства не позволяют вам приобрести цветной принтер и соответствующие расходные материалы), поэтому беспорядочные комментарии или отключение фрагментов кода путем заключения их в комментарии затрудняют восприятие.

Даже если вам не требуется распечатывать исходный код, всегда следует иметь в виду эти проблемы.

Стиль Уайтсмита (с отступами)

Стиль *с отступами* менее распространен, хотя и встречается. Фигурные скобки при этом имеют тот же отступ, что и код. Он получил название стиля *Уайтсмита*, поскольку применялся в примерах для Whitesmiths C – раннего компилятора с языка C.

```
int indented()
    {
    int a = 0, b = 0;
    while (a != 10)
        {
        b++;
        a++;
        }
    return b;
    }
```

За

- Соединяет блоки кода со скобками, в которые они заключены.

Против

- Многим не нравится привязка блоков к своим скобкам.

Другие стили скобок

Существуют и другие стили. Например, стиль *GNU* занимает промежуточное положение между расширенным стилем и стилем с отступами: скобки располагаются посередине каждого уровня отступа. Есть и гибридные стили: стиль кода ядра Linux наполовину представляет K&R, наполовину – стиль Олмана. Большинство программирующих на C# тоже комбинирует стили расположения. Можно извратиться и так:

```
  int my_worst_nightmare()
  {
int a = 0, b = 0;
while (a != 10) {
        b++;
        a++;
        }
return b;
  }
```

Мне доводилось видеть немало такого сюрреалистического кода, и вы сами при желании сможете соорудить что-нибудь столь же кошмарное.

Выясните, какие стандарты оформления кода есть для выбранного вами языка, и освойте каждый из них на практике. Оцените преимущества и недостатки каждого.

Единственно верный стиль

Поняв, что такое правильный стиль кодирования, на что он влияет и почему необходим, нужно выбрать для себя что-то подходящее. Вот тут и начинается столкновение. Последовательности одного религиозного течения в представлении ведут борьбу с проповедниками другого, порождая гражданские войны между программистами. Но настоящий мастер не ввязывается в эти мелкие дрязги, предпочитая взвешенный подход.

Если вы пишете код, пользуясь правильным стилем, неважно, как он называется. И спорить бессмысленно. Есть много правильных стилей, а качество и уместность каждого из них зависят от контекста и культуры.

Выберите какой-то один правильный стиль кодирования и придерживайтесь его постоянно.

Можно предположить, что если бы стандарт языка определял Единственный правильный стиль представления, то жить было бы существенно легче. В самом деле, весь код выглядел бы единообразно. Вместо того чтобы ссориться, мы бы занялись чем-то полезным. Сразу можно было бы схватывать суть кода, написанного любым другим программистом. Заманчиво, не правда ли?

В противовес этому следует заметить, что *конкуренция – ценная вещь*. Если бы существовала монополия одного стиля кодирования, как можно было бы узнать, что он – лучший? Наличие нескольких стилей кодирования заставляет нас думать и совершенствовать способы применения стилей. Стимулируется развитие принципов применения стилей. Итог: код, который мы пишем, становится лучше.

Однако это *не* довод в пользу программирования в своем особом личном стиле. Помните, что правильное представление должно быть *обычным* – таким, к которому привык читатель.

Стандарты кодирования

Существует ряд известных стандартов кодирования, используемых повсеместно.

Indian Hill

Полное название этого знаменитого документа – *Indian Hill Recommended C Style and Coding Standards*. Он не имеет никакого отношения к коренным жителям Америки, гордо стоящим на холмах; название происходит от прославленной лаборатории Indian Hill AT&T Bell.

GNU

Стандарты кодирования GNU имеют важное значение, поскольку они касаются большей части существующего бесплатного или с открытым исходным кодом программного обеспечения.

Их можно найти на веб-сайте проекта GNU (*www.gnu.org*).

MISRA

Британская Ассоциация разработчиков безотказного ПО для автомобильной промышленности (Motor Industry Software Reliability Association – MISRA) разработала известный набор стандартов для написания на C критически важного встроенного программного обеспечения. В него входит 127 правил, и есть ряд инструментов, с помощью которых можно проверить код на соответствие этим правилам. Правила в большей мере ориентированы на применение языка, нежели на оформление кода.

Проект *foo*

Почти для каждого существующего проекта установлен свой излюбленный стиль кодирования. Посмотрите вокруг, и вы обнаружите их тысячи. Например, для ядра Linux есть свои правила, так же как и для проекта Mozilla.

Внутрифирменные стили (и когда их придерживаться)

Во многих компаниях, разрабатывающих программное обеспечение, есть свой *фирменный* стиль кодирования, в котором среди прочего определены правила форматирования кода. Но зачем он нужен, ведь любой код, написанный в «правильном» стиле, легко читать и сопровождать? Зачем нужны эти бюрократические навороты, если понимание кода не вызывает трудностей?

Важность и полезность внутрифирменных стилей обусловлена рядом причин. Когда все танцуют под одну дудку, исходный код оказывается полностью единообразным и однородным. Что в этом хорошего? Улучшается качество кода и растет надежность разработки программного обеспечения. И вот почему:

- Любой код, выходящий за пределы организации, имеет аккуратное представление и согласованность, создавая впечатление хорошей продуманности. Наличие в одном проекте разнородных стилей создает впечатление неряшливости и непрофессионализма.
- Компания может быть уверена в том, что программы пишутся согласно единому стандарту, включающему в себя определенные идиомы и методологии. Это не гарантирует получение хорошего кода, но дает некоторую защиту от появления плохого кода.
- Компенсируются недостатки инструментария: по разному настроенные IDE будут вступать в конфликт между собой, разрывая код на части и досаждая форматированием. Стандарт создает ровную почву (и общего врага для всех программистов).
- Привлекательна возможность сразу оценить состояние кода, написанного коллегами, и быстро сделать необходимые изменения при сопровождении. Меньше времени тратится на чтение, а значит, берегутся финансовые ресурсы компании.
- Поскольку программисты перестанут непрерывно заново форматировать код в соответствии со своими эстетическими пристрастиями, система управления версиями станет намного эффективнее. Если один программист будет переформатировать код второго, чтобы привести его к «своему» стилю, это отразится на работе утилит, сравнивающих версии. Многие из них действуют довольно грубо и покажут массу несущественных различий в расстановке пробельных символов и скобок.

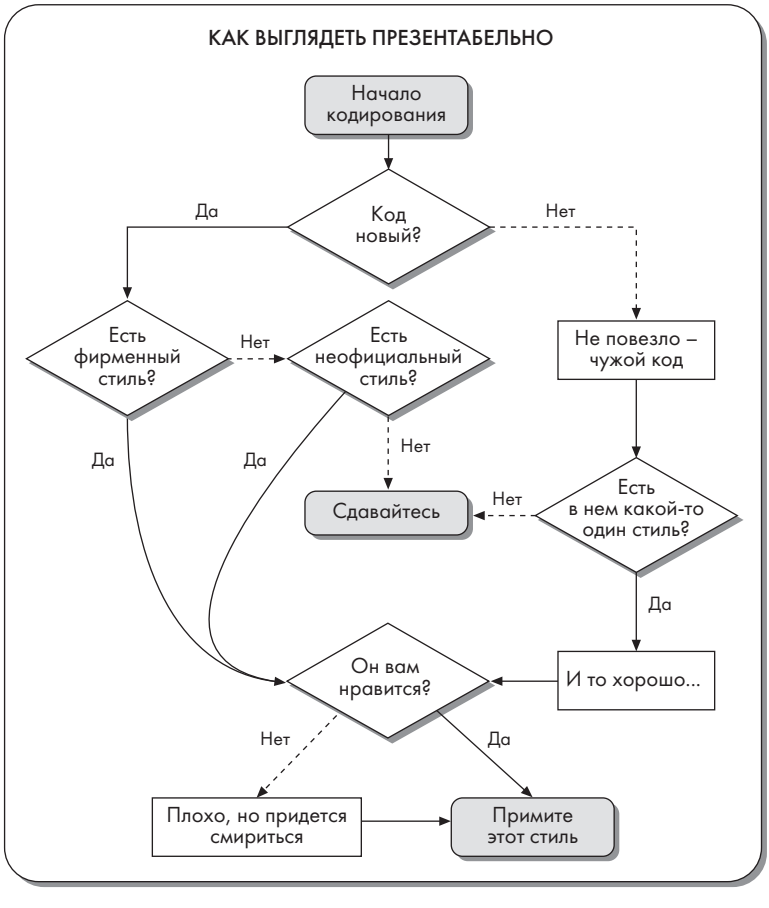
Такие внутрифирменные стандарты кодирования следует приветствовать. Даже если вы не согласны с предъявляемыми ими требованиями, например вам кажется, что ваши личные правила соблюдения отступов более привлекательны и понятны, это не должно иметь никакого значения. Выгода от наличия общего для всех стиля превосходит

ваши неудобства от необходимости его придерживаться. Если вы не согласны с принятым стандартом, все равно нужно его выполнять.

Если в вашей группе принят некий стандарт кодирования, придерживайтесь его. Свой собственный излюбленный стиль оставьте в стороне.

Ваш собственный стиль в большой мере основан на привычке и долгой практике. Поработав некоторое время в стиле, который принят в вашей фирме, вы обнаружите, что он стал для вас естественным и не вызывает неудобств.

Что делать, если нужно поработать над кодом, написанным вне стен вашей компании и в другом стиле? В таком случае благоразумнее *сохранить* этот стиль при написании нового кода. (Вот почему очень полезно работать в стиле, который легко распознать постороннему.) Альтернатива – переработать этот и прочие полученные вами файлы в стиле, принятом в вашей компании. В большинстве практических



случаев такой вариант трудноосуществим, особенно если вы регулярно получаете все новые и новые файлы из одного источника.

Придерживайтесь стиля полученного файла или проекта либо стиля вашей фирмы в тех случаях, когда между ними нет противоречий, а собственными предпочтениями пожертвуйте. Однако не отказывайтесь бездумно от своего стиля: взвесьте преимущества и недостатки одного и другого. А если в вашей компании нет внутреннего стандарта? Добейтесь, чтобы он был...

Установка стандарта

Вы получили задание предложить стиль форматирования кода в условиях отсутствия общепринятого в вашей фирме. Удачи вам! Можете не сомневаться, что у всех окажется собственное мнение по поводу того, каким должен быть этот стиль, а конечный результат полностью не удовлетворит никого. Таковы уж эти специалисты.

Выработка стандарта кодирования – задача тонкая, и решать ее нужно тактично, но непреклонно. Почему? Потому что приказной стиль в отношении программистов не принесет популярности ни вам, ни вашему стандарту. Если вам не удастся убедить их в важности стандарта, они не примут его и будут по-прежнему придерживаться своих индивидуальных способов оформления кода.

Сложность задачи зависит от особенностей членов вашего коллектива:

- Сколько всего программистов
- Как каждый в отдельности пишет код
- Насколько схожи между собой их стили программирования
- *Заинтересованы* ли они в наличии стандарта
- Готовы ли они к тому, чтобы изменить свой стиль

Если их стили программирования относительно близки, задача решается легко. Если они сильно различаются, вам предстоит нелегкая работа. Единодушия по поводу того, какой стиль лучший, обычно не наблюдается, но при этом признается, что одни стили лучше других. Вы должны постараться предложить достаточно детальные рекомендации по оформлению кода, с которыми согласилось бы возможно большее число программистов *и* которые облегчили бы им возможность совместной работы. Вот ряд практических советов для выполнения этого геркулесова труда:

Зачем это нужно?

Начните с уяснения объема предстоящей работы. На кого рассчитан будущий стандарт – на отдельную группу, подразделение компании или всю компанию в целом? От этого в большой степени зависит разработка и реализация.

Запомните: то, что хорошо для одного, не обязательно окажется лучшим стилем для группы программистов в целом. Разрабатываемые вами рекомендации нацелены не на удовлетворение *ваших* личных эстетических потребностей, а на то, чтобы стать стандартом кода, разрабатываемого целой группой, и облегчить взаимодействие в ней. Не упускайте это из виду, когда будете работать над стандартом.

Определите степень детализации, которой вы себя ограничите. Следует ли включить в итоговый документ только правила форматирования кода или также рекомендации по использованию средств языка? Поступите проще: один документ посвятите форматированию, а другой – применению языка.

Добейтесь поддержки

Постарайтесь вовлечь в работу всех членов своей группы, чтобы *они* чувствовали свою причастность. Если программисты будут считать себя соавторами стандарта, больше вероятность, что они станут его придерживаться.

- *Прежде чем* начать работу над стандартом, убедите всех в его необходимости. Постарайтесь объяснить всем преимущества единого стиля кода и неприятности, связанные с его отсутствием.
- Когда программистов достаточно много, *не* пытайтесь разрабатывать стандарт путем обсуждения на комиссии. Или, по крайней мере, сначала спрячьте все острые предметы в помещении. Выберите для решения задачи небольшую рабочую группу.

Когда стандарт будет близок к завершению, соберите комиссию для его обсуждения и принятия. Председатель этой комиссии должен обладать правом окончательного решения, иначе дело застопорится, когда полтора десятка программистов погрязнут в религиозных дебатах.

Создайте итоговый документ

Конечный продукт должен представлять собой общедоступный документ, а не некий ряд достигнутых соглашений. На этот документ можно будет потом ссылаться и предлагать его для ознакомления новым сотрудникам. В документе должны быть приведены все правила – возможно, с обоснованием наиболее спорных из принятых решений.

Введите в стандарт лучшее из практики

Отразите в стандарте лучшие из уже существующих в команде технологий – подтвердите людям, что они на правильном пути. Когда в стандарте нет ничего неожиданного, его легче принять. Если же включить в него произвольные положения, не связанные с существующей практикой, они встретят сопротивление.

Сосредоточьтесь на существенном

Сконцентрируйте свои усилия на том, что действительно важно и что позволит в наибольшей мере усовершенствовать выпускаемый вашей командой код. Не занимайтесь разработкой стандартов представления для C, C++ *и* Java, если вы пользуетесь в своей работе только C.

Избегайте опасных конфликтов

Оставьте редкие, но тяжелые случаи на личное усмотрение каждого, если фактически они не слишком существенны. Если кого-то очень волнует проблема разбиения оператора `if` на несколько строк, не спорьте и дайте людям возможность поступать так, как они хотят.

Не будьте слишком строги: иногда нарушение правил может быть вполне оправданным.

Действуйте поэтапно

Разумно создавать внутрифирменный стиль *по частям*. Для начала договоритесь о расстановке скобок и отступах – всего лишь. Даже этого добиться будет нелегко! Решив этот вопрос, двигаться дальше уже *значительно* легче; все остальные нововведения будут примерно в том же духе. В какой-то момент, когда код станет достаточно упорядоченным, можно прекратить придумывать новые правила.

Планируйте принятие

Составьте четкое представление о том, как принимать новый стандарт. Нужно быть реалистом. Стандарт должен *облегчить* людям жизнь, иначе им не станут пользоваться. Принятие стандарта должно быть в той или иной мере основано на мнении большинства: если Фред считает, что его формат оператора `switch` лучше, чем тот, с которым согласились все остальные, *тем хуже для Фреда*. Не пытайтесь осуществлять демократические процедуры – они неэффективны.

Не запугивайте стандартом и не вводите наказаний за отход от него. Это обычно плохо воспринимается. Лучше придумайте меры поощрения – хотя бы в виде похвалы при рецензировании кода.

В конечном счете, принятие стандарта зависит от авторитета тех, кто его внедряет. Либо его санкционируют сами программисты, либо руководству приходится осуществлять поддержку. Иначе это просто потеря времени.

Не правда ли, похоже на то, как уговаривают группу детей не ссориться и вести себя прилично? В результате после преодоления всех религиозных противоречий у вас на руках будет фирменный стандарт, который действительно окажет положительное влияние на качество выпускаемого вами кода. Когда стихнут обиды, вы не пожалеете о содеянном.

Религиозные войны?

Скорейший способ закончить войну – это проиграть ее.

Джордж Орвелл

Вести религиозные войны по поводу форматирования кода – пустая трата времени; есть гораздо более важные проблемы, заслуживающие вашего внимания. Но будьте осторожны: формат кода – не единственное больное место у программистов. Помимо него существуют выбор редактора, компилятора, методологии, Настоящего языка[1] и т. д.

Эти мелкие столкновения длятся годами. И будут происходить дальше. И победителя *никогда* не будет. Никто никогда не даст *правильный* ответ, потому что его не существует. Эти споры просто дают возможность одним людям попытаться навязать свое личное (тщательно сформированное) мнение другим, и наоборот. В конечном счете, мое мнение *не может* не быть правильным, потому что оно – *мое*. Это все равно что пытаться связать вместе макаронины – какое-то время забавно, но потом надоедает и совершенно бессмысленно. Обычно этим делом грешат не вполне зрелые программисты. (Ветераны уже выдохлись в этих спорах.)

Нужно понять главное: религиозные войны напрасно отнимают силы. Профессионалу следует воздерживаться от таких мелких споров. Конечно, не возбраняется иметь свое личное обоснованное мнение, но было бы дерзостью считать его единственно правильным.

Скажи «нет» религиозным войнам. Не вступай в них. Отойди в сторону.

Резюме

Ничто так не способствует успеху, как видимость успеха.

Кристофер Лаш

Представление – одна из главных характеристик, отличающих хороший код от плохого. Программист может многое узнать по внешнему виду кода, поэтому стоит позаботиться о его надлежащем форматировании. Важно уметь толково расположить код в соответствии с правилами существующего в фирме стандарта кодирования, обеспечив его максимальную понятность.

---

[1] На память приходит конференция по программированию на C/C++, участником которой я был несколько лет назад. Докладчик поделился своим открытием, что применение Pascal вместо C приводит к снижению количества ошибок (и облегчению их исправления), а труднее всего для исправления и многочисленнее ошибки при работе в C++. Реакция была замечательной – всеобщее возмущение!

Кто еще хочет подраться?

Формат кода – не единственный повод для сражений между программистами. Есть много религиозных вопросов, от которых лучше тактично уклоняться, сберегая свое здоровье. Остерегайтесь таких тем, как:

Моя ОС лучше твоей

...потому что она масштабируется от ручных часов до космического корабля инопланетян, требует перезагрузки раз в тысячу лет и выполняет почти все операции с помощью одной двухбуквенной команды.

Нет, *моя* лучше, потому что для работы с ней не нужно вводить никакого текста, в ней со вкусом подобраны цвета и справиться с ней может даже слепой. А то, чего в ней нельзя сделать, в большинстве цивилизованных стран все равно запрещено.

Мой редактор лучше твоего

...потому что он распознает миллион разных синтаксических систем, может редактировать файлы, написанные иероглифами, а любую из его 400 команд можно выполнить, нажав одновременно не более 10 клавиш. С ним можно работать на столе, из командной строки, через модем или электропроводку, а также замаскировавшись 128-разрядным шифром.

А *мой* все равно лучше, потому что встраивается в трусы и знает, что я хочу напечатать, еще до того, как я сам это пойму.

Мой язык лучше, чем твой

...потому что в нем реализован искусственный интеллект правительств большинства крупных стран и он настолько умен, что умеет интерпретировать произвольные телодвижения как осмысленный ряд команд.

А *мой* все равно лучше, потому что на нем можно писать хайку, а информация кодируется с помощью комбинаций пробельных символов.

Логично предположить, что тщательно структурированный код столь же тщательно спроектирован. С еще большей уверенностью можно утверждать, что неряшливо оформленный код и продуман был достаточно небрежно. Но дело не ограничивается способом форматирования исходного кода.

Способ оформления кода не единственное *качество*, которым хорошие программисты отличаются от плохих программистов. Мораль проста: *не нужно накалять атмосферу*. С этим справятся компьютеры (у нас в офисе нет отопления, потому что машины выделяют достаточно

тепла). Да, у вас есть свое мнение, и вы *готовы* отстаивать его и излагать свои взгляды, но не ставьте себе целью победить или доказать свою правоту, как не пытайтесь высокомерно продолжать действовать по-своему.

Хорошие программисты...

- Избегают бессмысленных споров и учитывают чужое мнение
- Достаточно скромны, чтобы понимать, что не всегда бывают правы
- Понимают значение формата кода для его удобочитаемости и стремятся писать как можно более понятный код
- Перейдут на корпоративный стиль, даже если он противоречит их личным представлениям

Плохие программисты...

- Обладают ограниченным кругозором и упрямы: *только моя точка зрения правильна*
- Вступают в спор со всяким и по всякому поводу; для них это способ показать свое превосходство
- Не выработали своего личного устойчивого стиля кодирования
- Переделывают чужой код в собственном стиле

См. также

Глава 3. Что в имени тебе моем?

Стандарт кодирования может определять способ образования имен.

Глава 4. Литературоведение

Хорошее представление – ключ к написанию самодокументирующегося кода.

Глава 5. Заметки на полях

Описываются способы написания комментариев; комментарии могут быть связаны с форматом кода.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 585.

Вопросы для размышления

1. Нужно ли менять формат старого кода, чтобы привести его в соответствие с новым кодом? Следует ли воспользоваться для этого инструментами переформатирования?
2. Распространена система, при которой исходный текст располагается в фиксированном количестве колонок. Каковы ее преимущества и недостатки? Есть ли в ней смысл?
3. *В какой мере* следует детализировать стандарт кодирования?
   a. Насколько строго следует придерживаться выбранного стиля? Какие наказания могут быть за уклонение от него?
   b. Допустим, что стандарт оказывается непомерно мелочным и ограничительным. Каковы могут быть последствия?
4. Если создается новый стиль представления, какое количество объектов или ситуаций следует регламентировать? Какие дополнительные правила представления следует установить? Перечислите их.
5. Что важнее – хорошее *представление* кода или хорошее *проектирование* кода? Почему?

Вопросы личного характера

1. У вас есть свой постоянный стиль?
   a. Работая с чужим кодом, какой стиль вы применяете – авторский или свой?
   b. В какой мере ваш стиль кодирования определяется автоматическим форматированием, применяемым в вашем редакторе? Служит ли это достаточной причиной для принятия определенного стиля?
2. Табуляция – дьявольское изобретение или благо? Объясните.
   a. Что вам известно о своем редакторе? Вставляет ли он символы табуляции автоматически? Какова величина табуляции в вашем редакторе?
   b. Некоторые *весьма* распространенные редакторы допускают отступы, состоящие одновременно из пробелов и табуляций. Затрудняет ли это сопровождение кода?
   c. Скольким пробелам должен соответствовать символ табуляции?
3. Есть ли у вас любимый формат кода?
   a. Опишите его несколькими простыми предложениями. Не упустите важное, например формат операторов `switch` и способ разбиения длинных строк.

   b. Сколько предложений вам потребовалось? Стало ли это для вас неожиданностью?
   c. Есть ли в вашей организации свой стандарт кодирования?
   d. Знаете ли вы, где его найти? Рекламируется ли он? Вы его читали?
      i. Если да: он вас удовлетворил? Сделайте честный анализ и сообщите свое мнение авторам документа.
      ii. Если нет: правильно ли это? (Обоснуйте свой ответ.) Существует ли общепринятый, но неписаный стиль кода? Можете ли вы оказать влияние на то, чтобы стандарт был принят?
   e. Нет ли у вас *нескольких* стандартов, например для разных проектов? Если да, то как в этих проектах используется общий код?
4. Много ли разных форматов кода вы испробовали?
   a. Какой из них вам больше пришелся по душе?
   b. В каком существовали наиболее строгие ограничения?
   c. Связано ли это между собой?

3

Что в имени тебе моем?

Существенные вещи должны иметь выразительные имена

В этой главе:

- Почему в качественном коде должны быть хорошие имена
- Что такое *хорошее имя?*
- Какие имена давать переменным, функциям, типам, пространствам имен, макросам и файлам

Когда я употребляю слово, – сказал Шалтай-Болтай довольно презрительно, – оно означает только то, что мне от него требуется, – не больше и не меньше.

Льюис Кэррол

В древних цивилизациях считали, что правильно назвать какую-то вещь – значит, овладеть ею. И это не было просто претензией на собственность. Иногда во власть имени верили так сильно, что ни за что не раскрыли бы своего имени незнакомцу, чтобы тот не смог причинить какого-нибудь вреда владельцу имени.

Имя значит очень много. Не нужно бояться имени, но недооценивать его значения не следует. Имя описывает:

Идентичность

Имя составляет основу нашего представления о личности. Примеры тому дает история – даже относящаяся к двум ты-

сячелетиям до нашей эры, поскольку в Библии есть примеры, когда давали имена каким-то местностям или детям в ознаменование событий. В большинстве культур до сих пор принято, чтобы женщина меняла фамилию, выходя замуж, но то, что некоторые женщины предпочитают не делать этого, показывает, какое значение они придают своему имени.

Поведение

Имя не только способствует идентификации, но оказывает влияние на поведение. Очевидно, имя не определяет действия субъекта, но влияет на взаимодействие с ним и восприятие окружающим миром. Каждому объекту не запрещается иметь несколько имен. Меня, например, по-разному зовут в зависимости от контекста: одним именем меня зовет жена,[1] другим – дочки, а в чатах я пользуюсь псевдонимом и т. д. Эти имена отражают разные виды отношений и выполняемыее мною роли.

Узнавание

Имя помечает нечто как отдельный объект. В результате из эфемерной идеи возникает конкретная сущность. Пока кто-то не ввел слово «электричество», никто не понимал, о чем идет речь, хотя существовало неясное представление о связанных с ним эффектах из наблюдения за молнией или опытами Бенджамина Франклина. Получив свое название, явление стало идентифицироваться как особая сила, поэтому легче стало его обсуждать. В баскской культуре считается, что, назвав что-то по имени, вы доказываете его существование: *Izena duen guzia omen da* – если у него есть имя, значит, оно существует. (Kurlansky 99)

В наше время назначение имен превратилось в огромный бизнес, в котором участвуют (с разным успехом) маленькие фирмы, крупнейшие корпорации и все, кто занимает между ними промежуточное положение. Для выпуска и рекламы своей продукции все эти организации нуждаются в новых и обращающих на себя внимание именах. Благодаря именам клиенты узнают о товарах и услугах.

Совершенно ясно, что имя имеет огромное значение.

Как программисты мы обладаем огромной властью над своими объектами, давая им имена. Плохое название объекта не просто сулит неудобства – оно может вводить в заблуждение и стать источником ошибок. В качестве элементарного примера приведем такой код C++:

```
void checkForContinue(bool weShouldContinue)
{
    if (weShouldContinue) abort();
}
```

[1] Оно зависит также от ее настроения в этот день.

Имя параметра (`weShouldContinue` – нужно продолжить) просто врет или, по крайней мере, имеет смысл, противоположный ожидаемому. Функция не будет работать предполагаемым образом, и в результате выполнение программы будет прекращено – весьма тяжелое последствие всего лишь неправильно названной переменной.

Палками и камнями можно поломать мне кости, но слова мне боли никогда не причинят. Как бы не так.

Зачем нужны хорошие имена?

Нужно тщательно выбирать имена для своих объектов. Ведь исходный код должен быть понятен. Имя создает путь к пониманию, контролю и овладению. Если правильно выбрать имя, по нему можно *понять назначение объекта*.

Хорошее имя – это очень важно. Считается, что человеческий мозг может одновременно хранить примерно семь фрагментов информации[1] (лично у меня меньше, потому что пара слотов явно не работает). Держать в голове информацию о программе и так трудно, поэтому не нужно создавать лишние трудности сложными системами наименований или туманными ссылками.

Четкие имена – верный признак хорошо сработанного кода. Умение давать хорошие имена – важная часть мастерства кодировщика; уж он постарается написать код, который легко читать.

Учитесь давать объектам прозрачные имена – они должны ясно описывать то, что за ними скрывается.

Каким объектам мы даем имена?

Задумаемся как программисты над тем, каким объектам мы даем имена и какие имена мы им даем. Сначала о том, *какие конструкции* чаще всего получают от нас имена, когда мы пишем код:

- Переменные
- Функции
- Типы (классы, перечисления, структуры, определения типов)
- Имена пространств C++ и пакетов Java
- Макросы
- Файлы с исходным текстом

Этот список далеко не исчерпывает все случаи; существуют другие объекты, более высокого уровня, которым следует давать осмысленные

[1] Так называемое «число Миллера», названное в честь исследователя в области психологии (Miller 56).

имена, такие как состояния конечных автоматов, части протоколов связи, элементы баз данных, исполняемые программные модули и т. д. Но для начала ограничимся первой перечисленной шестеркой.

Игра в названия

Какое назначить имя? Техника создания любого имени зависит от стандарта кодирования, которого вы придерживаетесь. Однако хотя стандарт и определяет некоторые правила составления имен, он не бывает настолько конкретным, чтобы определить *правильное* имя любого элемента, входящего в программу.

Чтобы дать хорошее имя, важно точно понимать объект, для которого оно предназначено. Если вы не понимаете сущности, которой собираетесь дать имя, какую роль она будет играть и зачем вообще она существует, можно ли рассчитывать, что вы дадите ей осмысленное имя? Плохие имена часто свидетельствуют о слабом понимании предмета.

Чтобы придумать хорошее имя, главное – четко понимать, для кого оно предназначено. Только в этом случае имя может стать осмысленным. Если не удается придумать хорошее имя объекту, спросите себя, понятно ли вам его назначение.

Прежде чем рассматривать конкретные категории создаваемых имен, важно выяснить, чем мы руководствуемся при выборе имени и почему то или иное имя является плохим или хорошим. Уделим внимание выяснению того, какими качествами обладают хорошие имена.

Описательность

Очевидно, что имя должно быть описательным. Для того оно и существует, чтобы описывать нечто. И тем не менее нередко встречаются замысловатые идентификаторы, по которым трудно определить, какие данные они описывают.

Даже точного наименования может быть недостаточно. Людям свойственно придерживаться первого впечатления о предмете – «по одежке встречают». Поэтому тщательный выбор имени может создать нужное первое впечатление. Выбирая имена, рассчитывайте на неопытного читателя, не владеющего такими обширными знаниями, как вы сами.

Не всегда легко найти хорошее описание. Если не удается подобрать хорошее имя, это может свидетельствовать о недостатках всей конструкции и необходимости внесения в нее изменений.

Техническая корректность

Современные языки программирования требуют выполнения некоторых правил при образовании имен. Большинство из них различает верхний и нижний регистры символов, запрещает *пробельные символы*

(пробел, табуляция, перевод строки) и любые другие, помимо буквенно-цифровых и некоторых других (например, символа подчеркивания). Заметные ограничения на длину идентификаторов перестали существовать.[1] Многие языки разрешают записывать идентификаторы с помощью Unicode, но сохраняется традиция ограничения символами ISO8859-1 (ASCII).

Возможны дополнительные ограничения. Стандарты C/C++ резервируют определенные диапазоны имен: глобальные идентификаторы не должны начинаться с символов `str`, за которыми следует буква нижнего регистра, а также начинаться с символа подчеркивания или совпадать с идентификаторами пространства имен `std`. Для написания надежного, корректного кода нужно помнить об этих ограничениях.

Идиоматичность

Если какая-то комбинация символов допустима в языке, это не значит, что она автоматически становится хорошим именем. Понятными читателю кажутся те имена, которые представляют собой языковые *идиомы*. Так же как беглое владение естественным языком предполагает знание его идиом, так и уверенное владение языком программирования требует применения идиом.

В некоторых языках применяется единое стандартное правило образования имен; обширная библиотека Java навязывает структуру, от которой трудно отказаться, в то время как в C и C++ степень единообразия меньше. Там существует несколько культур со своими особенностями: одни из них действуют в стандартных библиотеках, другие в Windows Win32 API.

Изучите правила образования имен в языке, с которым работаете. Еще важнее изучить идиомы этого языка. Есть стандартные способы формирования имен? Вот ими и пользуйтесь.

Тактичность

В ряде случаев качество имени может оцениваться по следующим параметрам:

Длина

Понятные, содержательные имена состоят из слов естественного языка. Программистам свойственно сокращать обычные слова, что может приводить к непониманию и путанице. Пусть имя будет длинным, но его смысл должен быть однозначным. Писать `a` вместо `apple_count` неправильно.

---

[1] Учтите, что в старых версиях C уникальность внешних идентификаторов определялась по первым шести символам, а регистр мог не иметь значения. Когда вы пишете код, точно определите, кто и как будет его обрабатывать.

Ясность имени предпочтительнее его краткости.

Однако есть случай, когда хороши короткие (даже однобуквенные) имена – в переменных цикла. Они уместны в маленьких циклах, где переменные типа `loop_counter` оказываются неоправданно большими и быстро утомляют.

Следует учитывать относительные достоинства коротких и длинных имен, особенно в зависимости от области действия переменной.

Стиль

Стиль именования тоже имеет значение. Подобно тому как на похоронах неуместны грубые шутки, дурно составленное имя может вызвать сомнения в профессиональности кода. Да, дурацкие имена заставляют читателя усомниться во вменяемости автора кода.

Избегайте шутливых имен, типа *blah* или *wibble*, или принятых среди гуру *foo* и *bar*. Очень легко проглядеть их, и первая улыбка сменяется неловкостью. (Объекты с такими именами часто возникают как временные заплатки, но сохраняются дольше, чем предполагалось.) Также очевидно, что профессиональный код не должен содержать бранных слов.

Пища для размышлений

Так что все-таки с этими *foo* и *bar*? Это слова из юмора гиков – совершенно бессмысленные и в то же время употребляемые особым образом. Обычно они служат заместителями произвольных объектов. Можно, например, написать: увеличим некоторую переменную *foo* следующим образом: ++*foo*;.

Эти слова обычно составляют целые группы. Их несколько, но чаще всего встречаются *foo*, *bar* и *baz*. Выбор может быть чисто случайным или взят из того варианта технического фольклора, которому вы отдаете предпочтение.

Этимология этих слов точно не установлена. Обычно их возводят к армейскому сленгу времен Второй мировой войны FUBAR (Mucked Up Beyond All Repair). Нет сомнений, что в готовом коде таких имен быть не должно.

Выбирайте для объектов правильные имена сразу и навсегда.

Технические подробности

В следующих разделах обсуждается, как выбирать имена в каждой из перечисленных категорий объектов. Даже если у вас многолетний опыт программирования, обзор существующих принципов формирования имен может оказаться полезен и для вас.

Имена переменных

Не будь переменная лишь электронной сущностью, ее можно было бы подержать в руке – своего рода программный эквивалент физического объекта. Название, которое этому соответствует, обычно должно быть существительным. Например, переменные в GUI-приложении могут называться `ok_button` и `main_window`. Даже таким переменным, которые не соответствуют объектам реального мира, можно давать имена существительные, например `elapsed_time` (истекшее_время) или `exchange_rate` (курс_обмена).

Не будучи «чистым» существительным, имя переменной часто оказывается отглагольной формой существительного, как, например, `count` (счетчик). Имя числовой переменной может описывать, как интерпретируется ее значение, например `widget_length` (длина_объекта). Имя булевой переменной часто является именем условного оператора, что естественно, поскольку ее значением будет «истина» или «ложь».

В объектно-ориентированных языках есть ряд соглашений по оформлению переменных-членов, чтобы было видно, что они – члены, а не простые локальные или (не дай бог) глобальные переменные. Это упрощенная форма *венгерской нотации*, которую некоторые программисты находят полезной.[1] Например, в C++ имена членов класса могут начинаться с символа подчеркивания, оканчиваться им или начинаться с `m_`. Первый способ считается предосудительным, поскольку связан с некоторым риском и неприятен.[2] Кроме того, несколько затруднительно читать переменные, имена которых начинаются или оканчиваются символом подчеркивания.

Некоторые программисты снабжают переменные-указатели суффиксами типа `_ptr`, а ссылки – суффиксами `_ref`. Это в некотором роде проникновение венгерской нотации и избыточность. Тот факт, что переменная является указателем, неявно выражен в ее типе. Если ваша функция столь велика, что такого рода украшение кажется вам полезным, то, скорее всего, эта функция слишком велика!

---

[1] Разумеется, эти способы именования не оказывают влияния на открытый API класса – ведь вы сделали все переменные-члены закрытыми, не так ли?

[2] Глобальные идентификаторы не должны начинаться с подчеркивания, за которым идет заглавная буква. В архаичных правилах именования C много таких странных требований.

Существует также практика назначения переменным имен в виде акронимов, которые должны представлять собой краткие «осмысленные» имена. Например, можно определить переменную так: `SomeTypeWithMeaningfulNaming stwmn(10);` (`НекоторыйТипСРазумнымИменем`). Если область видимости невелика, такое имя может быть понятнее, чем развернутый вариант.

Лучше всего, если правила различают имена типов и имена переменных. Имя типа часто начинается с заглавной буквы, а имя переменной со строчной. В результате можно увидеть такие объявления переменных: `Window window;`.

Предпочтительней такое правило именования, которое различает имена переменных и имена типов.

Венгерская нотация

Венгерская нотация – это спорное правило, требующее включать в имя переменной или функции данные об их типе – в надежде, что код станет легче читать и сопровождать. Она возникла в 1980-х годах в Microsoft и широко применяется в открытом Win32 API и библиотеке MFC, благодаря чему в основном и популярна.

«Венгерской» она названа по имени Чарльза Симонаи, венгерского программиста. Другая причина такого названия в том, что имена переменных выглядят в этой нотации так, как если бы были написаны по-венгерски: непривычным к Windows программистам дико видеть имена типа `lpszFile`, `rdParam` и `hwndItem`.

Существует несколько диалектов венгерской нотации, имеющих тонкие различия и не вполне совместимых, что еще ухудшает дело.

Имена функций

Если переменную хочется подержать в руке, то функция – это то, что с ней можно сделать: не держать же ее в руке вечно. Раз функция – это действие, логично, чтобы ее имя было представлено (или хотя бы содержало) глаголом. Функция, имя которой – существительное, непонятна. Что может делать функция `apples()`? Возвращать количество яблок, преобразовывать нечто в яблоки или сотворять яблоки из ничего?

Разумные имена функций не должны содержать слов *be*, *do* и *perform* (быть, делать, выполнять). Новички часто совершают ошибку, используя эти слова при попытке сознательно включить в имя глагол (*эта функция* выполняет *XXX*...). Это пустые звуки, не вносящие в имя никакого смысла.

При выборе имени для функции учитывайте, кто конечный пользователь, и скрывайте все, относящееся к внутренней реализации. (В этом и есть смысл функции как определенного уровня краткости и абстракции.) Никому не интересно, что в процессе своего выполнения функция пользуется списками, делает сетевые вызовы или создает новый компьютер и устанавливает на нем текстовый процессор. Если пользователь видит, что функция считает яблоки, значит, нужно назвать ее `countApples()`.

Давайте функциям имена с внешней точки зрения, в виде фраз, выражающих действие. Описывайте логическую операцию, а не способ реализации.

Данное правило можно нарушить только в одном случае – для простых функций запроса данных. Эти функции доступа вполне можно назвать

Употребление прописных букв

Большинство языков программирования запрещает использовать в идентификаторах пробелы и символы пунктуации, поэтому применяют различные ухищрения, чтобы составить идентификатор из нескольких слов. При этом правила использования прописных букв вызывают среди программистов бурные споры, которые можно сравнить с религиозными войнами по поводу выбора правильного редактора. В современных программах можно встретить ряд стандартных приемов:

camelCase

camelCase активно применяется в библиотеках Java и часто в текстах C++. Название возникло благодаря сходству прописных букв с горбами верблюда, а впервые этот стиль был, по-видимому, применен в языке Smalltalk в начале 1970-х.

ProperCase

Близкий родственник camelCase, отличающийся дополнительным выделением первой буквы. Другое название – PascalCase. Часто оба стиля используют совместно. Например, в Java стиль ProperCase используется для имен классов, а стиль camelCase – для членов. В методах Windows API и .NET применяется ProperCase.

использование_подчеркивания

Сторонников этого стиля мы находим среди авторов стандартной библиотеки C++ (взгляните на состав пространства имен `std`) и членов GNU.

Существует много других форматов. Например, можно применять ProperCase вместе с подчеркиванием или отказаться от прописных букв.

так же, как данные, которые они запрашивают. Иллюстрацией служит ответ (на стр. 597) на вопрос № 9 в конце этой главы.

При написании функции ее нужно подробно документировать (в составе спецификации или каким-нибудь общепринятым методом в коде программы). Тем не менее имя все равно должно ясно указывать на задачу, которую решает функция. Что делает какая-нибудь `void a()`? Да что угодно!

Имена типов

То, какие типы можно создавать, зависит от языка. В языке C существуют `typedef`, создающие синонимы для других имен типов. С их помощью можно создавать более простые и удобные имена. Понятно, что `typedef` должен создавать понятное имя. Даже если это локальное определение `typedef` в теле функции, у него должно быть содержательное имя.

В Java, C++ и прочих ОО-языках активно применяется создание новых типов (*классов*). C позволяет также определять составные типы, называемые `structs`. Хороший выбор имени типа имеет такое же важное значение, как выбор правильных имен переменных и функций для облегчения читаемости кода. Однако строгих правил именования классов нет, поскольку классы могут служить разным целям:

- Класс может описывать некий объект данных, меняющий свое состояние. В таком случае имя класса может быть существительным.

  Это может быть объект-функция (*функтор*) или класс, реализующий виртуальный интерфейс обратного вызова. Тогда его имя может быть глаголом, возможно, включающим имя известной конструктивной схемы. (Gamma et al. 94)

- Если в классе объединены и первое, и второе, то дать ему имя бывает затруднительно, а кроме того, он может быть просто плохо спроектирован.

  Классы интерфейсов (например, абстрактные классы C++ с чисто виртуальными функциями или интерфейсы в Java и .NET) обычно называют в соответствии с возможностями интерфейса. Часто встречаются такие имена, как `Printable` и `Serializable`. В .NET дополнительно применяется венгерская нотация в виде добавления `I` перед именами всех интерфейсов, что приводит к появлению таких имен, как `IPrintable`.

Ранее мы уже говорили о том, каких имен функций нужно избегать; аналогичные правила действуют и для классов. Например, плохим именем будет `DataObject` — и так ясно, что класс может содержать данные и будет создавать объект; нет необходимости снова это повторять.

Избегайте в именах лишних слов. В частности, в именах типов – таких слов, как class, data, object и type.

Класс плохих имен

Дурное имя класса может действительно сбить программистов с толку. Однажды мне довелось работать над приложением, в котором был реализован конечный автомат. В силу исторических причин базовый класс каждого состояния носил название `Window`, хотя более разумным было бы дать ему, к примеру, имя `State`. Это привело в замешательство нескольких программистов, которые впервые знакомились с кодом. Пуще того, базовый класс управляющего шаблона назывался `Strategy`, тогда как в действительности он не реализовывал никакого шаблона стратегии. Разобраться в существе было очень непросто. Если бы имена были выбраны лучше, было бы легче понять логику кода.

Следите за тем, чтобы описание относилось к *классу данных*, а не к *фактическому объекту*. Это тонкое, но важное отличие.

Пространства имен

Какое имя дать конструкции, специально созданной для сличения имен? *Пространства имен* C++ и C#, а также *пакеты* Java выполняют в основном функции группировки.

Кроме того, они служат предотвращению конфликтов *имен*. Если два программиста создадут какие-то программные объекты с одинаковыми именами и их код потребуется соединить, ничего хорошего из этого не выйдет. В лучшем случае общий код не удастся собрать. В худшем – во время выполнения могут произойти самые неприятные вещи. Благодаря размещению объектов в разных пространствах имен можно избежать засорения глобального пространства имен. Таким образом, это ценный инструмент в процессе именования.

Однако пространства имен сами по себе не исключают конфликтов; вы создадите свое пространство имен `utils` и столкнетесь с тем, что другой программист тоже создал пространство имен `utils`. Выход – в применении *системы имен*. В Java определена иерархия имен пакетов, вкладываемых друг в друга, как имена доменов Интернета: вы помещаете свой код в пакет с уникальным именем. Благодаря этому успешно решается проблема коллизий имен. Без этого правила пространства имен уменьшали бы вероятность коллизий, но не устраняли их полностью.

Выбирая имя пространства имен, старайтесь, чтобы оно имело отношение к содержимому. Если оно представляет собой весь интерфейс библиотеки, дайте ему такое же имя, как у библиотеки. Если содержимое является частью более крупной системы, выберите имя, которое описывает эту часть; `UI`, `filesystem` или `controls` – все они будут хорошими

именами. Не включайте в имя лишнее напоминание, что это коллекция объектов, например `controls_group` – неудачное имя.

Давайте пространствам имен и пакетам имена, логически связанные с их содержимым.

Имена макросов

Макросы – это зубодробительный инструмент в мире C/C++. Это средства поиска и замены базового текста, не признающие областей видимости. Они совершенно бестактны. Но есть такие орешки, которые без их помощи не вскрыть.

Последствия их работы могут быть катастрофическими, поэтому существует прочная традиция выделять их имена самым заметным образом – ПРОПИСНЫМИ БУКВАМИ. Следуйте этому правилу неукоснительно и не выделяйте прописными буквами никакие другие имена. Тем самым макросы постоянно напоминают больное место, чем они, по сути, и являются.

Поскольку это просто средства замены текста, давайте им уникальные имена, которые не могут встретиться в другом месте кода. В противном случае возникают катастрофы и беспорядки.

Полезно применять уникальные префиксы в виде имени проекта или файла. Например, имя `PROJECTFOO_MY_MACRO` значительно безопаснее, чем `MY_MACRO`.

Макросы в C/C++ всегда выделяют прописными буквами, чтобы сделать хорошо заметными, и тщательно выбирают их имена, чтобы избежать конфликтов. Никогда не выделяйте прописными буквами другие объекты.

Имена файлов

Выбор имен файлов исходного кода может заметно облегчить кодирование. В некоторых языках существуют строгие требования к именам файлов – имена файлов исходного кода Java должны соответствовать именам содержащихся в них открытых классов. Напротив, C и C++ допускают свободу, не накладывая никаких ограничений.[1]

Чтобы выбор имени файла был прост и очевиден, каждый файл должен состоять из одной логической единицы. Слишком объемистые файлы чреваты возникновением проблем в будущем. Разбейте свой код на как можно большее количество файлов; тогда не только проще будет дать им имена, но уменьшится число связей и структура проекта станет понятнее.

---

[1] Кроме тех, которые обусловлены ОС или файловой системой.

Файл C/C++, в котором определяется интерфейс для *widget*, нужно назвать `widget.h`, а не `widget_interface.h`, `widget_decls.h` или каким-то еще способом. Обычно каждому `widget.h` должен соответствовать `widget.cpp` или `widget.c` (см. «Все хорошо, что хорошо кончается» на стр. 86), где реализуется то, что декларировано в `widget.h`. Общее базовое имя логически связывает эти файлы. Это и очевидно, и общепринято.

Существует много других тонких, но важных правил именования файлов:

- Остерегайтесь употребления прописных букв. Некоторые файловые системы не способны различать регистр букв при поиске файлов по имени. А при переносе на платформу, где регистр имеет значение, ваш код может не скомпилироваться, если вы нетщательно соблюдали единообразное применение прописных букв. Проще всего избежать таких неприятностей, если использовать в именах файлов исключительно строчные буквы; как говорится, *не можешь сделать хорошо, сделай правильно*. (Разумеется, это не относится к Java, где в именах классов и интерфейсов применяется стиль PascalCase.)
- По той же причине, если ваша файловая система считает файлы `foo.h` и `Foo.h` разными, *не* злоупотребляйте этим. Следите, чтобы имена файлов, лежащих в одном каталоге, отличались не только регистром.
- Если в вашем проекте участвует несколько языков, не храните в одном и том же каталоге `foo.c`, `foo.cpp` и `foo.java`. Это вносит путаницу: из какого файла должен создаваться объектный файл `foo.o`, а из какого – выполняемый модуль `foo`?
- Постарайтесь дать всем создаваемым вами файлам различные имена, даже если они разбросаны по отдельным каталогам. Так проще определить, с каким файлом вы имеете дело. Тогда станет очевидным, какой заголовочный файл нужен, если вы напишете `#include "foo.h"`. При наличии двух файлов с одинаковыми именами новичку труднее разобраться в коде. Такие проблемы еще более осложняются по мере роста системы.

  Один из выходов – добавить к логическому имени файла данные о пути к нему. Разместите свои файлы так, чтобы можно было включать `library_one/version.h` и `library_two/version.h`, не вызывая недоразумений.

Выбор имен файлов существенно влияет на сложность кодирования. Однажды я работал над проектом C++, в котором большинство имен файлов точно совпадало с именами классов: класс `Daffodil` определялся в `Daffodil.h` (имена изменены, чтобы никого не обидеть). Однако ряд файлов именовался *несколько* иным способом, обычно путем сокращения, например класс `HerbaciousBorder` определялся в `HerbBdr.h`. В результате искать правильное имя файла для директивы `#include` было сложно и долго. Мало того, реализация класса `Daffodil` могла не цели-

ком лежать в `Daffodil.cpp`, поскольку какая-то ее часть могла быть в общем файле `FlowerStuff.cpp`, а то еще и в `Yogurt.cpp` – по совершенно непонятным причинам. Нетрудно догадаться, что поиск нужных фрагментов кода иногда превращался в кошмар. В таких случаях броузеры исходного кода бывают полезны, но лучше всего пользоваться проверенными временем методами эффективного именования.

Роза пахнет розой

Значение имени больше, чем может показаться на первый взгляд, и существует масса соображений при выборе имени для программного объекта. Какими главными принципами следует руководствоваться?

Чтобы выбрать хорошее имя, нужно:

- Соблюдать единообразие
- Связывать имя с содержимым
- Извлекать из имен практическую пользу

Все хорошо, что хорошо кончается

Суффикс – составная часть имени файла. Система сборки Java-программ требует, чтобы имена файлов исходного кода завершались на `.java`. Компиляторы C и C++ безразличны к суффиксам, но давать файлам заголовков имена с расширением `.h` настолько общепринято, что всякий иной способ режет глаз. Недостаток строгих правил приносит некоторые неудобства; есть несколько стилей именования файлов реализации C++, например стандартные суффиксы `.C`, `.cc`, `.cpp`, `.cxx` и `.c++`. Изредка встречаются файлы заголовков C++ с суффиксом `.hpp`. Ваше решение может зависеть от компилятора, личных вкусов и/или стандарта кодирования. Главное – соблюдать единообразие: выберите какую-то систему суффиксов и пользуйтесь ею постоянно.

Однажды мне довелось работать на платформе, которая не поддерживала суффиксы имен файлов. Определять на ней тип файла было сложно и неприятно.

Соблюдайте единообразие

Пожалуй, это главный принцип. Соблюдайте *единообразие* не только в собственном коде, но и сообразно принятой в компании практике. У меня не вызывает доверия качество интерфейса класса, если он выглядит так:

```
class badly_named : public MyBaseClass
{
```

```
public:
    void doTheFirstThing();
    void DoThe2ndThing();
    void do_the_third_thing();
};
```

При совместной работе многих людей очень легко придти к коду такого сорта – столь же единообразному, как результат работы генератора случайных чисел. Часто это оказывается симптомом более серьезной проблемы – возможно, программисты пренебрежительно относятся к базовым принципам проектирования кода, над которым совместно трудятся. В таких случаях большую помощь могут оказать обязательные стандарты кодирования и основная проектная документация.

Единообразие должно касаться не только применения прописных букв и форматирования, но и принципов *создания* имен. Имя задает неявную метафору. На всем пространстве программы или проекта эти метафоры должны быть последовательными. Способ именования должен быть глобальным.

Выберите единообразную систему именования и последовательно применяйте ее.

Единообразное именование приводит к интуитивно понятному коду, с которым проще работать, его проще расширять и сопровождать. В конечном счете, управлять таким кодом дешевле.

Связывайте имя с содержимым

Каждое имя, читаемое в своем контексте, должно быть абсолютно понятным. И имя всегда будет находиться в известном контексте, поэтому все лишнее, дублирующее контекстную информацию, можно убрать. Нужно стремиться к созданию кратких содержательных имен без лишних довесков.

Источником контекстной информации могут служить:

Область видимости

Объекты располагаются либо на верхнем, глобальном уровне, либо находятся внутри некоторого пространства имен, класса или функции. Выбирайте имя, оправданное в контексте своего уровня. Чем более узкой и конкретной является область видимости, тем проще создать в ней имя, имеющее понятный читателю смысл в контексте этой области. Если функция подсчитывает количество яблок на дереве и определена в классе `Tree` (дерево), нет смысла называть ее `countApplesInTree()`. Она однозначно описывается своим полностью квалифицированным именем: `Tree::countApples()`. Выбирайте для именуемых сущностей самые мелкие (и потому самые описательные) области видимости.

Во французском, как и в большинстве романских языков, есть две формы местоимения *2-го* лица: *tu* и *vous*. Выбор той или иной из них определяется степенью вашей близости с тем, к кому вы обращаетесь. Точно так же имя, с помощью которого вы обращаетесь к переменной, может зависеть от контекста, в котором она находится. Пример: разные имена переменных в открытых объявлениях функций и в их реализациях.

Тип

Все имеет свой тип, который можно узнать. Нет необходимости заново повторять тип в имени. (Повторение типа – смысл «венгерской нотации», из-за чего ее часто высмеивают.)

Неопытный программист может назвать переменную, содержащую строку адреса, как `address_string`. И какая польза от суффикса `_string`? Никакой, поэтому уберите его.

Степень необходимой детализации имени зависит от контекста его применения. Создавая имена, учитывайте контекстную информацию.

Извлекайте выгоду из выбора имени

Выбор имени дает вам возможность обеспечить большую выразительность, чем это можно сделать одними лишь синтаксическими средствами языка. Например, можно объединять объекты, используя для их имен общие префиксы. Другая возможность – сообщить, какие из параметров функции являются входными, а какие выходными, включив эту информацию в их имена.

Резюме

...и уповать на имя Твое, ибо оно благо...

Псалтирь 52:9

Наши предки знали в те времена то, что хорошие программисты знают сейчас: очень важно правильно выбрать имя. Хорошие имена не просто удовлетворяют эстетическую потребность – они передают информацию о структуре кода. Они служат важным средством, позволяющим облегчить понимание и сопровождение кода.

Главная причина применения языков высокого уровня при написании кода – возможность передачи информации, причем не компилятору, а людям, читающим код, т. е. другим программистам. Плохие имена могут служить причиной ошибок и заблуждений. В имени *заключена* сила, и опытные программисты понимают, какие факторы нужно учитывать при выборе имени для любой части своего кода.

Что можно и чего нельзя

Разнообразные советы, приведенные в этой главе, можно свести к ряду правил. *Избегайте* в своих именах таких качеств, как:

Загадочность

Есть много способов создания непонятных имен. Акронимы и сокращения вполне могут оказаться бессмысленными, а однобуквенные имена – слишком загадочными.

Многословие

Избегайте очень коротких имен, но и не создавайте таких переменных, как `количество_яблок_перед_тем_как_я_начал_их_есть`. Ни пользы, ни развлечения.

Неточность или обманчивость

Как ни банально, но помните, что имя нужно выбирать точно. Не нужно называть `widget_list` то, что не имеет никакого отношения к спискам. Не нужно называть что-то `widget`, если это контейнер для нескольких widget.

Орфографические ошибки – это минное поле всякой путаницы. Ищу переменную *ignoramus* и нигде не могу ее найти. Выясняется, что я по ошибке назвал ее *ignoramous*. Разобрались.

Двусмысленность или неясность

Не выбирайте имен, которые можно истолковать по-разному. Не выбирайте совершенно туманные имена типа `data` или `value`, если только обстановка не указывает точно, что они собой представляют. Избегайте неопределенных `temp` или `tmp`, если в этом нет *прямой* необходимости.

Не полагайтесь на имена, различающиеся использованием прописных букв или одним символом. Остерегайтесь имен, которые звучат сходным образом.

Не создавайте без особых причин локальные переменные с такими же именами, как объекты во внешней области видимости.

Заумь

Следует избегать симпатичных аббревиатур, остроумных, но трудно запоминаемых сокращений, а также толковательного применения числительных. Распространенное сокращение *i18n* для слова *internationalization* кажется полной чепухой непосвященным.

Напротив, выбирайте правильные имена, которые понятны, точны, конкретны, коротки и недвусмысленны. Пользуйтесь стандартными терминами и принципами. Выбирайте слова из предметной области и применяйте наглядные схемы составления имен. (Gamma et al. 94)

Хорошие программисты...

- Понимают важность выбора имен и занимаются этой проблемой
- Не забывают о выборе правильного имени для каждого создаваемого объекта
- Учитывают многие факторы: длину имени, понятность, контекст и т. д.
- Видят общую картину и выбирают имена в рамках проекта (или проектов)

Плохие программисты...

- Не заботятся о понятности своего кода
- Пишут *одноразовый* код, плохо продумывая его в погоне за скоростью
- Игнорируют естественную идиоматику языка
- Не стремятся к единообразию в именах
- Не заботятся об общей картине и не интересуются согласованностью своего кода с проектом в целом

См. также

Глава 2. Тонкий расчет

Обсуждаются стандарты кодирования, которыми можно руководствоваться при выборе имен. Рассказывается о *религиозных войнах*, причиной которых, несомненно, послужила венгерская нотация.

Глава 4. Литературоведение

Хороший выбор имен не заменяет хорошо документированного кода – имена являются неотъемлемой частью документации кода.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 594.

Вопросы для размышления

1. Хорошо ли выбраны следующие имена переменных? Ответом может быть *да* (объясните почему и в каком контексте), *нет* (объясните почему) или *не знаю* (объясните почему).
   a. `int apple_count`
   b. `char foo`
   c. `bool apple_count`
   d. `char *string`
   e. `int loop_counter`
2. В каких случаях оправдан такой выбор имен функций? Какие типы возвращаемых значений или параметров предполагаются? Какие типы возвращаемых значений делают такие имена бессмысленными?
   a. `doIt(...)`
   b. `value(...)`
   c. `sponge(...)`
   d. `d. isApple(...)`
3. Что важнее для системы именования – легкость чтения или легкость написания кода? Как можно облегчить то или другое?
   a. Сколько раз вы пишете один и тот же фрагмент кода? (Подумайте.) Сколько раз вы его читаете? Ответы на эти вопросы должны помочь определить относительную важность.
   b. Как вы поступаете при противоречиях в принципах именования? Допустим, вы работаете над кодом C++ в стиле camelCase и должны модифицировать библиотеку STL (используется_подчеркивание). Как лучше поступить в такой ситуации?
4. При каком размере цикла следует дать осмысленное имя переменной цикла?
5. Если в C `assert` является макросом, почему его имя пишут строчными буквами? Зачем дают макросам выделяющиеся имена?
6. Каковы достоинства и недостатки принципов именования стандартной библиотеки вашего языка?
7. Может ли надоесть имя? Правильно ли пользоваться одним и тем же именем локальной переменной в разных функциях? Правильно ли пользоваться локальными именами, которые перекрывают (и скрывают) глобальные имена? Почему?
8. Объясните сущность венгерской нотации. Каковы ее достоинства и недостатки? Согласуется ли она с современными принципами разработки кода?
9. Часто классы содержат члены-функции, предназначенные для *чтения* и *записи* значений определенных свойств. Какие схемы именования таких функций существуют и какие из них предпочтительней?

Вопросы личного характера

1. Искусны ли вы в выборе имен? Какими из предложенных правил вы пользуетесь на сегодняшний день? Занимаетесь ли вы составлением имен и применением этих правил сознательно или это происходит стихийно? В какой области вы могли бы улучшить свою практику?
2. Есть ли в вашем стандарте кодирования правила для выбора имен?
   a. Охватывает ли он все рассмотренные нами ситуации? *Достаточно* ли он полон? Есть ли от него польза или он представляет собой формальность?
   b. Насколько детализированы должны быть правила именования в стандарте кодирования?
3. Какое самое неудачное имя встретилось вам в последнее время? Были ли случаи, когда имена вводили вас в заблуждение? Как бы вы изменили их, чтобы избежать подобных проблем в будущем?
4. Приходится ли вам портировать код с одной платформы на другую? Как это повлияло на выбор вами имен файлов, других объектов и общей структуры кода?

4

Литературоведение

Техника написания самодокументируемого кода

В этой главе:

- Как документировать код
- Грамотное программирование
- Средства документирования

По-настоящему серьезное отношение к писательскому делу – одно из двух непременных условий. Второе, к сожалению, – талант.

Эрнест Хемингуэй

Современная сборная мебель (в плоской упаковке) – большое достижение, перед которым даже у опытного столяра возникает чувство смущения и благоговения. Как правило, она толково спроектирована и *в конце концов* превращается именно в то, что вам было нужно.

Во время сборки необходимо пользоваться прилагаемой инструкцией, иначе в итоге вы получите не мебель, а нечто вроде произведения современного искусства. Простота сборки в огромной мере определяется качеством инструкции. Плохая инструкция заставит вас мучиться, ругаться и постоянно разбирать детали, ошибочно соединенные между собой.

Жаль, что теперь не делают такие вещи, как раньше.

Исходный код выдвигает перед вами такие же проблемы. Верно, его делают не так, как раньше, но *никто* никогда и не восторгался перфокартами или COBOLом. Самое главное, что при работе с некоторыми программами вам придется мучиться, ругаться и постоянно разбирать детали, соединенные между собой по ошибке, если нет хорошей инструкции, объясняющей взаимное расположение блоков кода.

Хорошим кодом может быть только *хорошо документированный код*. Мы пишем код для того, чтобы изложить четкий набор инструкций – не только для компьютера, но и для тех несчастных, кому потом придется исправлять или расширять эти инструкции. На практике код никогда не пишут так, чтобы потом навсегда про него забыть. На протяжении всего цикла жизни программного продукта код модифицируется, развивается и сопровождается. Для этого необходима инструкция, руководство пользователя – документация.

Обычно считается, что для документирования кода нужно создать тонны документов, *описывающих* код, или написать уйму комментариев внутри кода.

И то и другое неверно. Большинство программистов с антипатией относится к текстовым процессорам и не любит писать пространные комментарии. Создание кода – трудная работа. И его документирование не должно быть еще более трудной работой. В организациях, профессионально производящих программное обеспечение, все, для чего требуется дополнительный труд, обычно не делается. А если и делается, то скверно.

Мне встречались программные системы, снабженные проектными спецификациями, замечаниями по реализации, руководствами по сопровождению и стилю. Нет ничего удивительного, что работать с таким кодом очень утомительно. Проблемы всякой сопроводительной документации заключаются в следующем:

- Нежелательно делать лишнюю работу. Составление документации отнимает много времени, и чтение ее – тоже. Программисты предпочли бы потратить это время на программирование.
- Все отдельные документы должны быть синхронизированы с любыми изменениями в коде. В большом проекте обеспечить это очень трудно. Альтернатива, которую обычно выбирают (не обновлять никакой документации), приводит к появлению опасно неточной и ошибочной информации.
- Большим объемом документов трудно управлять. Трудно найти нужный документ или конкретный фрагмент информации, который может располагаться в разных местах документа. Как и код, документация должна управляться системой контроля версий, чтобы при чтении определенной версии исходного кода обеспечивалось чтение соответствующей версии документации.

- Важную информацию, расположенную в отдельных документах, легко пропустить. Когда она располагается отдельно от кода и нет удобных указателей, можно упустить важные вещи.

Не пишите код, который нуждается во внешней документации. Он ненадежен. Пишите такой код, который понятен без посторонней документации.

Распространенный вариант – подробные комментарии, содержащиеся в коде, – нисколько не лучше. Обилие подробных комментариев не способствует созданию хорошего кода. В итоге получается не хорошая программа, а скверно отформатированная документация.

Как избежать этой напасти? С помощью *самодокументируемого кода*.

Самодокументируемый код

Идея звучит заманчиво. Но что такое «самодокументируемый» код? Следующая программа является самодокументируемой:

```
10 PRINT "Я очень маленькая и очень глупая"
20 GOTO 10
```

Гордиться тут, собственно, нечем. Сложная и имеющая практическую ценность программа требует большого мастерства. Практика показывает, что читать компьютерные программы гораздо сложнее, чем писать их. Всякий, кто знаком с Perl, это подтвердит: данный язык представлялся как предельный случай «напиши и забудь». И вправду, старый код Perl может казаться совершенно непостижимым. Однако непонятный код можно написать на любом языке, и особых стараний для этого не требуется.

Единственный документ, который описывает ваш код полностью и правильно – это сам код. Из этого не следует, что другое, лучшее описание невозможно, но чаще всего это единственная документация, которая оказывается в вашем распоряжении.

Следовательно, вы должны приложить все усилия, чтобы сделать эту документацию хорошей – такой, которую сможет читать каждый. Дело в том, что автор кода не единственный человек, которому необходимо в нем разбираться. Языки программирования служат для нас средством общения. Общение должно быть понятным. Благодаря ясности кода его качество растет, поскольку ошибки становятся менее вероятными (они оказываются более заметными), а сопровождение кода обходится дешевле – меньше времени требуется на его изучение.

Самодокументируемый код легко читается. Он понятен без привлечения дополнительной документации. Облегчить понимание кода можно многими способами. Одни приемы элементарны и входят в основы обучения программистов, другие более изощренны, и владение ими приходит с опытом.

Не судите о книге по переплету...

Файл с самодокументируемым кодом читается почти как хороший справочник. У него четкая структура, разделы, формат. Его легко читать с первой страницы и до конца, но можно открыть для справки в любом месте. Вот таким и должен быть наш код. Рассмотрим его по частям.

Введение

Во введении к книге рассказывается о том, что в ней содержится, задается тон и объясняются связи с внешним миром. Файл исходного кода должен начинаться с заголовочного комментария, в котором разъясняется, что находится внутри файла и к какому проекту относится этот файл.

Оглавление

Некоторые считают, что в заголовке файла должны быть перечислены все содержащиеся в нем функции, но я настоятельно советую не делать этого. Этот список быстро устаревает. В то же время большинство современных редакторов и сред разработки (IDE) позволяют перечислить содержание файла (все типы и классы, функции, переменные) с удобными ссылками на соответствующие фрагменты кода.

Разделы

Эта книга разделена на несколько частей. Файлы с исходным кодом тоже могут разделяться на крупные секции, например, когда файл содержит несколько классов или логических групп функций. Здесь полезно воспользоваться комментариями в качестве заметных границ. Обычно не следует заниматься художеством с помощью символов ASCII, но такие комментарии помогают логически разбить файл и облегчают поиск в нем.

Однако следует учитывать, что размещать в одном файле слишком много программных объектов не следует. Лучше всего, когда одному классу соответствует один файл. В крупных файлах, решающих много задач, трудно разбираться и очень сложно перемещаться по коду. (Если, следуя этому совету, вы получите слишком много файлов исходного кода, вам необходимо поработать над структурой кода более высокого уровня.)

Главы

Каждая глава книги – это самостоятельный текст с правильно подобранным заголовком. Файлы исходного кода обычно содержат ряд функций с правильно подобранными именами.

Абзацы

Внутри каждой функции операторы объединяются в блоки. Начальные объявления переменных составляют один логический блок, отделенный от последующего кода пустой строкой (по крайней мере, так делалось в старом коде C). Это не имеет отношения к синтаксису, а просто облегчает чтение кода.

Фразы

Фразы обычно соответствуют отдельным операторам кода.

Перекрестные ссылки и предметный указатель

Они также не входят в разметку файла исходного кода, но хороший редактор или IDE предоставляют возможности перекрестных ссылок. Научитесь пользоваться ими.

Это любопытная аналогия, но какое отношение она имеет к написанию кода? На самом деле многие полезные правила написания книг оказываются ценными и для написания хорошего кода. Освойте их, и ваш код будет легче читать. Разбейте код на секции, главы и абзацы. Применяйте форматирование для подчеркивания логической структуры кода. Пользуйтесь простыми короткими операторами – как и короткие предложения, их легче читать.

Пишите код, который может прочесть нормальный человек, причем без напряжения. Компилятор как-нибудь справится.

Вот пример простой функции, которую никак не назвать самодокументируемой. Что, по вашему мнению, она выполняет?

```
int fval(int i)
{
     int ret=2;
     for (int n1=1, n2=1, i2=i-3; i2>=0; --i2)
     {
         n1=n2; n2=ret; ret=n2+n1;
     }
     return (i<2) ? 1 : ret;
}
```

Это вполне реальный пример; бесчисленное количество строк готового кода выглядит примерно так же, принося мучения читающим их программистам. Напротив, следующий ниже код *является* самодокументируемым. Можно по первой же строке определить, какую задачу он решает.

```
int fibonacci(int position)
{
    if (position < 2)
```

```
    {
        return 1;
    }
    int previousButOne = 1;
    int previous       = 1;
    int answer         = 2;

    for (int n = 2; n < position; ++n)
    {
        previousButOne = previous;
        previous       = answer;
        answer         = previous + previousButOne;
    }
    return answer;
}
```

Обратите внимание на отсутствие комментариев в этой функции. Все, что происходит, понятно и без них. Комментарии только увеличили бы объем читаемого текста и стали бы лишней нагрузкой, затрудняющей сопровождение функции в будущем. Это важно, потому что даже самые короткие и изящные функции требуют в дальнейшем сопровождения.[1]

Техника написания самодокументируемого кода

Обычно считается, что самодокументируемый код должен содержать в себе обилие комментариев. Хорошие комментарии, несомненно, нужны, но ими дело не ограничивается. На самом деле нужно стремиться *избегать* комментариев путем написания такого кода, в котором в них нет необходимости.

Следующие разделы содержат описание важных приемов самодокументирования кода. Можно заметить, что они во многом перекрываются с другими главами в первой части книги. И это неудивительно – есть много общих признаков хорошего кода; достоинства отдельного метода проявляются в различных характеристиках качества кода.

Пишите простой код с хорошим форматированием

Формат представления оказывает огромное влияние на легкость понимания кода. Разумное представление передает структуру кода: функции, циклы и условные операторы становятся понятнее.

- Стремитесь к тому, чтобы ход «нормального» выполнения вашего кода был очевиден. Обработка ошибок не должна отвлекать от нормальной последовательности выполнения. Условные конструкции *if-then-else* должны иметь единообразный порядок ветвей (например, всегда помещайте ветвь «обычного» хода перед ветвью «обработки ошибок», или наоборот).

[1] Вы разобрались, что делает первый пример? Обе функции вычисляют член последовательности Фибоначчи. Какую из них вам удобнее читать?

- Избегайте большого количества уровней вложенных операторов. В противном случае код становится сложным и требует пространных пояснений. Принято считать, что у каждой функции должна быть *только одна* точка выхода; это известно как код *Single Entry, Single Exit* (*SESE, один вход – один выход*). Но обычно это ограничение затрудняет чтение кода и увеличивает количество уровней вложенности. Мне больше нравится приведенный выше вариант функции `fibonacci`, чем следующий вариант в стиле SESE:

```
int fibonacci(int position)
{
    int answer = 1;
    if (position >= 2)
    {
        int previousButOne = 1;
        int previous = 1;

        for (int n = 2; n < position; ++n)
        {
            previousButOne = previous;
            previous       = answer;
            answer         = previous + previousButOne;
        }
    }
    return answer;
}
```

  Я бы отказался от такой излишней вложенности в пользу дополнительного оператора `return` – читать функцию стало гораздо труднее. Весьма *сомнительна* целесообразность прятать `return` где-то в глубине функции, зато простые сокращенные вычисления в ее начале очень облегчают чтение.

- Остерегайтесь оптимизации кода, в результате которой теряется ясность базового алгоритма. Оптимизируйте код *только тогда*, когда становится ясным, что он мешает приемлемой работе программы. При оптимизации сделайте четкие комментарии относительно функционирования данного участка кода.

Выбирайте осмысленные имена

Имена всех переменных, типов, файлов и функций должны быть осмысленными и не вводить в заблуждение. Имя должно верно описывать то, что оно собой представляет. Если вам не удается найти осмысленное имя, то возникает сомнение в том, понимаете ли вы работу своего кода. Система именования должна быть последовательной и не вызывать неприятных неожиданностей. Следите за тем, чтобы переменная всегда использовалась только с той целью, которую предполагает ее имя.

Хороший выбор имен служит, вероятно, лучшим средством избежать лишних комментариев. Имена лучше всего позволяют приблизить код к выразительности естественных языков.

Разбивайте код на самостоятельные функции

То, как вы разобьете код на функции и какие имена им дадите, может сделать код понятным или совершенно непостижимым.

- *Одна функция, одно действие*. Повторяйте как заклинание. Не пишите сложных функций, умеющих варить кофе, чистить ботинки *и* угадывать задуманное число. В одной функции должно выполняться одно действие. Выберите имя, недвусмысленно описывающее это действие. Если имя выбрано правильно, не нужно ничего больше комментировать.
- Минимизируйте любые неожиданные *побочные эффекты*, сколь бы полезными они ни казались. Они потребуют дополнительного документирования.
- Пишите короткие функции. В них легче разбираться. Можно сориентироваться в сложном алгоритме, если он разбит на мелкие фрагменты с содержательными именами, но это не удастся сделать в бесформенной массе кода.

Выбирайте содержательные имена типов

Насколько это возможно, описывайте ограничения или поведение с помощью доступных возможностей языка. Например:

- Определяя величину, которая не будет меняться, назначьте для нее тип константа (используйте `const` в C).
- Если переменная не должна принимать отрицательных значений, воспользуйтесь беззнаковым типом (при наличии его в языке).
- Пользуйтесь перечислениями для описания связанного набора данных.
- Правильно выбирайте тип переменных. В C/C++ записывайте размер в переменные типа `size_t`, а результаты арифметических операций с указателями – в переменные типа `ptrdiff_t`.

Применяйте именованные константы

Код типа `if (counter == 76)` вызывает недоумение. Каково волшебное значение числа 76? В чем смысл этой проверки?

Практика *волшебных чисел* порочна. Они затеняют смысл кода. Гораздо лучше написать так:

```
const size_t bananas_per_cake = 76;
...
if (count == bananas_per_cake)
{
    // испечь банановый пирог
}
```

Если в коде часто встречается константа 76 (простите, `bananas_per_cake`), достигается дополнительное преимущество: когда потребуется изме-

нить содержание бананов в пироге, достаточно модифицировать код в одном месте, а не выполнять глобальный поиск/замену числа 76, что чревато ошибками.

Избегайте волшебных чисел. Пользуйтесь именованными константами.

Это относится не только к числам, но и к строкам-константам. Присмотритесь к *любым* литералам, имеющимся в вашем коде, особенно если они встречаются многократно. Не лучше ли использовать вместо них именованные константы?

Выделяйте важные фрагменты кода

Старайтесь выделить важный код на фоне обычного материала. В нужном месте следует привлечь внимание читателя. Для этого есть ряд приемов. Например:

- Разумно располагайте объявления в классе. Сначала должна идти информация об открытых объектах, поскольку именно она нужна пользователю класса. Закрытые детали реализации следует располагать в конце, поскольку они менее интересны большинству читателей.
- По возможности скройте всю несущественную информацию. Не оставляйте ненужного мусора в глобальном пространстве имен. В C++ есть идиома pimpl, позволяющая скрыть детали реализации класса. (Meyers 97)
- Не прячьте важный код. Не пишите в строке больше одного оператора и сделайте этот оператор простым. Язык *позволяет* писать очень изобретательные операторы цикла `for`, в которых вся логика укладывается в одной строке с помощью множества запятых, но такие операторы трудно читать. Избегайте их.
- Ограничьте глубину вложенности условных операторов. В противном случае за нагромождением `if` и скобок трудно заметить обработку действительно важных случаев.

Важные участки кода должны выделяться на общем фоне и быть легко читаемыми. Спрячьте все, что не должно интересовать клиентов.

Объединяйте взаимосвязанные данные

Вся связанная между собой информация должна находиться в одном месте. В противном случае вы заставите читателя не только скакать через обручи, но еще и разыскивать эти обручи с помощью ESP. API для каждой компоненты должен быть представлен одним файлом. Если взаимосвязанной информации оказывается слишком много, чтобы представить ее в одном месте, стоит пересмотреть архитектуру кода.

По возможности объединяйте объекты с помощью языковых конструкций. В C++ и C# можно объединять элементы внутри одного *пространства имен*. В Java средством объединения служит механизм пакетов. Связанные друг с другом константы можно определить в перечислении.

Старайтесь группировать родственную информацию. Делайте эту группировку наглядной с помощью средств языка.

Снабжайте файлы заголовками

Помещайте в начале файла блок комментариев с описанием содержимого файла и проекта, к которому он относится. Это не требует особого труда, но приносит большую пользу. Тот, кому придется сопровождать этот файл, получит хорошее представление о том, с чем он имеет дело.

Этот заголовок может иметь особое значение: большинство софтверных компаний по юридическим соображениям требует, чтобы в каждом файле с исходным кодом было заявление об авторских правах. Обычно заголовки файлов выглядят примерно так:

```
/*********************************************************
 * File: Foo.java
 * Purpose: Foo class implementation
 * Notice: (c) 1066 Foo industries. All rights reserved.
 *********************************************************/
```

Правильно обрабатывайте ошибки

Помещайте обработку всех ошибок в наиболее подходящий контекст. Если возникает проблема чтения/записи диска, ее нужно обрабатывать в том коде, который занимается доступом к диску. Для обработки этой ошибки может потребоваться сгенерировать другую ошибку (типа исключительной ситуации «не могу загрузить файл»), передав ее на более высокий уровень. Это означает, что на каждом уровне программы ошибка должна быть точным описанием проблемы *в его контексте*. Нет смысла обрабатывать ошибку, связанную со сбоем диска, в коде интерфейса пользователя.

Самодокументируемый код помогает читателю понять, где возникла ошибка, что она означает и каковы ее последствия для программы в данный момент.

Не выводите бессмысленных сообщений об ошибках. В зависимости от контекста представьте наиболее уместную информацию.

Пишите осмысленные комментарии

Итак, мы попытались избежать написания комментариев с помощью других косвенных методов документирования кода. Но после того как

Самосовершенствование

Как научиться более успешно писать самодокументируемый код? Обратимся снова к процессу написания книг и посмотрим, не найдется ли там чего-нибудь полезного.

Есть простое правило совершенствования писательского мастерства: *больше читайте, и вы станете писать лучше*. Критическое чтение работ признанных мастеров помогает понять, какие приемы хороши, а какие нет. При этом вы добавляете в свой арсенал новые приемы и идиомы.

Точно так же, читая большие объемы кода, вы совершенствуетесь как программист. После чтения достаточного количества хорошего кода запах скверного кода начинает чувствоваться за версту. Сотрудники таможни просматривают ежедневно такое количество паспортов, что при виде поддельного паспорта у них мгновенно срабатывает чутье. Даже тщательные подделки не проходят мимо их глаз. Точно так же плохой код бросается в глаза, когда вырабатывается чутье на его отличительные признаки.

Имея такой опыт, вы естественным образом станете применять хорошую практику в собственном коде. Вы сами начнете замечать, что написали плохой код, и потому будете испытывать неприятное чувство.

вы приложили все усилия, чтобы написать понятный код, все остальное нужно снабдить комментариями. Чтобы код было легко понять, его нужно дополнить *уместным* объемом комментариев. Каким именно?

Помещайте в код комментарии только в том случае, если не удается облегчить его понимание иными способами.

Сначала попробуйте воспользоваться другими приемами. Например, проверьте, нельзя ли сделать код понятнее, изменив имя или создав вспомогательную функцию, и таким образом избежать комментирования.

Практические методологии самодокументирования

В завершение этой главы мы сравним два конкретных метода документирования кода. Помните, что эти методы менее предпочтительны, чем те, которые мы уже рассмотрели. Как сказано у Кернигана и Плоэра, «нужно не документировать плохой код, а переписать его заново». (Kernighan Plaugher 78)

Грамотное программирование

Грамотное, или *литературное, программирование* (*literate programming*) – это экстремальная технология самодокументируемого кода, предложенная знаменитым специалистом в вычислительной технике Дональдом Кнутом. Он написал книгу под этим названием, в которой и описал эту технологию (Knuth 92). Это радикальная альтернатива традиционной модели программирования, хотя некоторые считают, что период грамотного программирования был крупной неудачей в карьере Д. Кнута. Но даже если не принимать эту технологию за единственно верное и правильное учение, кое-что полезное из нее можно извлечь.

Лежащая в основе идея проста: нужно писать не программу, а документ. Язык документации тесно привязан к языку программирования. Ваш документ в основном описывает то, что программируется, но при этом допускает компиляцию в нужную программу. Таким образом, исходный код – это документация, и наоборот.

Грамотную программу пишут почти как рассказ, который легко или даже интересно читать человеку. На нее не накладываются порядок и ограничения, требуемые синтаксическим анализатором. Но это не следует воспринимать как язык, в котором комментарии поставлены с ног на голову: скорее это поставленный с ног на голову метод программирования. Грамотное программирование представляет собой особый образ мышления.

Первоначально Кнут соединил TEX (язык разметки для набора документов) с C и создал систему под названием *WEB*. Инструментарий грамотного программирования анализирует файл программы и генерирует либо форматированную документацию, либо исходный код, который можно обработать традиционным компилятором.

Конечно, это еще один вид технологии программирования, как структурное программирование или объектно-ориентированное программирование. Качество документации при этом не гарантируется. Оно, как обычно, зависит от программиста. Однако в грамотном программировании акцент смещается на создание описания программы, а не на создание кода, который ее реализует.

По-настоящему достоинства грамотного программирования проявляются на этапе сопровождения продукта. Наличие документации хорошего качества и достаточного объема значительно облегчает сопровождение исходного кода.

У грамотного программирования много полезных качеств:

- Грамотное программирование делает упор на написание документации.
- Оно заставляет взглянуть на код с другой стороны, поскольку в процессе работы необходимо писать объяснения и обоснования.

- При внесении изменений в код появляется больше шансов, что произойдет соответствующее обновление документации, поскольку одно и другое удобно располагается рядом.
- Обеспечивается наличие всего лишь одного документа для всего объема кода. Всегда присутствует версия документации, соответствующая коду, с которым вы работаете; это и *есть* сам код.
- Грамотное программирование способствует включению в документацию элементов, обычно отсутствующих в комментариях. Например: описание применяемых алгоритмов, доказательства корректности, обоснование проектных решений.

Тем не менее грамотное программирование не панацея. У него есть ряд серьезных недостатков:

- Программы по этой технологии труднее писать, поскольку большинству программистов она кажется неестественной. Мы не склонны рассматривать код как печатный документ, который необходимо форматировать. Более привычно мысленно моделировать потоки управления и взаимодействующие объекты.
- Возникает необходимость в дополнительных этапах компиляции, что замедляет разработку программ. Хорошей инструментальной поддержки пока *не существует*.

  Обрабатывать грамотные программы весьма тяжело, поскольку компилятору необходимо извлечь все фрагменты программы и собрать их в правильном порядке. Очень удобно, что документ можно писать в любом порядке, но в C к коду предъявляются определенные требования, например директивы `#include` должны находиться в начале. В результате на практике приходится прибегать к некоторым компромиссам.
- В некоторых случаях документируется код, для которого это совсем не требуется. Случаи, когда не документируется большой объем простого кода, также часто встречаются. Тогда это уже нельзя назвать хорошей грамотной программой, и вся затея не имеет смысла.

  Если описывается *буквально все*, то в этом шуме можно пропустить важные куски документации.
- Кнут рассматривал *программиста как литератора*. Но есть много программистов, которые не напишут рассказа под страхом смерти, хотя пишут прекрасный код. Возможно, они представляют собой исключение, но не каждый хороший программист может проявить способности к грамотному программированию.
- Тесная привязка документации к коду может составить проблему. Например, вы фиксируете окончательную версию своего кода, не разрешая вносить изменения, но документация требует доработки. Изменить документацию – значит, изменить исходный код. В результате появляется окончательная версия исполняемого кода и окончательная версия документации к тому же коду, которые

нужно связать вместе, что представляет собой административную проблему.

В одной из последующих глав обсуждаются спецификации программного обеспечения. Как грамотное программирование относится к спецификациям? Грамотная программа не может заменить функциональную спецификацию с описанием работы, которую нужно сделать. Однако должна существовать возможность разработать грамотную программу по такой спецификации. Грамотная программа на практике скорее представляет собой комбинацию обычного кода со спецификацией проектирования и реализации.

Инструментарий документирования

Существует целый вид программных средств, занимающих промежуточное положение между технологией грамотного программирования и созданием внешних спецификаций. Эти инструменты генерируют документацию из исходного кода, вытаскивая из него блоки особым образом форматированных комментариев. Особенно модной такая техника стала после того, как Sun ввела Javadoc в качестве базовой компоненты платформы Java. Вся документация Java API генерируется с помощью Javadoc.

Чтобы лучше разобраться в том, как это работает, рассмотрим пример. Формат комментариев может немного отличаться, но в принципе документирование класса `Widget` осуществляется так:

```
/**
 * Это документация класса Widget.
 * Программа определяет начало комментария
 * с помощью особой последовательности '/**' .
 *
 * @author Имя автора
 * @version Номер версии
 */
class Widget
{
public:
    /**
     * Здесь документируется метод.
     */
    void method();
};
```

Средство документирования анализирует все файлы вашего проекта, извлекает документацию, строит перекрестную базу данных всей извлеченной информации и выводит милый документ со всей этой информацией.

Документировать можно практически любой код, который вы пишете: классы, типы, функции, параметры, флаги, переменные, пространства

имен, пакеты и т. д. Есть средства для получения многообразных данных, которые позволяют:

- Задать информацию об авторских правах
- Зафиксировать дату создания
- Получить информацию о перекрестных ссылках
- Пометить код как устаревший
- Дать синопсис для краткой справки
- Описать все параметры функции

Существует много средств документирования – как свободно распространяемых, так и коммерческих. Мы уже упомянули Javadoc; другими популярными инструментами являются NDoc для C# и превосходное средство Doxygen (*www.doxygen.org*).

Это отличный метод документирования, позволяющий комментировать код с разумной степенью детализации и не составлять при этом отдельной спецификации. Кроме того, документацию можно читать прямо в файле исходного кода, что бывает очень удобно.

Средства документирования имеют много достоинств:

- Как и грамотное программирование, этот подход поощряет составление документации и поддержание ее актуальности.
- Не требуется дополнительного шага для получения кода, допускающего компиляцию.
- Технология более естественна и не требует значительных изменений или долгого изучения. Поскольку код позволяет генерировать документ, не требуется искусственно делать код похожим на книгу или решать утомительные задачи структурирования текста.
- Средства документирования поддерживают богатые возможности поиска, перекрестных ссылок и выделения кода.

Следует, однако, учитывать последствия подхода к документированию, основанного на комментариях:

- В отличие от грамотного программирования он имеет практическую ценность только для документирования API, но не внутреннего кода. На уровне операторов нужно пользоваться обычными комментариями.
- Взглянув на файл исходного кода, трудно получить представление о его содержимом, поскольку оно испещрено бесчисленными комментариями, связанными с документированием. Приходится пользоваться обзорными функциями, предоставляемыми средством документирования. Результат может быть красиво отформатирован, но это может доставить неудобства при работе в редакторе кода.

Пользуйтесь инструментарием грамотного документирования для автоматической генерации документации по вашему коду.

Несмотря на мощь данного способа составления документации, он не гарантирует создание *хорошей* документации. Вот несколько полезных правил для избежания ошибок:

- Для каждого открытого элемента составьте описание из одного-двух предложений. Не перегружайте описание текстом. Длинные тексты отнимают время на чтение, и их трудно обновлять. Не занимайтесь болтовней.
- Описывайте переменные или параметры, если их назначение неясно. Но если оно с очевидностью следует из их имен, документирование излишне. Не нужно документировать каждую мелочь, если она не имеет особого значения. Программа включит этот элемент в документацию, но без текстового описания.
- Если одни параметры функции являются входными, а другие выходными, нужно отметить это в их описании. Синтаксический механизм для обозначения такого разделения есть лишь в немногих языках, поэтому в документации нужно указать его явным образом.
- Документируйте все пред- и постусловия функций, возбуждаемые ими исключения и побочные эффекты, если таковые имеются.

Резюме

Мастерство писателя заключается в создании контекста,
в котором могут думать читатели.

Эдвин Шлоссберг

Мы пишем код для того, чтобы передать информацию. Код без документации опасен и малоинформативен. Его трудно сопровождать. *Плохая* документация также создает проблемы: она либо вводит читателя в заблуждение, либо делает программу ненадежной и требующей дополнительных разъяснений.

Часто единственной документацией по программе оказывается ее собственный код. Если код самодокументируемый и написан ясно, это отчасти решает проблему. Самодокументируемый код не возникает из ничего; он требует тщательного обдумывания. Получаемый при этом код выглядит так, будто написать его было просто.

Грамотное программирование – это один из способов (весьма экстремальный) написания самодокументируемого кода. Менее экстремальный подход требует участия программных инструментов документирования. Эти инструменты легко создают документацию по API, но не всегда могут заменить письменные спецификации.

Хорошие программисты...	**Плохие программисты...**
• Стараются писать понятный самодокументируемый код	• Гордятся своим умением писать непостижимый спагетти-код

Хорошие программисты...

- Стараются составить лишь самую необходимую документацию
- Заботятся о программистах, которым придется сопровождать их код

Плохие программисты...

- Увиливают от составления всякой документации
- Не следят за своевременным обновлением документации
- Считают, что «если мне было трудно это написать, то пусть другие помучаются, пытаясь в этом разобраться»

См. также

Глава 3. Что в имени тебе моем?

Выбор правильных имен – мощное средство создания самодокументируемого кода.

Глава 5. Заметки на полях

Если приходится писать комментарии, то нужно делать это правильно.

Глава 19. Спецификации

Код должен быть самодокументируемым, но по многим причинам все же нужны отдельные спецификации.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 599.

Вопросы для размышления

1. Группировка взаимосвязанного кода делает эти связи более заметными. Как можно осуществить это группирование? Какие методы документируют эти связи наиболее заметным образом?
2. Следует избегать в коде *волшебных чисел*. Считать ли ноль магическим числом? Какое имя вы дали бы константе, представляющей ноль?
3. В самодокументируемом коде для передачи информации активно используется контекст. Приведите пример и покажите, как некоторое имя по-разному интерпретируется в различных функциях.
4. Можно ли реально рассчитывать на то, что человек, впервые видящий некоторый самодокументируемый код, сможет в нем полностью разобраться?
5. Если код действительно самодокументируемый, какой объем дополнительной информации необходим?
6. В каких случаях может потребоваться, чтобы в каком-то фрагменте кода мог разобраться кто-либо иной, кроме его автора?
7. Эту простую С-функцию *пузырьковой сортировки* можно усовершенствовать. Что именно в ней нехорошо? Напишите улучшенный, самодокументируемый вариант.

```
void bsrt(int a[], int n)
{
    for (int i = 0; i < n-1; i++)
        for (int j = n-1; j > i; j--)
            if (a[j-1] > a[j])
            {
                int tmp = a[j-1];
                a[j-1]  = a[j];
                a[j]    = tmp;
            }
}
```

8. Работа со средствами документирования кода обнаруживает ряд интересных проблем. Выскажите свое мнение по следующим вопросам:
   a. Следует ли при проверке документации *проверять* также и *код*, сравнивая с комментариями в файлах исходного кода, или *проверять спецификации*, сравнивая со сгенерированной документацией?
   b. Где следует документировать протоколы и прочие не относящиеся к API вопросы?
   c. Документируете ли вы закрытые/внутренние функции? Если это код C/C++, где поместить такую документацию – в файле заголовка или в файле реализации?

   d. Если система большая, что лучше – создать один большой документ по API или несколько меньших по отдельным областям? В чем преимущества каждого из подходов?

9. Если вы работаете с кодом, в котором отсутствует грамотная документация, и вам необходимо внести изменения или добавить новые методы или функции, что правильнее – снабдить их грамотными комментариями или оставить недокументированными?
10. Можно ли написать самодокументируемый код на языке ассемблера?

Вопросы личного характера

1. Какой из встречавшихся вам код был лучше всего документирован? В чем были его особенности?
   a. Было ли у этого кода много внешних спецификаций? Многие ли из них вы прочли? Можно ли быть уверенным, что вы достаточно знаете о коде, вовсе не прочтя их?
   b. В какой, по вашему мнению, мере качество этой документации было обязано личному стилю программирования автора, а в какой – требованиям и наставлениям той организации, в которой он работал?
2. Если вы программируете на разных языках, различаются ли ваши стратегии составления документации для каждого из них?
3. Каким образом вы выделили важные фрагменты в последнем коде, который писали? Постарались ли вы сделать малозаметной закрытую информацию?
4. Если вы работаете над кодом в составе группы, часто ли ее участники обращаются к вам с просьбой пояснить работу вашего кода? Смогли бы вы избавиться от таких отвлечений путем лучшего документирования кода?

5

Заметки на полях

Как писать комментарии к коду

В этой главе:

- Зачем нужны комментарии?
- Сколько комментариев требуется?
- Как писать эффективные комментарии

Комментарии свободны, но факты священны.

Чарлз Престуич Скотт

Комментарии похожи на личное мнение. Вы вправе иметь его, но это не значит, что оно справедливо. В данной главе мы ненадолго остановимся на деталях их составления. Составление комментариев – не такая простая задача, как может показаться.

О том, как писать комментарии, обычно рассказывается на самых начальных шагах обучения программированию. При этом сообщается, что комментарии облегчают чтение кода, и рекомендуется писать их побольше. Но на самом деле нужно больше заботиться об их *качестве*, чем о *количестве*. Комментарии – это спасательный трос, подсказка для памяти, путеводитель по коду. Они заслуживают уважительного отношения к себе.

Я настраиваю свой редактор кода так, чтобы он показывал комментарии зеленым цветом.

Благодаря этому после загрузки файла с исходным кодом я мгновенно получаю представление о качестве кода и о том, насколько легко будет с ним работать. Когда я вижу достаточное количество зеленого цвета в правой части экрана, у меня улучшается настроение. В противном случае мне хочется заварить себе крепкий кофе, прежде чем двигаться дальше.

Комментарии могут превратить плохой код в хороший, непомерно сложную и непостижимую логику – в чудесный набор прозрачных алгоритмов. Но не будем преувеличивать – есть вещи поважнее комментариев. Если вы написали действительно хороший код, ваши комментарии служат лишь *окончательной отделкой*, придающей продукту законченный эстетический вид, а не средством скрыть его изъяны и недостатки.

Хорошие комментарии позволяют избавиться от устрашающего впечатления при виде кода. Но это не волшебная приправа, превращающая испорченный продукт во вполне съедобный.

Что есть комментарий в коде?

Это важный раздел! Вопрос может показаться надуманным. Кто же не знает, что такое комментарии в коде? Тем не менее вопрос заслуживает бóльших размышлений, чем может показаться на первый взгляд.

Синтаксически комментарий представляет собой блок исходного текста, который игнорируется компилятором. Поместить в него можно что угодно – хоть имена своих внуков, хоть цвет любимой рубашки; компилятор и глазом не моргнет, встретив такое в файле.[1]

Семантически комментарии составляют разницу между темной лесной дорогой и хорошо освещенным шоссе. Комментарий – это аннотация лежащего рядом кода. Можно воспользоваться им как маркером, чтобы выделить конкретную область задач, или как средством документирования в заголовке файла. С помощью комментариев можно описать алгоритм, что поможет программисту, который будет сопровождать код (им можете оказаться вы сами), или разметить границы между функциями, что позволит легче ориентироваться в файле исходного кода.

Комментарии предназначены человеку, а не компьютеру. В этом смысле они представляют собой самый гуманистически ориентированный элемент в постройке, которой является программа. Они – как декоративные кирпичи в сравнении с бетонными блоками. Чтобы каче-

[1] Разумеется, та штука, которая заглотнет и выплюнет ваши комментарии, в каждом языке своя. В C/C++ есть зверь-препроцессор, который сожрет комментарии перед началом этапа компиляции. В других языках сам компилятор выкинет комментарии при лексическом анализе. В интерпретируемых языках обилие комментариев может замедлить выполнение, поскольку интерпретатору придется просмотреть имена всех ваших внуков.

ство комментариев было выше, нужно учитывать, каковы реальные потребности человека и как он читает код.

Комментарии в коде – не единственная помещаемая в нем документация. Комментарии – не спецификации. Это не проектные документы. И это не справочники по API.[1] Тем не менее они представляют собой важную форму документации, которая всегда физически связана с кодом (если кто-нибудь злонамеренно не уничтожит их). Близость их к коду приводит к тому, что комментарии чаще обновляют и чаще читают в контексте. Это механизм внутреннего документирования.

Порядочный программист обязан писать хорошие комментарии.

Как выглядят комментарии?

Конечно же, они зеленые... По крайней мере, у меня.

Комментарии C помещаются в блоках между комбинациями /* и */ и могут занимать произвольное количество строк. В C++, C99, C# и Java есть, кроме того, однострочные комментарии, следующие за //. В других языках есть аналогичные средства комментирования *блоками* и *в строке*, но синтаксис иной.

Конечно, это элементарный вопрос. Но в использовании маркеров комментариев встречаются тонкие различия. Ниже мы рассмотрим соответствующие примеры. Однако ко всем системам комментирования, изобретательно использующим тонкие синтаксические различия, следует относиться с осторожностью.

Сколько комментариев требуется?

Сильный текст всегда краткий.

Уильям Странк Мл.

Заботиться следует о качестве комментариев, а не об их количестве, поэтому важен не объем комментариев, а их содержание. О нем будет сказано в следующем разделе.

Тех, кто изучает программирование, заставляют писать комментарии, и в большом количестве. Но *слишком обширные* комментарии могут быть недостатком – важные фрагменты кода оказываются затерянными среди потока слов. Когда вам приходится продираться через пространные комментарии, вместо того чтобы читать сам код, качество последнего снижается.

Я бы сравнил это с искусством хорошего музыканта. Если вы играете в оркестре, то ваша задача не в том, чтобы при каждом удобном случае

[1] Если только вы не пользуетесь технологией, о которой рассказано в разделе «Грамотное программирование» на стр. 104.

поднимать как можно больше шума. Чем больше вы играете на своем инструменте, тем сложнее становится суммарное звучание и тем хуже музыка. Точно так же излишние комментарии запутывают код. Хорошему музыканту не нужно думать о том, *когда же ему остановиться и дать возможность сыграть кому-то другому*. Хороший музыкант играет свою партию, только когда она вносит что-то ценное в музыку. Смысл в том, чтобы сыграть тот *минимум*, который необходим для создания лучшего звучания. Простор составляет красоту. Комментарии нужно писать лишь тогда, когда они составляют действительную ценность.

Учитесь писать ровно столько комментариев, сколько необходимо. Отдайте предпочтение качеству, а не количеству.

Тот, кто будет читать ваши комментарии, может прочесть сам код, поэтому стремитесь в большей мере использовать для документирования *сам код*, а не комментарии. Во всяком случае к коду больше доверия – комментарии имеют скверную привычку врать. Рассматривайте операторы в своем коде как первый уровень комментирования и сделайте их самодокументируемыми.

На деле хорошему коду комментарии не нужны, потому что в нем все должно быть очевидно. Если вы даете функциям такие имена, как `f()` и `g()`, то неизбежно приходится описывать их в комментариях, но какая-нибудь `someGoodExample()` не требует никаких комментариев. Сразу видно, что это имя функции, дающей хороший пример.

Не пожалейте труда, чтобы ваш код не требовал поддержки в виде уймы комментариев.

Чем меньше комментариев, тем реже встретятся плохие комментарии.

Что помещать в комментарии?

Прежде чем станешь писать, научись порядочно мыслить!

Гораций

Лучше совсем без комментариев, чем плохие комментарии – они дезинформируют читателя и вводят его в заблуждение. Что же следует помещать в комментарии? Вот несколько основных советов, как повысить качество комментариев.

Описывайте «почему», а не «как»

Это ключевой момент, поэтому прочтите данный абзац дважды. Затем съешьте эту страницу. Комментарии не должны описывать, *как* работает программа. Это видно из кода. В конце концов, исчерпывающим описанием того, как работает код, является сам код. К тому же он на-

писан четко и ясно, правда? Вместо этого опишите, *почему* код написан так, а не иначе, или какую задачу решает нижеследующий блок операторов.

Постоянно следите за тем, как писать: `/* обновить структуру WidgetList из GlbWLRegistry */` или `/* сохранить сведения о виджете на будущее */`. Комментируя одно и то же, во втором варианте вы излагаете смысл кода, а в первом просто описываете его действие.

При сопровождении фрагмента кода его задача меняется реже, чем способ ее решения, и поддерживать актуальность комментариев такого рода гораздо проще.

Хорошие комментарии объясняют «почему», а не «как».

В комментарии можно также объяснить, почему вы выбрали тот или иной способ реализации. Когда есть две стратегии реализации и вы предпочли одну из них, возможно, стоит объяснить в комментарии, на чем основано ваше решение.

Не нужно описывать код

Бесполезность описательных комментариев бывает очевидна: `++i; // увеличить i`. Бывают более тонкие случаи: пространный комментарий, описывающий сложный алгоритм, за которым следует реализация самого алгоритма. Нет нужды старательно формулировать алгоритм обычным языком, если только он действительно не настолько сложен, что иначе в нем не разобраться. Не исключено, что в этом случае следует переписать алгоритм заново, а не комментировать его.

Один источник для каждого факта. Не копируйте код в комментариях.

Не подменяйте код

Если вы видите комментарий, сообщающий что-то, что может быть выражено средствами самого языка (например, `// доступ к этой переменной должен осуществляться только из класса foo`), попытайтесь выразить это синтаксическим способом.

Заметив за собой, что пишете уйму комментариев для объяснения работы сложного алгоритма, остановитесь. Сначала отшлепайте себя за попытку комментировать происходящее. Затем подумайте, нельзя ли изменить код или алгоритм, чтобы сделать их понятнее.

- Попробуйте разбить код на несколько функций, дав им хорошие имена, чтобы отразить логику программы.
- Не пишите комментарии, описывающие смысл переменной, – лучше дайте ей подходящее имя. Комментарий, который вы собира-

лись написать, часто и подскажет, каким должно быть имя этой переменной!

- Если вы документируете условие, которое должно выполняться всегда, может быть, правильнее написать оператор контроля.
- Не следует преждевременно оптимизировать (и тем самым делать менее понятным) свой код.

Обнаружив, что вы пишете многословные комментарии, описывающие ваш код, остановитесь и задумайтесь. Не признак ли это того, что существует некая проблема более высокого порядка?

Как сделать комментарии полезными

Для того чтобы поднять качество комментария, обычно нужно совершить несколько итераций – как и для кода. Ваши комментарии должны удовлетворять следующим требованиям:

Документировать необычное

Если какие-то фрагменты кода являются необычными, неожиданными или удивительными, опишите их в комментарии. Вы будете вознаграждены за это, если впоследствии, когда вы совершенно позабудете о сути проблемы, придется вернуться к этому коду. Если вы прибегли к особому обходному пути, скажем, чтобы решить проблемы с операционной системой, отметьте это в комментарии.

С другой стороны, незачем комментировать очевидное. Помните: *не копируйте код в комментариях!*

Говорить правду

Когда комментарий комментарием не является? Когда он лжет. Пусть вы никогда умышленно не пишете лжи, но очень легко случайно допустить искажение истины, особенно при модификации уже прокомментированного кода. Вносимые в код изменения легко могут сделать комментарии неточными. Раздел «Работа с комментариями» на стр. 126 описывает тактику борьбы с этим явлением.

Быть стоящими

Короткие загадочные комментарии могут быть остроумными, но *избегайте* их. Они только мешают и смущают. Избегайте непристойностей, шуток, которые понятны только посвященным, и чрезмерно критических комментариев. Никогда не известно, в чьих руках может оказаться код через месяц или год, поэтому не пишите комментариев, которые могут поставить вас в будущем в неловкое положение.

Быть понятными

Ваш комментарий должен служить аннотацией и объяснением кода. Избегайте двусмысленности. Будьте как можно более конкретны (не нужно писать диссертацию по каждой строчке). Если кто-то,

прочтя ваш комментарий, не понял его смысла, значит, вы сделали свой код только хуже и затруднили его понимание.

Быть вразумительными

Необязательно писать все комментарии законченными и грамматически правильными предложениями. Однако комментарий должен быть удобочитаем. Замысловатые сокращения обычно только заводят читателя в тупик, особенно если они не на его родном языке.

Думайте, что пишете в комментариях; не давите бездумно на клавиши. Прочтите комментарий снова в контексте кода. Ту ли информацию он содержит?

Случай из жизни

Однажды я консультировал компанию со смешанным составом программистов: для одних родным языком был английский, а для других – греческий. Все греки прекрасно говорили по-английски, но ни один из англоязычных программистов не знал греческого (что неудивительно).

Один из программистов-греков писал комментарии по-гречески и отказывался изменить язык, несмотря на высказанные ему вежливые просьбы. Англоязычные программисты были не в состоянии прочесть эти комментарии, потому что они были для них «китайской грамотой».

Не отвлекаться

Комментарии должны пояснять сопутствующий код, поэтому следует избегать в них лирических отступлений. Комментарии должны вносить *новую ценность*. Избегайте комментариев, в которых содержатся:

Описание былого

Нет необходимости записывать, как то или иное *делалось раньше*. Для этого есть система контроля версий. Не нужно копировать в комментариях старый код или старые алгоритмы.

Ненужный код

Не нужно удалять код путем заключения его в комментарии. Это приводит к путанице. Даже при отладке *в пожарном режиме* (без штанов, без отладчика и без `printfs`), *не* прячьте код, помещая его в блок комментариев. Воспользуйтесь директивами C `#ifdef 0 . . . #endif` или аналогичными. Такие конструкции допускают вложенность, и их назначение более понятно (что особенно важно, если потом вы забудете сделать приборку).

Символьное художество

Избегайте ASCII-картинок и аналогичных попыток выделить код художественным образом. Вот пример неудачной идеи:

```
aBadExample(n, foo(wibble));
//             ^^^
//             Моя любимая
//             функция
```

В редакторах с пропорциональными шрифтами это не имеет никакого смысла. Комментарии не должны увеличивать затраты труда на сопровождение!

Концы блоков

Некоторые программисты помещают комментарий в конце каждого блока управления, например пишут `// end if (a < 1)` после закрывающей скобки оператора `if`. Это избыточный вид комментария; его приходится отметать, чтобы действительно понять код. Конец блока должен располагаться на той же странице, где его начало, а форматирование должно быть таким, чтобы начало и конец были четко видны. Любых лишних слов нужно избегать.

На практике

Проиллюстрируем принципы комментирования следующим примером. Рассмотрим фрагмент кода C++. Даже если не критиковать идиоматическую сторону, код не вполне понятен.

```
for (int i = 0; i < wlst.sz(); ++i)
k(wlst[i]);
```

Да, здесь можно кое-что улучшить, чем мы и займемся. Код станет менее загадочным, если применить к нему разумное форматирование и добавить немного комментариев:

```
// Итерация по всем виджетам в списке
for (int i = 0; i < wlst.sz(); ++i)
{
    // Выведем этот виджет
    k(wlst[i]);
}
```

Гораздо лучше! Теперь совершенно ясно, что должен делать этот код. Но все же я удовлетворен не до конца. Если правильно выбрать имена функций и переменных, комментарии вообще не понадобятся, поскольку код опишет себя сам.

```
for (int i = 0; i < widgets.size(); ++i)
{
    printWidget(widgets[i]);
}
```

Обратите внимание, что я не стал переименовывать `i` во что-то более замысловатое. Это переменная цикла с очень небольшой областью видимости. Назвать ее `loopCounter` было бы излишеством и, пожалуй, *затруднило* бы чтение кода.

Ничего удивительного, что в итоге мы вообще остались без комментариев. Помните совет Кернигана и Плоера: плохой код нужно не комментировать, а переписывать заново. (Kernighan Plaugher 78)

Замечание об эстетичности

Несомненно, вам приходилось сталкиваться с горячими обсуждениями того, как нужно форматировать комментарии. Я не намерен наставлять вас на путь истинный в этом вопросе (его просто не существует), но есть ряд важных аспектов, которые нужно учитывать. Отнеситесь к ним как к общим принципам, сообразуясь с вашими личными вкусами, а не как к безусловным требованиям.

Единообразие

Все комментарии должны быть четкими и единообразными. Выберите определенную структуру для своих комментариев и пользуйтесь ею повсеместно. У каждого программиста существуют свои представления об эстетике, поэтому выберите то, что больше подходит вам. Если есть внутрифирменный стиль, пользуйтесь им, либо изучите какой-нибудь хороший код и заимствуйте его стиль.

Мелкие проблемы форматирования в комментариях могут показаться тривиальными. Например, следует ли начинать комментарии с прописных букв? Однако если прописные или строчные буквы в комментариях выбираются случайным образом, это свидетельствует о рыхлости кода и недостаточной продуманности его автором.

Четкие блочные комментарии

Большое достоинство редакторов, выделяющих синтаксис, состоит в их способности особым образом оформлять комментарии. Но не следует излишне полагаться на эту функцию. Возможно, кто-то будет читать монохромную распечатку вашего кода или работать с ним в редакторе без расцвечивания синтаксиса. Комментарии должны легко читаться и в этом случае.

Возможны несколько стратегий, особенно в отношении блочных комментариев. Размещение маркеров начала и конца (т. е. /* и */ в C и C++) на отдельных строках делает блоки заметнее. Один крайний символ, пущенный по левому краю блочного комментария, придает ему вид единого целого:

```
/*
 * Такой блочный комментарий
```

```
 * смотрится гораздо лучше
 * внутри массива кода
 */
```

Это выглядит значительно лучше, чем возможная альтернатива:

```
 /*
комментарий, занимающий
    несколько строк
без крайнего символа.
*/
```

Во всяком случае текст комментария должен быть выровнен, а не образовывать зубчатую линию.

Отступы в комментариях

Комментарий не должен вклиниваться в код, нарушая его логический поток. Сохраняйте для него такой же отступ, как у окружающего кода. Благодаря этому создается привязка комментария к определенному уровню кода. Мне всегда трудно читать код, подобный следующему:

```
void strangeCommentStyle()
{
    for (int n = 0; n < JUST_ENOUGH_TIMES; ++n)
    {
// Вот полезный комментарий к следующей строке.
        doSomethingMeaningful(n);
// По правде, он чрезвычайно сбивает меня с толку.
        anotherUsefulOperation(n);
    }
}
```

В цикле без фигурных скобок (и без того неудачный стиль) не пишите комментарий перед единственным оператором цикла – это может привести к самым неприятным последствиям. Если вам необходим комментарий в этом месте, заключите весь фрагмент в фигурные скобки. Это будет гораздо безопаснее.

Комментарии в конце строки

Обычно комментарии пишут в отдельных строках, но иногда короткий комментарий может *следовать* за оператором кода в той же строке. В таком случае полезно оставить пустое пространство между кодом и комментарием, чтобы четко их разграничить. Например:

```
class HandyExample
{
public:
    ... какой-то открытый код ...
private:
    int appleCount;           // Комментарии в конце строки
```

```
    bool isFatherADustman;    // отделите их
    int favoriteNumber;       // от кода
};
```

Это хороший пример того, как расположение комментариев может улучшить внешний вид кода. Если бы все комментарии шли сразу после объявления переменной, они образовали бы неаккуратную неровную линию, и для их чтения потребовалось бы больше вертеть глазами.

Помощь в чтении кода

Обычно комментарии помещают *над* кодом, который они описывают, а не под ним. Благодаря этому исходный код можно читать сверху вниз почти как книгу. Комментарий должен подготовить читателя к тому, что последует дальше.

В сочетании с пробельными символами комментарии помогают разбить код на «абзацы». Вначале комментарий объясняет, какова задача нескольких следующих строк кода, затем идет непосредственно код, потом пустая строка, затем очередной блок. При таком стиле комментарий с предшествующей пустой строкой создает впечатление начала абзаца, тогда как комментарий, втиснутый между двумя строками кода, более похож на оператор в скобках или на сноску.

Комментарии – часть повествования кода. Размещайте их так, чтобы порядок чтения был естественным.

Стиль должен обеспечивать легкость сопровождения

Разумно избрать такой стиль комментариев, чтобы их поддержка обходилась дешевле, иначе вам придется возиться с комментариями, вместо того чтобы писать код.

Некоторые кодировщики, пишущие на C, пишут блочные комментарии, ограниченные колонками звездочек с левого *и* правого краев. Возможно, это красиво, но слишком много труда нужно положить на то, чтобы выровнять абзац текста внутри таких границ. Вместо того чтобы двигаться к следующей задаче, приходится вручную выравнивать звездочки по правому краю. При использовании табуляции дело осложняется: когда другой программист откроет файл в редакторе, где установлен иной шаг табуляции, звездочки окажутся расположенными крайне уродливым образом.

Комментарии в конце строки, рассмотренные выше, требуют определенного труда для выравнивания. Сколько усилий вы готовы на это потратить – решайте сами. Всегда приходится выбирать некий компромисс между приятным видом исходного кода и усилиями по его сопровождению. Я предпочитаю немного потрудиться, чтобы код не выглядел уродливо.

Границы

Комментариями часто пользуются, чтобы разграничить секции кода. Тут берут верх художественные наклонности: программисты применяют различные системы, чтобы отличать главные комментарии (*новый раздел кода*) от второстепенных (*описание нескольких строк функции*). В файле исходного кода, содержащем реализации нескольких классов, основные разделы могут разграничиваться таким комментарием:

```
/*********************************************************
* реализация класса foo
*********************************************************/
```

Некоторые программисты любят помещать большие художественные комментарии перед каждой функцией. Другие отчеркивают функции длинными однострочными комментариями. Я предпочитаю просто пару пустых строк перед каждой функцией. Если ваши функции такие длинные, что требуются видимые маркеры их начала и конца, стоит пересмотреть структуру кода.

Не создавайте длинных строк-разделителей, чтобы выделить каждый комментарий. Таким способом вы ничего не выделите. Код нужно группировать с помощью отступов и структуры, а не символьного художества.

Тем не менее хороший выбор комментариев в качестве границ помогает быстро перемещаться по файлу.

Флажки

Комментарии могут также использоваться в качестве встроенных флажков в коде. Есть ряд общепринятых соглашений. В файлах, работа над которыми не завершена, вы тут и там встретите `//XXX`, `//FIXME` (исправить) или `//TODO` (сделать). Хорошие редакторы с подсветкой синтаксиса могут выделять такие комментарии. `XXX` отмечает опасный или требующий переработки код. `TODO` часто помечает отсутствие некоей функциональности, которую в будущем нужно реализовать.[1] `FIXME` указывает на фрагмент, в котором есть ошибки.

Комментарии в заголовке файла

Каждый файл с исходным кодом должен начинаться с блока комментариев, описывающих его содержимое. Это краткий обзор, предисловие, содержащие важную информацию, которую вы хотели бы сообщить тому, кто откроет этот файл. Если есть такой заголовок, любой программист чувствует доверие к содержимому файла; это свидетель-

[1] С комментариями `TODO` будьте осторожны. Может быть, лучше генерировать исключение `TODO`, не заметить которое нельзя. В этом случае, если вы забудете реализовать отсутствующий код, ваша программа выйдет на запланированный отказ в работе.

ство того, что файл создавался тщательно, а не является лишь черновиком некоего нового кода.

Снабжайте каждый файл исходного кода прологом в виде комментария.

Некоторые выступают за то, чтобы в этом заголовке был список всех функций, классов, глобальных переменных и т. п., содержащихся в данном файле. Но такой подход представляет катастрофу для сопровождения: комментарий быстро оказывается устаревшим. В заголовке файла *уместно* сообщить такие данные, как назначение файла (например, *реализация интерфейса foo*) и заявление об авторских правах, указывающее владельца и права копирования.

Если файл исходного кода автоматически создан в процессе сборки, необходимо каким-либо образом включить в заголовок комментарий, который четко информирует (БОЛЬШИМИ ПРОПИСНЫМИ БУКВАМИ) об источнике происхождения файла. Иначе кто-нибудь по ошибке станет его редактировать, не зная, что потеряет свой труд после очередной сборки.

Комментарий в нужном месте

В этой главе мы заняты в основном комментированием кода – тем, что мы фактически вводим в исходный код. Но по соседству бродят комментарии других видов:

Комментарии системы управления версиями

В системе управления версиями хранится история модификаций каждого файла за время существования проекта. Она связывает с каждой версией определенные метаданные – во всяком случае комментарии программиста, сделанные им при сохранении версии. Кроме того, могут записываться комментарии, сделанные при загрузке, если ведется учет файлов, находящихся в данное время в работе. В этих комментариях можно описать, что вы собираетесь изменить.

Такие комментарии очень ценны, и их нужно тщательно составлять. Они должны быть:

- Краткими (для быстроты просмотра журнала модификаций)
- Точными (не ошибитесь, иначе труд бесполезен)
- Полными (чтобы знать обо всех изменениях, не прибегая к сравнению версий вручную с помощью *diff*)

Регистрируйте, *что* было изменено и *по какой причине*, а не *как* было изменено. О том, какие изменения сделаны, можно узнать, сравнив версии.

Вот тут и место комментариям по поводу прошлого, а также ссылкам на систему контроля и исправления ошибок. Не поддавайтесь искушению поместить эти данные в комментарии к исходному коду. Помните: один факт – один источник.

Файлы README

Эти обычные текстовые файлы, лежащие в тех же каталогах, что и файлы с исходным кодом. Это полезная документация, занимающая промежуточное положение между формальными спецификациями и комментариями в коде. Они часто содержат практическую информацию, например о том, каково назначение каждого файла или какова иерархия файлов; обычно это короткие заметки.

Файлы README часто составляются поспешно и непродуманно либо плохо сопровождаются и являются устаревшими, что недопустимо. Заметив файл README, естественно загрузить его и посмотреть, какая полезная информация в нем есть. Наличие файла README свидетельствует о том, что кто-то сознательно собрал вместе файлы с исходным кодом; нашлось нечто, достойное включения в документ и упоминания.

В заголовке *не* должно быть информации, которая может быстро устареть, такой как сведения об авторах, модификаторах и дате последней модификации. Вряд ли ее станут часто обновлять, и она станет неверной. Эти сведения вам должна сообщить система контроля версий. Не нужно также излагать историю создания файла, описывающую все проведенные с ним модификации. Эти данные есть в системе контроля версий, и дублировать их здесь не нужно. Кроме того, если для того чтобы добраться до первой строки кода, нужно прокрутить 10 страниц истории модификаций, работать с файлом становится утомительно. По этой причине некоторые программисты помещают историю в конец файла, но и в этом случае файл оказывается излишне велик, медленно загружается и работать с ним обременительно.

Работа с комментариями

Комментариями удобно пользоваться при написании кода. Но не злоупотребляйте ими.

Помощь при написании программ

Распространенный подход к написанию программы – сначала смоделировать ее структуру в комментариях, а потом написать код под каждой строчкой комментария. Если вы работаете в таком стиле, то, закончив работу, спросите себя, по-прежнему ли вам нужны оставшиеся

комментарии. Оцените их в соответствии с обсуждавшимися здесь критериями и отредактируйте или удалите. Не двигайтесь дальше без проведения такой ревизии.

Альтернатива – написать новую программу, а потом уже добавить нужные комментарии. Есть опасность забыть довести дело до конца или что комментарии, когда их пишешь, слишком хорошо зная код, окажутся не лучшего качества. Опытный программист пишет комментарии в процессе работы. Практика показывает, какой объем комментариев оправдан.

Не бойтесь пользоваться флажками, типа рассмотренных выше `TODO`, в качестве пометок для самого себя. Это поможет вам не забыть ликвидировать досадные мелкие «хвосты». Такого рода комментарии легко обнаружить в коде и узнать, что еще требует доделки.

Заметки об исправлении ошибок

Частая, но спорная практика – помещение заметок в местах, где исправлены ошибки. Посреди какой-нибудь функции можно натолкнуться на комментарий такого типа:

```
// <ссылка на баг> - заменено на метод blah.foo2()
// поскольку blah.foo() некорректно обрабатывал
// <некоторое условие>
blah.foo2();
```

Написанные с лучшими намерениями (чтобы помочь разобраться в случившемся во время разработки), эти комментарии часто приносят больше вреда, чем пользы. Чтобы понять суть проблемы, вам придется найти ошибку в системе учета ошибок и загрузить предыдущую версию файла, чтобы выяснить, какие изменения были внесены. Редкие исправления ошибок нуждаются в такого рода работе, и новичку, возможно, лучше остаться в блаженном неведении. Такие комментарии на поздних стадиях разработки множатся в числе и при сопровождении засоряют исходный код побочными рассуждениями и устаревшей информацией, отвлекают от главного потока исполнения.

В пользу помещения такого рода комментария может говорить неочевидность исправления – чтобы кто-то, работающий над кодом в будущем, не вздумал вернуть прежнюю ошибку. Однако в таких исключительных случаях вы фактически *документируете нетривиальное место*, а не размещаете сообщение об ошибке.

Комментарии должны касаться настоящего, а не прошлого. Не описывайте того, что претерпело изменения, и не рассказывайте о том, как было раньше.

Устаревание комментариев

Комментарии разрушаются. Любой плохо сопровождаемый код подвержен разрушению; в нем появляются неприглядные заплаты и теряется

Случай из жизни

Однажды я работал над фрагментом кода, где был комментарий, извещавший о том, что функции A и B *пока не реализованы*. Обе они были мне нужны, поэтому я написал их реализацию. После этого я узнал, что функция B уже реализована, а функция A была не нужна, поскольку реализация B решала и ее задачу. Если бы написавший ее программист убрал свой ошибочный комментарий, я не потратил бы столько времени на *лишнюю* работу.

первоначальная стройность замысла. Однако комментарии деградируют, по-видимому, значительно быстрее, чем остальной код. Они устаревают вместе с кодом, который объясняют. Это может сильно досадить.

Простое решение может быть таким: если вы исправляете, добавляете или модифицируете любой код, исправьте, добавьте и модифицируйте все комментарии, которые к нему относятся. Не ограничивайтесь изменением пары строк. Следите за тем, чтобы любые изменения в коде не приводили к появлению ложных комментариев. Следствие: комментарии нужно писать так, чтобы их было легко обновлять, иначе их не будут обновлять. Комментарии должны быть четко связаны с разделом кода, который они поясняют, а не располагаться в произвольном месте.

Если вы изменили код, проверьте правильность комментариев, находящихся рядом с ним.

Еще одна скверная привычка – сохранение закомментированных блоков кода. Они станут источником путаницы, когда вы обратитесь к своему коду год спустя или когда им займется другой программист. Встретив код в блоке комментариев, вы станете размышлять, как он там оказался. Было ли это неосуществленным исправлением? Или добавлением, не доведенным до конца? Работает ли этот код? Все ли функции реализованы в остальном коде?

Либо поясните причину, по которой вы закомментировали код, либо удалите его вообще – если понадобится, вы всегда найдете его в системе управления версиями. Даже если вы считаете, что отключаете фрагмент кода временно, сделайте себе пометку, иначе вы можете забыть довести дело до конца.

Сопровождение и бессодержательные комментарии

Работая над старым кодом, лучше всего сохранить в нем любые дурацкие комментарии, если только они не несут явного вреда. Пусть они останутся как предупреждение тем, кто будет сопровождать этот код когда-нибудь в будущем – полезное свидетельство качества соответст-

вующего кода (точнее, его отсутствия). Конечно, если вы действительно хотите повысить качество этого кода, то и комментарии нужно подвергнуть переработке. Обнаружив комментарий, содержащий неверные сведения или ошибки, вы должны переписать его в процессе сопровождения данного кода.

Увидев предупреждающие флажки типа `XXX`, отнеситесь к ним со всем вниманием и осторожностью. Следите также за операторами вывода, которые закомментированы. Они явно указывают на то, что в этом месте кода были проблемы; изучите этот код особенно внимательно!

Помните о постепенной деградации комментариев. Если в комментарии сказано, что *нечто определено в* `foo.c`, нет никакой уверенности, что так оно и есть до сего дня. Всегда верьте коду и сомневайтесь в комментариях.

Резюме

Главное – описать увиденное так, чтобы больше не повторяться.

Делмор Шварц

Мы пишем много комментариев. Это вызвано тем, что мы пишем много кода. Важно научиться писать правильные комментарии, иначе наш код может потонуть под грузом бестолковых или устаревших комментариев.

Не нужно переоценивать значение комментариев; благодаря хорошим комментариям плохой код лучше не станет. Целью должно быть написание самодокументируемого кода, для которого не требуются никакие комментарии.

Хорошие программисты...

- Стремятся писать *малочисленные*, но добротные комментарии
- В комментариях объясняют *почему*
- Стараются писать хороший код, а не уйму комментариев
- Пишут полезные и разумные комментарии

Плохие программисты...

- Не понимают разницы между плохими и хорошими комментариями
- В комментариях объясняют *как*
- Пишут комментарии, не понятные никому, кроме них самих
- Пытаются подкрепить плохой код многочисленностью комментариев
- Помещают в исходные тексты лишнюю информацию (типа истории версий и т. д.)

См. также

Глава 2. Тонкий расчет

Система структурирования и расположения кода влияет на расположение комментариев.

Глава 3. Что в имени тебе моем?

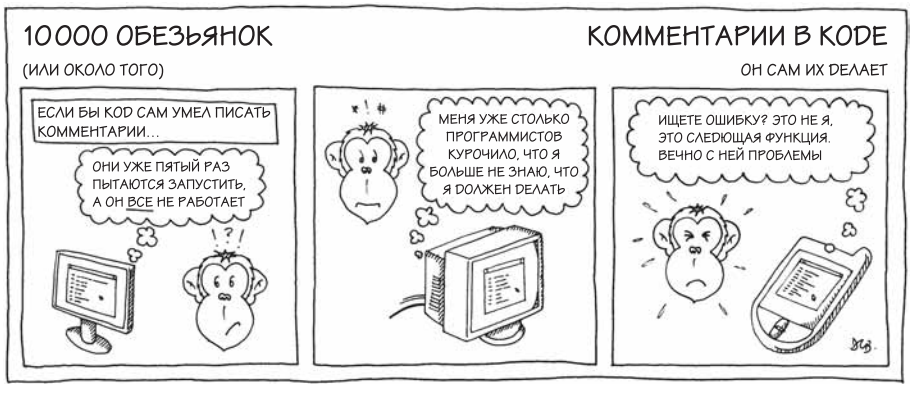
Один из аспектов самодокументируемого кода: правильный выбор имен.

Глава 4. Литературоведение

Описывает *самодокументирование кода* – прием, делающий пространное комментирование ненужным. Кроме того, описывает технологию *грамотного программирования*.

Глава 18. Защита исходного кода

Системы управления версиями хранят историю файлов, поэтому не стоит излагать ее в комментариях.



Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 605.

Вопросы для размышления

1. Как могут различаться *необходимость в* комментариях и их *содержание* в следующих типах кода:
   a. Язык ассемблера низкого уровня (машинный код)
   b. Сценарии командного интерпретатора
   c. Однофайловая среда тестирования
   d. Крупный проект C/C++

2. Есть инструменты для подсчета процента строк комментариев в исходном тексте. Насколько они полезны? Насколько точно могут они оценить качество комментариев?
3. Если вам встретился непонятный код, как лучше внести в него некоторую ясность – добавив комментарии с вашим пониманием его работы или переименовав переменные/функции/типы, дав им более содержательные имена? Какой подход может оказаться проще? Какой подход будет безопаснее?
4. Если вы написали блок комментариев к C/C++ API, куда его лучше поместить – в заголовочный файл, где объявляется функция, или в файл, где находится ее реализация? В чем преимущества и недостатки каждого из расположений?

Вопросы личного характера

1. Рассмотрите внимательно файлы исходного кода, над которыми работали в последнее время. Прочтите свои комментарии. Так ли они хороши, если быть честным? (Ручаюсь, что при чтении кода вам захочется сделать несколько изменений!)
2. Как вы добиваетесь того, чтобы ваши комментарии были действительно полезными, а не представляли собой личные заметки, никому, кроме вас, не понятные?
3. Придерживаются ли ваши коллеги одинакового стандарта написания комментариев, возможно, с небольшими различиями?
   a. Чьи комментарии лучше других? Почему? Чьи самые плохие? Есть ли корреляция между качеством их комментариев и качеством кода в целом?
   b. Не думаете ли вы, что если потребовать от членов вашей команды придерживаться определенного стандарта кодирования, то качество комментариев может вырасти?
4. Включаете ли вы в исходные файлы историю их модификации? Если да:
   a. Поддерживаете ли вы ее вручную? Зачем, если система управления версиями может автоматически делать это вместо вас? Насколько точно ведется история?
   b. Разумна ли такая практика *в действительности*? Как часто возникает потребность в таких данных? Каковы преимущества хранения их в исходном файле по сравнению с иным механизмом?
5. Добавляете ли вы свои инициалы или помечаете каким-то иным способом свои комментарии, которые делаете в чужом коде? Указываете ли вы дату комментирования? Когда и зачем вы это делаете – полезна ли такая практика? Сослужили ли вам когда-либо пользу чьи-то инициалы или отметка о дате?

6

Людям свойственно ошибаться

Борьба с неизбежным – ошибками в коде

В этой главе:

- Какие бывают ошибки
- Правильная обработка ошибок
- Как генерировать ошибки
- Программирование в условиях неопределенности

Мы знаем, что единственный путь избежать ошибки – обнаружить ее. И единственный путь обнаружить ее – иметь возможность свободно задавать вопросы.

Дж. Роберт Оппенгеймер

В какой-то момент жизни у каждого наступает прозрение: *мир устроен иначе, чем вам казалось*. Мой приятель Том выяснил это в годовалом возрасте, пытаясь залезть на кресло, которое было в четыре раза выше его самого. Он рассчитывал забраться на вершину. Результат оказался неожиданным: на него рухнула мебель.

Что-то нарушилось в мире? Или он неправильно устроен? Нет. Мир движется вперед уже миллионы лет и, похоже, не собирается останавливаться. Это *наши представления* о нем не верны и нуждаются в пересмотре. Как говорится, *неприятности неизбежны, поэтому нужно учиться справляться с ними*.

Мы обязаны писать код так, чтобы он работал в реальном мире со всеми его неожиданностями.

Это особенно тяжело, потому что *обычно* мир ведет себя предсказуемым образом и тем самым внушает нам ложное чувство безопасности. Человеческий мозг настроен на то, чтобы справляться с проблемами, и имеет встроенную систему безопасности. Если кто-то заложит вход в ваш дом кирпичом, ваш мозг обнаружит проблему, и вы остановитесь, а не ударитесь лбом об стену. Но программы не столь умны, и приходится сообщать им, где находятся кирпичные стены и что делать, сталкиваясь с ними.

Не стоит рассчитывать, что ваша программа всегда будет работать без сучка и задоринки. Мир не всегда действует согласно вашим предположениям: вы *должны* предусмотреть обработку всех аварийных ситуаций в своей программе. На словах это просто, но на практике требует массы усилий.

Откуда что берется

Готовиться к неожиданностям –
признак вполне современного интеллекта.

Оскар Уайлд

Ошибки были и будут. Неприятные результаты могут быть следствием почти любой операции. Это не то же самое, что дефект программы, поскольку вы заранее знаете, что может произойти ошибка. Например, вам нужно открыть файл базы данных, а он оказывается удален, или диск переполнен, и операция записи становится невозможной, или недоступен требуемый веб-сервис.

Если не написать код для обработки таких аварийных ситуаций, программа почти наверняка окажется *дефектной*; она не всегда будет работать так, как вы предполагали. Однако если аварийная ситуация возникает редко, дефект может оказаться трудно обнаружимым! О дефектах (багах) мы поговорим в главе 9.

Есть тысячи причин возникновения аварийных ситуаций, но все они попадают в одну из трех категорий:

Ошибки пользователя

Глупый юзер варварски обращается с вашей любимой программой. Возможно, он ввел неверные данные или попытался выполнить совершенно бессмысленную операцию. Хорошая программа укажет на ошибку и поможет пользователю исправить ее. Она не станет издеваться над ним или жаловаться непонятно на что.

Ошибка программиста

Пользователь действовал правильно, но код некорректен. Где-то есть дефект, недосмотр программиста, с которым пользователь ни-

чего не может сделать (кроме как постараться избегать в будущем). В идеале таких ошибок быть не должно.

Здесь возникает замкнутый круг: необрабатываемые аварийные ситуации приводят к ошибкам в программе. А эти ошибки могут приводить к возникновению аварийных ситуаций в других местах кода. Поэтому мы считаем защитное программирование важной технологией.

Исключительные обстоятельства

Пользователь действовал правильно, и программист ничего не напутал. Вмешалась неверная судьба, и мы столкнулись с чем-то, чего нельзя было избежать. Разорвалось сетевое соединение, кончились чернила в принтере или не осталось места на диске.

Необходима четкая стратегия действий для каждого типа ошибок в коде. Можно обнаружить ошибку и сообщить о ней пользователю во всплывающем окне, либо ошибку определит код промежуточного уровня и сообщит о ней коду клиента программным образом. В обоих случаях действуют одинаковые принципы: способ решения проблемы выбирает либо человек, либо ваш код – кто-то должен отвечать за обнаружение ошибки и принятие мер.

Обработка ошибок – дело серьезное. От нее зависит стабильность вашего кода.

Ошибки генерируются подчиненными компонентами и передаются наверх, чтобы их обработал обратившийся к компоненте код. Есть ряд способов извещения об ошибке, которые мы рассмотрим в следующем разделе. Чтобы управлять выполнением программы, мы должны уметь:

- Генерировать ошибку, если возникают какие-то неприятности
- Обнаруживать все сообщения об ошибках
- Обрабатывать их надлежащим образом
- Передавать ошибку дальше, если не можем обработать ее сами

Работать с ошибками тяжело. Возникшая ошибка часто не связана с тем, чем вы занимались в данный момент (большинство из них относятся к категории «непредвиденных обстоятельств»). Кроме того, обрабатывать их скучно – хотелось бы сосредоточиться на том, что *должна* делать программа, а не на неприятностях, с которыми она может столкнуться. Однако без надлежащей обработки ошибок программа оказывается хрупкой – построенной на песке, а не на каменном основании. При первых же порывах ветра она рухнет.

Механизмы сообщения об ошибках

Есть несколько стандартных стратегий передачи информации об ошибке в клиентский код. Вы можете встретить код, в котором применяется

любая из них, поэтому каждый из имеющихся диалектов должен быть вам понятен. Обратите внимание на особенности этих методов сообщения об ошибках и на преимущества тех или иных из них в отдельных ситуациях.

Для каждого механизма характерна своя мера *близости ошибки*. Ошибка близка (локальна) во *времени*, если обнаруживается вскоре после возникновения. Ошибка близка (локальна) в *пространстве*, если она обнаруживается вблизи или прямо на *месте* своего фактического проявления. Некоторые методы специально направлены на приближение ошибки, чтобы облегчить анализ происходящего (например, генерация кода ошибки). Другие методы стремятся увеличить меру локальности ошибки, чтобы избавить обычный код от логики обработки сбоя (например, исключительные ситуации).

Выбор того или иного механизма часто осуществляется в рамках архитектурного решения. Архитектор может решить, что необходимо создать однородную иерархию классов исключений или главный список общих кодов ошибок, чтобы унифицировать код, обрабатывающий ошибки.

Без обработки ошибок

Проще всего *не обращать внимания* на ошибки. Это лучшее решение, если вы хотите, чтобы ваша программа вела себя странным и непредсказуемым образом и время от времени аварийно завершалась.

Если вы встретились с ошибкой и не знаете, что с ней делать, то просто игнорировать ее – *неудачное* решение. Продолжить работу функции, вероятно, нельзя, но вернуться из нее, не выполнив контракт функции, значит, создать неопределенную и противоречивую ситуацию.

Не оставляйте сбойную ситуацию без внимания. Если вы не знаете, как справиться с проблемой, сигнализируйте об отказе вызвавшему коду. Не заметайте мусор под ковер в надежде, что все как-нибудь обойдется.

Альтернативой игнорированию ошибок будет немедленное прерывание работы программы при возникновении проблемы. Это проще, чем обрабатывать ошибки на всем протяжении кода, но вряд ли такое решение можно считать технически грамотным!

Возвращаемые значения

Следующий по простоте механизм – возврат функцией значений, соответствующих ее успешному/неудачному завершению. Возвращаемое булево значение дает простой ответ «да»/»нет». Более сложный подход состоит в нумерации всех возможных состояний и возврате соответствующего *кода*. Одно из значений означает *успех*; остальные представляют различные случаи аварийного завершения. Коды завершения могут быть общими для всего кода проекта, и тогда ваша функция

будет возвращать какое-то подмножество общего списка значений. При этом в документации должно быть указано, какие коды можно получить при вызове вашей функции.

Такой подход хорош, если функции не возвращают данных; получение же кодов ошибок *наряду* с данными вносит некоторую путаницу. Если функция `int count()` обходит связанный список и возвращает количество элементов в нем, каким образом она может сообщить о повреждении структуры списка? Есть три способа:

- Возвращать составной тип данных (или *кортеж*, *tuple*), содержащий как возвращаемое значение, так и код ошибки. В популярных языках типа C это выглядит неуклюже и встречается редко.
- Передать код ошибки через параметр функции. В C++ и .NET этот параметр передается по ссылке. В C доступ к переменной следует организовать с помощью указателей. Данный способ некрасив и неинтуитивен; нет синтаксических средств, чтобы отличить возвращаемое значение от параметра.
- Можно также зарезервировать некоторый диапазон возвращаемых значений для кодов ошибок. Например, в функции подсчета все отрицательные числа могут трактоваться как сигнал об ошибке; в качестве результата они все равно не имеют смысла. Отрицательные числа часто используются таким образом. Возвращаемым значениям типа указателя может быть присвоено специальное недопустимое значение, которым обычно является ноль (или `NULL`). В Java и C# можно возвращать нулевую ссылку на объект.

  Этот прием не всегда работает удовлетворительно. Иногда трудно найти диапазон для значений, представляющих ошибки – все возвращаемые величины одинаково имеют смысл и одинаково вероятны. Кроме того, побочным эффектом является сокращение диапазона допустимых успешных значений; использование отрицательных величин уменьшает количество возможных положительных значений на порядок.[1]

Переменные, содержащие состояние ошибки

В этом методе делается попытка решить конфликт между возвратом функцией значения и сообщением кода ошибки. Вместо того чтобы возвращать код состояния, функция устанавливает значение общей глобальной переменной кода ошибки. После вызова функции нужно посмотреть значение этой переменной и определить, был ли вызов успешным.

Использование общей переменной сокращает путаницу и сигнатуру функции, а на диапазон возвращаемых значений вообще не оказывает никакого влияния. Однако когда для сообщения об ошибках выделяет-

[1] Если пользоваться типом `unsigned int`, то количество доступных значений увеличивается вдвое за счет знакового разряда по сравнению с `signed int`.

ся отдельный канал, легко пропустить или сознательно игнорировать ошибку. Наличие общей глобальной переменной также создает большие трудности для параллельной работы потоков.

В стандартной библиотеке C в этой технологии участвует переменная `errno`. Ее семантика очень деликатна: перед вызовом любых средств стандартной библиотеки нужно вручную обнулить `errno`. В случае успеха ее значение не меняется; она устанавливается только в случае неудачи. Это часто является источником ошибок, а вызов всех библиотечных функций оказывается обременительным. К тому же не все функции стандартной библиотеки C пользуются `errno`, поэтому в работе с ней нет единообразия.

Функционально этот прием равносилен использованию возвращаемых значений, но если учесть его недостатки, то лучше им не пользоваться. Не пишите собственные сообщения об ошибках в таком стиле и с крайней осторожностью пользуйтесь уже имеющимися реализациями.

Исключения

Исключения (исключительные ситуации) – это средство языка для обработки ошибок; их поддерживают не все языки. Исключения помогают отличить нормальный ход выполнения от *особых* случаев, когда в функции возник сбой и она не может выполнить свой контракт. Когда код сталкивается с проблемой, с которой не может справиться, он стопорит свою работу и вырабатывает *исключительную ситуацию* – объект, представляющий ошибку. Исполнительная система языка автоматически движется обратно по стеку вызовов, пока не найдет код для обработки исключительных ситуаций. Тогда программа привлекает его, чтобы справиться с ошибкой.

Есть две модели функционирования, отличающиеся ходом событий после обработки исключительной ситуации:

Модель с прекращением исполнения

Выполнение продолжается далее после обработчика, перехватившего исключительную ситуацию. Такая модель существует в C++, .NET и Java.

Модель с возобновлением

Выполнение возобновляется в том месте, где была сгенерирована исключительная ситуация.

Первую модель легче понять, но она предоставляет ограниченный контроль. Она позволяет только *обработать ошибку* (выполнить код, обнаружив сбой), но не *преодолеть ситуацию* (что-то исправить и попытаться повторить действие).

Исключительную ситуацию нельзя игнорировать. Если она не перехвачена и не обработана, то будет передаваться по стеку вплоть до самого верха и, как правило, вызовет останов программы. Исполнительная система языка автоматически приводит все в порядок, разворачи-

вая стек вызовов. Благодаря этому исключительные ситуации оказываются более аккуратной и безопасной альтернативой самодельному коду обработки ошибок. Однако небрежный код генерации исключительных ситуаций может вызывать утечки памяти и проблемы с очисткой ресурсов.[1] Нужно следить за тем, чтобы код *корректно работал с исключительными ситуациями*. Подробнее о том, что это означает, рассказывается во врезке.

Код, который обрабатывает исключительную ситуацию, и тот, который генерирует ее, различны и могут находиться совсем в разных местах. Механизм исключительных ситуаций обычно предоставляется ОО-языками, и ошибки в них определяются как иерархия классов исключительных ситуаций. Обработчик может быть предназначен для перехвата вполне конкретного класса ошибок (принимая производный класс) или более общей категории ошибок (принимая базовый класс). Исключительные состояния представляют особую ценность для сигнализации об ошибках в конструкторах.

Поддержка исключительных ситуаций не проходит даром – она отражается на производительности программ. На практике потери несущественны и проявляются только в операторах, занимающихся обработкой исключений, – обработчики исключительных ситуаций мешают компилятору выполнить оптимизацию. Из этого не следует, что механизм исключительных ситуаций порочен; его издержки вполне оправданы по сравнению с последствиями отсутствия обработки ошибок!

Агитпоход за безопасность исключений

Прочный код должен хорошо *«держать» исключительные ситуации*. Он должен работать *корректно* (какой смысл в это вкладывается, обсуждается ниже), независимо от того, какие возникают исключительные ситуации. Это не должно зависеть от того, перехватывает ли сам код какие-либо исключительные ситуации.

Нейтральный в отношении исключительных ситуаций код пересылает все исключительные ситуации наверх, тому, кто его вызвал; сам он ничего не трогает и не меняет. Это важная концепция для таких общих программ, как код шаблонов C++; типы шаблонов могут генерировать самые различные исключительные ситуации, не известные тем, кто разрабатывает шаблоны.

[1] Например, можно выделить блок памяти, а затем неожиданно выйти из функции из-за возникшей исключительной ситуации. Выделенная память оказывается потерянной. Такие проблемы осложняют написание кода, который должен работать в условиях возможности генерации исключительных ситуаций.

Есть несколько уровней защиты от исключительных ситуаций. Они характеризуются теми гарантиями, которые даются вызывающему коду. Вот эти гарантии:

Базовые гарантии

Если в функции возникают исключительные ситуации (в результате выполняемых действий или вызова других функций), они не приводят к утечке ресурсов. Код сохраняет работоспособность (т. е. им можно корректно пользоваться), но состояние его может оказаться неопределенным. Пример: функция-член должна добавить в контейнер 10 элементов, но ее работа прервана исключительной ситуацией. Контейнер сохраняется в рабочем состоянии; при этом может оказаться, что в него не было добавлено ни одного объекта, либо добавлены все 10, либо только половина всех объектов.

Сильные гарантии

Они значительно строже, чем базовые. Если во время выполнения кода возникает исключительная ситуация, состояние программы абсолютно не меняется. Не изменится ничего – ни один объект, ни одна глобальная переменная. В предыдущем примере в контейнер не будет ничего помещено.

Гарантии невозбуждения исключительных ситуаций

Последний вид гарантий наиболее строг: операция не может создать исключительную ситуацию. Для нейтрального в отношении исключительных ситуаций кода это означает, что функция не должна делать ничего такого, что может генерировать исключительные ситуации.

Вид предоставляемых гарантий зависит исключительно от вас. Чем строже гарантии, тем шире возможности использования кода. Для реализации сильных гарантий обычно требуется иметь несколько функций, обеспечивающих гарантию отсутствия исключительных ситуаций.

Прежде всего, каждый деструктор должен обеспечивать гарантию невозбуждения исключений.[1] В противном случае невозможно обеспечить обработку исключений. При наличии исключительной ситуации деструкторы объектов автоматически вызываются во время развертывания стека. Генерация исключения во время обработки исключения недопустима.

---

[1] Так, по крайней мере, происходит в C++ и Java. В C# *деструктором* назван ~X(), что глупо, поскольку это лишь замаскированный финализатор. Генерация исключения в деструкторе C# приводит к другим последствиям.

Сигналы

Сигналы представляют собой более экстремальный механизм сообщения об ошибках, который широко используется средой исполнения для извещения работающей программы об ошибках. Операционная система перехватывает ряд аварийных ситуаций, таких как *исключения при операциях с действительными числами*, генерируемые математическим сопроцессором. Эти хорошо известные события вызывают отправку приложению сигналов, прерывающих его нормальную работу и влекущих переход в функцию, назначенную *обработчиком данного сигнала*. Программа может получить сигнал в любой момент, и код должен уметь его обработать. По окончании выполнения функции обработки сигнала работа программы продолжается с того места, где она была прервана.

Сигналы – это программный эквивалент аппаратных прерываний. Это концепция, пришедшая из UNIX, которая теперь реализована на большинстве платформ (базовая версия входит в стандарт ISO C [ISO99]). Операционная система предоставляет для всех сигналов разумные обработчики по умолчанию, одни из которых ничего не делают, а другие прекращают выполнение программы, выводя краткое сообщение об ошибке. Вы можете заменить эти обработчики собственными.

В число событий, о которых извещают сигналы C, входят завершение программы, запросы на приостановку/возобновление выполнения программы и математические ошибки. В некоторых средах базовый список событий значительно расширен.

Обнаружение ошибок

Как вы будете обнаруживать ошибку, зависит, очевидно, от механизма, который сообщит о ней. На практике это значит:

Возвращаемые значения

Успешно ли выполнилась функция, определяется по возвращаемому ею значению. Эта проверка на отказ тесно связана с самим вызовом функции; подразумевается, что, выполняя его, вы проверяете, был ли он успешным. Учитывать полученный результат или нет – дело ваше.

Переменные состояния ошибки

После вызова функции нужно рассмотреть значение переменной состояния ошибки. Для переменной `errno` в модели C нет необходимости проверять наличие ошибки после каждого обращения к функции. Сбросьте значение `errno`, а затем последовательно вызовите любое число стандартных библиотечных функций. После этого посмотрите на значение `errno`. Если оно соответствует коду ошибки, значит, одна из вызывавшихся функций дала сбой. Вы не узнаете, какая именно, но иногда можно воспользоваться и таким упрощенным способом обнаружения ошибок.

Исключительные ситуации

Если одна из подчиненных функций сгенерировала исключительную ситуацию, вы можете либо перехватить ее, либо игнорировать и позволить ей подняться на более высокий уровень. Обоснованное решение можно принять лишь тогда, когда известно, какие исключительные ситуации могут генерироваться. Такое знание может дать вам документация (если она заслуживает доверия).

В Java исключительные ситуации реализованы так, что эта документация находится в самом коде. Программист обязан написать для каждого метода *спецификацию* исключений, указав, какие исключительные ситуации могут генерироваться. Java – единственный из основных языков программирования, который предъявляет такие требования. Исключение, отсутствующее в списке, не сможет проскочить, потому что компилятор осуществляет статическую проверку с целью не допустить этого.[1]

Сигналы

Есть лишь один способ обнаружить сигнал: установить для него обработчик. Делать это необязательно. Можно не устанавливать никаких обработчиков сигналов и сохранить поведение по умолчанию.

По мере того как отдельные фрагменты кода объединяются в большую систему, может возникнуть необходимость в нескольких способах обнаружения ошибок, даже в рамках одной функции. Каким бы механизмом обнаружения вы ни воспользовались, важно помнить следующее:

Никогда не пренебрегайте поступающими вам сообщениями об ошибках. Если существует канал для сообщений об ошибках, значит, для этого есть причины.

Полезная привычка: всегда вставлять код для обнаружения ошибок, даже если ошибки никак не влияют на дальнейшую работу вашей программы. Тем самым вы дадите понять программисту, который будет сопровождать ваш код, что вам известно о возможности сбоев в функции, но вы сознательно решили их игнорировать.

Когда вы позволяете исключительной ситуации передаваться через ваш код, это не игнорирование – исключительную ситуацию игнорировать *нельзя*. Вы просто разрешаете ее обработку на более высоком уровне. Философия обработки исключительных ситуаций в этом отношении совсем иная. Не вполне понятно, как правильнее всего в этом случае составить документацию: написать блок `try/catch`, который просто заново сгенерирует исключение, написать в комментарии, что код

[1] C++ тоже поддерживает спецификации исключительных ситуаций, но не требует их обязательного применения. Принято уклоняться от них, в частности ради увеличения скорости. В отличие от Java, они вступают в силу на этапе исполнения.

безопасен в отношении исключений, или не делать ничего? Я предпочитаю документировать режим обработки исключительных ситуаций.

Обработка ошибок

Люби истину, но будь снисходителен к заблуждениям.

Вольтер

Ошибки неизбежны. Мы видели, как их искать и когда это делать. Вопрос следующий: как с ними поступать? Ответить на него нелегко. В значительной мере ответ зависит от обстоятельств и тяжести ошибки – можно ли исправить ситуацию и попробовать повторить операцию либо нужно двигаться дальше, несмотря ни на что. Часто такая возможность отсутствует: ошибка может свидетельствовать о начале агонии. Иногда лучше всего быстро навести порядок и завершить работу, пока не стало еще хуже.

Такого рода решения принимаются при наличии достаточной информации. Есть ряд ключевых сведений, которые нужно знать об ошибке:

Где она возникла

Это совсем не то место, где она станет обрабатываться. Где находится источник – в базовой системной компоненте или периферийном модуле? Эти данные могут присутствовать в сообщении об ошибке либо быть получены вручную.

Что вы пытались сделать?

Что спровоцировало ошибку? В этом может быть ключ ко всем восстановительным действиям. В отчете об ошибке такие сведения встречаются редко, но по контексту можно выяснить, какая функция была вызвана.

Почему возник сбой

В чем суть проблемы? Нужно точно узнать, что случилось, а не просто установить *класс* ошибки. Какая часть приведшей к ошибке операции успела выполниться? Прекрасно, если выполнилось *все* или *ничего*, но обычно программа оказывается в некоем промежуточном состоянии.

Когда это случилось

Это локализация ошибки по времени. Произошел ли отказ только что или это проблема двухчасовой давности, известившая о себе только сейчас?

Степень тяжести ошибки

Одни проблемы опаснее других, хотя при обнаружении они все равнозначны – не разобравшись с проблемой и не найдя ее решения, нельзя двигаться дальше. Степень тяжести обычно определяет вы-

звавший уровень исходя из того, насколько легко возобновить работу или найти обходной путь.

Как исправить ошибку

Решение может быть очевидным (например, вставить дискету и повторить операцию) или не совсем (например, потребуется модифицировать параметры функции, сделав их совместимыми). Чаще всего вывод делается на основании других источников информации.

От их полноты зависит стратегия обработки каждой ошибки. Пропущенный обработчик возможного сбоя приводит к появлению программного дефекта, который редко проявляется и с трудом локализуется, так что тщательно обдумывайте все сбойные ситуации.

Когда обрабатывать ошибки

Когда следует обрабатывать каждую ошибку? Не всегда это следует делать сразу при ее обнаружении. Есть две точки зрения.

Как можно скорее

Обрабатывайте каждую ошибку, *как только* обнаружите ее. Пока причина ошибки не устарела, сохраняется важная контекстная информация, и код обработки оказывается яснее. Это знакомая нам технология самодокументирования кода. Обработка ошибок вблизи их источника означает, что меньше кода будет выполнено при некорректном состоянии программы.

Обычно это лучший подход в том случае, когда функции возвращают код ошибки.

Как можно позднее

Напротив, можно попытаться всеми силами оттягивать обработку ошибок. При этом учитывается, что код, обнаруживший ошибку, часто не знает, как с ней поступить. Многое зависит от контекста. Когда не найден некий файл, то можно сообщить об этом пользователю, если это файл с загружаемым документом. Если же это файл индивидуальных настроек программы, то можно тихо проскочить дальше.

В таком случае лучше всего подходят исключительные ситуации; можно передавать исключение с одного уровня на другой, пока не определится, как поступить с ошибкой. Такое разделение обнаружения и обработки может быть более понятным, но приведет к усложнению кода. Вовсе не очевидно, что обработка ошибки умышленно откладывается на другое время, и, когда вы наконец займетесь ее обработкой, будет непонятно, где возникла ошибка.

Теоретически рекомендуется отделять «бизнес-логику» приложения от обработки ошибок. Но часто это невозможно, поскольку наведение порядка неотделимо связано с этой бизнес-логикой, и разделение их кодирования может вызвать дополнительные трудно-

сти. Тем не менее централизованная обработка ошибок имеет свои преимущества: вы знаете, где находится этот код, и можете реализовать всю политику аварийного завершения/продолжения работы в одном месте, не рассеивая ее по множеству функций.

Томас Джефферсон однажды заявил: «Лучше отложить, чем ошибиться». Его слова не лишены смысла; *наличие процедуры* обработки ошибок гораздо важнее, чем обработка *отдельной* ошибки. Тем не менее желательно стремиться к компромиссу, позволяющему уйти не настолько далеко, чтобы потерять смысл и контекст обрабатываемых ошибок, и не остаться столь близко, что обычный код оказывается загроможденным обходными маршрутами и тупиковыми путями обработки ошибок.

Обрабатывайте все ошибки в наиболее благоприятном контексте, когда становится ясно, как корректно с ней справиться.

Варианты реагирования

Допустим, вы перехватили ошибку. Вы готовы ее обработать. Что вы станете делать? Вероятно, то, что необходимо для корректной работы программы. Невозможно перечислить все существующие стратегии восстановления, но приведем наиболее распространенные типы ответных действий:

Регистрация в журнале

Во всяком достаточно большом проекте должны быть средства ведения журналов. В них заносится важная информация по ходу работы, и они служат начальной точкой расследования при возникновении проблем.

Журнал нужен, чтобы регистрировать интересные события в процессе разработки программы, иметь возможность глубже проникнуть во внутренние механизмы ее функционирования и восстановить ход ее выполнения. Поэтому все возникающие ошибки должны быть подробно описаны в журнале программы; это одни из самых интересных и информативных событий. Старайтесь регистрировать все относящиеся к делу данные, основываясь на приведенном выше списке.

Для действительно темных ошибок, предвещающих катастрофические последствия, бывает разумно заставить программу «позвонить домой» – переслать разработчикам снимок самой себя или копию журнала регистрации ошибок, чтобы они могли предпринять действия.

Как вы будете *потом* работать с журналом – уже другая задача.

Создание отчета

Программа должна сообщать пользователю об ошибке, только когда не остается ничего иного. Не нужно бесконечно бомбардировать поль-

зователя бесполезной для него информацией или раздражать бесчисленными глупыми вопросами. Вступайте в диалог с пользователем, только когда это действительно необходимо. Если ситуация допускает восстановление, не нужно посылать отчет. Обязательно зарегистрируйте событие, но не поднимайте из-за него шума. Обеспечьте пользователям возможность чтения журнала событий, если вам кажется, что у них может возникнуть такая потребность в будущем.

Бывают проблемы, решить которые может только пользователь. В таких случаях рекомендуется немедленно сообщить о проблеме, чтобы предоставить пользователю все возможности исправить положение или решить, как действовать дальше.

Конечно, такого рода сообщения возможны, если программа располагает средствами интерактивности. Программам, действующим независимо от пользователя, приходится справляться с трудностями самостоятельно; представьте себе стиральную машину, выводящую диалоговые окна!

Восстановление

Иногда единственным выходом оказывается немедленное прекращение работы. Но не всегда при возникновении ошибки программа обречена. Допустим, ваша программа сохраняет файл, а диск оказывается переполненным, и операция не может быть выполнена. Пользователь вправе рассчитывать, что программа сможет продолжить работу, и вы должны это обеспечить.

Если код сталкивается с ошибкой и не знает, что с ней делать, передайте ее выше. Скорее всего, у вызвавшего уровня найдутся средства для восстановления.

Игнорирование

Этот вариант я включил лишь для полноты. Надо полагать, что к этому моменту вы уже с негодованием отнесетесь ко всякому предложению игнорировать ошибки. Если же вы решите не забивать себе голову обработкой ошибок и, перекрестившись, двинетесь вперед – *удачи вам*. Это причина большинства дефектов во всех программных пакетах. Игнорирование сбоя, который может сорвать нормальную работу системы, неизбежно приводит к долгим часам отладки.

Можно, однако, написать код, который позволяет *ничего не делать* при появлении ошибки. Противоречие? Нет. Можно написать код, который справляется с ситуациями нашего непоследовательного мира, который корректно продолжает работу при наличии ошибки – но при этом код часто оказывается весьма сложным. Приняв такой подход, вы должны ясно сообщить об этом в коде, иначе его могут принять за неграмотно и некорректно написанный.

Игнорирование ошибок не экономит вашего времени. Вы потратите больше времени на выяснение причин некорректного поведения программы, чем вам понадобилось бы для написания обработчика ошибок.

Пересылка ошибки

Если возникает отказ в вызываемой функции, то вы, возможно, не сможете продолжить работу, но и не будете знать, как поступить. Единственный выход – сделать за собой уборку и передать сообщение об ошибке на более высокий уровень. Возможны варианты. Есть два способа пересылки ошибки:

- Экспортировать ту информацию об ошибке, которую сами получили (возвратить тот же код ошибки или передать дальше исключение).
- Преобразовать информацию, послав наверх более содержательное сообщение (возвратить другой код ошибки или перехватить исключение и сгенерировать свое собственное).

Попробуйте определить, связана ли ошибка с концепцией интерфейса модуля. Если да, можно спокойно передать ту же ошибку дальше. Если нет, можно преобразовать ее в соответствии с обстановкой, создав сообщение об ошибке, оправданное в контексте интерфейса вашего модуля. Это полезное применение техники самодокументирования кода.

Последствия для кода

Покажите мне код! Посвятим некоторое время изучению последствий обработки ошибок для нашего кода. Как будет продемонстрировано, написать хороший код обработки ошибок так, чтобы он не искалечил логику основной программы, непросто.

В первом примере мы рассмотрим стандартную структуру обработки ошибок. Это не самый изобретательный подход к написанию кода, устойчивого к ошибкам. Задача состоит в том, чтобы последовательно вызвать три функции, в каждой из которых может возникнуть сбой, и попутно выполнить некоторые промежуточные вычисления. Найдите, какие проблемы могут возникнуть в таком коде:

```
void nastyErrorHandling()
{
    if (operationOne())
    {
        ... какие-то действия ...
        if (operationTwo())
        {
            ... какие-то другие действия ...
            if (operationThree())
            {
                ... еще действия ...
            }
        }
    }
}
```

Синтаксически все прекрасно; код должен работать. На практике это не самый удобный стиль для сопровождения. Чем больше операций нужно выполнить, тем глубже становится вложенность кода и тем труднее его читать. Такого рода обработка ошибок быстро приводит к запутанному нагромождению условных операторов. Работа кода отражена не очень удачно; все промежуточные вычисления имеют, видимо, одинаковую степень важности, но тем не менее расположены на разных уровнях вложенности.

Можно ли избежать этих проблем? Да, есть несколько возможностей. В первом варианте мы избавимся от вложенности и сделаем структуру более плоской. Семантически код останется эквивалентным, но *дополнительно* увеличится сложность, что связано с появлением новой переменной состояния `ok`, управляющей потоком выполнения:

```
void flattenedErrorHandling()
{
    bool ok = operationOne();
    if (ok)
    {
        ... какие-то действия ...
        ok = operationTwo();
    }
    if (ok)
    {
        ... какие-то другие действия ...
        ok = operationThree();
    }
    if (ok)
    {
        ... еще действия ...
    }
  if (!ok)
  {
      ... уборка при возникновении ошибок ...
  }
}
```

Мы также ввели возможность сделать приборку, если возникнут какие-то ошибки. Достаточно ли этого для устранения следов всех сбоев? Возможно, нет. Действия, осуществляемые при уборке, зависят от того, насколько глубоко мы проникли в функцию к тому моменту, как прогремел гром. Есть два подхода к уборке:

- Выполнить небольшую приборку после каждой операции, которая может дать сбой, затем досрочно вернуться. При этом неизбежно дублируется код уборки. Чем больше работы вы проделали, тем больше объем наведения порядка, поэтому в каждой точке выхода растет объем того, что нужно вернуть на прежнее место.

  Если в каждой операции нашего примера выделяется какой-то объем памяти, в каждой точке преждевременного выхода придется осво-

бождать всю память, выделенную к тому моменту. Чем дальше, тем больше. В результате получается пространный код обработки ошибок с многочисленными повторениями, значительно увеличивающий размер функции и сложность ее понимания.

Сочиняем сообщения об ошибках

Ваш код неизбежно столкнется с ошибками, исправлять которые должен пользователь. Без человека тут не справиться: не будет же ваш код вставлять дискету или включать принтер! (Если он это сможет, вы станете богатым человеком.)

Если вы собираетесь пожаловаться пользователю, нужно учесть несколько обстоятельств:

- Пользователи мыслят иначе, чем программисты, поэтому представлять информацию нужно в том виде, который им понятен. Показывая оставшееся на диске пространство, вы можете напечатать: `на диске свободно 10K`. Но если свободного места нет вообще, ноль может быть неправильно интерпретирован как `OK` – к недоумению пользователя, который не может сохранить файл, когда программа сообщает ему, что все в порядке.
- Старайтесь делать свои сообщения как можно менее загадочными. Если они понятны вам, это не значит, что они будут понятны вашей бабушке. (И пусть ваша бабушка не будет работать с вашей программой – кто-нибудь с невысоким интеллектом почти наверняка ей воспользуется.)
- Не показывайте не имеющие смысла коды ошибок. Ни один пользователь не поймет, что ему делать, встретив «`код ошибки 707E`». Однако такие коды полезно указывать в качестве «дополнительной информации» – их можно сообщить в службу технической поддержки или поискать в Интернете.
- Делайте различие между серьезными ошибками и простыми предупреждениями. Пишите об этом в тексте сообщения (можно ставить префикс `Error:`) и помещайте соответствующие значки в окна сообщений.
- Задавайте вопросы (даже простые «*Продолжить: Yes/No*?»), только если пользователю должны быть совершенно понятны последствия его выбора. Сделайте пояснения, если это необходимо, чтобы было ясно, к чему приводит каждый ответ.

То, что вы можете показать пользователю, зависит от ограничений интерфейса и приложения, а также стиля, принятого для ОС. Если в вашей компании есть инженеры по интерфейсам, то такие решения должны принимать они, а вам следует с ними сотрудничать.

- Написать код уборки один раз и поместить его в конце функции, но он должен быть таким, чтобы убирать только там, где действительно что-то испорчено. Это более аккуратный код, но если в середине функции случайно произойдет возврат, до выполнения кода уборки дело не дойдет.

Если не следовать безусловно и неукоснительно принципу наличия в функциях *единственных точек входа и выхода* (*SESE*), то подойдет следующий пример, устраняющий зависимость от отдельной переменной, управляющей ходом выполнения кода.[1] Однако мы снова теряем код уборки. Код так прост, что фактические намерения хорошо видны:

```
void shortCircuitErrorHandling()
{
    if (!operationOne()) return;
    ... какие-то действия ...
    if (!operationTwo()) return;
    ... еще какие-то действия ...
    if (!operationThree()) return;
    ... еще действия ...
}
```

Объединение такого досрочного завершения с требованием выполнить уборку приводит к следующему подходу, чаще встречающемуся в системном коде нижнего уровня. Некоторые считают, что это *единственное* законное применение зловредного оператора `goto`. Я в этом не вполне убежден.

```
void gotoHell()
{
    if (!operationOne()) goto error;
    ... какие-то действия ...
    if (!operationTwo()) goto error;
    ... еще какие-то действия ...
    if (!operationThree()) goto error;
    ... еще действия ...
    return;
error:
    ... уборка после ошибок ...
}
```

В C++ можно избежать этого уродливого кода с помощью технологии *Resource Acquisition Is Initialization* (*RAII*) типа умных указателей (Stroustrup 97). Дополнительное достоинство – безопасная обработка

[1] Очевидно, это не SESE, но я утверждаю, что и предшествующий пример не был им. Точка выхода только одна – в конце, но ход управления таков, что моделирует досрочное завершение, и точек выхода с таким же успехом могло быть несколько. Это хороший пример того, как соблюдение правил типа SESE может приводить к появлению плохого кода, если тщательно не продумывать последствия.

исключений: когда выполнение функции прерывается досрочно и генерируется исключительная ситуация, ресурсы высвобождаются автоматически. Такие приемы позволяют избежать множества проблем, с которыми мы сталкивались выше, перемещая сложность в отдельный поток управления.

Тот же пример с использованием исключительных ситуаций, в предположении, что все вызываемые функции не возвращают код ошибки, а генерируют исключительные ситуации, выглядел бы так (в C++, Java и C#):

```
void exceptionalHandling()
{
    try
    {
        operationOne();
        ... какие-то действия ...
        operationTwo();
        ... еще какие-то действия ...
        operationThree();
        ... еще действия ...
    }
    catch (...)
    {
        ... уборка после ошибок ...
    }
}
```

Это элементарный пример, но он показывает, насколько удачно могут действовать исключительные ситуации. В хорошей конструкции кода блок `try/catch` может вообще не потребоваться, если гарантируется отсутствие утечки ресурсов, а обработка ошибок передается на более высокий уровень. К сожалению, написание хорошего кода с использованием исключительных ситуаций требует усвоения принципов, выходящих за рамки этой главы.

Подымаем скандал

Мы уже довольно долго занимаемся ошибками, которыми нас снабжают другие. Пора поменяться местами и самим выступить в роли нехороших мальчиков: будем генерировать ошибки. Если вы пишете функцию, в ней могут происходить неправильные вещи, о которых нужно сообщать вызывающему. И делать это необходимо – не проходите мимо сбоев. Даже если вы уверены, что вызвавший не будет знать, что делать с проблемой, вы должны проинформировать его. Не пишите лживый код, который якобы делает то, чего на самом деле он не выполняет.

Каким механизмом сообщения об ошибке воспользоваться? Это в значительной мере определяется архитектурой: соблюдайте установленные в проекте правила и употребляйте стандартные идиомы языка.

В языках, где такая возможность есть, обычно отдают предпочтение исключительным ситуациям, но применяйте их только тогда, когда это делается во всем проекте. Java и C# на самом деле не оставляют вам выбора; исключительные ситуации лежат глубоко в основе их среды выполнения. В архитектуре C++ иногда отказываются от этого средства ради переносимости на платформы, не поддерживающие исключений, или для совместимости со старым кодом C.

Мы уже рассмотрели стратегии передачи ошибок, генерируемых вызываемыми функциями. Теперь наша забота состоит в том, чтобы сообщить о новых проблемах, возникающих во время выполнения. Как вы станете обнаруживать ошибки – дело ваше, но при генерации сообщений о них нужно руководствоваться рядом правил:

- Во-первых, тщательно ли вы произвели уборку? В надежном коде не происходит утечка ресурсов и не возникают противоречивые состояния даже в случае ошибок, если только не оказывается *невозможным* корректно прибрать за собой. Такие случаи должны быть тщательно задокументированы. Подумайте, как поведет себя ваш код при новом вызове после возникновения такой ошибки. Необходимо гарантировать его работоспособность.
- Не помещайте лишнюю информацию в свои сообщения об ошибке. Выведите только полезные данные, которые сможет понять и использовать вызвавший.
- Корректно пользуйтесь исключительными ситуациями. Не генерируйте исключений, если возвращаемые значения необычны – они редки, но не ошибочны. Исключения должны сигнализировать только о тех случаях, когда функция не смогла выполнить свой контракт. Не применяйте их с целями, для которых они не предназначены (например, для управления потоком выполнения).
- Для перехвата ошибок, которые никогда не могут произойти в ходе обычного выполнения программы, т. е. подлинных ошибок программирования, можно воспользоваться оператором контроля `assert` (см. раздел «Ограничения» на стр. 45). Применение исключительных ситуаций в таких случаях тоже допустимо: некоторые механизмы контроля можно настраивать так, что они будут генерировать исключения в случае ошибок.
- Если какие-то тесты можно заранее провести на этапе компиляции, не нужно этим пренебрегать. Чем раньше будет обнаружена и исправлена ошибка, тем меньше неприятностей она принесет.
- Постарайтесь, чтобы ваши ошибки было сложно игнорировать. При первой же возможности вашим кодом *воспользуются* не так, как вы планировали. Исключения здесь полезны: нужно приложить специальные усилия, чтобы подавить исключение.

Какого рода ошибки нужно отслеживать? Очевидно, это зависит от того, какую задачу решает функция. Вот контрольный список общих типов ошибок, которые нужно проверять в каждой функции:

- Проверка всех параметров функции. Убедитесь, что вам передали корректные и допустимые данные. Для этого можно воспользоваться операторами проверки в зависимости от того, насколько строго был составлен контракт. (Не является ли его нарушением передача недопустимых параметров?)
- Проверка выполнения инвариантов в важных точках кода.
- Проверка допустимости всех данных из внешних источников перед их использованием. Содержимое файлов и данных интерактивного ввода должно быть разумным и не иметь пропусков.
- Проверка состояния возврата всех системных вызовов и вызовов подчиненных функций.

Исключение из правила

Исключительные ситуации – мощный механизм сообщения об ошибках. При правильном применении они могут значительно упростить код и способствовать его стабильности. Однако в неумелых руках они становятся опасным оружием.

Однажды я работал над проектом, где среди программистов было принято прерывать цикл `while` или завершать рекурсию путем генерации исключительной ситуации, используемой как оператор дальнего перехода. Идея кажется интересной, когда встречаешься с ней впервые. Но такая технология – издевательство над исключительными ситуациями: идиоматически они применяются для других целей. Множество критических ошибок появилось в результате того, что сопровождавший код программист не понимал последовательности передачи управления в сложном и нестандартно завершаемом цикле.

Следуйте идиомам своего языка и не пытайтесь сочинять остроумный код ради собственного развлечения.

Управление ошибками

Стандартный принцип для генерации и обработки ошибок требует наличия единообразной стратегии обработки сбоев, в каком бы месте они ни проявились. Вот общие административные соображения, относящиеся к возникновению, обнаружению и обработке программных ошибок:

- Избегайте ситуаций, *чреватых* ошибками. Лучше напишите код, который гарантированно будет работать. Например, чтобы не возникало ошибок выделения памяти, заранее позаботьтесь о резервировании достаточных ресурсов. Когда у вас обеспечен пул памяти, программа не может испытывать недостатка в памяти. Конечно,

это осуществимо, только если заранее известно, сколько нужно ресурсов, но часто именно так и бывает.

- Определите возможное поведение вашей программы или функции в необычных обстоятельствах. Исходя из этого установите, насколько устойчивым должен быть код и насколько тщательной – обработка ошибок. Не получится ли так, что функция, не привлекая внимания, сгенерирует скверные выходные данные согласно классическому принципу *GIGO*?[1]
- Четко распределите между компонентами обязанности по обработке тех или иных ошибок. Объявите это в интерфейсе модуля. Пусть ваш клиент знает, что будет работать всегда, а что может в один прекрасный день отказать.
- Проконтролируйте свой стиль программирования: *в какой момент* вы пишете код обработки ошибок? Не откладывайте его на будущее; обязательно что-нибудь пропустите. Не откладывайте написание обработчиков до того времени, когда тестирование выявит проблемы – это не инженерный подход.

Если вы пишете код, который может отказать, одновременно с ним пишите код для обнаружения и обработки ошибок. Не откладывайте это на будущее. Если вы все же вынуждены отложить обработку, по крайней мере, напишите оснастку для обнаружения ошибок.

- Если вы перехватили ошибку, что это – симптом или причина? Вы можете обнаружить источник проблемы, которую нужно исправлять тут же, или признак более застарелой проблемы. В последнем случае не нужно писать много кода в этом месте; лучше поместить его там, где он более уместен – в более раннем обработчике ошибки.

Резюме

Ошибка – от человека, раскаяние – от Бога, упрямство – от дьявола.

Бенджамин Франклин

Людям свойственно ошибаться (но у компьютеров это тоже хорошо получается). Исправление ошибок – святое дело.

На всякую написанную вами строчку кода должен приходиться надлежащий объем кода для доскональной проверки и обработки ошибок. Программа без строгой обработки ошибок не может быть надежной. В один прекрасный день случится какая-то неясная ошибка, и в результате программа рухнет.

---

[1] *Garbage In, Garbage Out* – если мусор на входе, то и результатом станет мусор.

Обработка ошибок и ситуаций отказа – тяжелый труд. Здесь программирование сталкивается с реалиями окружающего мира. Однако это абсолютно необходимая работа. По некоторым подсчетам 90% создаваемого кода занимается обработкой исключительных обстоятельств (Bentley 82). Это неприятная статистика, поэтому при написании кода будьте *готовы* к тому, что гораздо больше труда придется вложить в то, что может пойти вкривь и вкось, чем в то, что будет правильно работать.

Хорошие программисты...

- Сочетают хорошие замыслы с хорошей практикой кодирования
- Пишут код обработки ошибок одновременно с основным кодом
- *Досконально* рассматривают в своем коде все возможные случаи сбоев

Плохие программисты...

- Пишут код беспорядочно, не обдумывая его заранее и не пересматривая потом
- Игнорируют ошибки, возникающие во время написания кода
- Проводят длительные сеансы отладки после аварий программы, потому что никогда не задумывались, что может стать причиной ошибки

См. также

Глава 1. Держим оборону

Обработка ошибок в контексте – один из многих приемов защитного программирования.

Глава 4. Литературоведение

Самодокументирование кода гарантирует, что обработка ошибок становится неотъемлемой частью написания кода.

Глава 9. Поиск ошибок

Необработанные сбойные ситуации проявляются в виде ошибок в коде. Описываются способы борьбы с ними. (Лучше было бы не позволять им возникать.)

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 607.

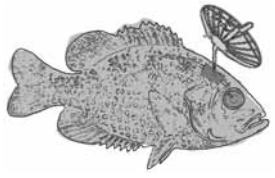
Вопросы для размышления

1. Эквивалентны ли такие механизмы сообщения об ошибках, как *возвращаемые значения* и *исключительные ситуации*? Докажите.
2. Можете предложить какие-нибудь реализации возвращаемых типов *кортежей*? Не ограничивайтесь каким-либо одним языком программирования. В чем достоинства и недостатки возвращаемых значений типа кортежа?
3. В чем различия между реализациями механизма исключительных ситуаций в разных языках?
4. Сигналы – это классический механизм UNIX. Нужны ли они попрежнему, когда мы располагаем современными технологиями типа исключительных ситуаций?
5. Какая структура кода лучше всего подходит для обработки ошибок?
6. Как бы вы поступили с ошибками, возникшими в вашем коде обработки ошибок?

Вопросы личного характера

1. Насколько полно реализована обработка ошибок в вашем нынешнем программном проекте? Как это отражается на стабильности программы?
2. Занимаетесь ли вы обработкой ошибок во время написания кода или она является неприятным отвлечением, которое вы откладываете на более позднее время?
3. Возьмите последнюю написанную вами функцию (достаточного размера) и тщательно проанализируйте ее код. Найдите все необычные ситуации и условия возможного возникновения сбоев. Все ли они учтены в коде обработки ошибок?

   Теперь предложите проанализировать свой код кому-то постороннему. Не стесняйтесь! Нашлись ли другие опасные ситуации? Почему? Как это характеризует код, над которым вы работаете?
4. Как вам проще справляться со сбойными ситуациями – с помощью *возвращаемых значений* или *исключений*? Вы уверены, что умеете писать код, корректно обрабатывающий исключительные ситуации?



II

Тайная жизнь кода

В этой части исследуются искусство и ремесло создания кода – повседневная работа программиста. Эти темы не являются строго охраняемыми секретами, тем не менее вы не часто услышите обсуждение ее специалистами или найдете что-либо написанное по этому поводу. Несмотря на это, овладение всеми секретами необходимо, если вы хотите писать хорошие программы; профессиональный кодировщик должен разбираться во всех этих вопросах.

Мы рассмотрим:

Глава 7. Инструментарий программиста

Обзор профессиональных инструментов и способов их применения.

Глава 8. Время испытаний

Код не может считаться завершенным, пока не проверено, что он решает поставленные задачи; он должен быть протестирован. Мы рассмотрим технологии проведения тестирования.

Глава 9. Поиск ошибок

Борьба с неизбежным: как найти и удалить ошибки, имеющиеся в коде.

Глава 10. Код, который построил Джек

«Сборка» кода: процедура создания выполняемых программ из исходного кода.

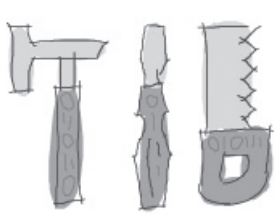
Глава 11. Жажда скорости

Рассмотрим детали, связанные с оптимизацией кода. Что, зачем, когда и как.

Глава 12. Комплекс незащищенности

Тернистая тема безопасности программ – как защитить код от умышленного злоупотребления и злонамеренных атак.

Таковы фундаментальные аспекты создания кода. В условиях промышленного производства программного обеспечения с его временными и прочими ограничениями эти навыки оказываются не просто полезными, но обеспечивают выживание. По мере накопления опыта они становятся вашей второй натурой, а ваши усилия направляются на более важные проблемы: архитектуру будущей системы, изменение требований заказчика и кого послать за очередной чашкой кофе.



7

Инструментарий программиста

Какие инструменты нужны для создания программ

В этой главе:

- Инструменты, используемые при создании кода
- Эффективная работа с инструментами
- Стандартные типы инструментов

Для нас опасны все средства искусства, более глубокого чем наше.

Дж. Р. Р. Толкиен

Чтобы успешно заниматься ремеслом, необходим хороший набор инструментов. У водопроводчика в ящике с инструментами лежит все, что может ему понадобиться в любой возникшей ситуации, иначе он рискует, что в следующий раз, когда у вас потечет кран, вы его не вызовете.

Важно не только *наличие* этих инструментов, но и высокое их *качество*; хороший мастер может оказаться бессилен, если у него плохой инструмент. Если вентили дрянные, они потекут, как бы ни старался ваш водопроводчик.

Конечно, мастером вас делает *умение* пользоваться этим инструментом. Сами по себе инструменты ничего не могут. В те времена, когда еще не было электроинструментов, столяры умели изготавливать изысканную

мебель. Инструменты были гораздо проще, но и их было достаточно мастеру, чтобы создавать произведения искусства.

То же самое относится к программированию. Чтобы хорошо выполнить работу, вам нужен соответствующий набор инструментов – таких, чтобы они вызывали у вас доверие и были пригодны для решения тех задач, с которыми вы столкнетесь; к тому же вы должны уметь ими пользоваться. Чтобы написать достойный код, нужны опытный мастер, хорошие инструменты и умение пользоваться этими инструментами.

Это важный вопрос. Благодаря умелому применению инструментов вы можете выделиться на общем фоне как действительно эффективно работающий программист. В отдельных случаях эти инструменты могут определить успех или провал вашего проекта. Неумолимое наступление промышленного производства программ требует, чтобы вы не упускали никаких средств, которые позволят вам писать лучший код и делать это быстрее и надежнее.

В последующих главах будут обсуждаться проблемы, связанные с конкретными инструментами. Здесь же мы обсудим тему *инструментов программирования* в целом. Программирование – это область деятельности, в которой обойтись без инструментов просто невозможно. Мы пользуемся ими ежедневно, даже не задумываясь об этом, и воспринимаем компилятор как нечто само собой разумеющееся – что-то вроде консервного ножа, который всегда под рукой. Но стоит ему споткнуться (например, на банке непривычной формы), и вы оказываетесь в затруднении, каким бы навороченным ни был этот прибор для открывания консервов. Простой и бесхитростный работающий консервный нож лучше, чем неработающая претенциозная штуковина.

Что такое инструмент программирования?

Для изготовления программного обеспечения мы пользуемся широким спектром инструментов – *программ, которые строят программы,* если не слишком вдаваться в философию. Все, с помощью чего мы создаем программное обеспечение, является в своем роде инструментом. Одни инструменты помогают писать код. Другие помогают писать *хороший* код. Третьи помогают привести в порядок тот запутанный код, который вы только что написали.

Есть инструменты на любой вкус, со своими принципами действия. Очевидно, следует учитывать платформу и среду, для которой они предназначены, но есть и другие характерные особенности.

Сложность

Некоторые инструменты представляют собой весьма сложные среды с множеством функций и возможностей настроек. Другие оказываются миниатюрными утилитами, решающими отдельные задачи. В каждом случае можно найти свои достоинства и недостатки:

- Насыщенный функциями инструмент замечателен, если вы *наконец-то* научите его одновременно варить вам кофе и приносить булочки. Но когда многочисленные сложные функции затрудняют его применение, пользы от него не так много.
- Простые инструменты легко осваивать: очевидно, какую функцию они выполняют. В конце концов, вы накапливаете их в большом количестве, по одному для каждой задачи. Но если попытаться связать их вместе, то из-за большого количества точек интерфейса их взаимодействие не всегда оказывается гладким.

Области применения инструментов могут весьма различаться: в одних случаях они выполняют весьма узкие задачи (например, поиск строк в текстовых файлах), в других поддерживают целые проекты (среда совместной разработки проекта).

Частота применения

Некоторыми инструментами пользуются постоянно; без них просто не обойтись. Другие извлекают из дальнего угла в редких случаях, когда в них возникает необходимость.

Интерфейс

Есть инструменты с красивыми *графическими интерфейсами пользователя* (*GUI*). Другие сделаны попроще – запускаются из *командной строки* (*CLI*) и направляют свои результаты в файл. Что выберете вы, зависит от устройства ваших мозгов и выработавшихся привычек.

Утилиты для Windows обычно графические и не предоставляют интерфейс командной строки. В UNIX принят противоположный подход, что облегчает автоматизацию и объединение в крупные комплексы с помощью скриптов. Интерфейс определяет способ, которым вы укрощаете мощь своего инструмента.

Интеграция

Из некоторых инструментов формируют длинные цепочки, которые часто включают в *интегрированные графические среды разработки* (*IDE*). Отдельные утилиты командной строки часто выдают результат в виде обычного текста в формате, допускающем ввод другими утилитами, благодаря чему они выступают в качестве фильтров данных.

Единые интерфейсы GUI бывают очень удобны в работе, а интеграция заметно повышает продуктивность программиста. С другой стороны, для того чтобы они работали так, как вам того хотелось бы, может потребоваться длительная настройка, и не всегда удается получить все возможности, доступные при задании командной строки вручную. Однако, несмотря на исключительную мощь утилит UNIX, наличие у них весьма загадочных интерфейсов затрудняет работу с ними.

Стоимость

Существует масса отличных бесплатных инструментов.[1] Но часто их бесплатность оборачивается иной стороной. Обычно бесплатные инструменты предоставляют документацию более низкого качества, слабый уровень поддержки или ограниченный набор функций. Но так бывает не всегда. Ряд бесплатных инструментов значительно превосходит свои коммерческие аналоги.

Сколько бы вы ни заплатили за любой инструмент из той или иной категории, более высокая цена не гарантирует вам более высокое качество. Мне приходилось работать с немыслимо дорогими инструментами, которые функционировали на удивление плохо. Поэтому рассмотрим еще один аспект инструментов.

Качество

Есть действительно отличные инструменты. И есть явно плохие инструменты. Мне приходилось работать с парой важных инструментов – таких, что я предпочел бы никогда больше с ними не сталкиваться: худо-бедно они как-то решают задачу, но при этом постоянно балансируют на грани краха. Но без них я не смог бы изготовить тот код, который требовался заказчику. Не счесть случаев, когда у меня возникало желание переписать их самому.

С учетом рассмотренных характеристик нужно выбрать для себя инструменты, сделав приемлемые компромиссы. Хотя очень важно привыкнуть к инструменту, которым вы часто пользуетесь, хорошо изучить его и эффективно применять, не стоит делать из него фетиш. Большинство пользователей Windows с презрением относятся к разработкам в стиле UNIX, а UNIX-специалисты смотрят свысока на тех, кто программирует в Windows и, следовательно, не может воспользоваться командной строкой. Будьте выше этого.

Я бы предложил вам поработать над достаточно крупным проектом в другой, малознакомой среде. Это поможет вам лучше понять, каким должен быть хороший набор инструментов, и более объективно оценить существующие – независимо от выбранной платформы.

А зачем они нужны – инструменты?

Невозможно создавать программы без базового набора программных инструментов; вы не сдвинетесь с места без редактора и компилятора. Есть ряд инструментов, без которых *можно* обойтись, но все же они

[1] «Free» имеет в мире программирования два значения: free как в *пиве* (вы получите его бесплатно) и free как в *свободе речи* (программное обеспечение open source, код которого можно читать и модифицировать). Что для вас важнее, зависит от степени вашего идеализма. См. врезку «Лицензии» на стр. 461.

очень полезны. Чтобы повысить свою продуктивность, качество кода и мастерство, полезно присмотреться к инструментам, которыми вы пользуетесь в данное время, и узнать, какие еще возможности они в себе таят.

Когда вы разберетесь в том, как работают ваши инструменты и для каких работ они лучше всего подходят, у вас появится больше возможностей создавать правильно работающий код – и делать это быстро. Чем толковее ваш инструмент, тем толковее вы сами как программист.

Изучите свои стандартные инструменты вдоль и поперек. Время, которое вы потратите на их изучение, незамедлительно окупится.

Отдадим себе отчет, для чего мы пользуемся инструментами. Инструменты не делают нашу работу *вместо нас*, но они дают нам *возможность* делать свою работу. Качество программного обеспечения всегда определяется компетентностью программиста, написавшего его. Вспомните об этом, когда ваш компилятор в очередной раз распечатает кучу страниц с ошибками. Вы написали этот код, а не кто-то другой!

Отношение программистов к выбору и применению инструментов оказывается весьма различным. Вероятно, сказываются какие-то глубокие психологические причины – что-то связанное с принадлежностью к «злым гениям». Столкнувшись с длинной новой задачей:

- одни программисты сами старательно пишут код для ее решения;
- другие пишут нечто на языке сценариев, чтобы работа выполнялась автоматически;
- третьи часами разыскивают уже написанный инструмент, который сделает работу вместо них.

При наличии инструмента, который *может* решить проблему:

- одни программисты играют с ним и так, и сяк, пока не добьются результата, близкого к нужному;
- другие тщательно изучают документацию, выясняя, что конкретно можно сделать, и уж *потом* начинают применять инструмент.

Как правильнее поступить? Все зависит от обстоятельств. Зрелость программиста отчасти определяется его умением различать ситуации, требующие индивидуального подхода, и правильно выбирать средства для решения текущей задачи. Все люди разные, и работать привыкли по-разному; возможно, ваши коллеги успешнее всего работают с инструментами, к которым вы не слишком расположены. Но все же, если кто-то систематически вручную переводит свой код C в язык ассемблера, вы вправе усомниться в его вменяемости.

Отнеситесь к инвестированию времени и средств в инструменты практически. Поразмышляйте над тем, как вы станете пользоваться инструментом. Поищите другой инструмент или напишите свой собственный, если считаете, что потраченное на это время *окупится*. Не следу-

ет тратить неделю на создание инструмента, который будет экономить вам один час каждый месяц. Но если инструмент позволит сберечь этот час ежедневно, то разумно потратить и неделю на его создание.

Относитесь к инструментам программирования прагматически; пользуйтесь ими, только если они облегчают вашу жизнь.

Электроинструменты

Программирование и применяемые в его процессе инструменты тесно связаны между собой, поэтому, чтобы стать суперпрограммистом, вы должны пользоваться суперинструментом. Что это значит?

Во-первых, нужно хорошо представлять себе, какие инструменты существуют. В следующем разделе мы рассмотрим список стандартных инструментов, которые должны быть под рукой у каждого программиста. Не нужно знакомиться со всеми до единого инструментами, присутствующими на рынке: во всяком случае это очень скучная тема для разговора за обедом. Важным шагом вперед будет хотя бы выяснение того, какие общие категории инструментов существуют. Это поможет вам сделать выбор между поиском инструмента для решения определенной задачи, написанием собственного инструмента или выполнением задачи вручную.

Потратьте некоторое время на изучение вопроса. Узнайте, где можно добыть какие-то из этих инструментов; существуют магазины, специализирующиеся на продаже инструментов программирования, и множество сайтов для их загрузки через Интернет. Возможно, у вас уже стоит нужный инструмент, которым вы никогда не пользовались или не догадывались о его ценности. Выясните, что умеют эти инструменты; в результате вы сможете воспользоваться ими с бóльшей эффективностью.

Узнайте, какие виды инструментов существуют. Выясните, где их можно взять, даже если в данный момент они вам не нужны.

Если появляется новый инструмент, постарайтесь изучить его; это правильный подход. Не исключено, что вам придется искать новые инструменты в случае начала нового проекта, перехода на другую платформу, возникновения неожиданных проблем или устаревания привычных для вас инструментов. Но не ждите, пока жизнь заставит вас сделать это – старайтесь, чтобы в вашем распоряжении всегда были лучшие из имеющихся инструментов.

Выделите часть своего времени для оттачивания навыков работы с инструментами – вы же читаете технические книги и журналы и посещаете курсы повышения квалификации! Вопрос важный, и уделяйте ему достаточно внимания.

Вот несколько простых шагов, позволяющих овладеть инструментом в совершенстве. Для каждого вида оружия из вашего арсенала разработчика программного обеспечения...

Выясните, каковы его возможности

Разберитесь с функциями инструмента – с тем, что он умеет *в действительности*, а не по *вашим предположениям*. Даже если вы не узнаете, как выжать из него все до последней капли (для этого могут понадобиться какие-нибудь таинственные параметры в командной строке), представление о его возможностях может оказаться полезным.

Есть ли какие-то конкретные задачи, с которыми инструмент не справляется? Возможно, он не поддерживает некоторые функции, реализованные у конкурентов. Разберитесь, каковы его ограничения, и тогда вы будете знать, в каких случаях вам понадобится более совершенный инструмент.

Научитесь им управлять

Если вам удалось воспользоваться инструментом и не выйти при этом на ошибку, не означает, что он сделал все *в точности*, как вы хотели. Вы должны уметь правильно пользоваться им и быть уверенным, что он сделает то, что вам требуется.

Какое место занимает инструмент в общей цепочке? От этого зависит его применение. Например, инструменты UNIX можно последовательно применять как фильтры в *конвейере*, *собирая* из отдельных мелких утилит более крупные.[1] Разобравшись с управлением каждым инструментом и взаимодействием их между собой, вы подниметесь на новый уровень работы с инструментарием.

Для каждого инструмента определите, как лучше им управлять – путем непосредственного вызова или с помощью щелчка где-то в дебрях GUI. Может ли он запускаться автоматически? Компилятор часто запускается не вручную, а в рамках системы сборки.

Выясните, для каких задач он пригоден

Выясните, какое место каждый инструмент занимает в контексте остальных имеющихся инструментов. Например, я могу создать в своем текстовом редакторе макросы с записью нажатий на клавиши, и это сэкономит мне много времени при выполнении повторяющихся действий. Часть таких модификаций я мог бы выполнить, вызвав sed.[2] Од-

---

[1] Если эта тема вам мало знакома, настоятельно советую ее изучить. Вполне можно начать с команды UNIX `man bash`, а также поискать страницы руководства по *pipelines*.

[2] sed – это потоковый редактор с интерфейсом командной строки, описанный в следующем разделе.

нако макросы в данном контексте будут эффективнее – ведь я уже нахожусь в редакторе и поэтому могу быстро их запустить.

Возможно, вы не знаете, как пользоваться уасс,[1] но если вам когда-либо понадобится написать синтаксический анализатор, при наличии этого инструмента ваша задача несравненно облегчится.

Для каждой задачи есть свой инструмент. Не стоит щелкать орехи кувалдой.

Убедитесь, что он работает

Каждый в какой-то момент может пострадать от недостатков инструментов. Например, ваш код не хочет работать, но все ваши поиски причин ошибочного поведения оказываются тщетными. В отчаянии вы начинаете тестировать что попало – оттуда ли дует ветер и прочно ли установлено освещение. По прошествии нескольких часов обнаруживается, что это ваш инструмент «выдает коленца».

Компиляторы могут порождать ошибочный код. Системы сборки могут неправильно определить зависимости. Ошибки встречаются в библиотеках. Научитесь проверять наиболее очевидные сбои, прежде чем рвать волосы на своей голове.

Наличие исходного кода ваших инструментов может помочь диагностировать возникшие проблемы, дав возможность точно узнать, как работает инструмент. Данный фактор бывает решающим при выборе набора инструментов.

Имейте четкие данные о том, как получить дополнительные сведения

Необязательно самому знать все. Фокус в том, чтобы знать того, кто знает все!

Определите, где находится документация к инструменту. Кто осуществляет поддержку? Как получить дополнительную информацию? Найдите руководства, комментарии к версиям, сетевые ресурсы, внутренние файлы подсказки и страницы руководства (man). Выясните, где они находятся и как добраться до них при необходимости. Есть ли в сетевых версиях удобные средства поиска и именные указатели?

Узнайте, как получить новые версии

Инструменты развиваются с невероятной скоростью – технологии в нашей индустрии меняются очень быстро. Одни инструменты прогресси-

[1] Генератор синтаксических анализаторов; не беспокойтесь, далее и о нем будет сказано.

руют быстрее других. Только вы успели установить очередной widgetizer, как авторы выпустили новую версию с длинной красной полосой сбоку.

Необходимо иметь свежую информацию об используемых вами инструментах, чтобы не остаться с устаревшим, содержащим ошибки и неподдерживаемым набором. Но следует проявлять осторожность и не гоняться бездумно за последней версией. Находиться на переднем крае небезопасно!

Новые версии могут содержать новые ошибки и продаваться по новым, более высоким ценам. Обновляйте продукт, только если он сулит существенные улучшения и показал свою стабильность. Проверьте сначала новый инструмент на своем старом коде, чтобы убедиться в его корректной работе.

Следите за выпуском новейших версий своего инструмента, но проявляйте осторожность при обновлении.

Какой инструмент необходим?

Ассортимент инструментов для разработки программ ошеломляет. Потребности, которые часто возникают, известны, и за годы развития набралось много инструментов, удовлетворяющих любые прихоти. Если какая-то задача возникает многократно, можете быть уверены, что кто-то уже написал инструментарий для ее решения.

Точный состав вашего инструментального набора зависит от рода вашей работы. Инструменты для встроенных платформ обычно уступают в богатстве возможностей инструментам для настольных приложений. Ниже мы рассмотрим стандартные компоненты. Одни из них очевидны в большей степени, другие – в меньшей.

Мы рассмотрим каждый класс инструментов в отдельности, но нужно помнить, что современные IDE объединяют частные программы под единым удобным интерфейсом. Несмотря на это удобство, важно разобраться в самостоятельной роли каждого инструмента, поскольку

- вы сможете более эффективно использовать отдельные функции;
- вы узнаете, каких полезных функций нет в вашей IDE.

Обычно IDE имеют модульную структуру, что позволяет заменять отдельные компоненты более эффективными или добавлять возможности, отсутствующие в оригинальной версии. Изучайте существующие разновидности инструментов и совершенствуйте возможности своей IDE.

Средства редактирования исходного кода

Гончару материалом служит глина, скульптору – камень, программисту – код. Это основной материал, с которым мы работаем, поэтому

необходимо выбрать лучшие инструменты для написания, редактирования и изучения исходного кода.

Редактор исходного кода

Пожалуй, самый главный инструмент – редактор, и даже компилятор имеет меньшее значение. Компилятор обращен лицом к компьютеру, а редактор – к вам. Вы им управляете. С ним вы работаете больше всего, поэтому выберите хороший редактор и изучите *досконально* его возможности. Эффективная работа с редактором резко повышает ваши возможности писать код.

Выбор редактора кода имеет решающее значение; он оказывает огромное влияние на то, как вы будете писать код.

Выбор Единственного настоящего редактора исходного кода – это старый спор, который мы не станем поднимать вновь. Вам должно быть удобно работать в редакторе, и он должен отвечать всем вашим требованиям. Если в вашей IDE есть встроенный редактор, это не значит, что он окажется для вас лучшим. С другой стороны, то, что он встроен, может стать чрезвычайным удобством. Я предъявляю к редактору исходного текста как минимум следующие требования:

- Полная подсветка синтаксиса (при поддержке нескольких языков, т. к. я работаю со многими языками)
- Простая проверка синтаксиса (например, подсветка непарных скобок)
- Хорошие средства *инкрементального поиска* (интерактивное окно, осуществляющее поиск одновременно с набором с клавиатуры)
- Запись действий с клавиатурой в макросы
- Широкие возможности настройки
- Работа на всех платформах, которыми я пользуюсь

Ваши требования и выбор редактора могут отличаться от моих, но мне кажется, что я привел разумный список самых важных возможностей. При этом стоит потратить некоторое время, чтобы научиться наиболее эффективно пользоваться всеми этими возможностями. Результатом будет рост производительности труда.

В зависимости от того, чем вы занимаетесь, полезными могут оказаться другие типы редакторов. Существуют редакторы двоичных файлов (обычно показывающие содержимое файлов в шестнадцатеричном виде) и редакторы для файлов специальных форматов, например XML.

Vim и Emacs – печально известные редакторы из мира UNIX, которые теперь доступны практически на любой платформе, чуть ли не на электрических тостерах. Они весьма отличаются от редакторов, включаемых в IDE.

Средства обработки исходного текста

Философия UNIX предполагает наличие огромного набора меленьких инструментов командной строки. В средах GUI есть свои эквиваленты для каждого из таких инструментов, но они редко располагают такой же мощью или возможностями легко соединяться вместе. Зато версии GUI гораздо легче осваиваются.

Следующие команды UNIX предоставляют мощный механизм для изучения и модификации исходного кода:

`diff`

Сравнивает два файла и выявляет различия между ними. Базовая версия diff выводит результат на консоль, но существуют более изощренные графические версии. Есть даже редакторы, позволяющие работать со сравниваемыми файлами, показывая их рядом и обновляя различия по мере редактирования текста. Совсем необычные варианты diff сравнивают одновременно три файла.

`sed`

Сокращение от stream editor – *потоковый редактор*. sed читает файлы построчно и применяет к ним заданное правило преобразования. С помощью sed можно менять порядок элементов, выполнять глобальный поиск и замену или вставлять в строки шаблоны.

`awk`

Представьте себе sed на стероидах. awk – это тоже программа поиска по шаблону, обрабатывающая текстовые файлы. Для этого в ней реализован полный язык программирования, поэтому на awk можно писать весьма развитые скрипты для сложных манипуляций с текстом.

`grep`

Ищет в файле символы, соответствующие шаблону. Шаблоны описываются с помощью *регулярных выражений* – особого мини-языка с символами-заместителями и гибкими критериями соответствия.

`find/locate`

Это средства поиска файлов в файловой системе. Поиск можно вести по имени, дате и ряду других критериев.

Это лишь вершина айсберга; существует множество других инструментов. Например, `wc` подсчитывает символы/слова. Другие ценные утилиты: `sort`, `paste`, `join` и `cut`.

Средства навигации в исходном коде

В действительно крупном проекте код можно уподобить городу. Даже те, кто планировал его, не знают всех улиц на окраинах. Немногие таксисты знают, как лучше всего добраться в нужное место. Обычно жители хорошо знакомы лишь со своими ближайшими окрестностя-

ми. Туристы теряют всякую ориентировку, как только выйдут из автобуса.

Существует разновидность инструментов, с помощью которых легче разобраться в коде, составить его схемы, выполнять несложный поиск, навигацию и делать перекрестные ссылки. Некоторые инструменты строят графические деревья вызовов, по которым можно следить, как передается управление в системе. Они могут создавать графические схемы или объединяться с редактором, чтобы обеспечивать автозавершение, подсказывать сигнатуру функции и т. п. Это неоценимый инструмент для работы с большим объемом кода или первого знакомства с уже устоявшимся проектом.

Хорошим примером бесплатных инструментов такого рода служат LXR, Doxygen и почтенный ctags.

Управление версиями

Мы не станем здесь распространяться на тему этих инструментов, поскольку о них рассказывается в разделе «Управление версиями исходного кода» на стр. 449. Скажем лишь одно: если вы *не будете* ими пользоваться, вам следует отрубить руки.

Генерирование исходного кода

Есть ряд средств для автоматического генерирования исходного кода. Некоторые из них хороши, другие приводят меня в ужас.

Один из примеров – это yacc, генератор синтаксических анализаторов LALR(1).[1] Нужно определить правила входной грамматики, а затем с помощью этого инструмента генерировать программы, которые могут проводить синтаксический анализ корректных текстов, соответствующих этим правилам. В итоге получается анализатор на C с ловушками, куда можно добавить свои функции для анализа объектов. Аналогичные задачи решает Bison.

Есть класс инструментов, генерирующих код, которые применяются при разработке интерфейса пользователя. Они выдают рабочий серверный код. Особенно часто их применяют со сложными инструментариями GUI типа MFC. Если для библиотеки требуется инструмент, который выполняет *столько тяжелой* работы, из этого можно сделать вывод, что библиотека чрезмерно сложна (или порочна в своей основе). Будьте осторожны!

Помощники, которые пишут кучу вспомогательного кода, который нужно потом читать и модифицировать, тоже требуют осторожного отношения. Вы должны серьезно разобраться в сгенерированном коде, прежде чем начать его переделывать, либо вас ждут неприятности.

---

[1] Загадочный технический (и неинтересный) способ выражения достаточно сложной грамматики.

Если же после модификации сгенерированного кода вы снова запустите помощник, вся ваша ручная работа будет, увы, стерта.

Можно даже самому писать скрипты, чтобы генерировать повторяющиеся разделы кода. Иногда это свидетельствует о том, что ваш код недостаточно тщательно спроектирован. Но порой такой технический подход оправдан. Мне приходилось писать на Perl скрипты, которые автоматически генерировали для меня код. Когда я сам пишу такой генератор, то доверяю сгенерированному им коду. Посторонний программист может отнестись к такому коду с недоверием, так же как к результату работы любого «чужого» генератора кода.

Декораторы кода

Эти инструменты обеспечивают единообразное форматирование кода, создавая расположение кода по принципу *наименьшего общего кратного*. Лично я считаю, что от них больше вреда, чем пользы – с одинаковым успехом они могут как разрушить важное и полезное форматирование, так и исправить его.

Средства построения кода

Наша задача не в том, чтобы битый день глазеть на красивый исходный код. Нужно еще заставить его что-то делать. Для этого мы прибегаем к ряду средств, в наличии и работоспособности которых нисколько не сомневаемся и не интересуемся при этом, как они справляются со своей задачей.

Компилятор

Это самый часто используемый инструмент программирования после редактора исходных текстов. Компиляторы преобразуют исходный код в исполняемый модуль, чтобы вы могли полюбоваться, как ваша программа отказывается работать. Этот инструмент используется столь часто, что совершенно необходимо научиться правильно им управлять. Известны ли вам все его параметры и возможности? Во многих компаниях есть особый *мастер сборки*, который следит за правильным применением всех инструментов сборки, но даже наличие такой должности не освобождает вас от ответственности за знание своего компилятора.

- Известно ли вам, какой уровень оптимизации следует применять и какое влияние он оказывает на генерируемый код? Это важно – в частности от вашего выбора зависит поведение кода в отладчике и выявление ошибок компиляции.
- Включаете ли вы вывод всех предупреждений при компиляции? Если нет, вашему поведению нет оправдания (исключая сопровождение старого кода, уже пестрящего предупреждениями). Предупреждения сигнализируют о возможности возникновения ошибок, а если они отсутствуют, это усиливает вашу уверенность в коде.

- Совместим ли ваш компилятор по умолчанию со стандартами? Со стандартом C++ (ISO 98), стандартом 1999 года C (ISO 98), со стандартом Java (Gosling et al. 00)) и стандартом ISO для C# (ISO 05)? Есть ли у компилятора нестандартные расширения, какие и как их отключить?
- Задана ли генерация кода для нужного набора команд CPU? Вы станете генерировать 386-совместимый код, когда известно, что ваша программа будет выполняться только на новейших чипах Intel? Постарайтесь, чтобы компилятор выдавал наиболее приемлемый код.

Кросс-компилятор создает исполняемые модули для платформы, отличной от той, на которой ведется разработка. Обычно это применяется для создания встроенного программного обеспечения (не будете же вы запускать Visual C++ на посудомоечной машине!).

Мне нужен инструмент...

Вам нужно выполнить задачу. Скучную и повторяющуюся. Как раз с этим лучше справился бы компьютер: будет меньше ошибок, не так монотонно и гораздо быстрее. Для чего вообще изобретали компьютеры? Как узнать, нет ли какого-нибудь инструмента, который сделает все вместо вас?

- Если вы найдете его в списке, значит, такой инструмент есть.
- Если его нет в списке, но вы уверены, что не вам первому пришлось столкнуться с такой проблемой, вероятно, можно где-то найти подходящий инструмент. Быстрый поиск в Интернете может дать самые неожиданные и приятные результаты.
- Если задача кажется вам уникальной, можете попытаться написать собственную программу для ее решения. Подробнее об этом на стр. 178 в разделе «Мастерим сами».

При поиске инструмента постарайтесь получить как можно больше дополнительной информации:

- Поинтересуйтесь у коллег, не сталкивались ли они с подобным.
- Займитесь поиском в Сети и телеконференциях.
- Обратитесь к производителям инструментов.

Собрав перечень имеющихся инструментов, сделайте обоснованный выбор исходя из критериев, предложенных в первом разделе. Чтобы принять решение, нужно определить свои требования. Нужен ли вам непременно бесплатный инструмент? Или важнее получить его *немедленно*? Трудно ли будет воспользоваться им другим членам вашей группы? Часто ли вы собираетесь им пользоваться – будут ли оправданы затраты?

Компилятор – это лишь одно звено длинной цепочки инструментов, в которую входят редактор связей, ассемблер, отладчик, профайлер и прочие манипуляторы объектных файлов.

В число популярных компиляторов входят gcc, Microsoft Visual C++ и Borland's C++ builder.

Редактор связей

Редактор связей (linker, компоновщик) тесно связан с компилятором. Из всех *объектных файлов*, созданных компилятором, он собирает единый исполняемый модуль кода. Редакторы связей C и C++ так тесно связаны с компилятором, что иногда две задачи выполняются одной и той же программой. В Java и C# редактор связей относится к среде исполнения.

При работе с редактором связей нужно знать:

- *Чистит ли* он двоичные файлы? То есть удаляет ли отладочные символы типа имен переменных и функций? С их помощью отладчик показывает важную диагностическую информацию, но при этом значительно увеличивается размер исполняемых модулей и замедляется их загрузка.
- Исключает ли он повторяющиеся разделы кода?
- Может ли он создавать библиотеки вместо исполняемых модулей? Есть ли возможность выбирать тип библиотек – *статически* или *динамически* загружаемых?

Среда сборки

Среда сборки включает в себя не только компилятор и редактор связей. При сборке применяются такие инструменты, как UNIX-программа make или средства сборки, включенные в IDE. Они автоматизируют процесс компиляции. Во многих проектах с открытым кодом для упрощения сборки применяются такие инструменты UNIX, как autoconf и automake.

Научитесь выжимать максимум из своей интегрированной среды сборки, но не ценой изучения работы с каждым отдельным инструментом. Мы подробнее обсудим этот вопрос в главе 10.

Тестовая цепочка

Обратите внимание, что это инструмент создания программного обеспечения, а *не* его отладки! Правильно организованное тестирование совершенно необходимо для создания надежного, высококачественного программного обеспечения. Этим этапом часто пренебрегают, вероятно, из-за связанных с ним трудозатрат, отвлекающих силы от главного – написания кода. Такой подход может нанести большой ущерб качеству программного обеспечения. Нельзя построить надежный

фрагмент кода, не проверив корректности его работы, а сделать это можно только путем создания тестов во время написания кода.

Существуют инструменты, облегчающие тестирование блоков, предлагая скелет для построения собственного проверочного кода. Эти инструменты легко интегрируются с системой сборки, поэтому тестирование становится важной частью процесса создания кода.

Помимо инструментов автоматизированного тестирования блоков, есть средства для генерации тестовых данных и составления контрольных примеров. Существуют и средства эмуляции целевых платформ, часто с возможностью моделирования конкретных аварийных ситуаций (нехватки памяти, высокой нагрузки и т. п.).

Инструменты для отладки и тестирования

Эти инструменты оценивают работу кода и способствуют обнаружению проблем как наглядно проявляющихся, так и имеющихся в потенциале. Подробнее мы рассмотрим их на стр. 231 в разделе «Спрей от ос, репеллент для мух, липучки...».

Отладчик

Наличие хорошего отладчика и умение пользоваться им может сберечь вам массу времени, которое было бы потрачено на выяснение причин непонятного поведения программы. Вы сможете проследить ветви выполнения программы, вмешаться в выполнение, получить текущие значения переменных, установить контрольные точки и препарировать работающий код различными способами. Это на порядок эффективнее, чем усеивать программу отладочными операторами `printf`!

`gdb` – это отладчик GNU с открытым исходным кодом, портированный едва ли не на все мыслимые платформы. `ddd` – его вполне законченный графический интерфейс. В каждой IDE или инструментальной цепочке есть свой отладчик.

Профайлер

К этому инструменту прибегают, когда программа выполняется недопустимо медленно. Профайлер осуществляет хронометраж отдельных секций работающего кода и определяет среди них узкие места. С его помощью находят участки, *достойные* оптимизации, и не тратят время на совершенствование кода, который редко выполняется.

Валидаторы кода

Валидаторы кода бывают *статическими* и *динамическими*. Первые из них изучают код подобно компилятору, находя в файлах с исходным кодом источники возможных проблем и ошибочное применение языка. Известным примером такого рода является `lint` – программа, выполняющая статическую проверку ряда стандартных ошибок коди-

рования на C. Их функции в значительной мере интегрированы в современные компиляторы, однако сохраняются отдельные инструменты для дополнительных проверок.

Динамические валидаторы модифицируют и настраивают код во время компиляции, а затем осуществляют проверку на этапе исполнения. Хорошим примером служат средства выделения памяти/проверки границ – они гарантируют правильное освобождение всей динамически выделяемой памяти и пресекают выход за границы памяти при обращении к элементам массивов.[1] Эти средства избавляют от долгой и кропотливой работы при поиске ошибок неясного происхождения. Часто они оказываются *гораздо* полезнее отладчиков, поскольку выступают в качестве профилактических, а не лечебных средств: ошибки обнаруживаются раньше, чем им удается нарушить работу программы.

Измерительные средства

Эти инструменты осуществляют экспертизу кода и обычно выступают в роли статического анализатора (хотя есть средства и для динамического измерения). Они осуществляют статистическую оценку качества кода. Статистика может легко ввести в заблуждение, но данные инструменты обладают хорошими возможностями для обнаружения наименее надежных областей. Полученная информация помогает выбрать области кода для углубленного анализа.

Измерения обычно снимаются отдельно по каждой функции. Основной метрикой считается *количество строк кода*, за ней следует отношение *объема комментариев к объему кода*. Особой пользы они не представляют, но есть множество других, более интересных метрик. *Цикломатическая сложность* служит оценкой сложности кода, учитывающей количество точек ветвления и возможных потоков команд. Высокое цикломатическое число свидетельствует о неясности кода, которая может быть связана с его неустойчивостью и наличием ошибок.

Дизассемблер

Этот инструмент работает с исполняемыми модулями, позволяя исследовать машинный код. Подобные функции поддерживаются и отладчиками, но развитые дизассемблеры могут попытаться восстановить код при отсутствии в нем символической информации и создать интерпретацию двоичного программного файла на языке высокого уровня.

Система учета ошибок

В хорошей системе учета ошибок есть общая база, в которой регистрируются ошибки, обнаруженные в вашей системе. С ее помощью ваши коллеги получают возможность регистрировать ошибки, делать запросы,

[1] Языки с более высокой степенью социальной ответственности, такие как Java, решают такие проблемы на уровне архитектуры языка.

назначать ответственных, делать комментарии и, в конечном счете, помечать ошибки как исправленные. Это важный инструмент, способствующий повышению качества продукта; необходимо систематически управлять исправлением ошибок, иначе они неуловимо проникнут в конечный продукт и сделают его дефектным. Кроме того, регистрация и хранение данной информации полезны при выполнении ретроспективного анализа проекта.

Средства поддержки языка

Для написания программ на языках высокого уровня требуется обширная поддержка. Реализация языка делает все необходимое, чтобы облегчить написание программ и избавить от необходимости погружаться в машинные коды.

Язык

Язык сам по себе является инструментом. Возможности, предоставляемые одними языками, могут отсутствовать в других. Соответствующие пробелы можно восполнить с помощью отдельных инструментов, обрабатывающих исходный код программы. Например, часто осуждаемый препроцессор C может оказаться весьма полезен; есть пакеты обработки текстов и для других языков. Инструменты создания общего кода (типа шаблонов C++), а также проверка пред- и постусловий служат другим примером полезных средств работы с языками.

Очень ценно владеть некоторым набором языков программирования. Выясните, в чем их отличия, для каких задач они наиболее пригодны и в чем состоят их слабости. Тогда для каждой возникшей задачи вы сможете выбрать наиболее подходящий язык.

Выучите несколько языков. В каждом вы обнаружите особый способ решения задач. Рассматривайте их как набор инструментов и выбирайте тот, который наиболее эффективен в конкретной ситуации.

Исполнительная система и интерпретатор

Большинство языков не требует особой поддержки на этапе исполнения. Интерпретируемым языкам необходим интерпретатор (или *виртуальная машина*), но компилируемым языкам достаточно вспомогательных библиотек. Эти библиотеки часто нераздельно связаны с самим языком.

Как допустимо выбрать иной компилятор, так же можно выбрать и иную исполнительную среду для языка, с другими характеристиками.

JVM (*виртуальная машина Java*) – стандартный интерпретатор языка. Стандартная библиотека C++ поддерживает язык, обеспечивая действия по умолчанию некоторых базовых функций языка. Аналогично язык C# полагается на поддержку среды .NET на этапе исполнения.

Компоненты и библиотеки

И это тоже инструменты! Благодаря повторному использованию программных компонент и наличию библиотек с нужными вам функциями вы избавляетесь от необходимости изобретать колесо заново. Хорошая библиотека повысит эффективность труда программиста не хуже любого другого программного инструмента.

Такие библиотеки могут охватывать самые разные области – от обширных уровней абстрагирования до ОС целиком, хотя иные из них очень просты, предоставляя всего лишь какой-нибудь скромный класс *данных*. Они заботятся о деталях и скрывают сложности реализации, чтобы вам не пришлось о них тревожиться. Не нужно тратить время на написание, тестирование и отладку собственных версий.

Все современные языки предоставляют некоторый уровень библиотечной поддержки. Прекрасный пример мощной расширяемой библиотеки – C++ STL. Язык Java и среда .NET поставляются с труднообозримым множеством стандартных библиотек. Существует масса библиотек сторонних производителей – как коммерческих, так и бесплатных.

Инструменты различного назначения

На этом история не заканчивается. Вы столкнетесь со многими другими инструментами. Раздел «См. также» на стр. 179 отмечает ряд других мест, где мы будем говорить об инструментах.

Вот еще некоторые интересные разновидности инструментов.

Средства документирования

Хорошая документация имеет большую ценность. Есть много средств, способствующих ее составлению, как в составе исходного кода, так и отдельно от него (о части из них рассказывается в разделе «Практические методологии самодокументирования» на стр. 103). Трудно переоценить значение хорошего текстового процессора.

Документацию пишут для того, чтобы ее читали. Большое значение имеют сетевые системы подсказки (основанные на высококачественных текстах).

Управление проектами

Средства для менеджмента и коллективной работы позволяют писать отчеты и следить за соблюдением графика, контролировать ошибки и активность членов команды. Объем задач, охватываемых инструментом управления, может быть таким, что обычным программистам он и не понадобится. Но более экзотические системы могут оказаться в центре разработки проекта и потребовать участия всех сотрудников.

Мастерим сами

Как быть, если нужный инструмент найти не удается, а выполнение работы вручную займет немыслимо много времени? Ничего не остается, как «смастерить собственный» инструмент. В этом нет ничего страшного. Если видно, что данная задача будет возникать снова и снова, то лучше потратить некоторое время на создание инструмента для ее решения, что в конечном итоге только сбережет ваше время.

Для одних задач легче разработать инструменты, для других – труднее. Постарайтесь оценить заранее реальность выполнения задачи, за которую беретесь, и окупаемость затрат на нее.

Вот обычные способы, которыми можно создать инструмент:

- Соединить уже существующие инструменты неким новым способом – обычно с помощью конвейерной технологии UNIX, хотя может потребоваться написать некий сопрягающий код. Сложные командные строки можно заключить в *сценарий оболочки* (или *пакетный файл*, если речь идет о Windows), чтобы не вводить их каждый раз заново.
- Воспользоваться *языком сценариев*. Большая часть самодельных инструментов пишется на том или ином языке сценария, чаще на Perl. Работать с ними легко и просто, а возможностей достаточно, чтобы изготовить необходимый инструмент.
- Написать полновесную программу с чистого листа. Делать это стоит лишь тогда, когда желательно иметь солидный инструмент для многократного использования. В противном случае ваши усилия могут оказаться неоправданными.

При написании инструмента необходимо учитывать:

- Целевую аудиторию – насколько велика важность отделки? Приемлемо ли наличие некоторых шероховатостей? Если инструмент предназначен для вас самого и вашего коллеги-технаря, можно особенно не стараться. Если же он может понадобиться другим, более нежным личностям, стоит позаботиться о его благообразии.
- Возможность расширения уже существующего инструмента (сделать оболочку для его команды или создать для него дополнительный модуль?)

Резюме

Дайте нам нужные инструменты, и мы доведем дело до конца.

Сэр Уинстон Черчилль

Инструменты обеспечивают возможность создания программного обеспечения. Хорошие инструменты значительно облегчают эту задачу.

Поставьте перед собой задачу оценить тот комплект инструментов, которым вы пользуетесь. Умеете ли вы правильно применять каждый из них? Нет ли в вашем комплекте упущений, которые следует восполнить? В полной ли мере вы используете то, что есть?

Инструмент хорош только в руках мастера. *Плохому танцору все мешает*. Плохой программист напишет плохой код, каким бы инструментом при этом ни пользовался. Более того, инструменты могут способствовать созданию исключительно скверного кода. Воспитание в себе профессионального и ответственного отношения к используемому инструментарию позволит вам подняться на более высокую ступеньку качества программирования.

Хорошие программисты...

- Предпочитают *один раз* изучить возможности нужного инструмента, а не повторять *одну и ту же* нудную работу
- Разбираются в различных цепочках инструментов и уверенно чувствуют себя с каждой из них
- Облегчают себе жизнь с помощью инструментов, но не делаются их рабами
- Все, чем они пользуются, рассматривают как инструмент, допускающий замену
- Продуктивные, потому что применение инструментов – их вторая натура

Плохие программисты...

- Умеют пользоваться лишь несколькими инструментами и пытаются их применить в каждой задаче
- Не решаются потратить время на изучение нового инструмента
- Начав работать в какой-то определенной среде разработки, фанатически придерживаются ее, даже не пытаясь выяснить, какие есть альтернативы
- Не пополняют свой инструментарий, столкнувшись с ценным новым инструментом

См. также

Глава 10. Код, который построил Джек

Процесс сборки программного обеспечения осуществляется с помощью инструментов. Вы можете себе представить компиляцию кода вручную?

Глава 13. Важность проектирования

Содержит раздел, посвященный *инструментам проектирования*.

Глава 18. Защита исходного кода

Глава посвящена применению *средств управления версиями*.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 612.

Вопросы для размышления

1. Что важнее – всем членам команды разработчиков пользоваться одинаковой IDE или каждому выбрать для себя наиболее подходящую среду? Каковы последствия использования разных инструментов в одной команде?
2. Какой минимальный набор инструментов должен быть в распоряжении каждого программиста?
3. Какие инструменты мощнее – основанные на командной строке или использующие GUI?
4. Есть ли строительные инструменты, не являющиеся программами?
5. Какое качество важнее всего для инструмента?
   a. Универсальность интерфейса
   b. Гибкость
   c. Настраиваемость

d. Мощь
e. Легкость освоения и применения

Вопросы личного характера

1. Какие инструменты входят в ваш стандартный набор? Какими из них вы пользуетесь ежедневно? Какими пользуетесь несколько раз в неделю? К каким обращаетесь в редких случаях?
   a. Насколько уверенно вы пользуетесь ими?
   b. Насколько полно используете возможности каждого инструмента?
   c. Как вы учились работать с ними? Пытались ли когда-нибудь совершенствовать свои навыки работы с ними?
   d. Известны ли вам более удачные инструменты?
2. Насколько современны ваши инструменты? Имеет ли для вас значение, что их версии не самые новые?
3. Чему вы отдаете предпочтение – интегрированным наборам (типа визуальной среды разработки) или цепочкам отдельных инструментов? В чем преимущества *противоположного* подхода? Насколько велик ваш опыт работы в *том и другом* стиле?
4. Соглашаетесь ли вы с настройками своего редактора по умолчанию или стремитесь переделать в нем все на свой вкус? Какой подход «правильнее»?
5. Как вы решаете, сколько потратить на покупку инструментов программирования? Как вы определяете, стоит ли инструмент своих денег?

8

Время испытаний

Черная магия тестирования

В этой главе:

- Зачем тестировать?
- Кто отвечает за тестирование?
- Как правильно тестировать?
- Виды тестирования

Все испытывайте, хорошего держитесь.

Первое послание
к Фессалоникийцам 5:21

Можно написать сколько угодно кода и быть уверенным только в одном: с первого раза он не заработает идеально. Это не зависит от того, сколь долго и тщательно вы разрабатывали код; ошибки обладают мерзким свойством проникать в любую программу. Чем больше кода вы напишете, тем больше ошибок в нем окажется. Чем быстрее вы будете писать, тем больше будет ошибок. Я еще не встречал действительно плодовитого программиста, код которого хотя бы с натяжкой можно было считать свободным от ошибок.

Как с этим борются? Путем тестирования кода. Это делается, чтобы найти имеющиеся проблемы, а после их решения сохранять уверенность в коде по мере того, как мы его модифицируем. Выпускать программное обеспечение, не протестировав его, – просто

самоубийство, каким бы замечательным программистом вы себя ни считали. Программное обеспечение, не прошедшее тестирования, неизбежно приведет к сбоям; тестирование – важная составная часть нашего ремесла. Очень многие профессиональные производители ПО недооценивают важность тщательного тестирования или пытаются провести его в течение жалкого времени, остающегося перед поставкой продукта. И это дает о себе знать.

Тестирование нельзя рассматривать как процедуру, относящуюся к самому завершению процесса разработки и служащую подтверждением того, что вы выпустили стоящий продукт. Если вы так воспринимаете его, то всегда будете выпускать очень слабый код. Тестирование – центральный элемент создания программного продукта. Только путем тестирования, не дожидаясь окончания работ, вы можете убедиться, что каждый фрагмент кода работает. Как еще можно в этом убедиться? Почему многие производители ПО считают, что могут обойтись без достаточного тестирования?

Терминология

Термин *bug* (программная ошибка) чрезвычайно многозначен и крайне неточен. Нет ничего проще, чем произносить слова, не понимая их действительного смысла. Использование специальной терминологии помогает лучше определить свои действия. Следующие определения навеяны стандартами IEEE (IEEE 84):

Ошибка (error)

То, что вы сделали неправильно. Деятельность, приведшая к появлению *дефекта* в программном продукте. Например, если вы забыли проверить условие в коде (скажем, размер массива C, перед тем как обратиться к его элементу), то это ошибка.

Неисправность (fault)

Неисправность – это следствие ошибки, введенной в программном продукте. Я сделал ошибку, и это привело к неисправности в коде. Первоначально проблема *скрыта*. Если написанный мною код никогда не будет выполняться, то эта неисправность никогда не проявится. Если выполнение часто передается коду, в котором есть неисправность, но всегда в таких условиях, что неисправность не проявит себя, мы никогда не заметим ее наличия.

Это обстоятельство весьма затрудняет отладку программ. Неисправная строка кода может годами прекрасно работать, пока вдруг не вызовет совершенно необъяснимое раздражение системы; странно будет подозревать код, который безупречно работал годами.

Неисправность можно обнаружить при анализе кода, но если программа работает, вы ничего не заметите.

Отказ (failure)

При столкновении с неисправностью может возникнуть *отказ*. Но не обязательно. Отказ как проявление неисправности – это то, что нас должно действительно беспокоить. Возможно, это единственное, на что мы обратим внимание. Отказ – это отклонение работы программы от технических требований и предполагаемого поведения. Здесь мы вступаем в область философии. Если в лесу рухнуло дерево, услышим ли мы звук падения? Если в работающей программе ошибка ничем не проявилась, есть в ней неисправность или нет? Попробуем ответить на этот вопрос с помощью определений.

Ошибка, баг (bug)

Баг – разговорное слово, часто используемое как синоним для неисправности (fault). По преданию первым компьютерным багом был *настоящий* баг (жук). Его обнаружил адмирал Грэйс Хоппер в 1947 году в Гарварде. Мотылек, застрявший между двумя электрическими реле машины Mark II Aiken Relay Calculator, вызвал аварию и отключение всей машины.

Проверка на подлинность

Два простых вопроса – *Что такое тестирование?* и *Зачем нужно тестирование?* – кажутся до боли очевидными. И тем не менее очень часто тестирование осуществляется неправильно или на неправильно выбранном этапе разработки. Умелое тестирование – это искусство. Реальное *проведение* тестирования – это уровень, которого многие программисты не достигают; от одного упоминания о тестировании они покрываются холодным потом. «В тестировании главное правило – это провести его» (Kernighan Pike 99).

Тестирование – это особый вид деятельности, не совпадающий с отладкой, хотя границы между ними четко не определены и их часто смешивают. *Тестирование* – это методология доказательства наличия или отсутствия неисправностей в программном обеспечении. *Отладка* – это действия, направленные на выяснение причин такого неисправного поведения. Тестирование приводит к отладке, а отладка приводит к устранению неисправностей, что требует повторного тестирования (с целью доказать действенность исправлений).

Тестирование – это не отладка. Не путайте между собой эти виды работы. Каждый требует особых навыков. Следите за тем, чем вы заняты – тестированием или отладкой.

Если вы хорошо программируете, то тестирование будет отнимать у вас *гораздо* больше времени, чем отладка. Поэтому данная глава предшествует той, где речь идет об отладке.

На протяжении всего процесса разработки программного продукта тестирование имеет разные цели:

- Большое количество *документов* проходит через стадию тестирования (которую обычно называют процедурой *рецензирования*). Благодаря этому, например, гарантируется, что спецификация технических требований корректно отражает потребности клиента, функциональные спецификации отражают технические требования, спецификации различных подсистем достаточно полно отражают функциональные спецификации и т. д.
- Затем, естественно, *код* реализации тестируется на машине разработчика. Тестирование проходит несколько уровней: начинают с построчного тестирования каждой функции, затем переходят к тестированию отдельных модулей, а потом к интегрированным тестам объединенных секций кода.
- Наконец, тестируется конечный *продукт*. Хотя на этом уровне косвенно тестируются (или *должны* тестироваться) все разработанные программные компоненты, цель этих тестов иная. Здесь нас интересует, работает или нет программа как единое целое в соответствии со спецификациями.[1]

  Тесты продукта могут проверять целый ряд вещей. Прежде всего они проверяют, что система функционирует предполагаемым образом. Проверяется также корректность установки (если это коробочный продукт) и его готовность к работе.

  Такого рода тестирование проводит отдел контроля качества (QA). Его задача – разобраться, что должен делать продукт, и проверить, делает ли он это, а также проверить выполнение установленных для него критериев качества.

В данной главе мы сосредоточимся на центральной части – как должны тестировать свой код разработчики программного продукта. Прочие виды тестирования составляют отдельные крупные темы, освещение которых не входит в задачи этой книги.

Кто, что, когда, зачем?

Чтобы тестирование программного обеспечения было эффективным, нужно разобраться, для *чего* мы тестируем, *кто* этим занимается, *что* подразумевает эта процедура и *когда* можно считать ее законченной.

---

[1] Поскольку очевидно, что корректное поведение было заранее тщательно зафиксировано в спецификации – или не так?

Контроль качества

QA: *контроль качества*. Звучит тягостно, правда? Но *кто* эти люди или *что* это за работа? Так называют и некую группу персонала, и технологию разработки. Чтобы разобраться с QA, нужно отделить обыденную речь и заблуждения от правильного определения.

Иногда путают QA с тестированием, однако между ними есть существенная разница. Тестирование служит выявлению ошибочной работы программного продукта, противоречащей спецификации. Задача QA – профилактика. Это система мер, воздействующих на процедуры и практику разработки с целью получить в итоге программное обеспечение высокого качества. Тестирование – лишь малая составная часть QA, потому что качество программного продукта определяется не только низким содержанием в нем ошибок. Контроль качества означает своевременный выпуск продукции, соблюдение бюджетных расходов, выполнение всех требований и ожиданий (последние могут не совпадать). К сожалению, современная программная индустрия выпускает не так много продукции высокого качества.

Кто отвечает за качество программного обеспечения? Подразделение компании, проводящее тестирование (часто оно называется отделом QA), – это группа людей, занятых тестированием продукта. Их мнение о том, допустимо ли выпустить продукт с текущим уровнем качества, является решающим. Это существенный элемент в общей картине качества, но не единственный. Все, кто участвует в процессе разработки, так или иначе определяют качество конечного продукта; качество – не наклейка, которую можно прицепить к готовому коду.

Ответственность за контроль качества программного продукта часто лежит на тех же людях, которые выполняют его тестирование. В противном случае за общий контроль качества отвечают менеджеры проекта, а тестеры тестируют сами по себе.

Зачем тестировать

Нам, разработчикам программного обеспечения, процедура тестирования нужна по ряду причин: чтобы обнаружить неисправности и устранить их, а также гарантировать отсутствие тех же неисправностей в следующих версиях.

Учтите, что тестирование не может доказать отсутствие неисправностей – только их наличие. Если при тестировании вы не обнаружили ошибок, это не значит, что их нет; просто вы пока не сумели их найти.

Тестирование может вскрыть только наличие ошибок. Оно не может доказать отсутствие неисправностей. Не поддавайтесь ложному чувству спокойствия, если код прошел ряд неадекватных тестов.

Тестирование в конце цикла разработки продукта может быть обусловлено другими мотивами. Наряду с *проверкой* корректности программного продукта и отсутствия в нем неисправностей может потребоваться проверить его *годность*, т. е. соответствие первоначальным техническим требованиям, как доказательство его готовности к выпуску. Проверка годности (validation) – один из видов приемо-сдаточных испытаний.

Кому тестировать

Программист, написавший код, обязан сам его протестировать. Зарубите это себе на носу и каждое утро по 10 минут разглядывайте его в зеркало.

Есть много программистов, которые неправильно представляют себе систему испытаний, существующую в условиях промышленного производства программ, а потому поспешно лепят код и сдают его в службу контроля качества, не потрудившись предварительно протестировать его самостоятельно. Это безответственный и непрофессиональный подход. В конечном итоге он потребует *больше* времени и усилий, чем при правильном тестировании. Включать непротестированный код в готовый продукт – полная глупость, а передавать его в службу контроля качества – немногим лучше. Действительно, задача службы контроля качества – тестирование, но тестирование *всего* продукта, а не тех новых строчек, которые вы написали. Служба контроля качества может найти какие-нибудь глупые ошибки, которые вы пропустили и которые могут проявиться непонятным и неожиданным образом. Но ее задача – поиск более фундаментальных ошибок, а не тех, что могли быть обнаружены ранее и являются результатом неаккуратной работы программистов.

Тестируйте каждый написанный вами фрагмент кода. Не рассчитывайте, что кто-то другой сделает это за вас.

В чем состоит тестирование

Разрабатывая программный продукт, мы создаем отдельные функции, структуры данных, классы и собираем из них работающую систему. Основная стратегия тестирования состоит в том, чтобы опробовать весь этот код и проверить его работоспособность, написав некий дополнительный код – *тестовый*. Это определенная оснастка, с помощью которой тестируемый объект вертят и так и сяк, провоцируя на реакцию и убеждаясь, что реакция правильная.

Тестовый код пишут для каждого уровня системы, проверяя каждый важный класс и функцию вплоть до объединенных структур, образованных из этих мелких частей. Для каждого теста нужно ясно определить:

- Какой именно фрагмент кода тестируется. Хорошо, если есть ясные модули с четко обозначенными границами; тогда вашими точками тестирования являются интерфейсы. Нечеткие и сложные интерфейсы делают нечетким и сложным тестирование.
- Метод, применяемый для тестирования (см. раздел «Типы тестирования» на стр. 195).
- В какой момент остановиться. Это сложный и важный вопрос, потому что иначе процесс может оказаться бесконечным. Когда можно принять решение, что проведено достаточно тестов?

Другая распространенная стратегия тестирования – это *внимательное изучение* кода с целью определения его корректности. Поскольку это делает человек, возможны ошибки; кроме того, необходимо четко определить требования. Проверка кода – это стандартный тип экспертизы (см. главу 20). Полезно пользоваться для проверки кода специальным инструментарием, но он не может автоматически выполнить за вас всю работу. Очень часто такая экспертиза проводится от случая к случаю. Предпочтительнее программные тесты; у них много преимуществ, которые мы обсудим в этой главе. Самое эффективное – сочетать оба метода.

Когда тестировать

Тестируйте код *по мере написания*, стараясь обнаружить ошибки как можно раньше. На самых ранних стадиях ошибки легче исправимы, оказывают меньше всего влияния на работу других программистов и наименее опасны. Тщательное тестирование на ранних стадиях работы – самый эффективный способ повысить качество программного продукта.

Стоимость ошибки растет по мере того, как она продолжает существование в процессе разработки,[1] поэтому важно начать тестирование кода как можно раньше – во время (или до начала) серьезных разработок программного обеспечения. Метод *разработки, основанный на тестировании* и популяризируемый некоторыми умными программистами, пропагандирует тестирование в качестве центрального элемента составления программы; тестовый код пишется прежде кода, который нужно тестировать!

[1] См. раздел «Экономика ошибок» на стр. 216, где подробнее обсуждается стоимость ошибок.

Чтобы сделать тестирование эффективным, его нужно начинать заранее, когда выявляемые ошибки еще не могут принести большого вреда. Тестовый код можно писать раньше, чем рабочий!

Это существенный момент, который необходимо включить в технологию программирования. Для каждого разрабатываемого фрагмента кода *сразу* пишите тест. Или сначала пишите тест. Убедитесь в том, что ваш код работает, и тогда можете свободно двигаться дальше. Если вы тут же не напишете тестовый код, то в вашем тылу останется непроверенный и потенциально содержащий ошибки код. Это подрывает стабильность всей вашей кодовой наработки. В конечном счете вам придется иметь дело с отладчиком, а это отнимает много времени.

Откладывая написание теста на будущее, вы утрачиваете выгодное положение – к тому времени вы либо подзабудете, что должен делать ваш код, либо тестировать придется отдельный модуль кода. Эффективность окажется уже не столь высока. Растет также вероятность, что вы вообще забудете написать тест.

Такая стратегия тестирования влечет глубокие последствия: думая о том, чтобы написать некий новый код, вы должны одновременно решать, как его протестировать. Это в лучшую сторону отражается на проектировании кода; почему – мы рассмотрим в разделе «Архитектура и тестирование» на стр. 202.

Каждый раз, обнаружив ошибку, проскользнувшую сквозь преграду ваших тестов, вы должны добавить новый тест в свой комплект (предварительно поругав себя за то, что его не оказалось на месте вовремя). Новый тест должен подтвердить, что вы корректно исправили ошибку. Он также выявит возможное появление той же ошибки в будущем; ошибки имеют обыкновение воскресать из мертвых, что часто случается при последующих модификациях кода.

Пишите тесты для всех выявленных ошибок.

Итак, мы пишем тесты как можно раньше, но как часто следует их прогонять? Столь часто, сколько позволяют человеческие силы, и даже чаще (если воспользоваться поддержкой со стороны компьютера). Чем чаще мы прогоняем тесты, тем больше шансов обнаружить ошибки. Это воплощается в стратегии *непрерывной сборки* (см. раздел «Автоматическая сборка» на стр. 256) и иллюстрируется мощью программных тестов (которые легко повторять многократно).

Прогоняйте свои тесты как можно чаще.

Тестировать легко...

Легко, если делать это плохо, иначе это *тяжелый труд*. Однако это не бездумный процесс. Чтобы проверить работоспособность конкретного фрагмента кода, требуется оснастка, которая демонстрирует:

- Корректность выходных данных при всевозможных допустимых вариантах входных.
- Обработку отказа при вводе любых недопустимых данных.

Выглядит невинно, но полностью выполнить такую проверку реально лишь для простейших функций. Обычно набор *допустимых* входных значений слишком велик, чтобы каждое из них можно было проверить отдельно. Приходится выбирать небольшой репрезентативный набор входных значений. Множество *недопустимых* значений обычно *гораздо* больше множества допустимых, поэтому в качестве недопустимых приходится проверять тоже какой-то набор представителей.

Проиллюстрируем это на двух примерах. Первую функцию тестировать легко:

```
bool logical_not(bool b)
{
    if (b)
        return false;
    else
        return true;
}
```

Допустимых входных значений два, недопустимых – нет. Тогда простой тест для этой функции может выглядеть так:

```
void test_logical_not()
{
    assert(logical_not(true) == false);
    assert(logical_not(false) == true);
}
```

Однако эта функция не выполняет ничего интересного. Теперь рассмотрим функцию для поиска наибольшего общего делителя (оставив в стороне вопрос о ее элегантности). Насколько труднее ее протестировать?

```
int greatest_common_divisor(int a, int b)
{
    int low = min(a, b);
    int high = max(a, b);

    int gcd = 0;
    for (int div = low; div > 0; --div)
    {
        if ((low % div == 0) && (high % div == 0))
            if (gcd < div)
                gcd = div;
```

```
        }
        return gcd;
    }
```

Этот фрагмент кода тоже невелик, но протестировать его гораздо труднее:

- Параметров всего два, но множество допустимых значений очень велико. Не представляется возможным проверить все комбинации значений; это заняло бы *слишком* много времени.[1] Добавление в функцию новых параметров увеличивает сложность экспоненциально.
- В функции есть цикл. Все виды ветвления (и цикл `for`) увеличивают сложность и вероятность ошибки.
- Есть несколько условных операторов. Придется организовать испытание так, чтобы код прошел через каждый вариант сочетания условий, и проверить, что они работают.

И это все для одной маленькой функции. Кстати, если вы заметили, в ней *есть* ошибка. Можете ее обнаружить? В случае успеха вам десять очков и звездочка.[2]

Читая код, очень легко обмануться и поверить, что он работает правильно. Если вы написали код, то при его чтении вы будете видеть то, что собирались написать, а не то, что написали на самом деле. Учитесь читать код с циничным недоверием.

Эти три проблемы не исчерпывают причин сложности тестирования кода. Есть много других факторов, увеличивающих сложность тестирования.

Размер кода

Чем больше код, тем больше места для потенциальных ошибок и тем больше отдельных маршрутов выполнения нужно проследить, чтобы убедиться в правильности кода.

Зависимости

Протестировать маленький фрагмент кода должно быть просто. Но если для своей работы оснастка тестирования должна подключить остальной код проекта, написание любых тестов отнимет слишком много труда и времени. В таком случае вы либо вообще не тестируете, либо тесты оказываются ограниченными, поскольку управлять

---

[1] Чем больше входные значения, тем дольше будет выполняться цикл. Предположив, что `int` имеет разрядность 32 бита (т. е. 2^{64} входных комбинаций) и есть прекрасная быстрая машина (допустим, что вызов функции занимает миллисекунду – чертовски мощный процессор!), обычный перебор займет около 600 миллионов лет! И это если не выводить результаты тестирования...

[2] О ней сказано в ответе на первый вопрос раздела «Вопросы для размышления» этой главы (стр. 615).

всеми подключаемыми компонентами кода слишком трудно. Это пример того, как *проект кода делает его тестирование невозможным*. Ниже мы изучим способы борьбы с этим (см. раздел «Архитектура и тестирование» на стр. 202).

Следующие два примера тоже демонстрируют случаи внутренних зависимостей кода.

Внешние исходные данные

Всякая зависимость состояния внешней части системы, в сущности, представляет собой еще один источник внешних данных. В отличие от параметров функции, не так легко заставить эти внешние источники подавать значения, необходимые для тестирования. Общей глобальной переменной нельзя присвоить произвольное значение без того, чтобы это повлияло на другие части работающей программы.

Внешние побудительные причины

Код может реагировать не только на вызовы функций. Особенно обременительно, если эти факторы возникают асинхронно (в любой момент) и с произвольной частотой.

- Класс может обрабатывать обращения из других частей системы, возникающие в произвольный момент.
- Аппаратные интерфейсы могут реагировать на изменения в физическом состоянии устройства.
- Обмен данными с другими системами может продолжаться любое время. Физические соединения подвержены воздействию помех, поэтому их качество может снижаться, а сетевые соединения бывают ненадежными.
- Код интерфейса пользователя воспринимает движения мыши. Создание автоматизированных условий физического тестирования GUI может вызвать затруднения.

Такие условия тяжело моделировать в искусственной среде тестирования, и они могут быть особенно чувствительны к временным параметрам (например, скорости двойных щелчков мыши или аппаратно генерируемым прерываниям).

Некоторые внешние факторы оказываются незапланированными: недостаток оперативной или дисковой памяти, отказ сетевых соединений. Необходимо гарантировать устойчивость работы своего кода в *наиболее* распространенных условиях окружающей среды.

Потоки

Многопоточность еще более осложняет тестирование, поскольку вмешательство кода, выполняемого одновременно, может происходить в произвольные моменты. Сложное взаимодействие последовательностей выполнения может привести к тому, что любые конкретные прогоны тестов оказываются невоспроизводимыми. Ошибки выполнения потоков, приводящие к блокировкам и недоступности

ресурсов, трудно смоделировать, но при возникновении они вызывают серьезные проблемы.

Поведение многопоточной программы на мультипроцессорных системах может отличаться от поведения однопроцессорных систем, моделирующих параллельные вычисления.

Эволюция

Программное обеспечение развивается, в результате чего тестирование может оказаться недействительным. Если технические требования точно не закреплены, ваши ранние тесты могут оказаться несостоятельными ко времени выпуска продукта, когда изменятся API, функциональность окажется совершенно иной, а полного набора тестов не будет, потому что разработка двигалась слишком быстро.

Нам необходимы стабильные интерфейсы как в нашем собственном коде, так и во внешнем коде, с которым мы связаны. В реальности это недостижимый идеал, поскольку код никогда не стоит на месте. Поэтому нужно строить небольшие гибкие тесты, которые можно модифицировать параллельно с кодом.

Аппаратные сбои

Ошибки бывают как в программах, так и в аппаратуре. Аппаратные ошибки чаще встречаются во встроенных средах, потому что в них вы ближе к железу. Диагностирование и исправление аппаратных ошибок может оказаться гораздо сложнее, чем программных: воспроизводить их трудно, а первое подозрение естественно падает на программы.

Особые случаи сбоев

Отказ кода может проявляться различными неожиданными и странными способами. Сбой программы не просто приводит к получению *неверных результатов* – приходится сталкиваться со многими вещами: бесконечными циклами, блокировками, недоступностью ресурсов, аварийным завершением, блокировками ОС и прочими мерзкими вещами, делающими тестирование разнообразным и волнующим занятием. Определенные программные ошибки могут даже вывести из строя оборудование![1] Пишите тесты, которые это проверяют.

Написание кода, осуществляющего тестирование, – нелегкая задача. По мере соединения компонент друг с другом и появления зависимостей между ними сложность программного продукта растет с экспоненциальной скоростью. Все эти проблемы сливаются и сильно осложняют вам жизнь. В какой-то момент написать структуру для *всеобъемлющего* тестирования программного продукта становится не только сложно, но и технически нереально. У вас не будет столько времени и ресурсов,

[1] Это не смешно. У процессора 68000 была недокументированная функция *останова и возгорания* – операция тестирования шины, которая быстро перебирала адресные линии, вызывая перегрев и возгорание микросхемы.

чтобы сгенерировать все необходимые тестовые данные и прогнать программу со всеми наборами входных данных и внешних условий. Метод полного перебора быстро становится неосуществимым, и возникает соблазн отказаться от тестирования в надежде, что ошибок не окажется.

Как бы тщательно вы ни проводили тестирование, создать программный продукт, лишенный ошибок, вам не удастся – для написания тестового кода нужно потратить столько же сил и изобретательности, сколько для обычного кода. Некоторые ошибки неизменно проскальзывают сквозь самые строгие процедуры тестирования (исследования показывают, что наиболее тщательно протестированные продукты содержат от 0.5 до 3 ошибок на 1000 строк кода) (Myers 86). На практике тестирование гарантирует не 100-процентную надежность программ, а лишь достаточную их адекватность решаемым задачам.

С учетом сказанного эффективнее всего будет сосредоточить свое внимание на ключевых тестах, рассчитывая с их помощью отловить большую часть программных ошибок. Ниже мы рассмотрим вопрос их выбора.

Типы тестирования

Есть много разновидностей программных тестов, и ни один из них не обладает превосходством над остальными. В каждом методе код рассматривается с определенной стороны и выполняется поиск определенного класса ошибок. Все тесты необходимы.

Блочное тестирование

Под *блочным тестированием* обычно понимают тестирование *модуля* кода (например, библиотеки, драйвера устройства или стека протоколов), но в реальности это тестирование неделимых единиц, таких как классы или функции.

Блочное тестирование проводится строго изолированно. Любой ненадежный внешний код, с которым взаимодействует блок, заменяется заглушкой, или имитатором; это обеспечивает поиск ошибок только в данном блоке, а не тех, которые обусловлены внешним влиянием.

Тестирование компонент

На следующем шаге – тестировании компонент – проверяется работоспособность одного или нескольких блоков в составе компоненты. Часто именно это подразумевают под *блочным тестированием*.

Комплексное тестирование

Этот тест проверяет комбинируемость компонент, подключаемых к системе, гарантируя их правильное взаимодействие.

Регрессивное тестирование

Это повторное тестирование, проводимое после исправлений или модификации программного обеспечения или его окружения. Регрессивные тесты проверяют, что программное обеспечение работает

так же, как прежде, и внесенные модификации не нарушили его функций. Если программное обеспечение ненадежно, изменения в одном месте могут привести к непонятным сбоям в другом. Регрессивное тестирование призвано бороться с этим.

Определить объем повторного тестирования бывает нелегко, особенно когда цикл разработки близится к концу. Для такого тестирования особенно полезны автоматизированные средства. Подробнее об этом будет сказано в разделе «Руками не трогать!» на стр. 203.

Испытание под нагрузкой

Испытания под нагрузкой проводятся с целью определить, выдержит ли ваш код объем данных, который, предположительно, может на него обрушиться. Легко написать код, который дает правильный ответ, но совсем другое дело, когда этот ответ требуется получить вовремя. Здесь могут вскрыться проблемы, связанные с производительностью системы, такие как неверный выбор размеров буферов, неэффективное использование памяти или неудачная схема базы данных. Испытание под нагрузкой проверяет, что программа должным образом «масштабируется».

Испытание пиковыми нагрузками

Испытание пиковыми нагрузками обрушивает на код *огромное* количество данных в течение короткого промежутка времени, чтобы посмотреть, как он с этим справится. Оно аналогично испытанию под нагрузкой и часто используется для интенсивно используемых систем. Испытание пиковымих нагрузками проверяет характеристики системы: насколько она выносliва к перегрузкам. Испытание под нагрузкой проверяет, может ли код справляться с *расчетными* нагрузками; испытание пиковыми нагрузками проверяет, не загнется ли код, если ему действительно хорошо достанется. При этом не предполагается, что код будет работать безупречно; лишь бы он прилично отработал отказ и смог восстановить нормальную работу.

Испытание пиковыми нагрузками позволяет определить пропускную способность системы – до какой степени можно ее прижать, прежде чем она сдастся. Особенно уместно такое испытание в многопоточных системах и системах реального времени.

Комплексное испытание под нагрузкой (прогон)

Прогон похож на испытание пиковыми нагрузками. Цель – заставить систему работать с высокой загрузкой в течение продолжительного времени – нескольких дней, недель или даже месяцев – и узнать, не отразится ли как-нибудь выполнение большого количества операций на функционировании системы. Прогон позволяет обнаружить проблемы, которые нельзя выявить другими способами: мелкие утечки памяти, приводящие в итоге к аварийному завершению, или снижение производительности за счет постепенной фрагментации внутренних структур данных.

Альфа, бета, гамма...

А как насчет альфа- и бета-тестирования? Эти термины встречаются часто, но относятся к несколько другому кругу проблем, чем рассмотренные выше. Их задачей, скорее, является тестирование окончательного продукта, а не отдельных участков *кода*. Тем не менее стоит остановиться на этих понятиях.

К счастью, формальных определений у них нет. У каждой компании существуют свои представления о том, что означают альфа- или бета-состояния ее программного продукта. Но обычно программы стадии «альфа» прочны, как лимонное желе, и взрываются при попадании на них света. Альфа- и бета-программы часто выпускают за пределы территории разработки для предварительной оценки клиентами, чтобы получить первоначальные отзывы и хоть какую-то уверенность в правильности направления.

Чаще всего термины интерпретируют так:

Альфа-версия

Первая стадия «законченного кода». Может иметь бесчисленное количество ошибок и быть совершенно ненадежной. Альфа-версия дает хорошее представление о том, каким будет конечный продукт, если не обращать внимания на очевидные недочеты.

Бета-версия

Она значительно превосходит альфа-версию, будучи почти свободной от ошибок и сохраняя нерешенными очень немногие проблемы. Она уже близка к конечному продукту. Бета-тестирование (т. е. тестирование *бета*-версии) служит для того, чтобы решить оставшиеся проблемы в кандидатах на окончательный выпуск. Обычно в бета-тестирование входят реальные полевые испытания.

Кандидат на выпуск

Это последняя стадия перед формальным выпуском программного продукта. Кандидаты на выпуск проходят верификацию и *тестирование качества* (validation), перед тем как пойти в производство. Кандидаты на выпуск – это внутренние сборки, обычно проверяемые только отделом контроля качества.

Альфа- и бета-версии, выпускаемые для публики, обычно имеют некоторые механизмы защиты (например, ограничение времени функционирования). Кандидаты на выпуск – это «чистые» сборки без такого рода ограничений.

Тестирование юзабилити

Гарантирует, что вашей программой сможет пользоваться даже тупой. Есть разные виды тестирования конечными пользователями, часто выполняемые в *особых лабораториях*, по специальным сценариям и с жестким контролем. Кроме того, используются *полевые испытания*, при которых можно узнать мнение пользователей о работе программы в реальных условиях.

При написании тестов для блоков и компонент есть два главных подхода к разработке контрольных примеров: тестирование методом *черного ящика* и тестирование методом *белого ящика*.

Метод черного ящика

Известен также как *функциональное тестирование*. В этом методе фактическая функциональность сравнивается с заявленной. Тестирующий не знает внутреннего устройства кода; он рассматривается как *черный ящик*. Разработчик и тестирующий могут быть никак не связаны друг с другом.[1]

Это метод ставит целью не проверку работы каждой строчки кода, а соответствие кода спецификации программы – что при подаче на вход черного ящика допустимых данных на его выходе будет получен нужный результат. Поэтому без ясных спецификаций и документированных API разрабатывать тесты черного ящика очень сложно.

Контрольные примеры для черного ящика можно начать разрабатывать, как только будет готова спецификация. Они существенно зависят от корректности спецификаций и незначительности изменений, вносимых в них после разработки тестов.

Метод белого ящика

Известен также как *структурное тестирование*. Это подход на базе изучения кода. Каждая строка кода подвергается систематическому исследованию с целью определения ее корректности. Если в черный ящик вы заглянуть не могли, то в этот – можете и должны. Поэтому тестирование методом белого ящика иногда называют тестированием методом *стеклянного ящика*. Его главная цель – протестировать написанные строки кода, а не проверять их соответствие спецификациям.

Существуют *статические* и *динамические* способы тестирования методом белого ящика. В статических тестах код не выполняется; он просматривается и проходится пошагово с целью определить, правильно ли он решает задачу. Динамические тесты выполняют код и нацелены на тестирование путей и ответвлений в попытке попасть на все строки кода и выполнить каждое решение. При этом может потребоваться несколько изменить код, чтобы принудитель-

[1] Однако не всегда эта мысль удачна; обычно программист лучше всего справится с написанием теста для кода, который сам разработал.

но пройти некоторыми путями. Осуществить такую модификацию бывает проще, чем сконструировать контрольный пример для всех возможных комбинаций обстоятельств.[1]

Тестирование методом белого ящика трудоемко и значительно дороже по сравнению с черным ящиком, а потому и прибегают к нему гораздо реже. Даже для планирования тестирования методом белого ящика нужно иметь завершенный код. Обычно вначале тестируют методом черного ящика и лишь затем – белого. Последствия ошибки, найденной на этой стадии, обходятся намного дороже. Приходится исправлять код, проводить тест черного ящика, а затем разрабатывать и проводить новые тесты белого ящика.

Существует инструментарий для модификации кода и замера результатов тестирования. Без такого инструментария метод белого ящика был бы вообще немыслим.

Время тестировать

Каждый из этих методов тестирования применяется на разных этапах процесса разработки. Это иллюстрируется таблицей, в которой отмечено, какие тесты наиболее существенны в каждый момент.

Стадия разработки	Черный или белый ящик?	Стандартные методы тестирования на данной стадии разработки	Кто проводит тестирование
Определение технических требований	Черный	Разработка тестов черного ящика	Разработчики, QA
Проектирование кода	Черный	Разработка тестов белого ящика	Разработчики, QA
Написание кода	Черный, белый	Блочный, компонентный, регрессивный	Разработчики
Интеграция кода	Черный, белый	Компонентный, комплексный, регрессивный	Разработчики
Альфа-стадия	Черный, белый	Регрессивный, нагрузки, пиковый, прогон, юзабилити	Разработчики, QA
Бета-стадия	Черный, белый	Регрессивный, нагрузки, пиковый, прогон, юзабилити	QA
Кандидат на выпуск	Черный, белый	Регрессивный, нагрузки, пиковый, прогон	QA
Выпуск	Черный, белый	Слишком поздно...	Пользователи (удачи им)

[1] Если вы модифицируете исходный код, то фактически тестируете неокончательный вариант, что вызывает некоторую тревогу.

Метод черного ящика проверяет, *не упущено ли* что-нибудь (какие-либо функции, предполагавшиеся в программном продукте), тогда как метод белого ящика обнаруживает ошибки в *реализации*. Полноценное тестирование программного блока требует применения обоих методов.

Выбор контрольных примеров для блочного тестирования

Если тестировать нужно, но полное тестирование невозможно, следует разумно выбрать комплект наиболее эффективных тестов. Для этого вам понадобится продуманный, методичный план. Можно выбрать подход *стрельбы дробью* – повесьте код на стену, а потом стреляйте по нему из всего, что попадется под руку...

Какие-то ошибки вы таким способом *можете* найти. Но в отсутствие продуманного пошагового подхода к тестированию вы никогда не сможете получить качественные тесты, которые обеспечат вас надлежащей уверенностью в своем коде. Вместо стрельбы дробью лучше выбрать винтовку с точным прицелом и тщательно выбирать объекты в коде, стреляя по выбранным отметкам и отмечая, насколько устойчивым оказывается код.

Куда стоит целиться? Куда направить град тестовых данных? Все допустимые значения вы все равно не сможете проверить, значит, нужно отобрать горстку самых подходящих. Необходимо выбрать тесты, которые вероятнее всего вскроют ошибки в программе, а не те, которые всякий раз указывают на одни и те же несколько проблем.

Напишите полный набор тестов, каждый из которых будет проверять определенный аспект кода. Пятнадцать тестов, которые демонстрируют одну и ту же ошибку, менее полезны, чем 15 тестов, демонстрирующих 15 разных ошибок.

Для этого вам нужно понимать, какие требования предъявляются к вашему фрагменту кода. Нельзя написать точный тест, если неясно, что

должен делать этот код. Он может очень хорошо работать – делая не то, что нужно.

При тестировании методом черного ящика можно предложить такие проверки:

Хорошие входные данные

Тщательно отберите *хорошие* входные данные, чтобы убедиться в правильной работе программного обеспечения в нормальных условиях.

Включите весь диапазон допустимых входных значений: некоторые средние значения, некоторые значения, близкие к нижней границе диапазона допустимых значений, и некоторые – близкие к его верхней границе.

Плохие входные данные

Столь же большое значение имеет ряд правильно подобранных *плохих* входных данных. Вы проверяете устойчивость приложения и отсутствие ложных ответов на ввод недопустимых данных.

Нужно рассмотреть все виды нехороших данных, в том числе:

- Слишком большие или слишком маленькие числовые значения (часто забывают об отрицательных значениях).
- Слишком длинные или короткие входные данные (классическим примером служит длина строки; посмотрите, как код реагирует на пустую строку, или проверьте массивы и списки разных размеров).
- Внутренне несовместимые значения данных (что это означает, зависит от контракта функции; может быть, она ожидает значения в определенном порядке).

Граничные значения

Проверьте *все* граничные случаи – они часто служат источником ошибок. Определите наибольшее и наименьшее допустимые значения или естественные границы ввода (например, те, которые изменяют характер поведения). В каждом из этих случаев проверьте поведение кода:

- На самом граничном значении
- На значениях, превышающих границу
- На значениях, несколько меньших границы

Тем самым вы убедитесь, что ваша программа работает правильно в крайних точках и отказывается работать там, где это и предполагается.

Граничные тесты обнаруживают частые ошибки, например ввод > вместо >= или неправильный выбор начального значения переменной цикла (отсчитывать с нуля или с единицы?). Для обнаружения такого рода ошибок нужны все три граничных теста.

Случайные данные

Не думайте долго – генерируйте случайные наборы входных данных. Это очень эффективная стратегия тестирования. Если вы сумеете написать инфраструктуру тестирования, в которой многократно автоматически генерируются и применяются случайные входные данные, это усилит ваши возможности выявления скрытых ошибок, о которых вы не догадались бы иначе.

Нуль

Если входные данные числовые, всегда проверяйте поведение при вводе нуля. По каким-то причинам программисты часто неправильно обрабатывают нули, которые выпадают из их рассуждений.

Указатели C/C++ часто принимают нулевые значения, когда значение *не определено* или *не задано*. Проверьте, корректно ли воспринимает ваш код поступающие на вход нулевые значения указателей. В Java можно с той же целью подавать ссылки на нулевой объект.

Архитектура и тестирование

Качество написанных вами блочных тестов в значительной степени зависит от качества интерфейса, который необходимо протестировать. Тестирование проще проводить, если вы пишете код именно в расчете на верификацию и контроль качества. Это достигается с помощью понятных API, уменьшения зависимости от других участков кода и недопущения жестких ссылок на другие компоненты. В результате оказывается проще поместить компоненту в тестовую среду и смоделировать ее. Если же, напротив, она тесно связана с другими секциями кода, вам придется переносить эти секции в среду тестирования и организовывать надлежащее взаимодействие секций с вашим блоком. Иногда это не только трудно, но просто невозможно, что ограничивает ваши возможности тестирования.

Архитектура кода должна облегчать его тестирование.

У этого правила есть полезный побочный эффект. Если структура кода предусматривает легкость его тестирования, она оказывается разумной, понятной и облегчающей сопровождение. Сокращается взаимодействие компонент и усиливается их связность. Структура становится более гибкой, легкой и облегчает включение в различные конфигурации. Код становится лучше.

Если код прошел тщательное тестирование, более вероятно, что он корректен.

Необходимо заранее проектировать код в расчете на тестирование. Не так просто взять старую компоненту и прикрепить к ней «тестируемый» интерфейс. Если есть масса прочего кода, зависящего от сущест-

вующего интерфейса, то такие модификации трудноосуществимы. Запомните: лучшая возможность написать действительно допускающий тестирование код – это писать тестовые блоки параллельно с кодом.

Есть ряд простых правил, соблюдение которых приводит к созданию кода, который действительно легко тестировать:

- Делайте каждый раздел кода независимым, избегая недокументированных и неясных зависимостей от внешней среды. Не употребляйте жестких ссылок на другие части системы; предпочитайте абстрактные интерфейсы, которые можно реализовать системными компонентами или тестовыми эмуляторами.
- Не пользуйтесь глобальными переменными (или объектами *singleton*, лишь прикрывающими глобальные переменные). Собирайте такие состояния в общие структуры, передаваемые в качестве аргументов.
- Не усложняйте свой код; разбивайте его на небольшие, доступные пониманию, ограниченные фрагменты, которые можно протестировать независимо.
- Обеспечьте доступность кода для изучения, чтобы можно было понять, чем он занят, выяснить его внутреннее состояние и проверить, что он работает в соответствии с расчетами.

Руками не трогать!

Невозможно заниматься целый день только управлением вручную всеми механизмами тестирования. Запуск вручную одного теста за другим плохо согласуется с моим представлением об эффективной работе программиста. Многократное повторение регрессивных тестов быстро наскучивает. И это не просто скучно, но медленно, неэффективно и подвержено ошибкам. Золотое правило тестирования: *автоматизируй*.

Насколько это возможно, автоматизируйте тестирование кода. Это быстрее и проще, чем выполнять тесты вручную, и гораздо надежнее: более вероятно, что тесты станут выполняться регулярно.

Если тесты не требуют вмешательства оператора, их можно запускать на фазе проверки процедуры сборки. В начале работы с только что собранной программой можно быть уверенным, что тесты блоков успешно проведены; гарантируется отсутствие нелепых ошибок программирования и сохранение работоспособности прежнего кода при сделанных нововведениях.

Выполняйте тестирование автоматически в ходе процедуры сборки.

Можно собрать отдельные фрагменты тестирующего кода и организовать их в виде единой автоматизированной процедуры, управляющей

тестированием и помещающей его результаты в одном месте. Такая структура может контролировать, какие тесты проведены, а в более сложном варианте вести журнал всех проводившихся тестов. Существуют популярные средства для таких целей, например JUnit – стандартная система тестирования для Java.

Высокий уровень автоматизации приносит выгоды при регрессивном тестировании. Если вы модифицировали старый код и хотите убедиться, что при этом ничего случайно не повредили, можно автоматически прогнать весь набор тестов; в конце вы получите *положительный* или *отрицательный* ответ. Разумеется, ценность результата регрессивного тестирования определяется качеством тестов, в нем участвующих.

Автоматизация – это действительно основа надежной разработки кода. Если в данный момент у вас отсутствует автоматизированный набор тестов, выступающий в качестве непрерывного регрессивного теста вашего кода, то создайте его. Качество вашей работы быстро повысится.

К сожалению, не все тесты *можно* сделать автоматическими. Блочное тестирование библиотечных функций осуществляется относительно легко, а вот автоматически тестировать интерфейс пользователя очень трудно. Как вы станете моделировать щелчки мыши, проверять перевод строки на язык урду или воспроизведение нужного клипа?

Анатомия провала

Проблема не в том, чтобы никогда не падать,
а в том, чтобы каждый раз подниматься.

Конфуций

Как вы поступите, если тестирование обнаружит ошибку в программе? Прежде чем сломя голову бросаться отлаживать ее, задержитесь и оцените проблему. Это особенно важно, если вы не хотите (или не можете) исправить ошибку сразу. Чтобы точно определить суть проблемы, что позволит вам или другому разработчику в будущем вернуться и решить ее, действуйте следующим образом:

1. Запишите, что вы пытались сделать в тот момент и какие действия вызвали сбой.
2. Попробуйте их повторить. Выясните, можно ли воспроизвести проблему, как часто она возникает и не выполняются ли в то же время какие-то другие действия.
3. Опишите сбой. Полностью. Будьте предельно конкретны. Включите следующее:
   - контекст проблемы;
   - простейший способ ее воспроизведения;
   - сведения о повторяемости и частоте возникновения;

- версию программного обеспечения, точный номер сборки и использовавшееся оборудование;
- все остальное, что предположительно могло бы иметь отношение.

4. Запишите все и не теряйте! Занесите данные в систему контроля ошибок, даже если это простая ошибка в коде, которую вы собираетесь исправить сами (см. ниже раздел «Справлюсь ли я сам?»)
5. Напишите простейший контрольный пример, демонстрирующий сбой, и добавьте его в набор автоматически выполняемых тестов. Этим вы гарантируете, что ошибка не будет потеряна или забыта, а когда она будет исправлена, но не повторится в новых разработках.

Запомните: тестирование – это *не* отладка, и данные действия – *не* отладка! Вы же не пытаетесь вскрыть причину сбоя, не изучаете код, а просто регистрируете сведения, которых достаточно для описания проблемы другому разработчику.

Самый лучший сбой – повторяющийся. В самом деле хорошо, когда код многократно дает сбой: тогда легко заново вызвать проблему, проследить ее источник и убедиться, что ошибка исправлена. Плохи ошибки, возникающие нерегулярно, даже случайно, поэтому их трудно описать. Самый кошмар, когда ошибки проявляются раз в тысячу лет, да еще в зависимости от того, в какую сторону дует ветер.

Справлюсь ли я сам?

Для *поиска* ошибок требуются методичность и систематичность. Методичность и систематичность нужны также для учета и управления устранением ошибок. До того как код выпущен или попал в систему контроля версий, единственный, кто испытывает неприятности из-за этой ошибки, – вы сами. Но как только вы выпустили код на свободу, он начинает жить своей жизнью. И ошибки в нем касаются уже не только вас. По мере вступления в игру новых участников правила меняются:

- Программист находит проблемы, глядя на код – свой собственный или чужой.
- Интегратор кода обнаруживает ошибки в ходе соединения отдельных компонент.
- Отдел QA обнаруживает ошибки во время тестирования.

Когда много людей обнаруживает ошибки и при этом кто-то пытается одновременно их исправить, необходима правильная процедура администрирования всех этих действий. Иначе возникнет неразбериха, и разработка станет общим кошмаром.

Система контроля ошибок

Наше главное оружие в борьбе с ошибками – *система контроля ошибок*. Это особая база данных, интерфейс которой доступен всем, кто принимает какое-то участие в процессе тестирования.

По ходу обнаружения ошибок и действий по их устранению в эту базу вносятся данные, отражающие текущее состояние программного продукта. Таким образом, этот инструмент становится составной частью *технологии борьбы с ошибками* в проекте. Вот основные действия, выполняемые системой:

Отчет об ошибке

Обнаружив ошибку, создайте для нее запись в базе данных в виде *отчета об ошибке*. Он становится полноправным членом клуба ошибок с личным членским номером. В дальнейшем данный номер уникальным образом идентифицирует ошибку. Теперь пропустить ошибку нельзя. С ней *нужно* что-то сделать, прежде чем выпускать продукт.

Создание отчета также предупреждает других участников о том, что найдена ошибка; столкнувшись с ней, им не придется вносить снова ту же информацию.

Назначение ответственного

В отчете об ошибке указывается конкретное лицо. Оно становится ответственным за устранение ошибки (самостоятельно или назначив для этого кого-то другого). Если не назначить «хозяина» ошибки, каждый программист будет считать, что ее исправит кто-то другой, и ошибка будет благополучно существовать дальше.

Установление приоритетов

Система контроля ошибок позволяет также помечать, какие ошибки наиболее существенны. Очевидно, что частое аварийное завершение при запуске гораздо серьезнее, чем эпизодическое смещение кнопочки в сторону на один пиксел.

Определив, какие ошибки являются критическими, а какие – мелкими неудобствами, разработчики могут планировать свою работу, выбирая ошибки, которые требуют устранения в первую очередь. Система может поддерживать несколько уровней тяжести ошибки: критически важные, средней или низкой очередности или запрос дополнительных функций.

Пометка проблемы как решенной

Разработчик делает такую пометку после внесения исправлений. Тем самым отчет об ошибке не закрывается, но помещается в группу подлежащих проверке. Проверить, что ошибка исправлена, должен тот, кто создал отчет, хотя он может возложить эту задачу на кого-то другого. В любом случае – по очевидным причинам – тестирование не должен проводить тот, кто сделал исправление.

Закрытие отчета

После проверки отчет можно закрыть, оставив его лишь для истории (или статистики проекта).

Есть разные ситуации, в которых отчет может быть закрыт; например, если выяснится, что это не ошибка, а свойство системы

и, возможно, совершенно допустимое поведение. Тестеры тоже могут ошибаться.

Можно не закрывать отчет, с которым вы не намереваетесь работать, а *отложить* его с пометкой, что ошибка будет исправлена в следующей версии.

Запрос к базе данных

У системы контроля ошибок можно запрашивать различную информацию:

- Разумеется, можно сделать список всех незакрытых отчетов, упорядочив его по версии программного продукта, ответственному исполнителю, приоритету и пр.
- Можно выяснить, какие ошибки назначены для исправления вам.
- Можно создать отчет по ошибкам, исправленным в каждой версии программного продукта. Это полезно при подготовке *сопроводительного текста к новому выпуску*.
- Можно посмотреть статистику по проекту: сколько отчетов об ошибках поступило во время разработки, сколько ошибок исправлено и каково соотношение между скоростью обнаружения и исправления. Статистика в графическом виде позволит получить хорошее представление о состоянии процесса разработки.

Модификация записи

Можно открыть отчет и изменить содержащиеся в нем данные, в том числе:

- Добавить комментарии в связи с обнаружением новых данных.
- Прикрепить файлы журналов с данными вывода, иллюстрирующими проблему.
- Пометить, что данный отчет является дубликатом для некой ошибки, чтобы избежать путаницы в дальнейшем.

Существует масса пакетов контроля за ошибками – как коммерческих, так и бесплатных, например популярная система Bugzilla, входящая в проект Mozilla.

Обсуждение ошибок

По мере приближения конца сроков разработки совещания по поводу *оставшихся ошибок* (*bug review*) становятся нормой жизни и происходят практически еженедельно. Эти обзоры проводятся, когда функции программы реализованы, но ошибки не устранены – затянувшаяся финишная прямая процесса разработки. На этих совещаниях все заинтересованные стороны могут узнать о ходе проекта, планируются оставшиеся ремонтные работы и весь программный проект направляется к окончательному выпуску.

Состав участников довольно пестрый:

- Программисты, участвующие в проекте. (В конце концов, им придется делать исправления.)
- Представители группы тестирования, которые могут рассказать о контексте, в котором возникают ошибки, направят совещание в нужное русло. (Чаще всего в их обязанности и входит созыв данного совещания.)
- Менеджеры продуктов, которые получают представление о состоянии дел и принимают решения *на месте*.
- Сотрудники коммерческих и маркетинговых подразделений, которым придется продавать этот продукт с дефектами. (Их оценка вредности той или иной ошибки определяет, что нужно исправить обязательно, а что можно втихаря оставить.)

Во время совещания поочередно обсуждают каждую из незакрытых ошибок, список которых получают из системы контроля ошибок. Кто-нибудь из тестирующих или разработчиков при необходимости может представить дополнительную информацию, а затем принимаются коммерческие решения по поводу важности проблемы. Обсуждаются неприятные ошибки, исправление которых затягивается, с отчетом о ходе работы над ними. В случае затруднений могут быть выделены дополнительные ресурсы.

При таком разнообразии участников совещание может легко свернуть в сторону, поэтому его должен вести председатель с крепкими нервами, который не даст отклониться от темы. А тема – это отчеты об ошибках и что с ними делать, а не конкретные поправки в коде. Программисты любят обсуждать технические вопросы и пытаются решить во время совещания все проблемы. Это должно пресекаться.[1]

Резюме

Тестирование – необходимый элемент для создания хорошего программного продукта. В целом, чем больше тестирования, тем лучше – хотя на качестве конечного продукта отражается и *качество* тестов. Плохие тесты обнаружат мало ошибок, и в результате вы выпустите продукт с дефектами.

Тестирование осуществляется на разных уровнях разработки, начиная с отдельных функций, интеграции компонент и кончая полностью собранным продуктом. На каждом этапе должен существовать методический подход к поиску и управлению ошибками в программе.

Каждый программист обязан протестировать свой код. У отдела QA есть достаточно своих проблем, и он не должен разбираться с вашим неряш-

[1] Тактика успешного ведения совещаний описывается на стр. 435 в разделе «Это судьба».

ливым кодом. Нельзя начинать заниматься тестированием и проблемами качества, когда разработка подходит к концу – это нужно делать с самого начала, разрабатывая и прогоняя тесты одновременно с кодом.

Хорошие программисты...

- Пишут тесты для всего своего кода (возможно, даже *прежде* самого кода)
- Тестируют на *микроуровне*, чтобы не осложнять тестирование на *макроуровне* глупыми ошибками кодирования
- Заботятся о качестве продукта и несут за него ответственность, принимают участие в общей процедуре тестирования

Плохие программисты...

- Не считают тестирование важной и необходимой частью разработки программного продукта, а также *своей обязанностью*
- Передают непротестированный код в отдел QA и очень удивляются, когда обнаруживаются ошибки
- Осложняют себе жизнь тем, что обнаруживают ошибки слишком поздно, не проводя тестирования на ранних этапах, когда выявить ошибки гораздо проще

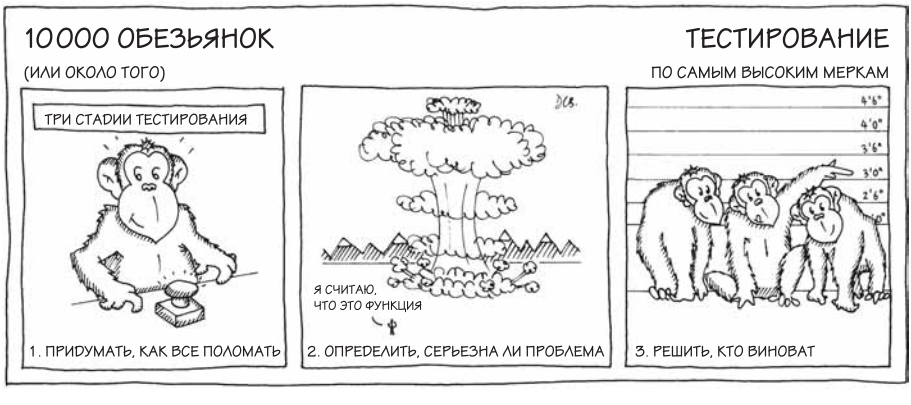
См. также

Глава 9. Поиск ошибок

Что делать при обнаружении сбоя – процедура поиска и исправления ошибок.

Глава 20. Рецензия на отстрел

Анализ кода как прием тестирования – ручная форма статического анализа кода.



Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 615.

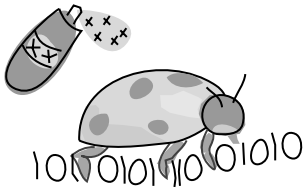
Вопросы для размышления

1. Напишите комплект тестовых примеров для кода определения наибольшего общего делителя (`greatest_common_divisor`), приведенного в начале главы. Постарайтесь проверить как можно больше случаев. Сколько отдельных контрольных примеров у вас получилось?
   a. Сколько успешно прошло?
   b. Сколько не прошло?
   c. С помощью своих тестов найдите ошибки и исправьте код.
2. Каковы различия между тестированием приложения электронной таблицы и системы автоматического пилотирования самолета?
3. Нужно ли тестировать *тестовый код*, который вы пишете?
4. В чем разница между тестированием, проводимым программистом, и тестированием в отделе QA?
5. Нужно ли писать набор контрольных примеров для каждой отдельной функции?
6. Существует *методика*, требующая сначала писать тесты, а потом код. Какого рода тесты вы стали бы писать?
7. Следует ли писать тесты для C/C++, чтобы проверить правильность обработки параметров, имеющих значение `NULL` (нулевых указателей)? В чем ценность такого теста?
8. Первоначально тестирование может происходить не на той платформе, для которой пишется код – она может быть недоступна для вас. Что лучше – отложить тестирование, пока вы не *получите* в свое распоряжение целевую платформу, или начать его сразу?

   Если код предназначен для работы в другой среде (например, на мощном сервере или во встроенном устройстве), как обеспечить адекватность тестирования?
9. Как определить, что проведено достаточно тестов? Когда можно *остановиться*?

Вопросы личного характера

1. Для какой части своего кода вы пишете тесты? Вас это удовлетворяет? Входят ли ваши тесты в процедуру автоматизированной сборки? Как вы проводите тестирование остального кода? Адекватно ли оно? Собираетесь ли вы изменить свою процедуру?
2. Какие у вас отношения с сотрудниками отдела QA? Какое мнение у них сложилось о вас?
3. Как вы обычно поступаете, обнаружив ошибку в своем коде?
4. Пишете ли вы отчет по каждой обнаруженной в коде проблеме?
5. Какой объем тестирования обычно осуществляют инженеры проекта?



9

Поиск ошибок

Отладка: что делать, когда дела идут плохо

В этой главе:

- Откуда берутся ошибки?
- Какие бывают ошибки?
- Технология отладки: поиск и исправление
- Инструменты отладки

Я не терпел поражений. Я просто нашел 10 000 способов, которые не работают.

Томас Эдисон

Недостатки есть у каждого. Кроме меня, пожалуй. Целый день я должен сидеть и разбираться со скучными проблемами в чужом коде. Наш отдел тестирования обнаружил, что наше программное обеспечение дает сбой, когда выполняет *то-то и то-то*. Поэтому я копаюсь в системе, нахожу код, в котором программист Федя напортачил три года назад, исправляю его, отправляю тестерам, а они обнаруживают новый сбой.

Конечно, сам-то я никогда бы таких элементарных ошибок не сделал. Мой код абсолютно надежен. Безупречен. Никакого жира и холестерина. Я и строчки не напишу без тщательного планирования, не буду считать оператор готовым, пока не проверю все особые ситуации, и я печатаю так аккуратно, что мне никогда не случалось набрать = вместо == в операторе `if`.

Я абсолютно надежен. Без всяких сомнений.

Хотя, может быть, это не совсем так.

Реальные факты

Не думаю, что кто-нибудь усаживает студентов-программистов и объясняет им, как устроена жизнь: «*Видишь ли сынок, есть птички и пчелки. Ах да, забыл про жучков.*» «Жучки», или «баги», – это неизменная мрачная сторона создания программ, неизбежная реальность. Печально, но это так. Существуют целые отделы, даже специальности, поставленные на борьбу с ними.

Всем известно, как множатся сбои в выпущенном ПО. Почему ошибки обнаруживаются с такой ужасающей регулярностью и в таких количествах? Это обусловлено человеческой природой. Программы пишутся людьми. Люди ошибаются. Вольно или невольно ошибаются по ряду причин. Ошибаются, потому что недостаточно хорошо разобрались в системе, над которой работают, или неправильно поняли, что им нужно реализовать, но чаще всего делают ошибки по невнимательности. Большинство ошибок делается по глупости. Однажды я встретился с очень простой иллюстрацией этого факта. Сыграйте в такую игру:

- Дерево, вырастающее из желудя, называется...
- Звук, издаваемый лягушкой, называется...
- Пар, поднимающийся над костром, называется...
- Белок яйца называется...

Желтком, правильно? А теперь подумайте. Если вы не попались на эту удочку, то, возможно, лишь потому, что были настороже, т. к. я вас предупредил. (Все равно можете засчитать себе очко.) Но кто станет предупреждать вас всякий раз перед тем, как вы соберетесь написать строку кода с ошибкой? Если бы такой человек существовал, ему следовало бы присудить сразу мешок очков, которых ему хватило бы на всю жизнь.

Все мы, программисты, достойны осуждения за плохое качество программ, которые мы пишем. Мы все виновны. Так и жить с вечным чувством вины или можно что-то предпринять? Ответов может быть два. Первый: *это не ошибка, а функция*. Придумайте объяснение и не обращайте внимания. При появлении сбоя отвечаем словами великого философа Барта Симпсона: «Это не я. Кто-нибудь видел, как я это делал? Вы все равно ничего не докажете!» (Simpsons 91). Возлагать вину можно на что угодно – на странности компилятора, ошибки ОС, редкие климатические явления и своенравность компьютеров. Или, как я отмечал в начале главы, можно обвинять коллег. Хорошо программировать, когда тебе все как с гуся вода.

Мы все же примкнем к другому течению, которое признает, что ошибки в программах *не* являются абсолютно неизбежным явлением. Многие глупые ошибки можно легко обнаружить и даже предотвратить, и от-

ветственный программист должен принять для этого меры. Наше главное оружие – защитное программирование и правильное тестирование. В данной главе мы посмотрим, какие успешные технологии отладки можно применить, если ошибки все же проскользнут в программу.

Природа этого зверя

Вопреки распространенному мнению термин *жучок* (*bug*) был в ходу еще до появления компьютеров. В 1870-х годах Томас Эдисон говорил о жучках в электрических цепях. История с релейной вычислительной машиной Mark II Aiken Гарвардского университета является первым зарегистрированным случаем компьютерного жучка. В 1945 году, когда первые компьютеры были таких размеров, что занимали целую комнату, в одно такое помещение залетела бабочка и, сев на какие-то провода, вызвала аварию системы. В журнале оператора была сделана запись о *первом реальном случае обнаружения компьютерного жучка*. Это событие увековечено экспонатом Смитсоновского института.

«Баги» – неприятная вещь. Но что, в сущности, они собой представляют? Номенклатуру этих явлений мы привели в разделе «Терминология» на стр. 184. Полезно определить, какие разновидности багов встречаются, выяснить, откуда они берутся, как им удается сохраниться и как от них можно избавиться.

Взгляд с высоты птичьего полета

Программные ошибки распадаются на несколько крупных категорий, знание которых облегчит дальнейшие действия. Природа некоторых ошибок такова, что их труднее отыскать, и это связано с тем, к какой категории они относятся. Слегка отстранившись и прищурясь, мы обнаружим следующие три класса ошибок:

Сбой при компиляции

Крайне досадно, когда код, на который вы потратили столько времени, отказывается компилироваться. Это значит, что вам придется вернуться и исправить противную маленькую опечатку или несоответствие типов параметров, а затем снова компилировать и только потом приниматься по-настоящему тестировать свой труд. Как ни странно, это лучший тип ошибок, с которым вы можете столкнуться. Почему? Просто потому, что их проще всего обнаружить и исправить. Они проявляются сразу и наиболее очевидны.[1]

Чем дольше поиск ошибок, тем дороже их исправление; это проиллюстрировано в разделе «Экономика ошибок» на стр. 216. Чем раньше вы сможете найти и исправить каждую ошибку, тем скорее вы

[1] При условии, что у вас правильно налажена среда компиляции, которая останавливается при обнаружении ошибки и дает разумные диагностические сообщения.

сможете продолжить работу и тем меньшим объемом неприятностей и затрат обойдетесь. Ошибки компиляции очень легко заметить (на них просто трудно не обратить внимания) и обычно легко исправить. Вы не сможете запустить программу, пока не устраните их.

Чаще всего сбой при компиляции бывает вызван глупой синтаксической ошибкой или обычной невнимательностью, например при вызове функции с параметром неверного типа. Сбой может быть вызван ошибкой в make-файле, ошибкой этапа сборки (например, отсутствием реализации функции) или даже нехваткой дискового пространства на машине, где осуществляется сборка.

Аварии этапа исполнения

После устранения ошибок компиляции у вас появляется исполняемый модуль, который вы и запускаете. Через некоторое время он аварийно завершает работу. Вы тихо ругаетесь и вините во всем нечистую силу. После 60-й аварии вы грозитесь выкинуть компьютер в окошко. Бороться с такими ошибками значительно труднее, чем с ошибками компиляции, но все же достаточно просто.

Причина в том, что, как и ошибки компиляции, эти ошибки вопиюще очевидны. С программой, которая перестала выполняться, не поспоришь. Нельзя сделать вид, что авария – это «функция». После того как программа откинула копыта, можно вернуться и подумать, где она пошла неверным путем. Легче, когда вы знаете, какие данные были введены перед сбоем и что сделала программа перед тем, как аварийно завершиться. Дополнительные сведения можно получить с помощью специальных инструментов (о чем ниже).

Неожиданное поведение

Это действительно неприятно, когда ваша программа не отправилась в мир иной, но ее явно понесло куда-то не туда. Вдруг она начинает делать непредсказуемые вещи. Вы ждете голубой квадрат, а появляется желтый треугольник. Код выполняется как ни в чем ни бывало, нимало не смущаясь вашим отчаянием. Откуда взялся желтый треугольник? Может быть, программу разрушили полчища воинственных COM-объектов? Почти наверняка окажется, что где-то в недрах кода, выполнявшегося полчаса назад, есть мелкая логическая ошибка. Удачи вам в поиске.

Сбой может появиться в результате одной неверной строки кода, а может возникнуть тогда, когда будут объединены друг с другом несколько взаимодействующих между собой модулей, исходные предположения которых не вполне совместимы.

Взгляд с поверхности земли

Если мы спустимся на землю и внимательнее посмотрим на ошибки этапа исполнения, обнаружится несколько новых групп ошибок. Здесь они перечисляются по степени нарастания болезненности – от занозы в пальце до отрубания головы.

Синтаксические ошибки

Обычно их *перехватывает* компилятор во время сборки, но некоторые грамматические ошибки проскакивают незамеченными. В результате программа ведет себя странно и непредсказуемо. В языках типа C часто встречаются такие синтаксические ошибки:

- В условных выражениях вместо `==` пишут `=` или вместо `&&` ставят `&`
- Забывают точку с запятой или ставят ее в неверном месте (классический случай – после оператора `for`)
- Забывают заключать группу операторов цикла в фигурные скобки
- Нарушают парность круглых скобок

Проще всего избежать попадания в одну из этих ловушек, если включить вывод компилятором всех предупреждающих сообщений. Современные компиляторы обнаруживают и показывают массу таких проблем.

Компилируйте код при включенном выводе компилятором всех предупредительных сообщений. Тем самым вы обнаружите потенциальные проблемы раньше, чем столкнетесь с ними реально.

Ошибки сборки

Не будучи, *по существу*, ошибкой этапа исполнения, ошибка сборки может проявиться только во время выполнения. Будьте бдительны и никогда не доверяйте своей системе сборки, какой бы надежной она вам ни казалась. В нынешние просвещенные времена ошибки в компиляторах встречаются редко. Но может оказаться, что вы выполняете не тот код, который хотели собрать.

Я несколько раз сталкивался с такой ситуацией: система сборки не выполняла повторную компиляцию программы или общей библиотеки (возможно, из-за отсутствия в make-файлах необходимых зависимостей или влияния временной метки старого выполняемого файла). И всякий раз, тестируя сделанные мной модификации, я, неведомо для себя, продолжал выполнять старый, содержащий ошибки код. Обмануть систему сборки можно разными способами, но хуже всего, если вы не заметите, что она сработала неправильно – как пораженная проказой конечность.

Иногда это обнаруживается достаточно поздно. Поэтому, если у вас есть хоть какие-то подозрения, разумно полностью очистить проект и выполнить сборку с самого начала. Это должно устранить возможные проблемы системы сборки.[1]

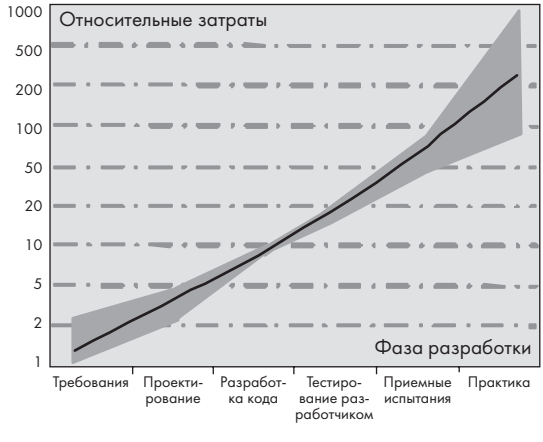
[1] Предполагается, что вы уверены в средствах чистки своей системы сборки. Для полной надежности удалите весь проект и загрузите его заново из системы контроля версий. Либо можно вручную удалить все промежуточные объектные файлы, библиотеки и исполняемые модули. В больших проектах оба варианта крайне утомительны. *C'est la vie.*

Экономика ошибок

Искусство отладки тесно связано с темой предыдущей главы – *тестированием* кода. Тестирование выявляет сбои, которые требуют отладки. Я разделил эти темы на две разные главы, потому что это разные отрасли знаний. Однако их совместное применение составляет фундамент разработки надежного программного обеспечения.

Безумное напряжение промышленного производства программ требует, чтобы код изготавливался быстро и дешево. Эта спешка приводит к тому, что программные продукты полны ошибок, в результате чего их выпуск чрезвычайно затягивается. Задержка выпуска ПО – большая проблема; это не просто неприятно, но может составить угрозу существованию всей компании.

На практике чем дольше вы избегаете тестирования и сохраняете ошибки, тем хуже становится ситуация; данный график иллюстрирует растущее влияние ошибок, сохраняющихся в течение процесса разработки. Он отражает среднюю стоимость поиска и исправления ошибки в зависимости от того, на какой стадии производства она обнаружена (Boehm 81).



Видно, как резко растет стоимость со временем (обратите внимание, что ось издержек имеет логарифмический масштаб). Более того, чем ближе плановый срок завершения проекта, тем меньше времени остается на тщательное тестирование. Дополнительное давление поджимающих сроков значительно осложняет отладку – в таких условиях растут шансы на то, что каждое исправление внесет в код новую ошибку.

Чтобы поберечь свою шкуру и нервы, начинайте тестирование на ранних стадиях и выполняйте его тщательно. Устраняйте все найденные ошибки как можно быстрее, пока они не приведут к еще большим неприятностям. Для этого существуют опробованные технологии; обратите, например, внимание на разработку, управляемую тестированием (test-driven development), один из элементов быстрой разработки программного обеспечения.

Элементарные семантические ошибки

Большинство сбоев времени исполнения обусловлено очень простыми ошибками, вызывающими некорректное поведение. Классическим примером служит использование неинициализированных переменных, которое бывает трудно выявить: поведение программы оказывается зависящим от случайного значения, оказавшегося в памяти, отведенной для переменной. Иногда программа работает прекрасно, а иногда дает сбой. Вот другие элементарные семантические ошибки:

- Сравнение значений переменных с плавающей запятой на (не)равенство[1]
- Выполнение вычислений без учета возможного числового переполнения
- Округление ошибок при неявном преобразовании типов (часто теряется знак `char`)
- Объявление переменной `unsigned int foo`, которая затем используется в коде `if (foo < 0)` — приехали!

Такого рода семантические ошибки часто обнаруживаются средствами статического анализа.

Семантические ошибки

Это коварные ошибки, которые не ловятся средствами контроля, их гораздо труднее выявить. Семантическая ошибка может быть низкоуровневой, например использование не той переменной, отсутствие проверки допустимости входных параметров или неверно составленный цикл. Уровень безголовости может оказаться выше: некорректный вызов API или нарушение внутренней целостности объекта. В эту категорию попадают многие ошибки, связанные с памятью; их бывает чертовски трудно обнаружить благодаря способности изменять и калечить выполняющийся код, из-за чего он ведет себя совершенно непредсказуемым и необъяснимым образом.

Программы часто ведут себя странно. Утешает одно: они делают ровно то, что мы им приказали.

Самая приятная разновидность ошибок времени выполнения — это воспроизводимые ошибки. Если ошибку можно повторить, гораздо проще написать тесты и проследить ее источник. Сбои, которые происходят нерегулярно, обычно связаны с ошибками памяти.

Взгляд из глубины

Разложив все ошибки по полочкам, рассмотрим их вблизи и обсудим некоторые из распространенных типов семантических ошибок:

---

[1] Этого делать нельзя: арифметика с плавающей точкой слишком приблизительна, чтобы точное сравнение имело какой-то смысл.

Ошибки сегментации

Ошибки сегментации, известные также как *ошибки защиты*, происходят при обращении программы к тем участкам памяти, которые не были выделены для использования в программе. Они приводят к тому, что ОС аварийно завершает ваше приложение и выводит некое сообщение об ошибке, обычно с полезной диагностической информацией.

Такие ошибки легко возникают при опечатках во вводе кода с указателями или при некорректных операциях с указателями. Типичная опечатка в C, приводящая к ошибке сегментации, — `scanf("%d", number);`. Отсутствие `&` перед `number` приводит к тому, что `scanf` пытается писать по адресу памяти, соответствующему случайному значению числа, в результате чего программа исчезает, оставив после себя облачко дыма. Но еще больший конфуз происходит, если значение числа соответствует допустимому адресу памяти. Тогда код продолжит работу как ни в чем ни бывало, пока не потребуется память, в которую вы только что произвели запись, и тогда ваша судьба непредсказуема.

Выход за границы памяти

Происходит при записи в память за границами участка, выделенного структуре данных, которой может быть массив, вектор или некая другая пользовательская конструкция. Записывая значения куда попало, можно покалечить данные в другой части программы. Если у операционной системы нет системы защиты (что чаще бывает во встроенных системах), можно даже испортить данные другого процесса или самой ОС. Ой.

Выход за границы памяти случается часто и обнаруживается с трудом. Симптомом обычно служит случайное неожиданное поведение, возникающее гораздо позже, чем ошибочное действие, иногда после выполнения тысяч других команд. Если повезет, выход за границы памяти придется на недопустимый адрес, и тогда возникнет ошибка сегментации, которую трудно не заметить. По возможности старайтесь применять безопасные структуры данных, которые избавят вас от таких катастроф.

Утечка памяти

Это постоянная угроза для языков, в которых нет сборки мусора.[1] Когда вам требуется память, вы должны вежливо попросить ее у системы (с помощью `malloc` в C или `new` в C++). Потом, когда она больше не нужна, будьте любезны вернуть ее (с помощью `free` и `delete` соответственно). Если вы поступите неприлично, забыв освободить память, программа станет постепенно захватывать все больше

[1] Наличие сборки мусора тоже не гарантирует отсутствие утечки памяти. Создайте ссылки двух объектов друг на друга, а потом обе удалите. Сборщик мусора должен быть довольно изощренным, иначе они останутся навечно.

и больше ограниченных ресурсов компьютера. Сначала это может оказаться незаметным, но постепенно реакция компьютера будет становиться все более замедленной по мере учащения переноса страниц памяти на жесткий диск и обратно.

Есть два других родственных класса ошибок: *лишнее* освобождение блока памяти, что приводит к непредсказуемым сбоям в среде, и небрежное управление другими ресурсами, такими как файловые указатели и сетевые соединения. (Запомните: все, что вы захватили вручную, должно быть вручную и освобождено.)

Исчерпание памяти

Такой риск всегда есть, как есть риск израсходовать указатели памяти или другие управляемые ресурсы. Возможно, это редкий случай (у современных компьютеров столько памяти, что такое событие маловероятно), что не служит оправданием для пренебрежения возможностью такого сбоя. В неряшливом коде отсутствуют соответствующие проверки, в результате чего программа ведет себя очень неустойчиво при критических нагрузках. По этой причине всегда проверяйте результаты операций выделения памяти или обращений к файловой системе.

Есть операционные системы, которые *не* возвращают ошибку при неудачном запросе памяти – всякое выделение возвращает указатель на зарезервированную, но не выделенную страницу памяти. Когда в дальнейшем программа пытается получить доступ к этой странице, ОС перехватывает это обращение и действительно выделяет странице память, благодаря чему программа успешно продолжает работу. Все идет хорошо, пока не кончится доступная память. Тогда ваша программа получит сигнал ошибки – спустя много времени после того, как состоялось соответствующее выделение памяти.[1]

Математические ошибки

Они встречаются в разном виде: исключения операций с плавающей точкой, некорректные математические конструкции, переполнение/потеря значимости или выражения, дающие сбой (например, при делении на нуль). Даже при попытке вывести число с плавающей точкой, но фактически передав целое в `printf("%f")`, можно получить в программе математическую ошибку.

Зависание программы

Обычно происходит вследствие ошибочной логики программы: чаще всего причиной оказываются бесконечные циклы с неверно заданными условиями завершения. Кроме того, в многопоточных приложениях встречаются взаимная блокировка и ситуации гонки,

[1] Это как раз характерно для Linux, по крайней мере, пока не закончится адресное пространство виртуальной памяти. В таком случае `malloc` может возвратить 0, но система внезапно рухнет.

когда код, управляемый событиями, ждет событие, которое никогда не может произойти. Однако обычно несложно прервать выполняемую программу, посмотреть, где застрял код, и определить причину зависания.

В различных ОС, языках и окружениях есть свои способы сообщения о таких ошибках, использующие разную терминологию. В некоторых языках исключаются целые классы ошибок из-за отсутствия средств, которыми пользователь может себе навредить. Например, в Java нет указателей и проверка каждого обращения к памяти осуществляется автоматически.

Борьба с вредителями

Выдергивать сорняки из программ трудно. Нужно обнаружить ошибку, распознать проблему, искоренить все следы нежелательного поведения, проверить, не проникла ли ошибка в другие места, и постараться не испортить ничего в коде, проводя все эти действия. Даже первый шаг – обнаружение ошибки – представляет собой большое затруднение: человек совершает ошибки при письме, но не меньшее их количество он делает при чтении. Читая свой текст или код, я, естественно, вижу то, что *хотел* написать, а не то, что написал в *действительности*. Ошибки в коде не бросаются в глаза. От компилятора помощи тоже немного; он действует весьма педантично. Компилятор сделает в *точности* то, о чем вы просили, а не то, что вы подразумевали.

Есть программисты, которые по сравнению со своими коллегами значительно реже (на 60%) допускают ошибки в коде, быстрее отыскивают и исправляют ошибки (тратя 35% обычного времени), а при исправлении вносят меньше новых ошибок (Gould 75). Как им это удается? Они обладают естественной способностью более внимательно относиться к решаемой задаче и концентрироваться на микроскопическом уровне кода, который пишут, в то же время не упуская из виду общую картину.

Таково *искусство* отладки; в значительной мере это мастерство, которое нужно освоить. Умение эффективной отладки приходит с опытом. И такого опыта у каждого *будет* предостаточно.

Самое главное правило отладки: *думай головой*. Контролируй, чем ты занят. Не крутись без толку, меняя куски кода в надежде, что все вдруг заработает.

Следуйте золотому правилу отладки: думать головой.

Есть два пути истребления вредителей: *обходная дорога* на скорую руку и теоретически правильная *столбовая дорога*. Знать нужно оба пути: иногда обходная дорога кажется быстрым способом решить проблему, а в действительности отнимает больше времени, а иногда следование по главной дороге отнимает больше сил, чем требуется на самом деле.

Обходная дорога

Ошибка очень проста. Причина очевидна. Стоит ли долго размышлять? Иногда для успеха достаточно небольшой правки; несколько простых тестов быстро находят проблему. Так стоит ли городить огород? Возможно, что и нет, но не надейтесь, что вам всегда будет так везти. Очень часто программисты пытаются исправлять ошибки мелкими поправками, заплатками, нашлепками, не особенно задумываясь над смыслом своих действий. В результате польза бывает сомнительной: они просто скрывают исходную проблему с помощью тьмы других ошибок.

Если вы приняли решение починить код на скорую руку, установите для себя жесткие временные границы. Не тратьте полдня на то, чтобы «попробовать еще один способ». После того как отведенное время вышло, переходите к методическому подходу, описываемому ниже.

Установите разумное временное ограничение на «бессистемную» отладку, и если она окажется безуспешной, переходите на более методичный способ.

Если ваши гадания будут удачны и вы найдете ошибку, снова включайте в работу мозги. Читайте «Как исправлять ошибки» на стр. 228 и изменяйте код осторожно и продуманно. Даже если ошибку удалось легко найти, ее исправление может оказаться не столь очевидным.

Правильный путь

Для отладки лучше применять более методичную и продуманную технологию, которая учитывает, что задача устранения неисправности имеет две разные стороны: *поиск* ошибки и ее исправление.

Каждая из сторон имеет свои трудности. Часто забывают о второй части и считают, что раз ошибка найдена, то исправление ее окажется очевидным. Это не так. Ниже я подробно расскажу об обоих аспектах и очерчу разумный подход к решению задачи. Но сначала о некоторых главных принципах, которым подчиняются действия при отладке:

- Трудность обнаружения ошибки зависит от того, хорошо ли вам знаком код, в котором она притаилась. Трудно залезть в код, который вы видите впервые, и принимать какие-то решения, не зная его структуры и принципов работы. Поэтому если вы вынуждены отлаживать незнакомый код, потратьте сначала некоторое время на его *изучение*.

Изучите отлаживаемый код – трудно найти ошибки в коде, который вам непонятен.

- Насколько легко проводить отладку, зависит также от степени вашего контроля над средой выполнения – в каком объеме вам доступны действия с программой и изучение ее состояния. Для встроенного

программного обеспечения отладка бывает значительно затруднена по причине более скудной инструментальной поддержки. Возможно также, что ваша среда обеспечивает плохую защиту от вашей собственной глупости; ваши мелкие ошибки будут иметь крупные последствия.

- Одно из самых мощных средств в арсенале отладчика – недоверие к чужому коду в совокупности со здоровой дозой цинизма. Причиной ошибочного поведения может быть все, что угодно, и, приступая к диагностике, вы должны проверять даже самых маловероятных кандидатов.

Когда вы ищете ошибку, не верьте никому. Проверьте самые невероятные причины, вместо того чтобы сходу отвергнуть их. Не принимайте ничего на веру.

Охота за ошибками

Как искать ошибки? Если бы существовала какая-нибудь простая трехшаговая процедура, мы бы ее выучили и стали выпускать идеальные программы. Нет такой процедуры, и наши программы такие, какие они есть. Попробуем разобраться в опыте поиска ошибок, накопленном человечеством.

Ошибки этапа компиляции

Мы рассмотрим их первыми, потому что с ними относительно легко бороться. Когда компилятор натыкается на что-то неприятное для него, он обычно не просто останавливается, чтобы сообщить об этом, но, пользуясь случаем, оглашает все, что думает о жизни в целом, и разражается целой серией новых сообщений об ошибках. Так его учили; найдя ошибку, компилятор пытается по возможности продолжить синтаксический анализ. Это ему редко хорошо удается, но чего еще можно ожидать при таком коде, как ваш?

Результатом является то, что последующие сообщения компилятора могут иметь весьма случайный и необоснованный характер. Смотреть нужно только на самое первое сообщение об ошибке и решать проблему с ней. Однако не мешает просмотреть и остальные ошибки; там могут быть другие полезные сообщения об ошибках, но не всегда.

Если сборка продукта не прошла, смотрите на первую ошибку компилятора. Последующие сообщения заслуживают гораздо меньшего доверия.

Даже первое сообщение компилятора об ошибке может оказаться загадочным или вводящим в заблуждение; это зависит от качества компилятора (если смысл ошибки невозможно понять, попробуйте другой компилятор). Код стандартных шаблонов C++ может провоцировать

некоторые компиляторы на безудержный поток сообщений об ошибках, содержащих таинственные заклинания.

Обычно синтаксическая ошибка находится на строке, указанной компилятором, но иногда – на *предшествующей* ей и делающей бессмысленной следующую; компилятор замечает это и выражает неудовольствие.[1]

Пример 1: графическая утилита

Программа

Небольшая утилита с графическим интерфейсом.

Проблема

Программа была переработана для придания ей «современного вида» – новые значки и расположение элементов. Старый интерфейс предполагалось сохранить как опцию в настройках. Во время модернизации все работало прекрасно почти до самого момента выпуска, когда кто-то попытался воспользоваться старым интерфейсом. Программа аварийно завершалась, когда начинало рисоваться окно, но прежде чем оно отображалось полностью.

История

К счастью, эта проблема была возобновляемой. Программу запустили в отладчике и нашли место сбоя, которое оказалось где-то глубоко в библиотеке интерфейса пользователя в коде отображения графики.

Изучение вопроса навело на мысль, что используется недопустимая графика. Программа пыталась показать значок, находящийся по нулевому адресу памяти, т. е. аварийное окончание вызывал нулевой указатель. По стеку вызовов мы определили, какое изображение нужно было показывать. Взглянув затем на каталог с графикой старой версии, мы обнаружили, что именно этого значка в нем и нет.

Очевидно, что сбой происходил в операции загрузки значка в конструкторе окна, возвращавшей нулевой указатель, который свидетельствовал о том, что загрузки значка не произошло. Но проверка возвращаемого значения отсутствовала – автор предположил, что графика всегда будет в положенном месте.

[1] C++ может иногда разыграть красивый фокус: предыдущая строка оказывается в другом файле! Если пропустить ; в конце объявления класса в заголовочном файле, то первая строка в файле реализации окажется бессмысленной. Ошибка, о которой сообщает компилятор, оказывается весьма загадочной.

Исправление должно быть двояким:

- Устроить проверку значений, возвращаемых всеми процедурами загрузки значков, чтобы они более элегантно обрабатывали случаи отсутствия графики.
- Поместить отсутствующую графику в нужные каталоги.

Время, потраченное на исправление

На изучение проблемы, исправление ошибки и проверку исправленного кода потребовалось несколько часов.

Полученные уроки

- Проверяйте значения, возвращаемые *всеми* функциями, даже если вам кажется, что в них не может быть сбоев.
- Тестируйте функции программы как можно раньше, особенно редкие условия, которые будут возникать не часто.

Ошибки компоновки в целом гораздо более понятны. Редактор связей сообщит вам, что отсутствует некая функция или библиотека, так что лучше подсуетиться и найти ее (или написать). Иногда компоновщик жалуется на таинственные проблемы, связанные с таблицей виртуальных методов C++; обычно это признак отсутствия деструктора или чего-либо аналогичного.

Ошибки этапа исполнения

Ошибки этапа исполнения требуют более тонких действий. Если в программе ошибка, это может означать, что содержащееся где-то в коде условие, которое вы считали выполненным, на самом деле таковым не является. Поиск ошибки состоит в последовательной проверке того, что вы считали истинным, до обнаружения места, где это условие не выполняется. Нужно создать модель реальной работы кода и сравнить ее с предполагавшейся вами работой. Единственный разумный путь – методически применять этот способ.

Отладка – это методичная работа, медленно сужающая кольцо вокруг места нахождения ошибки. Не следует относиться к ней как к игре в угадайку.

Научный метод – это процесс, с помощью которого ученые получают точное представление об окружающем мире. Это сходно с тем, что пытаемся сделать мы. Научный метод заключает в себе четыре этапа:

1. Наблюдение явления.
2. Формирование гипотезы, объясняющей явление.
3. Предсказание результатов других наблюдений на основе предложенной гипотезы.
4. Проведение экспериментов, подтверждающих предсказания.

Несмотря на то, что мы пытаемся *избавиться* от феномена ошибки, а не построить его модель, нам необходимо понять суть ошибки, чтобы действительно устранить ее. Научный метод представляет собой хорошую основу для отладки, в чем вы убедитесь, изучив предлагаемые ниже действия.

Идентифицировать ошибку

Все начинается с обнаружения того, что программа делает не то, что нужно. Может быть, она аварийно завершается или выводит желтый треугольник вместо синего квадрата – вы видите непорядок и должны его выправить. Прежде всего нужно послать отчет об ошибке в базу данных (см. раздел «Система контроля ошибок» на стр. 205). Это особенно важно, если вы находитесь в процессе отладки какой-то другой ошибки или у вас нет времени, чтобы сразу разобраться с ошибкой. Регистрация ошибки гарантирует, что она не будет забыта. Не пытайтесь ограничиться мысленной отметкой о необходимости вернуться к проблеме позже – вы обязательно забудете это сделать.

Прежде чем бросаться на поиски ошибки, вызвавшей обнаруженный вами сбой, определите сущность аномального поведения, с которым вы столкнулись. Опишите проблему как можно подробнее, ответив на следующие вопросы: зависит ли ошибка от времени? оказывают ли на нее влияние данные ввода, загруженность системы или состояние программы? Если вы не разберетесь в сути ошибки до того, как начнете ее исправлять, вы просто станете модифицировать код, пока не исчезнут симптомы. Причина ошибки окажется замаскированной, и та же неисправность проявится где-то в другом месте.

Работал ли код раньше? Поищите в своей системе контроля версий последнюю версию, работавшую без сбоев, и сравните тот код с нынешним.

Воспроизвести ошибку

Это нужно сделать в дополнение к описанию ошибки. Опишите набор действий, уверенно приводящий к возникновению данного сбоя. Если таких способов несколько, документируйте их все.

Первый шаг на пути установления места ошибки – это определение способа ее уверенного воспроизведения.

Хуже, если ошибку не удается воспроизвести. Необходимо расставить ловушки для сбора всевозможной информации на тот случай, если сбой возникнет снова. Если сбой возникает нерегулярно, тщательно накапливайте все сведения, получаемые в момент сбоя.

Определить место ошибки

Это серьезное дело. След взят; теперь нужно воспользоваться всеми полученными данными, чтобы найти зверя и точно установить, где он

залег. Легко сказать! Это процесс, в котором нужно отсеять все, что не имеет отношение к сбою или явно работает правильно – в духе Шерлока Холмса. По ходу дела выясняется потребность в дополнительной информации – чем больше ответов, тем больше возникает вопросов. Возможно, придется сочинить какие-то дополнительные тесты. Возможно, придется порыться в малоприятных потрохах кода.

Проанализируйте, что вам стало известно относительно сбоя. Не делая скороспелых выводов, составьте список подозреваемых участков кода. Попробуйте подметить какие-то закономерности событий, которые подскажут возможную причину. Если есть возможность, ведите учет входных и выходных данных, иллюстрирующих проблему.

Расследование хорошо начать с того места, где проявляет себя ошибка, хотя обычно она реально находится совсем в другом месте. Запомните: если отказ возникает в каком-то модуле, из этого необязательно следует, что именно *этот* модуль во всем виноват. Определить место аварийного завершения легко; отладчик сообщит вам, в какой строке произошел сбой, значения всех переменных в тот момент и кто вызывал эту функцию. Если аварийного завершения нет, начните с того места, в котором программа ведет себя неправильно. Двигайтесь оттуда в обратном направлении, следуя порядку выполнения кода, и проверяйте, что в каждой точке код делает именно то, чего вы от него ждете.

Начните с известного места, например с точки аварийного завершения программы. Затем двигайтесь в обратном направлении в сторону причины, вызвавшей сбой.

Есть несколько стандартных стратегий поиска ошибок:

- Самое худшее – менять что-то произвольным образом и проверять, не исчезнет ли сбой. Это незрелый подход. (Хотя профессионал может попытаться придать ему наукообразный вид!)
- Гораздо лучше стратегия *разделяй и властвуй*. Допустим, вы сузили область поиска до одной функции, в которой 20 шагов. После 10-го шага выведите промежуточный результат или установите контрольную точку и изучите ее в отладчике. Если значение правильное, значит, ошибка находится в тех командах, которые лежат ниже; в противном случае она выше. Займитесь этой частью команд и повторяйте операцию, пока не загоните ошибку в угол.
- Еще один способ – *формальный прогон* (*dry run*). Не рассчитывая, что интуиция поможет вам найти ошибку, поработайте сами вместо компьютера и выполните контрольную трассировку программы, вычисляя все промежуточные значения и конечный результат. Если ваш результат не совпадет с тем, который дает программа, ошибка в коде – он не делает то, чего вы от него хотите. Конечно, это требует времени, но зато эффективно, потому что доказывает, что ваши предположения не верны.

Уясните проблему

Когда вы найдете место, где таится ошибка, нужно разобраться в сути реальной проблемы. Если это простая синтаксическая ошибка, например = вместо == (*уф!*), последствия не слишком страшны. Если это более сложная семантическая проблема, убедитесь, что она вам понятна, как и все способы, которыми она может проявиться, и лишь потом действуйте дальше – возможно, вы обнаружили лишь часть проблемы.

Часто ошибка оказывается очень тонкой: код делает именно то, что должен *и* что вы предполагали, когда писали его! Проблема заключается в ложных допущениях (помните, как они опасны?). Тот, кто пишет функцию, и тот, кто к ней обращается, вполне могут предполагать различное поведение функции отдельных особых случаях. Выполните обратную трассировку и точно разберитесь, в чем причина проблемы и нет ли других фрагментов кода, содержащих ту же ошибку.

Если вам показалось, что вы нашли причину ошибки, досконально исследуйте ее и убедитесь, что не ошиблись. Не принимайте безрассудно первую же гипотезу.

Это важнейший принцип борьбы с ошибками. Иначе вы пополните число тех программистов, которые создают ошибок *больше*, чем устраняют их во время отладки.

Создайте тест

Напишите контрольный пример, демонстрирующий ошибку. Вам стоило сделать это еще на этапе «Воспроизведите ее». Если вы не сделали этого тогда, то теперь точно настало время. Используя накопленные знания, сделайте тест достаточно строгим.

Исправьте ошибку

Теперь самое простое: нужно исправить эту чертову штуку! Здесь действительно не должно быть никаких трудностей: вы точно знаете, почему происходит сбой, и можете сами вызвать его появление. При таких данных исправление обычно оказывается плевым делом. Часто программисты считают, что исправлять ошибки трудно; это потому, что они пропускают первые два шага.

В следующем разделе мы более подробно рассмотрим, как исправлять ошибки.

Докажите, что вы ее исправили

Теперь вы знаете, зачем вам нужен контрольный пример. Прогоните его снова и докажите, что мир стал более совершенным. Этот контрольный пример можно добавить в комплект регрессивных тестов, чтобы гарантировать отсутствие возникновения той же ошибки в будущем.

Отладка заканчивается лишь тогда, когда вы докажете, что ошибка устранена и проблема решена навсегда.

Все! Игра закончена, миссия выполнена. Все прекрасно. И все же...

Когда все средства бесполезны

Иногда можно перепробовать все перечисленное, и ничего не помогает: можете плакать и скрежетать зубами, биться головой об стену без всякого результата. Если такое случается, бывает полезно объяснить всю проблему кому-то постороннему. Где-то во время объяснения все вдруг становится на свои места и обнаруживается та ключевая информация, на которую почему-то все время не обращали внимания. Попробуйте и убедитесь сами. Это одна из причин, почему *программирование в паре* оказывается таким успешным методом.

Как исправлять ошибки

Этот раздел, как вы заметите, гораздо меньше предыдущего. Как ни странно. Обычно главную часть проблемы составляет *поиск* проклятой ошибки. После того как вы вычислили, где она лежит, исправить ее можно очевидным образом.

Но не поддавайтесь при этом ложному чувству безопасности. Мысль не должна останавливаться после того, как источник ошибочного поведения выявлен. Очень важно при осуществлении исправления не испортить ничего в программе – оказывается, что, потянувшись выдрать из клумбы сорняк, можно легко потоптать и саму клумбу.

Проявляйте крайнюю осторожность при исправлении ошибок. Следите, чтобы ваша модификация не покалечила ничего остального.

Модифицируя код, всегда задумайтесь *о возможных последствиях своей модификации*. Следите за тем, замкнуто ли исправление в пределах отдельного оператора или оно влияет на близлежащие участки кода. Не скажутся ли ваши изменения на коде, который вызывает эту функцию? Не изменят ли они каким-то незаметным способом поведение функции?

Убедитесь в том, что вы действительно нашли *корень* проблемы, а не просто скрываете один из ее *симптомов*. Тогда можно быть уверенным, что исправление делается в нужном месте. Проверьте, нет ли аналогичных ошибок в близких модулях, и при необходимости исправьте их тоже.[1]

---

[1] Это объясняет, почему нехорошо *программировать методом копирования и вставки*, т. е. методом дублирования кода. Здесь таится опасность: вы безрассудно копируете ошибки, а потом не сможете исправить их в одном-единственном месте.

Исправляя ошибку, проверьте, не повторяется ли она в близких разделах кода. Уничтожьте ошибку раз и навсегда: исправьте все ее дубликаты немедленно.

Наконец, постарайтесь сделать выводы из своей ошибки. Учиться необходимо, иначе мы обречены на вечное повторение одних и тех же ошибок. Что это – простая ошибка программирования, которую вы часто повторяете, или нечто более фундаментальное, например неправильное применение алгоритма?

Пример 2: повешен, проволочен и четвертован

Программа

Встроенное программное обеспечение для управления бытовым электронным устройством.

Проблема

Случайные блокировки, возникающие в среднем раз в неделю при непрерывной работе. Выражаются в полном зависании устройства; оно не реагирует на действия пользователя, нет соединения с сетью, даже отсутствует обработка прерываний – процессор полностью замирает. Последнее было особенно неприятно, поскольку затрудняло поиск причин.

История

Блокировка возникала так редко, что ее было очень трудно отследить. В попытке найти причину мы провели ряд тестов, запуская каждый примерно на неделю. Сначала мы опробовали несколько схем применения устройства в надежде, что какая-то из них быстрее выведет его на сбой, и это поможет определить его причину. Никакой разницы между этими тестами не обнаружилось.

Характер блокировки позволил предположить, что она была связана с особенностями аппаратуры. Мы запускали это программное обеспечение на материнских платах разных версий, с разной периферией и разными процессорами. Шли недели, но мы не приблизились к разгадке, зато потеряли немало волос (а в оставшихся появлялось все больше седины). Какую бы конфигурацию мы ни задавали, программа работала в течение примерно недели, а потом замирала.

Следующим ходом было удаление из системы отдельных секций кода. После многочисленных тестов мы свели проблему к одной компоненте; при ее наличии неизбежно возникала блокировка, а в отсутствие блокировки не было. Наконец-то, удача!

Выяснить, почему эта компонента вызывала такие проблемы, оказалось не просто. Она была построена на основе библиотеки стороннего разработчика, которая, в свою очередь, была основана на базовой библиотеке ОС. Мы узнали, что вышла обновленная версия этой базовой библиотеки ОС, но библиотека стороннего разработчика не была соответственно перекомпилирована. Таким образом, мы постоянно компоновали свой продукт с сомнительным кодом. Хотя теоретически разницы не должно было быть – изменение библиотеки ОС предполагалось совместимым на *уровне исполняемого кода*, – тем не менее перекомпиляция библиотеки стороннего разработчика решила проблему окончательно.

Время, потраченное на исправление

Весь процесс занял четыре месяца. В течение этого периода подключались и уходили разные участники, было потрачено много ресурсов на тестирование, привлекалось разнообразное оборудование и было проведено немыслимое количество совещаний. Этот «жучок» оказался с ядовитым жалом, причинив компании много неприятностей (не говоря об издержках).

Полученные уроки

При изменении какой-либо из компонент соберите заново всю программную платформу во избежание трудноуловимых несоответствий версий.

Из каждой исправленной ошибки делайте выводы. Можно ли было ее избежать? Можно ли было обнаружить ее быстрее?

Профилактика

Всякий скажет вам, что «предохраняться – эффективнее, чем лечиться». Лучший способ не множить ошибки – это не допускать их возникновения изначально. К несчастью, я не надеюсь, что мы когда-нибудь достигнем этого идеала. Поскольку программирование подразумевает решение задач, оно всегда будет сложным занятием; вы не только должны правильно решить задачу, но прежде всего должны понимать задачу в целом. Несмотря на это, тщательно проводимое защитное программирование позволяет избежать многих проблем. Хороший стиль программирования подразумевает дисциплину и внимание к деталям. Тщательное тестирование предотвращает проникновение ошибок в окончательные версии программных продуктов.

Писать об этом можно много, но все указания по профилактике сводятся к одному простому совету: *думайте головой*. И хватит об этом.

Спрей от ос, репеллент для мух, липучки...

Есть много полезных средств отладки, и глупо отказываться от них. Одни *интерактивны* и позволяют изучать код во время его выполнения, другие *неинтерактивны* и часто действуют как фильтр кода или анализатор, выводящий информацию об анализируемой программе. Научитесь ими пользоваться, и это неизмеримо сократит время, которое вы тратите на отладку.

Отладчик

Это самый известный инструмент отладки; его назначение отражено в названии. Отладчик (debugger) – это интерактивное средство, с помощью которого можно заглянуть во внутренности выполняемой программы и поковыряться в них. Вы сможете следить за порядком выполнения кода, изучать значения переменных, устанавливать в коде *контрольные точки*, в которых будет прерываться выполнение, и даже прогонять произвольные секции кода по своему выбору.

Отладчики бывают самого разного размера и сложности; одни представляют собой инструменты командной строки, другие – графические приложения. На какой бы платформе вы ни вели разработку, хотя бы один отладчик для вашего случая найдется (при этом вездесущий gdb теперь перенесен, кажется, на все мыслимые платформы).

Отладчик действует на основании *символов*, сохраненных в выполняемом модуле (это часть внутренней информации компилятора, которая обычно удаляется на финальной стадии сборки). С их помощью он предоставляет вам данные об именах функций и переменных и местонахождении файлов исходного кода.

Отладчики – мощное и полезное средство, но мне представляется, что ими часто пользуются неправильно или в избыточной мере, что *мешает* правильной отладке. Программисты легко увлекаются слежением за работой программы, уходят в сторону, уделяя внимание значениям ненужных переменных или пошагово выполняя не те функции, которые требуются, забывая об общей проблеме, которую они должны решить. Бóльший упор на осмысление проблемы может скорее привести к обнаружению ошибочного кода, чем охота за ним в отладчике.

Столкнувшись с поведением, которое вы не можете объяснить, умеренно пользуйтесь отладчиками. Не привыкайте к тому, чтобы сразу бросаться на них, не попытавшись сначала понять, как работает ваш код.

Средство проверки доступа к памяти

Этот интерактивный инструмент (Memory Access Validator) ищет в работающей программе утечки памяти и выход за пределы допустимых границ. Он бывает чрезвычайно полезен – вскрывает множество ошибок освобождения памяти, о которых вы и не подозревали.

Трассировщик системных вызовов

Средства трассировки системных вызовов, например `strace` в Linux, показывают все системные вызовы, выполняемые приложением. Это хороший способ узнать, как программа взаимодействует со своим окружением. Он особенно полезен, если возникает подозрение, что она застревает в ожидании какой-то внешней активности, которой не происходит.

Дамп памяти

Этот термин (core dump) применяется в UNIX для обозначения генерируемого ОС мгновенного снимка программы в момент ее аварийного завершения. Термин происходит от древних машин с памятью на *ферритовых сердечниках* (*ferrite core*); файл дампа до сих пор называется *core*. Он содержит копию памяти программы в момент ее гибели, состояние регистров ЦП и стек вызовов функций. Дамп памяти можно загрузить в анализатор (часто им оказывается отладчик) и получить большой объем полезной информации.

Журналирование

Средства журналирования позволяют программным образом генерировать информацию о программе во время ее выполнения. Некоторые системы журналирования позволяют назначать для выводимых данных приоритеты (например, отладочные, предупредительные, неустранимые) и производить при работе программы фильтрацию выводимых сообщений в соответствии с выбранным уровнем. Журнал работы программы содержит историю ее работы, по которой можно выяснить обстоятельства, приведшие к сбою.

Не прибегая к специальным средствам журналирования (входящим в операционную среду или полученным из библиотек сторонних разработчиков), можно достичь того же результата, оснастив свой код соответствующими операторами вывода в интересующих вас точках. Однако их вывод может перемежаться с обычной выдачей программы, и их нужно тщательно удалить при выпуске окончательной версии.

Иногда оказывается невозможным разместить даже самые скромные команды вывода. Однажды при запуске нового образца аппаратуры у меня не оказалось иных средств диагностики, кроме одного 8-сегментного индикатора на светодиодах и осциллографа, подключенного к свободной системной шине. Просто удивительно, как много информации можно при желании запихнуть в несколько лампочек!

Журналирование имеет свои недостатки: оно может замедлить выполнение программы, увеличить ее размер и даже привнести в программу ошибки. Некоторые системы журналирования, в которых при аварии уничтожается буфер с диагностическими сообщениями, бесполезны для выяснения причин аварийного завершения программы. Разберитесь в работе своего механизма журналирования и всегда направляйте сообщения диагностики в небуферизованный поток вывода.

Статический анализатор

Это неинтерактивный инструмент, анализирующий ваш исходный код на предмет возможных проблем. Многие компиляторы проводят элементарный статический анализ, если задать для них максимальный уровень вывода предупреждений, но хорошие инструменты анализа имеют значительно больше возможностей. Существуют продукты, которые обнаруживают проблемный код и случаи неопределенного поведения или непереносимых конструкций, указывают на рискованные приемы программирования, снимают метрики кода, проверяют выполнение стандартов кодирования и создают автоматические наборы тестов.

Применение средств статического анализа может устранить многие ошибки до того, как они проявят свое вредоносное действие – полезное средство страховки. Неплохая идея – воспользоваться статическим анализатором иной компании, чем та, которая изготовила ваш компилятор: менее вероятно, что обе компании сделают одинаковые допущения или ошибки.

Резюме

Я точно помню ту минуту, когда понял,
что с этого момента большую часть своей жизни
потрачу на поиск ошибок в собственных программах.

Морис Уилкс

Как смерти и налогов, мы при всех стараниях не сможем избежать ошибок. Конечно, эффект первых двух обстоятельств можно умерить, если не упускать из виду ни один новый крем против морщин или применять хитрые финансовые схемы, но если вы не знаете, как поступить, столкнувшись со сбоями в вашей программе, ваш код обречен.

Отладка – это искусство, которому можно научиться. Оно основано не на догадках, а на методическом поиске и разумных исправительных мерах.

Хорошие программисты...

- Лишают ошибки питательной среды; пишут код так, чтобы препятствовать их появлению
- Понимают работу своего кода и пишут тесты, старательно проверяющие его функционирование
- Ищут ошибки методически и тщательно, а не бросаются на поиски сломя голову

Плохие программисты...

- Не занимаются отладкой; совершают беспорядочные действия и тонут в море плохого кода
- Полжизни проводят в отладчике, пытаясь понять, что же делает их код
- Столкнувшись со сбоем, пытаются скрыть его – активно стремятся избежать отладки

Хорошие программисты...

- Сознают скромность своих сил и просят посторонних о помощи, если сами бессильны найти ошибку
- Изменяют код с осторожностью, даже если речь идет о «простых» исправлениях

Плохие программисты...

- Нереалистично оценивают качество своего кода и свою способность исправлять ошибки
- «Исправляют» ошибки, маскируя их симптомы, а не доискиваясь до истинных причин

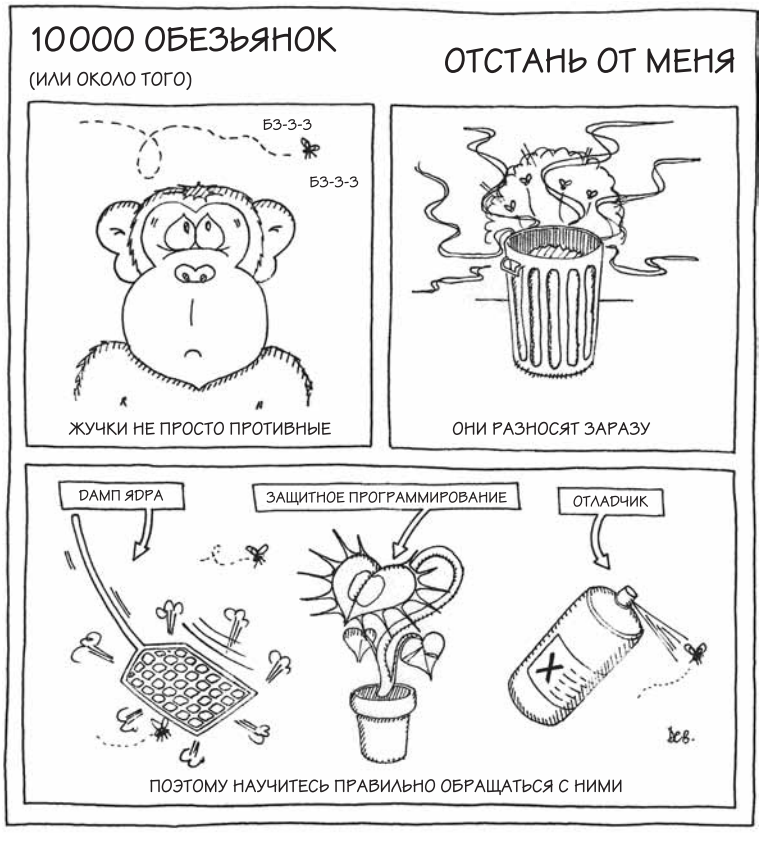
См. также

Глава 1. Держим оборону

Как помешать проникновению ошибок в ваш код.

Глава 8. Время испытаний

Нельзя исправить ошибку, не убедившись в ее существовании. Тщательное тестирование – это средство профилактики, предотвращающее проникновение ошибок в окончательный выпуск продукта.



Глава 20. Рецензия на отстрел

Анализ кода помогает обнаружить и устранить ошибки, а также указать на проблемные области, которые могли бы пройти незамеченными.

Контрольные вопросы

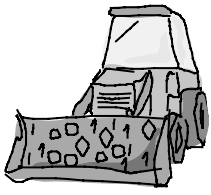
Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 622.

Вопросы для размышления

1. Следует ли требовать, чтобы ошибки исправлял тот программист, который написал код? Или программист, обнаруживший ошибку, лучше сможет ее исправить?
2. Как определить, что лучше – воспользоваться отладчиком или подумать?
3. Чтобы искать и исправлять ошибки в незнакомом коде, следует сначала изучить его. Однако темпы работы организаций, производящих программное обеспечение, часто не позволяют уделить достаточно времени изучению и освоению ремонтируемой программы. Какая стратегия будет оптимальной в таких условиях?
4. Предложите технологию, препятствующую возникновению ошибок утечки памяти.
5. В каких случаях оправдана кавалерийская атака на поиск и устранение ошибки в противоположность более методическому подходу?

Вопросы личного характера

1. Каким количеством отладочных технологий/инструментов вы обычно пользуетесь?
2. Какие стандартные проблемы и ловушки существуют в языках, которыми вы пользуетесь? Какие меры вы предпринимаете против появления соответствующих ошибок в вашем коде?
3. Какие ошибки чаще встречаются в вашем коде – сделанные по невнимательности или связанные с более тонкими проблемами?
4. Умеете ли вы пользоваться отладчиком на своей рабочей платформе? Служит ли он вашим повседневным инструментом? Опишите свои действия в случае:
   a. обратной трассировки;
   b. изучения значений переменных;
   c. изучения значений полей в структуре;
   d. выполнения произвольной функци;
   e. переключения контекста потока.



10

Код, который построил Джек

Способы превращения исходного кода в исполняемый

В этой главе:

- Как происходит сборка программ?
- Различные модели программных языков для строительства ПО
- Механизмы хорошей системы сборки
- Сборка финальных версий

Все, что вы строили годами,
может быть разрушено в одну ночь.
Но все равно строить нужно.

Мать Тереза

Программист (*Geekus maximus*) обычно обитает в своей естественной среде, т. е. сгорбившись в неверном сиянии монитора, вводя глубокомысленные комбинации символов пунктуации в текстовом редакторе. Иногда этот пугливый зверь покидает свою берлогу, чтобы совершить вылазку за кофе или пиццей. Затем он быстро возвращается в свое убежище и продолжает священнодействовать над клавиатурой.

Если бы программирование заключалось только во вводе языковых конструкций, наша профессия была бы гораздо легче, хотя возникла бы опасность, что нас заменят пресловутой бесконечной стаей обезьян с соответствующим бесконечным количеством текстовых редакторов. Вместо этого приходится

пропускать наш исходный код через компилятор (или интерпретатор), чтобы получить то, что работает так, как нам хотелось бы. Неизменно оно отказывается это делать. Смыть и повторить.

Задача, состоящая в том, чтобы преобразовать тщательно отшлифованный код на языке высокого уровня в исполняемый модуль, который можно распространять среди пользователей, обычно называется *сборкой кода* (*building code*), хотя в большинстве случаев этот термин используют наравне с *компиляцией* (*making*, *compiling*).

Процедура сборки является базовой – нельзя разрабатывать код, не выполняя сборки. Поэтому важно понимать, что́ в нее входит и как работает ваша система сборки – лишь тогда можно иметь хоть какое-то доверие к генерируемому коду. Здесь играет роль ряд довольно тонких факторов, особенно если базовый код достигает внушительных размеров. Любопытно, что почти все учебники программирования благополучно пропускают эту тему; они ограничиваются примерами программ, состоящих из единственного файла, что не отражает реальной сложности процедуры сборки.

Многие разработчики полагаются на систему сборки своей интегрированной среды, но это не снимает с них обязанности разобраться, как она работает. Очень удобно, когда можно нажатием кнопки сгенерировать весь код, но если вы не знаете, какие опции передаются компилятору C или на каком уровне находятся вспомогательные средства, сохраняемые в объектных файлах, то вы слабо контролируете ситуацию. То же самое происходит, когда вы вводите в командной строке одну команду *сборки*. Вы должны понимать, какие скрытые действия производятся, и только тогда сможете многократно осуществлять надежную сборку.

Языковые барьеры

Есть несколько разновидностей языков программирования, в каждой из которых существует своя механическая процедура построения исполняемой программы из исходного кода. Модели построения различаются по сложности, и у каждой есть свои сильные и слабые стороны.

Три основных механизма – это *интерпретируемые* языки, *компилируемые* языки и языки, *компилируемые в байт-коды*. Они представлены на рис. 10.1.

Интерпретируемые языки

Коду, написанному на интерпретируемом языке, не требуется проходить через особую фазу сборки. Написав некий код, нужно лишь сообщить интерпретатору, где он лежит; интерпретатор станет анализировать код и выполнять инструкции в реальном времени. Распространенными интерпретируемыми языками являются Perl, Python и JavaScript. Большинство ОО-языков является интерпретируемыми, в основ-

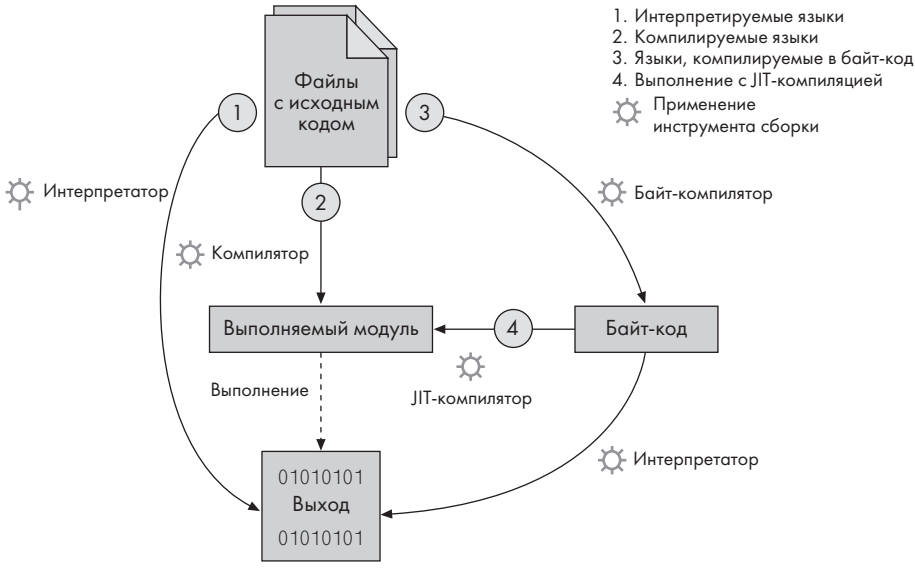


Рис. 10.1. *Методы сборки и выполнения языков программирования*

ном потому, что они разработаны не так давно – когда компьютеры смогли выполнять интерпретацию с разумной скоростью.

Главное достоинство интерпретируемых языков – скорость разработки программ, обусловленная отсутствием промежуточной стадии *компиляции*; все изменения можно очень быстро проверить. Кроме того, достигается независимость от платформы – интерпретаторы популярных языков работают на многих платформах. Ваша программа сможет работать всюду, куда был перенесен интерпретатор.

Но у интерпретируемых программ есть недостатки: они выполняются медленнее, чем компилированные эквиваленты, поскольку на этапе исполнения нужно прочесть, проанализировать, интерпретировать и выполнить каждую отдельную команду кода. Это большая работа. Современные машины настолько быстры, что проблемы интерпретации возникают только в приложениях, требующих особенно интенсивных вычислений. Существуют также различные технологии интерпретации, увеличивающие производительность кода: в некоторых языках исходные тексты компилируются перед выполнением (что увеличивает время запуска программы) или применяют компиляцию *Just-In-Time* (*JIT*), когда каждая функция компилируется непосредственно перед ее исполнением (что замедляет первое обращение к каждой функции). Для большинства программ такие накладные расходы приемлемы, а работа в режиме JIT-компиляции неотличима от выполнения обычного скомпилированного кода.

Действительно ли мы собираем программы?

«Сборка» часто используется в качестве метафоры программирования, равносильной тому, что происходит в «нормальной» сборочной промышленности. Между ними есть много глубоких параллелей, поскольку в том и другом случае мы имеем дело со строительными процессами. И мы действительно наблюдали в известной мере частичное совпадение и сотрудничество между двумя отраслями в виде движения «паттернов программирования» (см. «Шаблоны проектирования» на стр. 334), примером чего служит книга Кристофера Александра (Christopher Alexander) об архитектуре (Alexander 79).

Важно понять пределы этой метафоры и ее реальную ценность. В конце концов, нет идеальных метафор. Хотя это абстрактный вопрос, лежащий несколько в стороне, он важен, поскольку сравнение неизбежно вызовет предвзятость нашего подхода к разработке. Метафора хороша к месту; в иных случаях она может быть неудачна и даже вредна.

Положительное

Как и в физическом процессе строительства здания, мы начинаем с нуля и осуществляем строительство, помещая слои структуры один на другой. До начала строительства должны быть выполнены сбор технических требований и тщательное проектирование и разработка архитектуры. Можно особенно не заниматься планированием, если нужно построить садовую беседку, но было бы безумием рассчитывать на то, что без предварительного планирования может вырасти небоскреб; для его появления необходимы серьезные предварительные усилия по проектированию и планированию. Здесь параллель с созданием программного обеспечения достаточно близка.

Отрицательное

В остальных областях метафора оказывается довольно слабой. Например, базовые уровни программной конструкции модифицировать значительно легче, чем фундамент здания. Гораздо дешевле разрушить программное сооружение, чем физическое. Это говорит о том, что в области программирования возможности создания прототипов и исследования шире, чем в физическом мире.

Строительство реальных сооружений основано на твердых инженерных принципах: они отражены в законодательных актах и предполагают легальную ответственность. Многие фирмы-производители программного обеспечения в глаза не видели никаких инженерных принципов.

Самое страшное

В целом процедура разработки похожа на процесс физического строительства, включая в себя замысел системы, проектирование, реализацию и пробную эксплуатацию. Но в данной главе мы озабочены несколько иным – компиляцией и процедурами, входящими в такого рода строительные задачи. Здесь метафора тоже оказывается не вполне удачной. Каждый раз, взяв новый экземпляр исходного кода, вы «строите» его, создавая выполняемую программу – об этом здесь и идет речь. Обратите внимание на два разных употребления термина «build» (сборка, строительство).

Процесс построения программного продукта подчинен своим правилам: если вы модифицируете функцию, то должны потом заново построить систему. В противоположность этому, если вы покрасили двери, вам не нужно заново возводить стены своего дома.

Языки сценариев часто являются интерпретируемыми. У этих языков очень быстрый цикл разработки благодаря терпимому отношению к сомнительному коду (в языке нет строгих правил и слабая типизация) и отсутствию сложных функций. Языки сценариев часто используются в качестве связующей среды для более удобного вызова других утилит. Примерами языков сценариев служат сценарии оболочки UNIX, пакетные файлы Windows и Tcl.

Компилируемые языки

В компилируемых языках для преобразования файлов исходного кода в машинные инструкции на целевой платформе применяется последовательность инструментов. Обычно целевая платформа совпадает с платформой разработки, однако разработчики встроенного ПО часто работают на ПК, а целевыми могут быть самые разные машины, для чего применяются *кросс-компиляторы*. Крупные проекты компилируются поэтапно: отдельные файлы с исходным кодом компилируются в промежуточные *объектные файлы*, а затем эти объекты компонуются в окончательный исполняемый модуль. Эту модель иллюстрирует метафора выпекания пирога, показанная на рис. 10.2, где отдельные ингредиенты (исходные файлы) перемешиваются (компилируются) и наконец запекаются все вместе (компонуются).

Из компилируемых языков наиболее популярны C и C++, хотя компилируется большинство структурированных языков. Совершенно естественно, что скомпилированное приложение будет выполняться быстрее, чем его интерпретируемый аналог (по крайней мере, в отсутствие JIT-компиляции), хотя на практике вы этого не заметите; большинство

Рис. 10.2. *Кухня компиляции*

приложений не требует интенсивных вычислений и большую часть времени стоит в ожидании данных от пользователя, с диска или от сети.

Для компилируемых языков процедура сборки сложнее, чем при интерпретации, поэтому возможностей для возникновения сбоев в них больше. Для каждой платформы, на которой планируется выполнять приложение, оно должно быть скомпилировано отдельно.[1]

Языки, компилируемые в байт-коды

Компилируемые в байт-коды языки занимают промежуточное положение между интерпретируемыми и компилируемыми языками. Они предполагают этап компиляции, но не создают обычной исполняемой программы. Результатом является файл, содержащий *байт-коды*: псевдомашинный язык, который может выполняться на *виртуальной машине*. Примером таких языков служат Java и C#.

[1] Целевые платформы различаются по типам процессоров и операционных систем. Могут иметь значение и другие факторы, такие как имеющееся периферийное оборудование.

Распространено заблуждение, что байт-коды выполняются *обязательно* медленнее, чем соответствующие откомпилированные двоичные модули. Это не всегда так. JIT-оптимизатор может принимать разумные решения относительно того, какой код может значительно ускорить его работу (например, подстраиваясь под конкретную аппаратную часть, на которой выполняется программа).

Будучи компромиссным решением, байт-компиляторы частично наследуют как преимущества, так и недостатки других подходов. Байт-код может выполняться на любой платформе, на которую перенесена виртуальная машина, поэтому достигается переносимость ПО (хотя степень охвата разных платформ системами исполнения зависит от языка).

Делаем слона из мухи

Модели компиляции (и компиляции в байт-коды) сложнее всего в обсуждении, поэтому рассмотрим, что включает в себя компиляция программного обеспечения. Просто поразительно, как мало начинающих программистов действительно понимает это, поэтому мы начнем с базовых принципов. Если эта тема вам знакома, можете пропустить ее.

Для того чтобы лучше понять суть, лучше рассмотреть каждую ручную операцию, а не рассчитывать, что IDE выполнит за вас всю работу сборки. Все должно стать понятным из следующего рассказа в пяти частях о разработке простой программы:

1. Вы начинаете новый проект на C. Он избавит от всех невзгод, связанных с разработкой программного обеспечения, и откроет новую эру мирового сотрудничества. Однако единственное, что у вас вначале имеется, – это один файл, содержащий `main`. Надо же с чего-то начинать.

   Собрать и запустить эту программу из одного файла легко – просто вводите `compiler main.c`,[1] и он выплевывает исполняемый модуль, который можно запускать и тестировать. Все просто.

2. Программа растет. Вы делите ее на части и помещаете каждую в отдельный файл, содержащий определенный функциональный блок. Сборка по-прежнему не вызывает затруднений. Теперь вы просто печатаете `compiler main.c func1.c func2.c`. Снова на выходе оказывается исполняемая программа, которую вы тестируете, как и раньше. Никаких сверхусилий.

3. Вскоре у вас возникает понимание того, что некоторые части кода в действительности оказываются особыми компонентами с собственными проблемами – почти что самостоятельными библиотеками. С этими разделами кода проще работать, если поместить их в отдельные каталоги, сгруппировав похожие секции кода. Вот теперь

[1] Очевидно, `compiler` нужно заменить той командой, которая запускает ваш компилятор C; это лишь гипотетический пример.

проект начинает разрастаться. Эту новую структуру кода можно собрать, скомпилировав вручную каждый отдельный файл исходного кода и вызывая при этом компилятор таким образом, чтобы он не строил исполняемый модуль, а создавал промежуточные объектные файлы. После этого компилируется `main.c` и компонуется со всеми промежуточными объектными файлами. Для этого может также потребоваться задать компилятору включаемые файлы из некоторых других каталогов. Задача становится несколько сложнее.

При каждом изменении кода в одном из новых каталогов вам придется запускать команду компиляции в *этом* каталоге, а потом еще раз выполнять окончательную команду «скомпоновать все». И все вручную. Кроме того, если вы измените заголовочный файл, используемый другими каталогами, придется заново компилировать все *те* каталоги тоже. Если вы забудете это сделать, редактор связей может выдать целый ряд непонятных сообщений об ошибках.

Чтобы избавиться от ввода многочисленных команд в командной строке, можно написать *сценарий оболочки* (или *пакетный файл* в Windows), который обойдет все каталоги и в каждом запустит необходимые команды сборки. Когда исчезнут вся эта муторная работа и скучный набор параметров компиляции, вы сможете снова заняться серьезным делом – разработкой кода, и на душе у вас при этом будет покойно, потому что не нужно запоминать все это многословие, требующееся для сборки.

4. В дальнейшем эти подкаталоги действительно становятся самостоятельными библиотеками; они начинают использоваться в других проектах. Вы пригладите код, чтобы его использование стало немного удобнее, добавите рассчитанную на пользователя документацию, а затем измените команды сборки так, чтобы генерировать *библиотеки общего доступа*, а не объектные файлы. Для этого потребуются дополнительные изменения в сценарии сборки, но они относительно невелики и не слишком обременительны.
5. Некоторое время разработка продолжается в том же духе. Объем кода быстро растет. Создаются многочисленные каталоги и подкаталоги. Несмотря на то что файловая структура выглядит довольно скромно, время, затрачиваемое на сборку, начинает вызывать проблемы – когда вы запускаете сценарий сборки, заново компилируются все исходные файлы, даже если в них нет никаких изменений. Возникает мысль о том, чтобы самостоятельно следить за всеми изменениями и запускать сборку в подкаталогах тоже самостоятельно (возможно, создавая для этого в каждом каталоге особые сценарии сборки). Но теперь проект вырос настолько, что легко пропустить какие-то зависимости. Это приведет к трудно разрешимым проблемам сборки или еще более тонким проблемам (например, некоторые ошибки не препятствуют выполнению сборки, но получаемая в результате программа ведет себя некорректно).

> Теперь дальнейшая разработка становится под угрозу срыва. Вы перестаете доверять системе, которая осуществляет сборку кода. Она ненадежна. Выполняемый модуль вызывает доверие лишь тогда, когда каталоги полностью очищены от промежуточных модулей и сборка выполнена с чистого листа.

И тут появляется инструмент, который как раз предназначен для работы в таких ситуациях. Классическим решением является программа командной строки с оригинальным названием make (Feldman 78). Она вместо вас займется всеми промежуточными объектными файлами и правилами компиляции и, что главное, учтет зависимости между файлами. Эта программа получает указания в виде make-файлов, содержащих необходимые правила сборки. Она смотрит на временны́е отметки исходных файлов и проверяет, какие из них изменились с момента последнего запуска make, а потом перекомпилирует только их, а также те, которые от них зависят. Это более интеллектуальная версия сценариев, которые мы писали раньше, специально ориентированная на задачу компиляции и повторной компиляции программного обеспечения.

С течением лет появилось много разновидностей скромной программы make, часто оборудованных милым графическим интерфейсом. GNU Make – один из наиболее широко используемых инструментов (он бесплатный и отличается высокой гибкостью). Если вы пока не посвящены в Культ make, то во врезке «Make: краткий обзор» на стр. 248 приводится объяснение ее основных функций.

Существует множество других распространенных систем сборки, например SCons, Ant, Nant и Jam. Все они ориентированы на конкретную среду сборки (например, Nant применяется для сборки проектов .NET) или на определенное свойство (многие пытаются упростить синтаксис make, который весьма вычурен!).

Выполнение сборки

Мы рассмотрели некоторые главные моменты построения сборки программы в этом засасывающем болоте создания программного продукта. В сущности, в любой процедуре сборки программы есть один или несколько файлов с исходным кодом на входе и некая выполняемая программа на выходе. Результатом может быть даже полный дистрибутив программы, включая выполняемый файл, файлы подсказки, программу установки и т. д., которые будут тщательно упакованы и готовы к записи на CD.

Подобно той собирательной истории, у которой я беззастенчиво позаимствовал название данной главы, по мере развития и совершенствования нашего программного обеспечения развивается и совершенствуется процесс его сборки. Возможно, в вашем случае процесс сборки не был вначале таким элементарным, как в приведенном примере, но обычно

оснастка для сборки сначала проста и развивается параллельно с кодом, для которого она создана. В крупном проекте часто участвует внушительная процедура сборки, требующая (но не всегда имеющая) адекватной документации. Мы рассмотрели компиляцию единственного файла исходного кода – фундаментальную стадию процесса, а теперь на ее основе возведем целое здание дополнительных действий.

Процесс сборки *не ограничивается* компиляцией исходных файлов. В него может также входить подготовка текстовых регистрационных файлов по шаблонам, создание интернационализированных строк для интерфейса пользователя или преобразование графических файлов

Сроки и условия

Следующие термины составляют основу терминологии построения программного продукта:

Исходный код

Исходный код (source code) физически содержится в файлах, которые вы пишете, и обычно составляется на языке высокого уровня. С помощью надлежащих средств эти языковые конструкции могут быть преобразованы в работающую программу.

Компиляция

Исходный код преобразуется в выполняемый модуль одним из двух способов. Первый состоит в *компиляции* его в выполняемую программу. Альтернативой является *интерпретация* исходного кода в реальном времени – языковая среда времени исполнения выполняет синтаксический анализ и исполняет исходный код при запуске программы.

Сборка

Это не вполне точный термин, который часто используют как синоним *компиляции*. Компиляция составляет один из этапов построения программы, тогда как сборка описывает весь процесс полностью. Термин *make* тоже используется неоднозначным образом; более того, это также название распространенного инструмента сборки программ.

Объектный код

Объектный код хранится в *объектном файле*. Это компилированный вариант файла исходного кода. Объектный код нельзя выполнить непосредственно; он зависит от других файлов кода (большинство программ состоит из нескольких исходных файлов). Объектный файл должен быть *скомпонован* с другими объектами, и тогда он может образовать *исполняемый* модуль.

Библиотека

Библиотека кода похожа на объектный файл; это собрание компилированного кода, но сама она не является программой. В библиотеке хранится связанное собрание полезной функциональности, которую можно включить в любую программу. Библиотеки бывают *статическими* и *динамическими*. Первые компонуются как объектные файлы, вторые динамически загружаются приложением во время выполнения.

Машинный код

На некоторых стадиях результатом компиляции становится *машинный код*, а не объектные файлы. В таком виде исходный код представляет собой точные команды процессора для программы. Машинный код преобразуется в реальные команды ЦП с помощью *ассемблера*, почему его также называют *кодом ассемблера*.

Некоторые библиотеки ОС низкого уровня и встроенные программы написаны на языке ассемблера, но обычно мы работаем на языках высокого уровня, а ассемблер оставляем компилятору в качестве его внутреннего механизма.

Компоновка

Редактор связей (*linker*) соединяет один или несколько *объектных файлов* (а возможно, и библиотек) в конечный исполняемый модуль или частично скомпонованную библиотеку кода.

Выполняемый модуль

Результат компиляции или компоновки. Это самостоятельная программа, которую можно непосредственно выполнять на компьютере.

в особый формат. Практически все такие виды деятельности можно включить в систему сборки и выполнять при обычном ее осуществлении. Это предполагает, что все инструменты должны иметь возможность выполняться в составе сценариев – запускаться некоторой другой программой (например, make).

Необходимо рассматривать свою систему сборки не отдельно, а как часть всего дерева исходного кода. Make-файлы должны храниться в системе управления версиями, как и исходные коды, и составлять часть программы. Это важно – без них вы не сможете собрать приложение.

Считайте систему сборки частью дерева исходного кода и ведите их совместно. Они тесно связаны между собой.

Make: краткий обзор

Make – одна из наиболее широко применяемых систем сборки в мире программирования. Промчимся вихрем по ее организации и возможностям.

Make управляется *make-файлами*, которые обычно располагаются в каталогах по соседству с исходным кодом, который они собирают. Эти make-файлы содержат правила, описывающие сборку приложения. Каждое правило описывает *цель* (*target*, программу или промежуточную библиотеку), детали, от которой она зависит, и как ее создавать. Комментариям в файле предшествует #. Вот короткий пример, в котором для сборки источника используется гипотетическая программа `compiler`:

```
# Первое правило утверждает: ".o-файлы можно собирать
# из .c-файлов, и вот команда для этого."
# $< и $@ - это специальные символы для файлов источника
# и приемника. Да, синтаксис make бывает загадочным...
%.o: %.c
    compiler -object $@ $<
# Это правило говорит: "программа myapp собирается из трех файлов .o,
# и вот как компоновать их вместе"
myapp: main.o func1.o func2.o
    linker -output $@ main.o func1.o func2.o
```

Это общая идея. Если сохранить текст в файле с именем `Makefile` и выполнить команду `make myapp`, произойдет загрузка и синтаксический анализ этого файла. Так как `myapp` зависит от нескольких `.o`-файлов, то сначала они будут собраны из соответствующих `.c`-файлов с помощью указанных правил. Затем будет выполнена команда редактора связей, которая создаст приложение.

Есть много способов приукрасить эту процедуру и сделать ее более управляемой. Например, в make-файлах можно определять переменные, и тогда правило `myapp` будет выглядеть изящнее:

```
OBJECT_FILES=main.o func1.o func2.o
myapp: $(OBJECT_FILES)
    linker -output $@ $(OBJECT_FILES)
```

Подробное обсуждение всех деталей применения make выходит за рамки этой книги, но они должны быть известны каждому разработчику. Есть очень много полезных функций. Средства сборки с GUI фактически представляют собой оболочки над этими функциями, скрывая детали составления make-файлов. Обычно они легче настраиваются, но могут препятствовать созданию особо сложных конфигураций сборки.

Что должна уметь хорошая система сборки?

Ниже перечислен ряд качеств хорошей системы сборки.

Простота

Система сборки должна быть доступна для *всех* программистов, а не только особых специалистов. Любой разработчик должен иметь возможность выполнить сборку, иначе он не сможет работать. Если система слишком сложна, она практически бесполезна. Ей должны быть присущи следующие свойства:

Простота освоения

Это означает, что новый разработчик, присоединившийся к команде, должен быстро разобраться в том, как собрать пакет. Пока он не освоит процедуру сборки, от него будет мало прока. Мне приходилось работать в компаниях, где только *посвященным* дозволялось изучать и осуществлять процедуру сборки. Такая позиция не только непродуктивна, но и опасна – что произойдет, если по каким-то причинам уйдут те, кто действительно умеет выполнять сборку кода?

По мере развития программного пакета он становится все больше и труднее для понимания. Если система сборки развивается параллельно, она также становится все больше и труднее для понимания. По мере расширения своих средств сборка становится все более хитроумной и загадочной. Боритесь с ее усложнением.

Простота установки

Установка системы сборки предполагает следующие действия:

- Взять чистый ПК, на котором установлен свежий экземпляр ОС.
- Установить все необходимые программы (компиляторы, трансляторы, система управления версиями, инсталляторы, а также патчи и пакеты обновления).
- Установить все необходимые библиотеки (обратив внимание на правильность версий).
- Создать нужную среду для осуществления сборки (для этого может потребоваться организовать деревья каталогов, установить переменные окружения, получить необходимые лицензии и т. п.).

Без четких инструкций по установке нельзя быть уверенным в стабильности и возобновляемости процедуры сборки.

Привычность

Лучше всего применять стандартные, хорошо знакомые инструменты сборки – такие, которые привычны разработчикам и не требуют

значительных усилий в освоении. Сложные инструменты, в которых никто толком не разбирается, вызывают беспокойство.[1]

Единообразие

Важно, чтобы все пользовались одной и той же системой сборки. В противном случае вы будете собирать разные пакеты. С виду разные механизмы сборки могут показаться эквивалентными (*Я работаю в своей IDE, а он пользуется make-файлами*), но в таком случае затрудняется сопровождение и возникает опасность ошибок. Могут незаметно вкрасться мелкие различия, например разные опции компиляторов, и исполняемые модули не будут одинаковыми.

Это согласуется с требованием хранить систему сборки рядом с деревом исходного кода. Если система сборки физически становится частью кода, то ее невозможно обойти или избежать.

Все программисты, участвующие в проекте, должны пользоваться единой системой сборки. Иначе они будут собирать разные программные пакеты.

Кажется, все совершенно очевидно, но наделать ошибок очень легко. Даже если вы *все пользуетесь* одними и теми же make-файлами, вы можете не обратить внимания на другие различия – несоответствие версий библиотек, инструментов или сценариев сборки может привести к тому, что в итоге будут собираться разные программы.

Повторяемость и надежность

Сборка должна быть детерминированной и надежной. Набор исходных файлов должен легко определяться перед сборкой. Две сборки, проведенные с одним и тем же набором файлов, должны дать совершенно одинаковые исполняемые модули – сборка должна быть *повторяемой*.

Правильная система сборки позволяет многократно создавать физически идентичные бинарные файлы.

Тогда вы сможете пометить этот набор исходных файлов в системе контроля версий как определенную версию своего продукта (либо сделать резервную копию этих файлов) и в любой момент в будущем выполнить идентичную сборку.

Это очень важно: один из ваших клиентов может обнаружить существенную ошибку в старой версии программы, и если вы не сможете найти ту версию и собрать точно такую же программу, как у клиента, то

---

[1] Я испытываю органическое недоверие ко всем средствам, более сложным, чем GNU Make, но это может характеризовать скорее меня, чем эти сложные средства. Мне *вполне* достаточно того, что дает GNU Make!

не сможете воспроизвести аварийную ситуацию, а тем более исправить ошибку.

Вы должны быть в состоянии достать дерево исходных файлов трехлетней давности и правильно собрать его заново.

Если процедура сборки выдает двоичный файл, который не удается собрать повторно, это должно насторожить. Когда результат сборки зависит от фазы луны, есть над чем призадуматься. В этой связи необоснованное применение `__DATE__` в C или какой-то иной информации в исходных файлах, подверженной изменениям, следует свести к абсолютному минимуму.

Сборка всегда должна действовать идеально – она должна быть *надежной*. Если она иногда завершается аварией или генерирует неработоспособный двоичный файл, это просто опасно. В таком случае у вас не может быть уверенности, что вы тестируете правильный исполняемый модуль и что ваша фирма выпускает работающий продукт. Проблемы сборки существенно осложняют разработку.

Сборка должна проходить практически незаметно: вам нужно только знать, на какой рычажок нажать, чтобы с уверенностью получить готовый продукт.

Атомарность

Идеальная система сборки должна принять *исходные файлы* без какой-либо правки и скомпилировать их все без какого-либо вмешательства с вашей стороны. Никаких специальных действий перед сборкой не должно быть. Недопустимо, чтобы посреди сборки вам приходилось запускать какое-то другое приложение и проталкивать файл. Сборка должна запускаться *одной* командой. В результате вам не придется ничего дополнительно держать в голове, рискуя позабыть это в один прекрасный день. Все тайны сборки должны быть сохранены в надежном месте – ее скрипте. Сборка всегда может быть повторена. Она надежна.

Правильная сборка выглядит как одна операция. Достаточно нажать кнопку или набрать одну команду.

Если такой идеал для вас недостижим (в силу веских причин), то все равно постарайтесь сократить количество ручных операций. Все ручные операции должны быть полностью документированными. Допустимо (и даже желательно) разбить процедуру на следующие этапы:

1. Получение чистого исходного кода.
2. Сборка кода.
3. Создание дистрибутива из результатов.

Обратите внимание на разделение *сборки* кода и *получения* исходных файлов; можно сделать так, чтобы одна и та же команда сборки созда-

Рассказы бывалого человека

Повторяемость сборки имеет важное значение; вы должны быть в состоянии воссоздать любую выпущенную версию вашего продукта. Иначе у вас могут возникнуть неприятности. Мне довелось работать в компании, где возникла именно такая проблема.

Разработчики сделали изменения в коде установленной у клиента программы, но не перенесли их в свой экземпляр кода в системе контроля версий. Таким образом, у клиента оказалась «неофициальная» версия продукта. Когда позднее клиент обнаружил критическую ошибку, программисты не смогли воспроизвести ее у себя. При этом никто не мог понять причин, потому что о сделанной ранее правке экземпляра клиента к тому времени все уже позабыли.

Почему это произошло? Потому что несравнимо легче сделать модификацию на скорую руку, чем действовать, как полагается, т. е. исправить ошибку в основном экземпляре кода, протестировать его, выпустить официальную версию, передать ее клиенту и получить надлежащую санкцию на установку. Когда бизнес клиента зависит от вашей программы и все его производство останавливается и ждет, чтобы вы исправили ошибку, на вас оказывается огромное давление с целью заставить сделать скоропалительную заплатку.

вала разные версии продукта в зависимости от того, какой исходный код будет выбран. *Упаковка* программы также выделена в отдельный этап; во время разработки не всегда стоит тратить время на создание полного инсталляционного пакета.

Борьба с ошибками

В конце разработки, когда осядет пыль вокруг готового кода, ошибок при сборке *не будет*. Но в процессе работы что-то будет не получаться постоянно. Система сборки должна уметь справляться с ошибками и оказывать вам помощь.

- Если возникла ошибка, система сборки не должна продолжать работу. Она должна остановиться и сообщить вам все данные о том, что сломалось и где. Если сборка будет продолжена, наверняка появятся другие проблемы как следствие первой пропущенной ошибки. Разобраться с ними будет очень трудно. В ваших интересах не нарушать этого правила.
- При ошибке во время сборки система должна удалить все незавершенные объекты. Иначе при следующем запуске сборки она решит, что эти файлы целые, и возьмет их в работу. В результате потом мо-

гут возникнуть проблемы с поиском таинственно скрывающихся ошибок.

- Сборка должна проходить *без лишних сообщений*. Это зависит не столько от процедуры сборки, сколько от вашего исходного кода.[1] Если код генерирует ошибки компилятора, значит, в нем нужно разбираться. Переделайте код так, чтобы компилятор работал молча. За многочисленными дурацкими предупреждениями *можно просмотреть* более коварные сообщения.

  Для большей уверенности активируйте вывод компилятором всех предупредительных сообщений; отключение их вывода не устраняет проблему, а скрывает ее.

  Этой рекомендации нужно следовать с самого начала: позаботьтесь о процедуре сборки, начиная проект. Если вы *включите вывод всех предупредительных сообщений* только тогда, когда уже будет написано много кода, вас захлестнет лавина сообщений. Естественной реакцией будет скорее вновь отключить этот вывод и сделать вид, что ничего не было. Упростить себе жизнь любыми средствами. Если уж вы собираетесь делать что-то, нужно делать это с самого начала.

Механика сборки

За этими соображениями качества стоят практические вопросы системы сборки. Чтобы поговорить о них конкретно, мы подробно обсудим make, конкретную систему сборки и make-файлы, но не пугайтесь – за исключением различий в синтаксисе, другие системы сборки следуют аналогичным соглашениям (даже красивые графические пакеты).

Выбор целей

Make-файлы определяют *правила*, описывающие, как нужно собирать *цели*. (Другие системы сборки работают очень похожим образом, даже если есть различия в терминологии.) Система умеет определять, какие требуются промежуточные цели, и тоже собирает их. В одном make-файле может быть указано несколько целей. Благодаря этому одна система сборки может генерировать несколько разных выходных объектов, например:

- Различные программы (часто встречается, когда код двух программ содержит общие компоненты и потому они располагаются в одном дереве исходного кода для сборки).
- Сборка приложения для разных целевых платформ (например, версий для Windows/ Apple/Linux или настольной машины/PDA).

---

[1] Сделать это нетрудно – достаточно отключить вывод предупреждений компилятором, и сборка пройдет без лишнего шума. Но это *неправильный* способ решения проблемы.

- Различные версии продукта (скажем, *полная версия* или *демонстрационная*, в которой отсутствуют возможности сохранения/печати результатов).
- Сборка разработчика (с поддержкой отладки, регистрации в журнале и операторами контроля, прерывающими работу).
- Различные *уровни* сборки (только внутренних библиотек, всего приложения, полного дистрибутива).

Может потребоваться неким образом скомбинировать эти цели, например собрать демонстрационную версию для PDA.[1] Дерево исходного кода можно организовать так, что каждая из этих целей будет собираться из одних и тех же файлов. Вам тогда нужно будет не просто ввести команду `make`, а набрать `make desktop` или `make pda`, и соответствующий исполняемый модуль появится на выходе. (Имя после `make` — это правило, которое она должна собрать.)

Это значительно лучше, чем заводить для каждой цели свое дерево исходного кода. Одновременное сопровождение нескольких деревьев, в которых бóльшая часть кода совпадает, — утомительная и чреватая ошибками задача. Очень легко, модифицировав код, позабыть сделать это со всеми его экземплярами.[2]

В чем же различие между этими целевыми правилами? Фактические различия могут быть такими:

- Собираются разные файлы (скажем, `save_release.c` или `save_demo.c`).
- Компилятору передаются различные макроопределения (например, компилятор определяет макрос `DEMO_VERSION`, чтобы выбрать правильный код `#ifdef` в `save.c`).
- Используются различные опции компилятора (например, включающие поддержку отладки).
- Для сборки выбираются разные наборы инструментов или типы окружения (например, выбирается компилятор, соответствующий целевой платформе).

Поскольку можно создать любое количество целей, учитывающих самые разнообразные различия, система может стать сложной и громоздкой. Некоторые варианты настройки можно задать с помощью параметров конфигурации сборки. Конфигурирование можно частично осуществлять при сборке кода и даже во время выполнения. Такой

---

[1] В этом случае механизм меняется: одновременно можно собрать только одну цель, поэтому «демонстративность» станет конфигурацией сборки, а не целью. Ниже будет сказано о конфигурациях.

[2] Обратите внимание на отличие такого опасного способа от хранения нескольких ветвей проекта в системе контроля версий. Системы контроля версий предоставляют средства для разнесения модификаций по ветвям и сравнения ветвей между собой.

Жизнь после make

Значительная часть рассматриваемых здесь вопросов относится к циклу разработки в стиле C, в котором компилятор генерирует объектный код и библиотеки из исходных файлов, а затем из них собирается окончательный исполняемый модуль. Для некоторых языков характерна другая модель. В Java процедура сборки значительно проще; компилятор javac берет на себя функции make, автоматически выполняя проверки зависимостей. Он более стесняет вас, навязывая определенную структуру дерева сборки, но в результате облегчает вам жизнь.

Простым программам, написанным на Java, не нужна сложная система сборки: всю сборку можно успешно выполнить одной командой `javac`. Однако в проектах Java покрупнее часто применяют make. Мы уже знаем, что сборка не ограничивается компилированием исходного кода. Необходим механизм для подготовки вспомогательных файлов, выполнения автоматизированных тестов и создания конечного дистрибутива. Make – это удобная среда для того, чтобы избавиться от такой работы, поэтому не стоит считать ее избыточной.

способ предпочтительнее, если он сокращает количество различных существующих сборок, требующих тестирования.

Уборка

Для каждого целевого правила, которое вы определили, должно быть соответствующее правило *чистки*, которое аннулирует все результаты сборки – удаляет выполняемый модуль программы, промежуточные библиотеки, объектные файлы и все прочие файлы, созданные во время сборки. Дерево исходного кода должно вернуться в свое исходное состояние, что относительно легко проверяется.[1]

Отсюда следует, что система сборки, которая физически изменяет исходные файлы, плоха – как потом обращать эти изменения? Следует использовать исходные файлы в качестве шаблона и записывать все изменения в другой выходной файл.

Правила чистки (clean) – это полезное средство для уборки. Они позволяют легко убрать весь мусор и выполнить сборку с чистого листа, если кажется, что вас преследуют темные силы сборки.

---

[1] Выполните сборку, затем чистку, а потом проверьте, отличается ли состояние дерева от исходного.

Для каждого правила сборки напишите соответствующее правило чистки, которое отменяет всю операцию.

Зависимости

Как может система сборки узнать, что одни файлы зависят от других? Без экстрасенсов это трудновато, поэтому мы попросим помочь тех, кто знает.

Вы сами указываете сведения о зависимостях в правилах, содержащихся в make-файлах. Make может построить из зависимостей дерево и двигаться по нему – прочесть временные метки всех файлов и выяснить, какие участки нужно собрать заново после проведенной модификации.

Это достаточно просто для правила сборки исполняемого модуля – нужно лишь указать, какие объектные файлы и библиотеки входят в него. Однако не хотелось бы скрупулезно указывать зависимости для каждого из исходных файлов; наверняка в них включено много других файлов директивой `#include`, а в тех есть свои `#include`. Список будет внушительный. Легко можно ошибиться при наборе, а также отстать от времени: добавить еще один `#include` и забыть при этом изменить make-файл.

Кому же все-таки известна вся информация о зависимостях? Компилятору; это тот элемент системы сборки, который реально прослеживает все зависимости между исходными файлами. У каждого хорошего компилятора есть опция, заставляющая его выдать все данные о зависимостях. Задача в том, чтобы написать для make правило, которое соберет эту информацию, поместит ее в правильно форматированный файл, а потом включит *его* в дерево зависимостей.

Автоматическая сборка

Если ваша сборка устроена в виде единой процедуры, запускаемой одной командой, вы легко можете организовать ночные сборки полного исходного дерева.[1] Во время регулярных ночных сборок полная процедура применяется ко всему коду, наработанному в течение дня. Это очень полезная практика, дающая многие преимущества:

- Каждое утро вы получаете свежий экземпляр своего творения. Разработчики целый день занимаются своими делами и забывают синхронизировать свой код с кодом коллег. Данная технология без труда обеспечивает тестирование интеграции и проверяет, что все элементы должным образом согласуются друг с другом.
- Проблемы сборки выявляются на ранних стадиях без вашего дополнительного труда. Сев утром за рабочий стол с чашечкой кофе в ру-

[1] Запускать команды в нужное время позволяют утилита cron в UNIX и Планировщик (Scheduled Tasks) в Windows.

ке, вы видите, в каком состоянии дерево исходного кода – допускает ли оно сборку. Вы сразу увидите, где требуются исправления, а не станете ждать, когда завершится ваша собственная сборка.

- В ночную сборку можно добавить автоматические регрессивные и нагрузочные тесты. Это хороший способ проверить дееспособность кода, прежде чем кто-то попытается запустить его. Днем у вас может не быть времени, чтобы проводить полное тестирование с каждой сборкой – теперь же вы никогда его не пропустите. Это мощное средство проверки.
- Ночную сборку можно использовать как мерило продвижения вашего проекта. Выложите результаты ночного тестирования, и по мере того как будет проходить все больше тестов, у разработчиков будет расти чувство внутреннего удовлетворения.
- Можно использовать ночную сборку для выпуска окончательных версий продукта. Ее результаты можно считать надежными, потому что на них не могли повлиять неправильно введенные команды, настройки или другие ошибки, связанные с человеческим фактором.
- Наличие такой сборки показывает, что вы действительно умеете собирать продукт и процедура сборки являет собой единое целое. Без автоматической сборки нельзя быть уверенным, что сборка не зависит от каких-то дополнительных факторов, например, когда один из разработчиков вручную чистит дерево сборки.

Организуйте автоматическую процедуру сборки своего программного продукта. Проверяйте с ее помощью работоспособность вашего кода.

Автоматическая сборка особенно полезна, если система велика (сборка занимает не один час) или над ней одновременно работает много людей (и не у каждого разработчика в любой момент есть самый свежий вариант исходного кода системы).

Полезная практика для ночных сборок – запись выводимых процедурой сборки сообщений в *журнал*, который всем доступен для просмотра. Можно даже рассылать его электронной почтой, если сборка оказалась неудачной, чтобы обратить внимание на проблемы. Очень важно следить за тем, что происходит при каждом проведении сборки, особенно при возникновении сбоев.

Ночная сборка становится как бы пульсом разрабатываемого проекта. Если сборки удачны, значит, код развивается правильно и успешно. Есть важное правило, которому следуют во многих проектах: *не порть дерево исходного кода* – внесение разработчиком в систему кода, из-за которого срывается ночная сборка, влечет суровое и неприятное наказание (желательно, с общественным порицанием). Другое правило: *если сборка не прошла, это общая проблема*. При сбое ночной сборки все разработчики должны отложить все свои дела и заняться устранением ошибки.

Можно довести автоматизацию сборки до крайней степени, заставив ее выполняться при любом изменении исходного дерева. Такая тактика называется *непрерывной интеграцией* и служит мощным средством проверки правильности кода и возможности его сборки в любой момент.

Конфигурация сборки

Хорошая система сборки позволяет настраивать некоторые аспекты каждого проведения процедуры. Настройку можно осуществлять с помощью параметров IDE, но в make-файлах она обычно заключается в определении переменных. Переменные могут быть получены из разных мест:

- Унаследованы из среды
- Заданы в командной строке make
- Явно определены внутри make-файла

Переменные конфигурации обычно используются следующим образом:

- Определяется переменная `PROJECT_ROOT`, указывающая на корень дерева сборки. Она позволяет системе сборки узнать, где искать различные файлы, например определить пути к файлам заголовков. Не хотелось бы жестко прописывать место, в котором находится дерево сборки на машине разработчика. В этом случае его нельзя было бы никуда переместить и нельзя было бы работать одновременно с двумя деревьями.
- Другие переменные могут указывать, где находится каждая из внешних библиотек (благодаря чему можно опробовать при сборке разные их версии).
- Переменные могут определять, какого типа сборку нужно выполнить (например, *для разработчиков* или *окончательную*).
- Команды для вызова каждого инструмента сборки (компилятора, редактора связей и пр.) можно поместить в переменные. Благодаря этому можно опробовать разные параметры командной строки или инструменты разных производителей.

Значения по умолчанию можно устанавливать в make-файле. Это решает две задачи: документирование возможных вариантов и *возможность опускать* значения параметров конфигурации.

Рекурсивное применение make

Исходный код обычно располагается в разных каталогах. Если в большом проекте сбрасывать все файлы в один каталог, проект быстро станет неуправляемым. Раз ветвится дерево исходного кода, система сборки тоже будет ветвиться. Такая вложенность не осложняет жизнь и позволяет более гибко организовать сборку.

make-файл, находящийся в одном каталоге, может вызывать make-файлы, находящиеся в дочерних каталогах, с помощью новых команд

`make` – точно так же, как он вызывает компилятор. Это распространенный прием, известный как *рекурсивная make*; система сборки спускается в каждый подкаталог и собирает в нем компоненты, а потом возвращается в верхний каталог и собирает компоненты в нем. В результате можно ввести команду `make`, находясь в корневом каталоге проекта, и собрать весь продукт, либо ввести ее, находясь в подкаталоге, и выполнить частичную сборку. Что хотите, то и соберете.

Рекурсия make помогает разделить пакет на компоненты и собирать их раздельно, но при этом вносит некоторые проблемы. Она выполняется медленнее (так как запускает много дочерних процессов для обхода подкаталогов), и, поскольку каждая дочерняя make видит только свою часть всего дерева, сведения о зависимостях могут оказаться неверными. Будьте осторожны, столкнувшись с рекурсивной make – предпочтительней нерекурсивные системы сборки. (Подробнее об этом см. ответ на вопрос 7 этой главы на стр. 628.)

Отпусти меня...

Некоторые сборки имеют особую важность и требуют более тщательной подготовки. Это *финальные сборки*, выполняемые со специальной задачей, а не в ходе разработки. Финальная сборка может быть связана с волнующим событием: выпуском бета-версии, первым официальным выпуском продукта или выпуском обновления продукта. Это также может быть внутренняя контрольная версия разработчика или промежуточная версия для отдела тестирования; такие версии не выходят за стены компании, но отношение к ним такое же серьезное, как к выпускаемым версиям – что-то вроде учебной тревоги перед окончательным выпуском.

Если система сборки изготовлена тщательно, для выпуска окончательной версии не требуется специальной подготовки. Однако к этим важным сборкам нужно подходить продуманно, чтобы исключить влияние каких-либо проблем при сборке окончательного исполняемого модуля. Вот главное, что нужно учитывать при проведении финальной сборки:

- Финальные сборки должны проводиться на чистом дереве исходного кода, а не на частично собранном дереве кого-то из разработчиков. Начните с чистого листа. Необходимо точно знать, в каком состоянии находятся файлы исходного кода, участвующие в сборке. Не следует полагаться на то, что файлы на машине Джо в «достаточно хорошем» состоянии.
- Перед началом сборки выполняется специальная процедура, во время которой указывается, какой исходный код с какими конкретными версиями файлов включить в выпуск. Они помечаются особым образом, обычно с помощью тегов или меток в системе кон-

троля версий. После этого набор файлов для сборки финальной версии можно получить в любой момент.

- У каждой финальной сборки есть собственное название, которое вы ей дадите; оно может быть красивым кодовым названием или просто номером сборки. Оно должно совпадать с той меткой, которой вы пометили исходные файлы. Если два разработчика договорились, что при поиске ошибки они имеют в виду «сборку номер пять», то между ними все согласовано. Если вы работаете с пятой сборкой, а я нашел ошибку в шестой, то как можно быть уверенным, что мы встретим одни и те же проблемы?

Финальные сборки всегда выполняются из чистого исходного кода. Позаботьтесь, чтобы потом всегда можно было получить этот чистый исходный код из архива или системы управления версиями.

Дерево познания (исходного кода)

Весь код находится в дереве исходного кода: файловой структуре, хранящей каталоги и файлы с исходным кодом. Структура этого дерева определяет простоту работы с кодом. В аккуратно организованной иерархии гораздо легче разобраться, чем в беспорядочной куче файлов. Располагая исходные файлы в виде определенной структуры, мы можем облегчить себе разработку. Эта древовидная структура тесно связана с системой сборки, потому что система сборки физически является частью дерева исходного кода (поэтому слова «дерево сборки» и «дерево исходного кода» означают одно и то же). Модификация одного требует вмешательства в другое.

Мы разбиваем код на отдельные модули, библиотеки и приложения. В правильном дереве исходного кода отражена эта структура. Отдельные части кода нужно аккуратно отобразить в файлы, а каталоги использовать в качестве механизма их логической группировки. Это помогает управлять проектом, в котором участвуют несколько разработчиков; каждый из них может работать в своем отдельном каталоге на разумном и безопасном расстоянии от тех, в которых работают другие.

Библиотеки

Поместите все библиотеки в свои отдельные каталоги. Создайте такую структуру каталогов, которая отделит интерфейс библиотеки (общедоступные файлы заголовков) от закрытых деталей реализации. Хорошим решением будет разместить открытый API в каталоге, который входит в путь поиска компилятора, а закрытые заголовки хранить в другом месте.

Приложения

Здесь структура проще: нет общедоступных файлов как таковых, а есть просто набор файлов с исходным кодом, которые компонуются с библиотеками. Но и тут лучше разместить каждое приложение в собственном каталоге, чтобы обозначить его границы. Если приложение настолько велико, что состоит из отдельных частей, их нужно разместить по подкаталогам или даже библиотекам и собирать отдельно. Дерево сборки при этом должно отражать структуру программы.

Код сторонних разработчиков

Дерево исходного кода должно четко отделять *ваш код* от работы сторонних разработчиков. Проекты все чаще включают в себя чужой код: стандартные библиотеки берут где-то в другом месте (у коммерческих производителей, из открыто разрабатываемых проектов или даже из других подразделений компании). Эти внешние файлы нужно держать отдельно.

Прочие данные

Документация по программе может располагаться в дереве исходного кода. Поместите ее в каталоги, соседствующие тем, где находится относящийся к ней код. То же касается графики и различных вспомогательных файлов.

- После сборки кода могут потребоваться дополнительные действия по созданию пакета, например подготовка CD, добавление документации, включение лицензионной информации и т. п. Этот этап тоже следует автоматизировать.
- Каждую сборку нужно сохранить в архиве для возможного контроля в будущем. Очевидно, исполняемый модуль финальной сборки нужно сохранить в том виде, в каком он поставляется пользователю (в виде Zip-файла, саморазворачивающегося EXE или ином). По возможности следует сохранить и дерево сборки в его финальном состоянии, но часто оно слишком велико для этого.
- По крайней мере, следует сохранить *журнал сборки*, точную последовательность введенных команд и полученные на них ответы. Эти журналы позволят вам вернуться к старым сборкам и выяснить, какие ошибки компилятора были пропущены и что именно происходило во время сборки. Это может оказаться полезным, если появится сообщение о сбое в старой версии, выпуск которой давно прекращен.
- У каждой финальной версии есть *сопроводительная записка*, в которой описываются сделанные модификации. Она может быть адресована клиенту или другим лицам в зависимости от назначения сборки. Эти записки тоже нужно сохранить. Обычно в записке опи-

сываются изменения, выполненные после предыдущей финальной версии, и перечисляются обновления, не попавшие в последний печатный вариант документации, отмечаются известные проблемы, даются инструкции по обновлению и т. д. Это важная часть процедуры выпуска финальной версии, которую нельзя пропустить.

- При выполнении окончательной сборки нужно правильно выбрать параметры компилятора – они могут отличаться о тех, которые устанавливались во время разработки. Например, отключается поддержка отладки. Следует также выбрать уровень оптимизации кода. Во время разработки оптимизация может отключаться, потому что она занимает так много времени, что для больших деревьев сборки становится неприемлемой. Следует учесть, что, затребовав максимальный уровень оптимизации, вы рискуете столкнуться с ошибками компилятора, которые приведут к сбоям, поэтому уровень нужно выбирать осторожно и тестировать полученный код.

  При использовании разных параметров компилятора для обычной и финальной сборок проявляйте осторожность. Регулярно тестируйте финальные сборки, не дожидаясь приближения окончательного срока сдачи продукта. Старайтесь минимизировать различия между финальной и обычной сборками.

Проводите тестирование финальной конфигурации своего приложения, а не только рабочих сборок. Небольшие различия между ними могут отрицательно сказаться на поведении кода.

Ввиду того что осуществление финальной сборки является относительно сложной задачей, которую очень важно выполнить правильно, обычно назначается ответственный за нее (он может быть кодировщиком или входить в группу контроля качества). Этот человек выполняет все финальные сборки для проекта, чтобы гарантировать высокое качество каждой из них. Окончательные сборки столь же зависимы от правильного выполнения процедуры, сколь от качества системы сборки.

Мастер на все руки

Во многих организациях есть конкретный работник, занимающийся сборкой, которого часто называют *мастером сборки* (*buildmaster*). Его работа состоит в сопровождении системы сборки. В его обязанности могут входить планирование и реализация графиков сборки, либо он выполняет чисто технические функции. Мастер сборки должен глубоко разбираться в системе сборки. Он может настраивать систему, добавлять в нее нужные новые цели, сопровождать сценарии ночной сборки и т. д. Мастер сборки также ведет документацию по системе сборки, а возможно, и администрирует систему контроля версий.

Мастер сборки выполняет финальные сборки и поэтому часто занимается слежением за стабильностью компонент. В его обязанности вхо-

дит обеспечение надежности и безопасности процесса выпуска окончательной версии продукта.

Мастер сборки не всегда нанимается на полный рабочий день; иногда его обязанности по совместительству выполняет кто-то из программистов.

Резюме

Ломать проще, чем строить.

Латинская пословица

На первый взгляд кажется, что в процедуре сборки программного обеспечения нет ничего сложного, если правильно выбрать инструменты. Но этими инструментами нужно уметь пользоваться. Качество системы сборки имеет первостепенное значение; в отсутствие безопасной, надежной процедуры сборки разработать крепкий код невозможно. Выпустить надежный конечный продукт еще сложнее – для этого требуются тщательность и точность процедуры. Важно разбираться в том, как работает система сборки, даже если не требуется ежедневно вносить в нее изменения.

Правильное выполнение сборки не простая задача, и пресловутая бесчисленная стая обезьян нас, программистов, не заменит. А обезьянки пусть спорят, какой из их бесконечного числа редакторов лучше.

Хорошие программисты...

- Разбираются в том, как работает их система сборки, как ей пользоваться и как ее расширять
- Создают простые, целостные системы сборки и размещают их рядом с исходным кодом
- Автоматизируют как можно больше операций сборки
- Применяют ночные сборки для обнаружения проблем с интеграцией

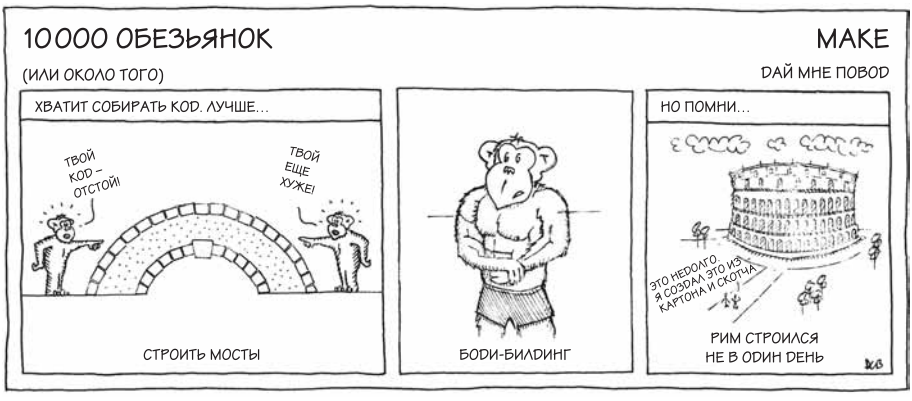
Плохие программисты...

- Не обращают внимания на то, как действует система сборки, а потом сталкиваются с глупыми проблемами сборки
- Не обращают внимание на отсутствие безопасности и надежности в своей системе сборки
- Почти с враждебностью требуют, чтобы новички освоили их причудливую систему
- Лепят финальную сборку, не придерживаясь строгой процедуры выпуска продукта

См. также

Глава 9. Поиск ошибок

Описывает, что делать с ошибками при сборке.



Глава 18. Защита исходного кода

Дерево сборки хранится в системе контроля версий и тесно связано с кодом.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 624.

Вопросы для размышления

1. Зачем при наличии удобной интегрированной среды обращаться к утилите командной строки make, если среда позволяет собрать проект, нажав одну кнопку?
2. Почему необходимо рассматривать извлечение исходного кода как отдельный этап перед сборкой?
3. Куда следует записывать промежуточные (например, объектные) файлы, возникающие при сборке?
4. Следует ли при добавлении в систему сборки автоматизированного набора тестов запускать его автоматически после сборки или вводить для запуска тестов отдельную команду?
5. Какую сборку выполнять ночью – *отладочную* или *финальную*?
6. Напишите для make правило, которое будет автоматически генерировать информацию о зависимостях с помощью компилятора. Покажите, как пользоваться этой информацией в make-файле.
7. Рекурсивная сборка – популярный метод создания модульной системы сборки, охватывающей несколько каталогов. Однако ей свойственен ряд недостатков. Опишите их и предложите альтернативные методы.

Вопросы личного характера

1. Умеете ли вы выполнять различные виды компиляции с помощью своей системы сборки? Каким образом можно собрать из одного и того же источника и с помощью одних и тех же make-файлов отладочную или финальную версию продукта?
2. Хороша ли система сборки в вашем текущем проекте? Обладает ли она качествами, о которых говорилось в этой главе? Можно ли ее усовершенствовать? Легко ли будет:
   a. Добавить новый файл в библиотеку?
   b. Добавить новый каталог в код?
   c. Удалить или переименовать файл с кодом?
   d. Добавить новую конфигурацию сборки (например, для демонстрационной версии)?
   e. Собрать две конфигурации по одному экземпляру исходного кода, не зачищая дерево в промежутке между сборками?
3. Приходилось ли вам когда-либо создавать систему сборки с самого начала? Чем вы руководствовались при ее конструировании?
4. Всем приходится иногда сталкиваться с недостатками системы сборки. При написании сценария сборки можно точно так же допустить ошибки, как и при программировании обычного кода.

   С какими сбоями системы сборки вы сталкивались и как смогли исправить или даже предотвратить их?

11

Жажда скорости

Оптимизация программ и составление эффективного кода

В этой главе:

- Почему важна эффективность кода
- Проектирование эффективного кода
- Повышение эффективности имеющегося кода

В этой жизни есть вещи куда важнее скорости.

Махатма Ганди

Наша культура – это культура быстрого питания. Не только наш обед запаздывает на сутки; наш автомобиль должен быть быстрым, а развлечение – мгновенным. И код должен выполняться со скоростью молнии. Мне нужен результат. Здесь и *немедленно*.

Ирония заключается в том, что для создания быстрых программ требуется много времени.

Оптимизация – это угроза для разработки программного обеспечения, как отмечал известный ученый Уильям Вульф: «Во имя эффективности (при этом часто без достижения желаемого результата) совершается больше преступлений, чем в силу любых других причин, включая элементарную глупость». (Wulf 72)

Оптимизация – избитая тема, по поводу которой высказались уже все, и рекомендации

предлагаются одни и те же. И несмотря на это, существует масса недоработанного кода. Идея оптимизации правильна, но программисты *обычно* понимают ее неверно: они отвлекаются соблазном эффективности, пишут плохой код, стремясь к производительности, оптимизируют там, где в этом нет необходимости, или выбирают неправильные типы оптимизации.

Этими проблемами мы и займемся в данной главе. Мы по-новому взглянем на привычные вещи. Не беспокойтесь: когда речь идет об оптимизации, разговор оказывается непродолжительным...

Что такое оптимизация?

Оптимизация подразумевает улучшение чего-либо, усовершенствование. В нашем деле она обычно означает, что «код станет выполняться быстрее», если мерить производительность программы с секундомером. Но это лишь одна сторона. У разных программ разные требования, и что «хорошо» для одной, не обязательно «хорошо» для другой. На практике оптимизация программного обеспечения может иметь разный смысл:

- Увеличение скорости выполнения программы
- Уменьшение размера исполняемого модуля
- Совершенствование качества кода
- Повышение точности результатов
- Сокращение времени запуска
- Увеличение производительности обработки данных (не обязательно связанное с ростом скорости выполнения)
- Уменьшение расходов на хранение данных (например, уменьшение размера базы данных)

Народная мудрость в отношении оптимизации суммирована Джексоном (M. A. Jackson) в злобных законах оптимизации:

1. Не занимайтесь ею.
2. (*Только для специалистов*) Пока не занимайтесь.

Иными словами, всеми силами избегайте оптимизации. Не занимайтесь ею с самого начала и задумайтесь только в конце, если окажется, что код работает недостаточно быстро. Это упрощенная точка зрения – в какой-то мере верная, но способная оказаться вредной и гибельной. На самом деле, производительность должна приниматься во внимание с самого начала разработки, прежде чем будет написана хоть одна строка кода.

Эффективность кода определяется рядом факторов, в число которых входят:

- Платформа, на которой исполняется код
- Конфигурация установленного продукта
- Решения по архитектуре программного пакета

Рассказы бывалого человека

Однажды я столкнулся с тем, что написанный мною модуль работал невообразимо медленно. Я сделал профилирование и свел проблему к единственной строчке кода. Она вызывалась часто и добавляла в буфер один элемент.

Дальнейшее рассмотрение показало, что буфер (который я не писал, а получил в готовом виде) расширял себя на один элемент всякий раз, когда оказывался полным! Иными словами, при каждом добавлении элемента осуществлялось выделение памяти, копирование памяти и освобождение памяти целого буфера. Естественно, я на такое не рассчитывал.

Это пример того, как появляются неоптимальные программы: в результате роста. Никто ведь не пишет умышленно хромые программы. Но при объединении компонент в более крупную систему легко сделать неверные предположения о скоростных свойствах кода и столкнуться с неприятными неожиданностями.

- Конструкция модулей на нижнем уровне
- Исторические ограничения (например, необходимость взаимодействия со старыми частями системы)
- Качество каждой строки исходного кода

Некоторые из этих факторов являются основополагающими для программной системы в целом, и когда программа будет написана, решать проблему повышения ее эффективности будет трудно. Обратите внимание на то, что отдельные строки кода играют незначительную роль в сравнении с другими факторами, определяющими производительность. Проблемы производительности нужно рассматривать на каждом этапе разработки и решать по мере возникновения. В некотором смысле оптимизация, не будучи особо планируемым видом деятельности, должна быть предметом забот на всем протяжении разработки.

Об эффективности работы программы нужно думать с самого начала – не надейтесь, что в конце разработки вам удастся повысить ее ценой небольших изменений.

Но не пользуйтесь этим как предлогом, чтобы писать мудреный код исходя из своих представлений о том, что работает быстро, а что нет. Интуитивные представления программистов о том, какая часть кода снижает эффективность программы, часто оказываются ошибочными независимо от накопленного опыта. Ниже мы рассмотрим некоторые практические решения этой дилеммы написания кода.

Но сначала золотое правило. Прежде чем хоть в какой-то мере заниматься оптимизацией, вспомните следующий совет:

Корректность кода гораздо важнее его скорости. Что толку быстро получить результат, если он неверный!

Стоит потратить больше сил и времени на обеспечение корректности кода, а не его скорости. Любая последующая оптимизация не должна нарушить правильность кода.

От чего страдает оптимальность кода?

Чтобы улучшить свой код, нужно знать, отчего он может работать медленно, неоправданно расти в размере или иным образом терять качество. В дальнейшем это поможет нам предложить некоторые методы оптимизации кода. Но сейчас полезно уяснить, с чем мы собираемся бороться.

Сложность

Неоправданная сложность губит код. Чем больше работы должен сделать код, тем медленнее он будет выполняться. Сокращение объема работ или разбиение задачи на несколько более простых и коротких могут значительно повысить производительность.

Косвенность

Превозносится как решение всех проблем программирования и формулируется в виде известной программистской максимы: *любую проблему можно решить, добавив уровень косвенности*. Но косвенность также часто является причиной медлительности кода. Эта критика часто звучит со стороны процедурных программистов старого закала в адрес современного ОО-проектирования.

Повторение

Повторение обязательно снижает эффективность кода, но часто его можно избежать. Оно случается в разных формах, например, если не сохранять в буфере результаты трудоемких вычислений или вызовов удаленных процедур. При каждом повторно выполненном вычислении вы теряете эффективность. Повторение участков кода неоправданно увеличивает размер выполняемого модуля.

Плохой проект

Это неизбежно: плохое проектирование приводит к плохому коду. Например, если разнести далеко друг от друга связанные блоки (скажем, разместив их в разных модулях), то их взаимодействие замедлится. Плохой проект может привести к возникновению фундаментальных, очень тонких и сложных проблем с производительностью.

Ввод/вывод

Взаимодействие программы с внешним миром – ее ввод и вывод – это известное узкое место. Если выполнение программы блокируется в ожидании ввода или вывода данных (пользователем, с диска или через сеть), она неизбежно теряет в производительности.

Это далеко не полный список проблем, но он дает представление о том, на что нужно обращать внимание при написании оптимального кода.

Доводы против оптимизации

В прежние времена оптимизация имела огромное значение, поскольку первые компьютеры работали крайне медленно. Чтобы заставить программу выполниться в течение разумного промежутка времени, требовались высокое мастерство и тонкая шлифовка отдельных машинных команд. Сейчас такое мастерство утратило былую ценность; революция, осуществленная персональными компьютерами, изменила характер разработки программ. Часто мы располагаем избытком вычислительной мощи – в полную противоположность минувшим временам. Может сложиться впечатление, что оптимизация больше не нужна.

Но это не совсем так. Индустрия программного обеспечения по-прежнему ставит нас в положение, когда требуется высокопроизводительный код, и если допустить небрежность, то в последний момент могут потребоваться невероятные усилия по оптимизации. Но если ситуация позволяет, предпочтительнее не прибегать к оптимизации кода. У оптимизации есть *масса* недостатков.

За скорость всегда приходится платить. Оптимизация кода – это выбор одного желательного качества в ущерб другому. В каком-то отношении код неизбежно пострадает. При правильном выборе улучшается то качество, которое признано более ценным. Приведу основные причины, по которым следует избегать оптимизации кода:

Потеря легкости чтения кода

Очень редко оптимизированный код читается так же легко, как его более медленный аналог. По самой своей сути оптимизированная версия не является простой и непосредственной реализацией логики. Ради производительности вы жертвуете легкостью чтения и аккуратностью конструкции кода. Чем сильнее «оптимизирован» код, тем он уродливее и труднее для понимания.

Рост сложности

Более изобретательная реализация, в которой могут применяться особые тайные средства (увеличивающие взаимозависимость модулей) или специфичные для платформы методы, увеличивает сложность. Сложность – враг хорошего кода.

Сложность сопровождения/расширения

Вследствие возросшей сложности и трудности чтения сопровождение кода затрудняется. Если алгоритм неясен, ошибки в коде труднее обнаруживаются. Оптимизация – верный способ добавить новые ошибки, которые будет трудно обнаружить в силу того, что код становится менее ясным и прослеживаемым. Оптимизация ведет к появлению опасного кода.

Кроме того, она становится препятствием на пути расширения вашего кода. Оптимизация часто основывается на дополнительных допущениях, ограничивая общность и возможности дальнейшего роста.

Появление конфликтов

Часто оптимизация базируется на особенностях платформы. Определенные операции становятся быстрее на одной платформе, но не затрагивают другие. Выбор оптимальных типов данных для одного процессора может привести к замедлению работы на других.

Дополнительные затраты

Оптимизация – это лишний труд. Спасибо, у нас и так достаточно забот. Если код работает правильно, мы должны обратить свое внимание на более насущные проблемы.

Оптимизация кода отнимает много времени, и работать с подлинными причинами не так легко. Если вы оптимизировали не то, что требовалось, вы зря потратили свои ограниченные силы.

Поэтому оптимизация не должна занимать слишком высокое место в списке ваших забот. Сравните, что для вас важнее – оптимизация, или исправление ошибок, или добавление новых функций, или поставка продукта в срок. Часто оптимизация оказывается неуместной или невыгодной. Если вашей первейшей задачей становится написание эффективного кода, вам вообще едва ли потребуется его оптимизировать.

Альтернативы

Часто оптимизацию кода проводят, когда в ней по-настоящему нет необходимости. Есть ряд альтернатив, к которым можно прибегнуть, не меняя имеющегося высококачественного кода. Попробуйте следующие решения, *прежде чем* зацикливаться на оптимизации:

- Нельзя ли смириться с существующим уровнем производительности – действительно ли он настолько вас не удовлетворяет?
- Запустите программу на более быстрой машине. Это может показаться слишком простым, но если вы в какой-то мере влияете на выбор платформы, на которой будет выполняться программа, экономически более выгодным может оказаться выбор быстрого компьютера, чем трата времени на совершенствование кода. С учетом средней продолжительности реализации проекта можно быть уверенным, что к моменту его окончания процессоры станут заметно быстрее. А если не намного быстрее, то количество ядер ЦП на том же физическом пространстве точно удвоится.

  Не все проблемы решаются повышением скорости ЦП, особенно если узким местом является не скорость выполнения, а, скажем, скорость обмена данными с внешней системой. Иногда быстрый ЦП может привести к значительному снижению производительности: быстрое выполнение может усугубить проблемы блокировки потоков.

- Поищите аппаратные решения. Добавьте специальный блок для операций с действительными числами, чтобы ускорить вычисления, увеличьте кэш процессора, объем оперативной памяти, повысьте скорость работы сети или поставьте быстрый контроллер жестких дисков.
- Перенастройте целевую платформу, чтобы снизить нагрузку на ЦП. Отключите фоновые задачи или ненужное оборудование. Старайтесь, чтобы ваши процессы экономно расходовали память.
- Выполняйте медленный код асинхронно, в фоновом режиме. Добавлять новые потоки в последнюю минуту – рискованное занятие, если вы не сильны в этом вопросе, но тщательно разработанные потоки могут очень неплохо справляться с медленными операциями.
- Поработайте с элементами интерфейса пользователя, от которых зависит субъективное восприятие скорости. Пусть кнопки GUI меняются мгновенно, даже если код, запускаемый ими, выполняется секунду или более. Для медленных задач сделайте индикаторы прогресса: программа, которая замирает во время долгой операции, кажется зависшей. Визуальное отображение хода операции создает лучшее впечатление качества работы.
- Проектируйте системы так, чтобы они не требовали вмешательства оператора, и тогда никто не обратит внимания на скорость выполнения. Сделайте программу пакетной обработки с толковым интерфейсом, который позволит ставить задачи в очередь.
- Попробуйте применить другой компилятор с более агрессивной оптимизацией либо нацельте свой код на специализированную версию процессора (со всеми дополнительными командами и расширениями), чтобы выжать из него максимальную производительность.

Рассмотрите альтернативы оптимизации; не удастся ли вам повысить эффективность программы иными способами?

Нужна ли оптимизация

Уяснив, что с оптимизацией связаны опасности, не выкинуть ли навсегда из головы идею оптимизировать свой код? Не стоит. Хотя оптимизации следует по возможности избегать, однако во многих случаях она весьма важна. И есть области, в которых она оказывается просто *необходимой*.

- Программирование игр всегда требует высокой эффективности кода. Несмотря на огромный прогресс в мощности ПК, рынок требует все более реалистичной графики и эффективных алгоритмов искусственного интеллекта. Добиться этого можно только путем предельной эксплуатации всех возможностей исполнительной среды. Это очень напряженная область деятельности: если появляется новый, более

быстрый образец аппаратуры, программистам игр все равно приходится выжимать из его производительности все до последней капли.

- Программирование *цифровой обработки сигналов* (*DSP*) в высшей степени зависит от высокой производительности. Цифровые процессоры сигналов – это специальные устройства, оптимизированные для быстрой цифровой фильтрации больших объемов данных. Без высокой скорости они были бы никому не нужны. Программирование DSP обычно в меньшей мере полагается на оптимизирующие свойства компилятора, поскольку здесь требуется высокая степень контроля над работой процессора в каждый момент времени. Программисты DSP достигли большого мастерства в получении максимальной производительности от этих устройств.
- Среды с ограниченными ресурсами, например глубоко вложенные платформы, могут иметь трудности с достижением приемлемой производительности на имеющейся аппаратуре. Чтобы добиться терпимого качества обслуживания или уместить код в ограниченную память устройства, может потребоваться тщательная оптимизация кода.
- Системы *реального времени* требуют своевременного выполнения и возможности завершения операций в течение заданного временного интервала. Алгоритмы, гарантирующие выполнение в течение фиксированного времени, требуют тщательной доводки.
- Численное программирование – в финансовом секторе или научных исследованиях – требует высокой производительности. Эти громадные системы работают на крупных компьютерах со специальной поддержкой работы с числами, в том числе векторных операций и параллельных вычислений.

В программировании общих задач об оптимизации можно не беспокоиться, но есть много ситуаций, когда она имеет решающее значение. Требования к производительности редко указываются в проектной документации, но клиент может выразить неудовольствие, если ему покажется, что программа работает слишком медленно. Если другие способы не помогают, а код выполняется недостаточно быстро, придется заняться его оптимизацией.

Аргументов в пользу оптимизации меньше, чем против нее. Если нет особой необходимости оптимизировать код, не занимайтесь этим. Но если приходится оптимизировать, нужно знать, как делать это правильно.

Выясните, действительно ли есть необходимость оптимизировать код, но лучше с самого начала писать эффективный код высокого качества.

Технические подробности

Каким же образом выполнять оптимизацию? Гораздо важнее уяснить правильный *подход* к оптимизации, чем выучить список конкретных

методов. Не бойтесь, мы рассмотрим ниже некоторые приемы программирования, но их следует рассматривать в контексте более широкой процедуры оптимизации.

Вот шесть этапов для повышения скорости работы вашей программы:

1. Убедитесь, что программа слишком медленно работает и требует оптимизации.
2. Определите, какая часть кода самая медленная, и нацельте на нее свои усилия.
3. Проверьте производительность кода, выбранного для оптимизации.
4. Оптимизируйте код.
5. Протестируйте оптимизированный код и убедитесь, что он сохранил работоспособность (очень существенно).
6. Проверьте, насколько выросла скорость, и решите, что делать дальше.

Может показаться, что это слишком трудоемко, но иначе вы просто зря потратите время и силы и получите искалеченный код, который работает ничуть не быстрее. Если ваша цель не в повышении скорости выполнения программы, а другая, соответствующим образом поправьте предложенную процедуру. Например, если есть проблема нехватки памяти, определите, какие из структур данных пожирают всю вашу память, и займитесь в первую очередь ими.

Важно приступать к оптимизации, имея перед собой четкую цель, потому что чем сильнее оптимизирован код, тем труднее его читать. Решите, какой уровень производительности вам нужен, и остановитесь, приблизившись к нему. Всегда есть соблазн пойти еще дальше, выжать еще немного дополнительной мощности.

Чтобы выполнить оптимизацию корректно, вы должны позаботиться об отсутствии влияния внешних факторов на работу вашего кода. Если условия будут меняться, вы не сможете правильно сравнить характеристики своего кода. Здесь помогают два важных приема:

Оптимизируйте код отдельно от всякой прочей работы, чтобы результаты одной работы не оказывали влияния на другую.

...и...

Оптимизируйте окончательные версии программы, а не промежуточные сборки.

Промежуточные сборки могут работать совсем не так, как окончательные, в силу наличия данных для трассировки, символов объектных файлов и т. д.

Рассмотрим каждый из этапов оптимизации более подробно.

Убедитесь, что нужна оптимизация

Прежде всего нужно проверить, действительно ли требуется оптимизация. Если производительность кода приемлема, нет смысла ковырять его. Как сказал Кнут (цитируя C.A.R. Hoare): «Следует забыть о недостаточной эффективности примерно в 97% случаев; необдуманная оптимизация – корень всех зол». Есть столько убедительных оснований не делать оптимизацию, что самый быстрый и безопасный метод оптимизации – убедиться, что она вам не требуется.

Свое решение вы должны принимать на основании технических требований к программе или исследования ее юзабилити (usability). После этого вы можете определить, что для вас важнее – оптимизация или добавление новых функций и исправление ошибок.

Определите самую медленную часть кода

В этом месте большинство программистов совершает ошибку. Если вы собираетесь заниматься оптимизацией, нужно определить те точки, в которых она имеет смысл. Согласно некоторым исследованиям, в среднем 80% времени программы проводят в 20% своего кода (Boehm 87). Данное правило известно как *правило 80/20*.[1] Это относительно небольшая часть кода, и она может оказаться незатронутой вашей оптимизацией, в результате чего все ваши усилия окажутся малоэффективными, поскольку оптимизированным окажется код, который редко выполняется.

Возможно, вы обнаружите, что какая-то часть программы может быть относительно легко оптимизирована, но если она редко выполняется, то в оптимизации нет смысла; это тот случай, когда понятность кода предпочтительнее скорости его работы.

Как же определить, на что следует обратить внимание? Наиболее эффективно воспользоваться *профайлером*. Такая программа проводит хронометраж работы различных участков вашей программы. Она покажет вам, куда уходит 80% времени работы программы, поэтому вы узнаете, куда направить свои силы.

Вопреки распространенному заблуждению, профайлер *не* покажет вам части кода, которые работают медленно. На самом деле, он сообщит вам, где процессор проводит большую часть времени, а это не совсем одно и то же.[2] Не исключено, что программа проводит больше всего времени в нескольких совершенно приличных функциях, которые никак нельзя усовершенствовать. Не все доступно оптимизации; иногда приходится считаться с законами природы.

---

[1] Некоторые идут еще дальше, считая, что правило должно гласить «90/10».

[2] Код всегда выполняется с фиксированной скоростью, зависящей от тактовой частоты ЦП, количества прочих процессов, которыми манипулирует ОС, и приоритетом процесса.

Существует масса превосходных программ для измерения эффективности – коммерческих и бесплатно распространяемых. Затраты на покупку хорошего профайлера оправдывают себя: оптимизация может отнять у вас много времени. Но это и дорогой продукт. Если профайлер окажется для вас недоступен, есть ряд других технологий хронометража, которыми можно воспользоваться.

- Разместите сами в своем коде контроль времени исполнения. Проверьте, чтобы источник временных отсчетов был точен и чтобы затраты на чтение времени не слишком влияли на производительность программы.
- Подсчитайте число вызовов каждой функции (некоторые библиотеки для отладки обеспечивают поддержку такого рода действий).
- Воспользуйтесь ловушками, предоставляемыми компилятором, для вставки собственного кода, ведущего учет входа в функции и возврата из них. Такие возможности предоставляются многими компиляторами; в некоторых профайлерах реализован такой же механизм.
- Воспользуйтесь счетчиком команд; периодически прерывайте программу в отладчике, чтобы выяснить, где находится управление. Это труднее осуществить в многопоточных программах, и такой способ очень медленный. Если у вас есть доступ к среде выполнения, можно написать вспомогательные программы для автоматизации такого тестирования – фактически собственный вариант профайлера.
- Проверьте, как влияет отдельная функция на скорость выполнения программы, заставив ее выполняться медленнее. Если подозрения в причинах замедленности падают на определенную функцию, попробуйте вызвать ее два раза подряд вместо одного и посмотрите, как это влияет на время выполнения.[1] Если программа станет выполняться на 10% дольше, то функция занимает примерно 10% времени работы. Это может быть самым примитивным способом хронометража.

Во время профилирования нужно вводить реальные входные данные, моделирующие реальные события. Характер выполнения кода может в очень большой степени зависеть от того, какие данные и как подаются на вход, поэтому потрудитесь составить правдоподобные наборы входных данных. Еще лучше взять входные данные от реальной системы.

Попытайтесь провести профилирование при разных наборах входных данных и посмотрите, отличаются ли при этом результаты. Возьмите самый простой набор данных, трудный набор и несколько обычных

[1] Это не означает, что функция непременно станет работать вдвое медленнее. Эффективность повторяющихся участков кода может вырасти за счет буферизации файловой системы или кэширования памяти ЦП. Эту оценку следует воспринимать как очень приблизительную – скорее качественную, чем количественную.

наборов. Тем самым вы убережетесь от оптимизации, нацеленной на отдельные необычные наборы входных данных.

Тщательно отберите входные данные для профилирования, с тем чтобы они отражали реальный мир. В противном случае может оказаться, что вы оптимизируете те части программы, которые в обычных условиях не выполняются.

Несмотря на то что профайлер (или аналогичные средства) может послужить хорошей отправной точкой для выбора объектов оптимизации, можно легко упустить ряд фундаментальных проблем. Профайлер показывает лишь, как выполняется код в текущей реализации, и способствует усовершенствованиям только на уровне кода. Стоит взглянуть на общие проблемы проекта. Неудовлетворительная производительность может быть обусловлена не особенностями реализации отдельной функции, но более глубокими ошибками проектирования. В таком случае решение проблемы потребует дополнительных усилий. Это показывает, насколько важно правильно спроектировать код с самого начала с учетом требуемых показателей эффективности.

Не ограничивайтесь профайлером при поиске причин недостаточной эффективности программы; они могут оказаться более глубокими.

Этот этап покажет вам, какие участки кода следует исправить, чтобы добиться наибольшего повышения эффективности. Займемся этими участками.

Тестирование кода

Мы выделили три этапа тестирования в процедуре оптимизации. Для каждого участка кода (объекта оптимизации) мы: 1) проверим его эффективность, прежде чем начать модификацию, 2) убедимся, что код корректно работает после оптимизации, и 3) протестируем его эффективность после модификации.

Программисты часто упускают из виду вторую из проверок, которая должна подтвердить, что оптимизированный код работает корректно во *всех* возможных случаях. Легко проверить обычный режим работы, но, как правило, не возникает желания проверять все редкие ситуации. Именно поэтому могут возникать непонятные сбои на поздних этапах разработки, поэтому здесь недопустимы послабления.

Эффективность кода *нужно* проверять до и после модификации, чтобы убедиться в том, что ваши модификации возымели эффект, притом положительный; иногда «оптимизация» невольно оказывается *деградацией*. Хронометраж можно осуществить с помощью профайлера или самодельных измерительных инструментов.

Обязательно проведите измерения до и после оптимизации.

Вот несколько очень важных советов, касающихся проведения замеров времени:

- Используйте одни и те же входные данные в тестах, проводимых до и после оптимизации. В противном случае ваши тесты окажутся бессмысленными; вы будете сравнивать хрен с апельсином. Лучше всего воспользоваться автоматизированным контрольным примером (см. «Руками не трогать!» на стр. 203) – с такого же рода реальными репрезентативными данными, как при профилировании.
- Выполняйте тесты в обычных одинаковых условиях, чтобы исключить влияние на результат таких факторов, как загрузка ЦП или объем свободной памяти.
- Ваши тесты не должны зависеть от того, насколько быстро пользователи вводят данные, поскольку скорость работы пользователей может меняться в широком диапазоне. Постарайтесь автоматизировать все, что только удастся.

Оптимизация кода

Ниже будут рассмотрены некоторые конкретные приемы оптимизации. Ускорение может быть достигнуто в результате как простой переработки небольшого раздела кода, так и серьезных изменений на уровне проекта. Задача в том, чтобы оптимизировать код, не поломав его при этом.

Определите, какие способы существуют для оптимизации данного кода, и выберите из них лучший. Вносите изменения по одному; это менее рискованно и позволяет определить, от чего в большей степени зависит производительность. Иногда наибольший эффект оптимизации дают совершенно неожиданные факторы.

После оптимизации

Не забудьте провести тестирование оптимизированного кода, чтобы убедиться в успешности сделанных модификаций. Если оптимизация не удалась, отмените сделанные изменения. Тут вам поможет система управления версиями, с помощью которой вы вернетесь к предыдущему варианту кода.

Отмените также те изменения, которые *незначительно* оптимизировали код. Лучше иметь ясный код, чем скромные результаты оптимизации (если, конечно, у вас нет абсолютной необходимости ускорить код и отсутствуют другие способы сделать это).

Методы оптимизации

Мы долго обсуждали общие вопросы; пора заняться конкретными деталями. Следуя описанной выше процедуре оптимизации, вы убедились, что программа работает неэффективно, и нашли код, больше всего повинный в этом. Необходимо переделать его. Что для этого нужно?

Существуют различные виды оптимизации. Какой из них подойдет в вашем случае, зависит от конкретной проблемы, вашей цели (например, увеличить скорость или уменьшить размер кода) и требуемой степени оптимизации.

Можно выделить две категории оптимизаций: изменение *проекта* и изменение *кода*. Изменения на уровне проекта обычно оказывают более значительный эффект на производительность, чем модификации на уровне кода. Неудачный проект может гораздо больше снизить эффективность, чем несколько неудачных строк исходного кода, поэтому конструктивные изменения, хотя и более трудные, дают большую отдачу.

Чаще всего возникает задача повысить скорость выполнения. Для повышения скорости предлагаются следующие пути:

- Заменить медленный код быстрым
- Реже обращаться к медленному коду
- Откладывать выполнение медленного кода до того момента, когда оно реально потребуется

Другие распространенные задачи оптимизации – уменьшение требуемой памяти (в основном путем изменения представления данных, модификации схемы работы с памятью или сокращения объема данных, одновременно находящихся в памяти) или уменьшение размера выполняемого модуля (путем сокращения функциональности или удаления повторяющегося кода). Как будет продемонстрировано, эти задачи часто противоречат одна другой: рост скорости часто происходит за счет увеличения потребления памяти, и наоборот.

Конструктивные изменения

Имеются в виду *макрооптимизации* – крупные изменения, направленные на улучшение внутренней конструкции вашего программного продукта. Исправлять плохой проект тяжело. Чем ближе срок завершения проекта, тем менее вероятно, что вы станете вносить конструктивные изменения; это слишком рискованно.[1] В итоге мы ограничиваемся заделкой обнаружившихся трещин, делая небольшие заплатки на уровне кода.

Если отважиться на конструктивные изменения, то для оптимизации можно прибегнуть к следующему:

- Ввести дополнительное кэширование или буферизацию, чтобы ускорить доступ к данным или избежать длительных повторяющихся вычислений. Можно заранее вычислить величины, которые вам потребуются, и запомнить их, чтобы иметь к ним мгновенный доступ.
- Создать пул ресурсов для сокращения расходов на выделение памяти объектам. Например, можно заранее выделить память или дер-

[1] Увы, обычно обращают внимание на то, что эффективность программы недостаточна, лишь тогда, когда завершение проекта уже на носу.

жать открытыми файлы, вместо того чтобы многократно открывать и закрывать их. Такой прием часто используется для ускорения выделения памяти; процедуры выделения памяти старых ОС проектировались для применения в простом однопоточном режиме. Их блокировки очень сильно тормозят многопоточные приложения.

- Пожертвовать точностью ради скорости, если это допустимо. Например, отказаться от вычислений с числами с плавающей точкой. В некоторых устройствах отсутствует *блок для вычислений с плавающей точкой* (*FPU*) и применяется их программная эмуляция, которая медленна. Можно выбрать библиотеки арифметики с фиксированной точкой, чтобы избежать эмуляции, но потерять при этом точность вычислений. Особенно просто это делается в C++ с помощью абстрактных типов данных.

  Точность связана не только с типами данных; эту тактику можно развить гораздо глубже, выбирая те или иные алгоритмы или качество представления результирующих данных. Можно также оставить решение на усмотрение пользователя – выбрать режим *медленной, но точной* работы или *быстрой, но приближенной*.
- Изменить формат хранения данных или их представления на диске, сделав его более подходящим для скоростной обработки. Например, ускорить синтаксический анализ текстового файла, воспользовавшись двоичным форматом. Передавать или хранить файлы в сжатом виде, чтобы повысить эффективность их передачи по сети.
- Применить распараллеливание задач и многопоточность, чтобы одно действие не ждало окончания выполнения другого. По мере того как иссякают возможности роста скорости процессоров, их производители все чаще выпускают процессоры с несколькими ядрами и несколькими конвейерами. Чтобы воспользоваться их преимуществами, ваш код должен быть спроектирован с учетом многопоточной модели. Борьба за оптимизацию активно осваивает это направление.
- Эффективно организовать потоки: избегать или удалять лишние блокировки. Они препятствуют параллельному выполнению, требуют дополнительных накладных расходов и часто приводят к взаимным блокировкам. С помощью статического анализа проверьте, какие блокировки вам необходимы, а какие нет.
- Не злоупотреблять исключениями (exceptions). Они могут помешать компилятору выполнить оптимизацию[1] и нанести ущерб быстродействию, если применяются слишком часто.
- Отказаться от определенных средств языка, если это поможет сократить объем кода. Некоторые компиляторы C++ позволяют

[1] Как и функции, блоки `try/catch` представляют собой препятствие для оптимизатора. Через них не видно, есть ли возможность оптимизации, поэтому некоторые приемы ускорения могут оказаться не реализованными.

отключить RTTI и исключительные ситуации, тем самым уменьшив размер исполняемого модуля.

- Ограничить функциональность: самый быстрый код тот, который вообще не выполняется. Функция будет работать медленно, если она делает много вещей, которые не нужны. Уберите все лишнее. Перенесите его в какое-нибудь другое место. Откладывайте выполнение всякой работы до того времени, когда она действительно понадобится.
- Пожертвовать качеством конструкции ради скорости. Например, сократить косвенность и увеличить связывание. Для этого можно нарушить правила инкапсуляции: предоставить открытый интерфейс для закрытой реализации класса. Разрушая границы между модулями, вы наносите непоправимый ущерб всему проекту. По возможности попытайтесь сначала прибегнуть к менее разрушительным средствам.

Обозначения для сложности

Алгоритмическая сложность – это мера того, насколько хорошо масштабируется алгоритм: какова зависимость времени его выполнения от объема входных данных. Это качественная математическая модель, позволяющая быстро сравнить эффективность различных способов реализации. Она не показывает конкретное время выполнения (оно сильно зависит от скорости ЦП, настроек ОС и т. д.).

Сложность определяется как объем работы, которую должен осуществить алгоритм: количество выполненных элементарных операций. Элементарными операциями считаются арифметические действия, присваивание, проверка, чтение/запись данных и т. п. Алгоритмическая сложность не показывает, сколько именно операций должно быть выполнено, а отражает связь этой величины с размерностью задачи. Обычно нас интересует поведение алгоритма в худшем случае, т. е. самый большой объем работы, который ему может потребоваться проделать. Для сравнения неплохо также знать сложность в лучшем случае и в среднем.

Алгоритмическая сложность обозначается с помощью «О большого» – системы, придуманной немецким специалистом по теории чисел Эдмундом Ландау. Задача с размером входных данных `n` может иметь сложность:

`O(1)`**: порядка** `1`

Это алгоритм с *постоянным временем*. Каков бы ни был объем входных данных, выполнение задачи всегда требует одного и того же времени. Это лучший показатель эффективности из всех возможных.

$O(n)$**: порядка** n

Сложность алгоритма с *линейным временем* пропорциональна размеру входных данных. Поиск в связанном списке требует обхода все большего количества узлов по мере роста списка; количество операций прямо пропорционально длине списка.

$O(n^2)$**: порядка** n **в квадрате**

В этом случае дело с производительностью гораздо хуже: сложность растет быстрее, чем объем входных данных. Алгоритм с *квадратичным временем* может быть совсем не плох, когда объем входных данных для него невелик, но для больших наборов данных он может выполняться очень долго. Алгоритм пузырьковой сортировки имеет сложность $O(n^2)$.

Порядок сложности может быть любым; например, алгоритм быстрой сортировки имеет сложность $O(n \log n)$, что хуже, чем $O(n)$, но гораздо лучше, чем $O(n^2)$. Простой способ оптимизации медленного алгоритма пузырьковой сортировки – это замена его алгоритмом быстрой сортировки, тем более что множество его реализаций находится в свободном доступе.

Обозначения в виде «О большого» не содержат констант или членов более низкого порядка. Вы едва ли встретите сложность типа $O(2n+6)$. Когда n достаточно велико, эти константы и члены более низкого порядка становятся несущественны.

Основные оптимизации уровня проекта включают в себя усовершенствование *алгоритмов* или *структур данных*. В большинстве случаев неудовлетворительная скорость или чрезмерный расход памяти обусловлены недостатками одного или обоих из указанных факторов, над которыми и следует работать.

Алгоритмы

Алгоритмы оказывают огромное влияние на скорость выполнения программы. Функция, которая удовлетворительно работает в небольшом локальном тесте, может не справиться с реальными данными. Если профилирование показывает, что ваш код значительную часть времени проводит в некоторой функции, нужно заставить ее работать быстрее. Можно работать с ней на уровне кода, выжимая все, что можно, из каждой команды. Но лучше заменить весь алгоритм, найдя более эффективный его вариант.

Рассмотрим конкретный пример. Пусть некоторый алгоритм выполняет цикл 1000 раз. Каждый проход тела цикла занимает 5 миллисекунд. Вся операция, таким образом, длится около 5 секунд. Допустим, модифицировав код внутри цикла, вы сократите каждую итерацию на 1 мс; это даст вам выигрыш в 1 секунду. Неплохо.

Но лучше, если найдется алгоритм, у которого каждая итерация длится 7 мс, но зато их требуется повторить всего 100 раз. Тогда вы сэкономите 4,5 секунды, что существенно лучше.

Поэтому при оптимизации старайтесь модифицировать целые алгоритмы, а не возиться с отдельными строчками кода. В вычислительной науке разработана масса алгоритмов, и вы всегда добьетесь наиболее значительного прироста эффективности путем выбора более удачного алгоритма, если только у вас не какой-то очень специфичный код.

Лучше заменить медленный алгоритм быстрым, чем пытаться улучшить реализацию имеющегося алгоритма.

Структуры данных

Структуры данных тесно связаны с выбором алгоритмов; некоторые алгоритмы требуют определенных структур данных, и наоборот. Если программа требует слишком много памяти, изменение формата хранения данных может поправить дело, хотя часто это отрицательно сказывается на скорости выполнения. Если нужно быстро провести поиск в списке из 1000 элементов, не храните их в линейном массиве со скоростью поиска $O(n)$ – воспользуйтесь бинарным деревом (которое больше) и поиском за $O(\log n)$.

Если вы предпочтете другую структуру данных, вам вряд ли придется самостоятельно реализовывать новое представление. Для большинства языков существуют библиотеки, поддерживающие все стандартные структуры данных.

Модификация кода

Наконец, мы боязливо приблизились к самому противному: оптимизации на микроуровне – мелкому, близорукому ковырянию в коде. Есть много способов атаки на исходный код с целью повысить его эффективность. Только опыт покажет, какой из них эффективнее в конкретной ситуации: одни модификации полезны, другие не очень или даже имеют отрицательный результат. Некоторые изменения кода могут помешать компилятору выполнить оптимизацию и привести к неожиданно плохим результатам.

Первая задача простая: включите оптимизацию в компиляторе или повысьте ее уровень. Оптимизацию часто отключают при внутренних сборках, потому что она может выполняться очень долго, на порядок увеличивая время сборки крупного проекта.[1] Попробуйте изменить параметры оптимизации и посмотрите, какой эффект это будет иметь.

[1] Она включает в себя сложный анализ кода с целью выявления возможных способов ускорения работы и выбора наиболее подходящих из них.

Многие компиляторы позволяют определить цель оптимизации – ускорение работы или уменьшение размера кода.

Есть несколько типов оптимизации самого низкого уровня, которые нужно знать, но по возможности избегать. Это те модификации, которые компилятор может выполнить вместо вас. Если оптимизатор включен, он сразу станет искать такие возможности – достаточно активизировать оптимизацию, и вы получите полную поддержку в этой области. Едва ли вам потребуется применять такую оптимизацию вручную, и это хорошо: она затрудняет чтение кода, поскольку совершенно видоизменяет его базовую логику. Выполняйте такого рода оптимизацию только тогда, когда совершенно *уверены*, что она необходима, что оптимизатор не сделал ее сам и что нет лучших альтернатив.

Развертывание циклов

Если у цикла очень короткое тело, то структура, образующая цикл, может обойтись дороже, чем сама циклическая операция. Избавьтесь от накладных расходов, сделав структуру плоской – замените цикл из 10 итераций на 10 последовательных операторов.

Развертывание циклов может быть частичным, что бывает оправдано для больших циклов. Можно выполнить в каждой итерации четыре операции и каждый раз увеличивать счетчик цикла на четыре. Однако такой прием оказывается затруднителен, если количество итераций цикла не кратно количеству развернутых операций.

Встраивание кода

При выполнении коротких операций накладные расходы на вызов функции могут оказаться чрезмерными. С помощью разбиения кода на функции достигаются значительные преимущества: более понятный код, единообразие в результате многократного использования и возможность модификации в отдельной области. Однако для увеличения быстродействия можно освободиться от функций и объединить вызывающий код с вызываемым.

Для этого есть несколько способов. При наличии соответствующей поддержки в языке можно сделать это в исходном коде (в C/C++ с помощью ключевого слова `inline`), при этом в значительной мере сохраняется читаемость кода. В противном случае придется соединять код самостоятельно, либо многократно копируя функцию, либо делая это с помощью препроцессора.

Затруднительно встраивание обращений к рекурсивным функциям – как узнать, когда нужно остановиться? Попытайтесь заменить рекурсию другими алгоритмами.

Встраивание часто открывает возможность дальнейшей оптимизации на уровне кода (ранее невозможной внутри границ функции).

Свертывание констант

Вычисления с использованием констант можно проводить во время компиляции, что сокращает объем действий, проводимых на этапе выполнения. Простое выражение типа `return 6+4;` можно свести к `return 10;`. В результате упорядочения членов большого выражения можно свести вместе две константы, что позволит сократить выражение.

Обычно программисты не пишут таких очевидных вещей, как `return 6+4;`. Однако такого рода выражения часто встречаются после расширений макросов.

Перенос действий в этап компиляции

На этапе компиляции можно не только свертывать константы. Можно статически проверять условия и удалять лишнее из кода. От ряда проверок можно вообще избавиться, например убрать проверку отрицательных значений, использовав беззнаковые типы данных.

Понижение прочности

Речь идет о замене операции другой, которая выполняется быстрее. Это особенно эффективно на ЦП со слабой поддержкой арифметики. Например, заменить умножение или деление целых чисел на сдвиг или сложение; `x/4` можно преобразовать в `x>>2`, если эта операция выполняется быстрее на вашем процессоре.

Вложенные выражения

Устранение одинаковых вложенных выражений позволяет избежать повторного вычисления выражений, значения которых не изменились. В коде

```
int first = (a * b) + 10;
int second = (a * b) / c;
```

выражение `(a * b)` вычисляется два раза. Достаточно одного. Можно выделить общее выражение и заменить его на

```
int temp   = a * b;
int first = temp + 10;
int second = temp / c;
```

Удаление «мертвого» кода

Не пишите лишнего кода; уберите все, что не является необходимым для работы программы. Статический анализ покажет функции, которые никогда не вызываются, и участки кода, которые никогда не получают управления. Удалите их.

Это были весьма неприятные типы оптимизации, но те, что приводятся ниже, более приемлемы. Они нацелены на ускорение работы программы.

- Если обнаружится, что некая медленная функция вызывается многократно, постарайтесь обращаться к ней реже. Сохраните ее результат и пользуйтесь этим значением. Код может стать менее понятным, но программа заработает быстрее.
- Перепишите эту функцию, выбрав другой язык. Например, перепишите критическую Java-функцию на C, воспользовавшись интерфейсом Java Native Interface (JNI). Обычные компиляторы все еще превосходят JIT-интерпретаторы по скорости.

  Не будьте наивны, думая, что один язык быстрее другого – многих программистов удивило, сколь малую выгоду приносит использование JNI. Часто утверждают, что ОО-языки гораздо медленнее соответствующих процедурных. Это неправда. Плохой ОО-код может быть таким же медленным, как и плохой процедурный код. Если вы станете писать код в ОО-стиле на C, то он, скорее всего, будет работать *медленнее*, чем код, правильно написанный на C++; компилятор C++ сгенерирует более эффективный код для вызова методов, чем вы сможете сделать это вручную.
- Повысьте эффективность, изменив порядок расположения частей кода.

  Откладывайте работу до того момента, когда ее выполнение становится совершенно необходимым. Не открывайте файл, пока он вам не понадобится. Не вычисляйте выражение, значение которого вам может не понадобиться; подождите, пока оно не будет востребовано. Не вызывайте функцию, пока код может работать без нее.

  Размещайте проверки как можно выше в функции, чтобы избавиться от ненужной работы. Если проверку, которая приводит к досрочному выходу из функции, можно разместить в начале функции или где-то в середине, лучше сделать это в начале. Чтобы не терять зря времени, делайте проверки раньше.

  Выведите вычисления, дающие неизменный результат, за рамки цикла. Самый незаметный источник этой проблемы встречается в условии цикла. Если вы напишете `for (int n = 0; n < tree.appleCount(); ++n)`, но `appleCount()` пересчитывает 1000 объектов при каждом обращении, цикл окажется очень медленным. Выполните подсчет до входа в цикл:

```
int appleCount = tree.appleCount();
for (int n = 0; n < appleCount; ++n)
{
    ... какие-то действия ...
}
```

  Однако выполните сначала профилирование, чтобы убедиться в том, что этот цикл действительно тормозит. Это прекрасный пример того, что оптимизация зависит от конкретной среды исполнения: в C# новый вариант может оказаться *медленнее*, потому что неоп-

тимизированный код представляет собой паттерн, знакомый JIT-компилятору, и он может сам оптимизировать его.

- Пользуйтесь *справочными таблицами* (*lookup tables*) для сложных вычислений, экономя время за счет памяти. Например, вместо того чтобы писать тригонометрические функции и вычислять потом их значения, выполните вычисления заранее и поместите в массив. Отобразите входные значения в индексы этого массива.
- Пользуйтесь *укороченными вычислениями* (*short-circuit evaluation*). Размещайте проверки, отрицательный результат которых наиболее вероятен, в самом начале, чтобы сэкономить время. Если вы пишете условное выражение `if (condition_one && condition_two)`, то сделайте так, чтобы условие `condition_one` с большей вероятностью оказывалось невыполненным, чем `condition_two` (если только `condition_one` не выступает в качестве защиты для верности `condition_two`).
- Не нужно изобретать колесо – пользуйтесь стандартными функциями, которые давно оптимизированы. Авторы библиотек тщательно шлифуют свой код. Но учтите, что библиотека может быть оптимизирована по иным параметрам, чем те, которые вам нужны. Например, встроенный продукт может быть оптимизирован по объему необходимой памяти, а не по скорости.

Оптимизация на уровне кода с целью сокращения объема памяти может осуществляться такими способами:

- Создание архивированных исполняемых модулей, самораспаковывающихся при выполнении. Размер выполняемой программы может и не уменьшиться, но памяти для хранения понадобится меньше.[1] Это может оказаться полезным, если программу нужно хранить во флэш-памяти небольшого размера.
- Выделение стандартного кода в общую функцию и устранение дублирования кода.
- Перенесение редко используемых функций в другое место. Поместите их в динамически загружаемую библиотеку или в отдельную программу.

Разумеется, предельная оптимизация может быть достигнута, если вы перепишете не удовлетворяющий вас участок кода на ассемблере – тут вы получите *полный* контроль над процессором и возможность любых действий (включая самоубийственные). Это последнее средство, в котором практически никогда нет необходимости. Современные компиляторы производят вполне приемлемый код, и время, потраченное на написание, отладку и сопровождение «оптимизированных» разделов кода, не окупается достигаемым ростом эффективности.

---

[1] При этом может проявиться приятный побочный эффект в виде ускорения запуска программы: сжатый исполняемый модуль быстрее загружается с диска.

Как писать эффективный код

Если лучше всего *отказаться* от оптимизации, то как достичь того, чтобы потребности в повышении эффективности кода вообще не возникало? Для этого нужно *проектировать* код с учетом его *эффективности* и планировать надлежащее качество его работы с самого начала, а не обстругивать его в последнюю минуту.

Некоторые утверждают, что это опасный путь. Действительно, он таит в себе определенные угрозы, если не проявлять осторожность. Пытаясь оптимизировать код в процессе написания, вы спускаетесь на более низкий уровень. В результате возникает неприятный хакерский код с массой улучшающих эффективность приемов низкого уровня и потайных интерфейсов.

Как же примирить эти две, как может показаться, противоположные точки зрения? Без большого труда, потому что на самом деле они не противоречат одна другой. Есть две взаимодополняющие стратегии:

- Писать эффективный код
- Оптимизировать код потом

Если вы поставите себе задачу *сразу* писать ясный, хороший, эффективный код, не потребуется впоследствии существенно его оптимизировать. Есть мнение, что нельзя заранее знать, потребуется ли оптимизация, поэтому нужно писать *как можно проще*, а оптимизацию проводить только в том случае, если профилирование выявит узкие места.

Недостатки такого подхода очевидны. Если известно, что вам нужна структура данных с эффективными средствами поиска (поскольку программа должна быстро осуществлять поиск), следует выбрать бинарное дерево, а не массив.[1] Если такие требования перед вами не стоят, *тогда* выберите наиболее подходящее средство, которое решит задачу. Оно необязательно должно быть простейшим; обычный массив в C – это структура данных, которой трудно управлять.

Проектируя каждый модуль, не гонитесь бездумно за эффективностью – тратьте силы на оптимизацию, только если в этом есть необходимость. Уясните, какие требования предъявляются к производительности, и проверяйте на каждом этапе, удовлетворяют ли ваши решения этим требованиям. Если вы знаете, какой уровень производительности необходим, проще проектировать код, удовлетворяющий требованиям эффективности. Кроме того, вы можете написать конкретные тесты, чтобы проверить, достигнут ли заданный уровень производительности.

---

[1] Однако, как всегда, не все так просто. Массивы могут обеспечивать лучшую связность кэша, в то время как узлы бинарного дерева легко оказываются раскиданными по всей памяти. Стоит рассмотреть такой вариант, как сортированный массив (время тратится при вставке элементов). Мерить, мерить и еще раз мерить.

Пессимизации

Если не делать тщательных измерений, то ваши оптимизации легко могут оказаться отнюдь не оптимальными. Оптимизация, идеальная для одной ситуации, может резко снизить эффективность в другой. Приведем наглядный пример: оптимизация строк copy-on-write (копирование при записи).

В 1990 году данная оптимизация часто применялась к реализациям стандартных библиотек. Программы, интенсивно работавшие со строками, тратили огромные ресурсы при копировании длинных строк – как в отношении загрузки процессора, так и расхода памяти. Копирование длинных строк требует дублирования и перемещения большого объема данных. Автоматически генерируются многочисленные экземпляры строк, создаются временные объекты, которые вскоре выкидываются – в действительности они никогда не модифицируются. Дорогостоящая операция копирования требует излишних расходов.

При оптимизации copy-on-write (COW) строковый тип данных преобразуется в некий *умный указатель*: действительные данные строки хранятся в виде (скрытого) общего представления. В итоге копирование строки сводится к простому копированию умного указателя (прикреплению нового умного указателя к общему представлению) вместо дублирования всего содержимого строки. Только в случае модификации строки осуществляется копирование ее внутреннего представления, а также изменение умного указателя. Такая оптимизация позволяет избежать выполнения многочисленных операций копирования.

COW хорошо работала в однопоточных программах; она заметно увеличивала скорость работы. Однако при использовании COW-строк в многопоточных программах возникли проблемы. (На самом деле, проблема появляется даже в однопоточных программах, если класс строк COW собран с поддержкой многопоточности.) Реализация требует очень консервативной блокировки потоков во время операций копирования, и эти блокировки становятся *существенно* узким местом. Внезапно программа, которая раньше летала, начинает ползти. Оптимизация COW оказалась существенной пессимизацией.

В условиях многопоточности значительно лучшей производительности удалось достичь, когда вернулись к классической реализации строк и стали писать более тщательный код, который реже осуществлял автоматическое копирование строк. К счастью, в настоящее время изготовители библиотек C++ предоставляют более толковые варианты классов строк, которые поточно-ориентированы и быстро работают.

Вот некоторые простые конструктивные решения, которые увеличивают производительность и облегчают последующую оптимизацию:

- Минимизируйте применение функций, которые реализованы на удаленных машинах либо требуют обращения к сети или медленной системе хранения данных.
- Уясните, где будет устанавливаться программный продукт и как предполагается запускать программу, чтобы учесть эти условия при проектировании.
- Пишите *модульный* код, чтобы можно было оптимизировать отдельные блоки, не переписывая заново остальные.

Резюме

Прогресс технологии одаряет нас все более совершенными средствами для движения вспять.

Олдос Хаксли

Высокая эффективность кода не столь важна, как некоторым кажется. Хотя иногда *приходится* засучить рукава и поковыряться в коде, в целом нужно активно избегать проведения оптимизации. Поэтому до начала работы над программным продуктом выясните, какие требования предъявляются к его производительности. На каждом этапе разработки проверяйте, обеспечивается ли требуемое качество. Тогда оптимизация вам не понадобится.

Если вы все же проводите оптимизацию, делайте это в высшей мере методично и осторожно. Поставьте себе четкую цель и после каждого шага проверяйте, приблизились ли вы к ней. Опирайтесь на твердые данные, а не на свои догадки. Разрабатывая код, заботьтесь об эффективности своих решений, но не в ущерб качеству кода. Оптимизация на уровне кода оправдана только в случае особой необходимости.

Хорошие программисты...

- Прибегают к оптимизации только в случае крайней необходимости
- Проводят оптимизацию методически, подходя к ней взвешенно и осторожно
- Рассматривают альтернативные решения и возможности конструктивных усовершенствований, прежде чем решиться на оптимизацию на уровне кода
- Стремятся к тому, чтобы оптимизация не ухудшила качество кода

Плохие программисты...

- Начинают оптимизировать, не удостоверившись, что код недостаточно эффективен
- Бездумно набрасываются на код, который *кажется* им узким местом, не выполнив предварительных замеров и исследований
- Не задумываются над общей картиной: каковы будут последствия их оптимизации для остальных участков кода и моделей работы
- Считают, что скорость важнее качества кода

См. также

Глава 1. Держим оборону

Оптимизации, удаляющие «лишний» код, часто входят в конфликт с дополнительным защитным кодом.

Глава 4. Литературоведение

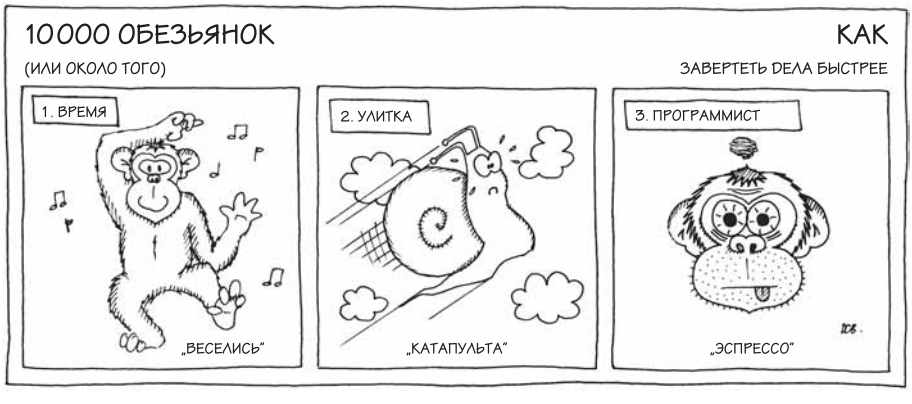
Оптимизация кода часто противоречит особенностям самодокументируемого кода.

Глава 13. Важность проектирования

Высокая производительность должна *закладываться* в код с самого начала проекта.

Глава 19. Спецификации

Требования к производительности должны быть тщательно определены до того, как начнется разработка, чтобы было понятно, в каком объеме может потребоваться оптимизация.



Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 633.

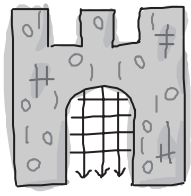
Вопросы для размышления

1. Оптимизация – это поиск компромисса, т. е. отказ от одного положительного свойства кода в пользу другого, более желательного. Опишите, какого рода компромиссы возникают в стремлении увеличить производительность программы.
2. Посмотрите на альтернативы оптимизации, перечисленные в разделе «Доводы против оптимизации» на стр. 271. Опишите, допускаются ли при этом компромиссы, и если да, то какие.

3. Объясните значение следующих терминов и связь между ними:
   - Производительность (Performance)
   - Эффективность (Efficiency)
   - Оптимизированность (Optimized)
4. Какие наиболее вероятные узкие места могут быть в медленно работающей программе?
5. Как устранить необходимость в оптимизации? Какие методы позволяют избежать появления неэффективного кода?
6. Как влияет на оптимизацию наличие нескольких потоков?
7. Почему мы пишем *неэффективный* код? Главное, что мешает нам сразу воспользоваться высокопроизводительными алгоритмами?
8. Тип данных `List` реализован с помощью массива. Какова будет алгоритмическая сложность следующих методов `List` в худшем случае?
   a. Конструктор
   b. `append` – добавление нового элемента в конец списка
   c. `insert` – вставка нового элемента в заданном месте между двумя существующими элементами
   d. `isEmpty` – возвращает `true`, если в списке нет элементов
   e. `contains` – возвращает `true`, если в списке есть указанный элемент
   f. `get` – возвращает элемент с указанным индексом

Вопросы личного характера

1. Насколько важна производительность кода в проекте, над которым вы работаете в данное время (только честно)? Чем обусловлено данное требование к производительности?
2. Когда вы последний раз проводили оптимизацию:
   a. Применяли ли профайлер?
   b. Если да, какой прирост эффективности он показал?
   c. Если нет, как вы выяснили, было ли достигнуто улучшение?
   d. Проверили ли вы работоспособность кода после проведенной оптимизации?
   e. Если да, насколько тщательно вы провели тестирование?
   f. Если нет, то почему? Можете ли вы быть уверены, что код по-прежнему правильно работает во *всех* случаях?
3. Если вы еще не занимались оптимизацией кода, над которым работаете в данное время, попробуйте угадать, какие его части самые медленные и какие потребляют больше всего памяти. Теперь пропустите свой код через профайлер и проверьте, насколько верны оказались ваши предположения.
4. В какой степени заданы требования к производительности вашей программы? Есть ли у вас конкретный план проверки того, что ваш код удовлетворяет этим критериям?



12

Комплекс незащищенности

Как писать защищенные программы

В этой главе:

- Угрозы безопасности для действующего кода
- Как взломщики могут злоупотребить вашим кодом
- Технология снижения уязвимости кода

Безопасность обычно оказывается предрассудком. Ее не существует в природе... Жизнь интересна, только когда она – смелое приключение.

Хелен Келлер

Не столь далеко в прошлое ушли времена, когда доступ к компьютеру был редкой возможностью. Во всем мире существовала лишь горстка машин, принадлежавших нескольким организациям, и работали на них небольшие группы высококвалифицированных специалистов. В те дни компьютерная безопасность состояла в надевании лабораторного халата и предъявлении пропуска охране у входа.

Возьмем сегодняшний день. Мы носим в карманах мини-компьютеры такой вычислительной мощности, о которой операторы прежних лет даже и не мечтали. Компьютеры во множестве присутствуют повсюду, и, как правило, между ними установлена хорошая связь.

Объем данных, циркулирующих в компьютерных системах, растет с фантастической скоростью. Мы пишем программы для хранения, обработки, интерпретации и передачи этих данных. Наше программное обеспечение не должно допускать попадания этой информации в чужие руки: к злоумышленникам, случайным людям или просто в эфир. Это очень важно; утечка из компании сверхсекретной информации может привести ее к финансовому краху. Нельзя, чтобы конфиденциальные личные данные (например, данные вашего банковского счета или кредитной карты) стали известны постороннему, который может ими воспользоваться. В большинстве компьютерных систем требуется некоторый уровень безопасности.[1]

Кто несет ответственность за безопасность создаваемого программного обеспечения? К несчастью, это *наша* забота. Если мы не уделим пристальное внимание безопасности результатов своего труда, то будем выпускать ненадежные, дырявые программы и пожнем все плоды этого.

Безопасность программного обеспечения – это большая проблема, с которой мы обычно плохо справляемся. Едва ли не каждый день становится известно об обнаружении новой уязвимости в системе защиты популярного продукта или компрометации систем в результате действия вирусов.

Это слишком обширная тема, чтобы подробно осветить ее в данной книге. Работа в этой области требует большой подготовки и опыта. Однако даже элементарные ее проблемы не находят адекватного отражения при обучении разработке программного обеспечения. Задача данной главы – осветить задачи системы безопасности, исследовать возникающие проблемы и рассказать о некоторых базовых технологиях защиты кода.

Риски

Лучше терпеть насмешки над чрезмерностью своих страхов, чем попасть в беду, переоценив свою безопасность.

Эдмунд Берк

Кому может понадобиться атаковать вашу систему? Тому, кто захочет воспользоваться тем, чем вы располагаете. Это могут быть:

- Вычислительные мощности.
- Возможность отправки данных (например, для рассылки спама).
- Ваши конфиденциальные данные.
- Ваши возможности, например определенное программное обеспечение, которое у вас установлено.

[1] Как мы увидим далее, это не зависит от того, работают ли они с секретными данными. Если у малозначительной компоненты есть открытый интерфейс, она несет в себе риск для безопасности системы в целом.

- Ваше подключение к представляющим интерес удаленным системам.

Вас могут атаковать просто потому, что вы кому-то не понравились и он хочет навредить вам, нарушив работу вашего компьютера. С одной стороны, вокруг полно злоумышленников, ищущих легкой добычи, а с другой – сама программа может ошибочно направить информацию не тому, кому она предназначалась. Тот, к кому она попала, может воспользоваться утечкой и нанести вам вред.

Следует учитывать, какими важными ресурсами вы располагаете. Есть ли у вас особо деликатная информация или специальные возможности, за которыми могут охотиться? Обеспечьте их защиту.

Чтобы понять, какого типа атакам вы можете подвергнуться, важно провести различие между защитой целой компьютерной *системы* (состоящей из нескольких компьютеров, сети и ряда взаимодействующих приложений) и написанием отдельных защищенных *программ*. То и другое – важные аспекты компьютерной безопасности; они сливаются между собой, поскольку оба необходимы. Второе – это часть первого. Достаточно одной небезопасной программы, чтобы сделать всю компьютерную систему (или сеть) незащищенной.

Вот стандартные риски системы защиты и способы компрометации действующих компьютерных систем:

- Вор, укравший ноутбук или PDA, может прочесть все незащищенные конфиденциальные данные. Украденное устройство может оказаться настроенным для автоматического входа в закрытую сеть, позволяя благополучно пройти все системы защиты вашей компании. Это серьезная угроза системе безопасности, которой не так просто противодействовать при написании кода! Но мы можем писать системы так, чтобы тот, кто украл компьютер, не получал к ним доступа сразу.
- Недостатки процедур ввода данных могут допускать злонамеренное их использование (эксплойт, exploit), что приводит к различным видам компрометации – вплоть до полного захвата машины атакующим (мы продемонстрируем это в разделе «Переполнение буфера» на стр. 303).

  Особенно неприятно проникновение через незащищенный открытый сетевой интерфейс. Если уязвимостью в GUI могут воспользоваться только те, кто фактически пользуется этим интерфейсом, то незащищенность системы, подключенной к открытой сети, может привести к тому, что к вам может попытаться вломиться кто угодно.
- *Подъем привилегий* (*privilege escalation*) возникает, когда пользователю с ограниченными правами доступа удается обмануть систему и получить себе права более высокого уровня. Атакующий при этом может быть законным пользователем или злоумышленником, проникшим в систему. Его конечной целью является получение

привилегий *суперпользователя*, или *администратора*, которые дают атакующему полный контроль над машиной.

- Если данные передаются в незашифрованном виде и проходят по незащищенным каналам (например, через Интернет), тогда любой встретившийся по дороге компьютер может прочесть не предназначенные для него данные – подобно прослушиванию телефонных разговоров. Разновидность этого случая носит название атаки *человек посередине* (*man-in-the-middle attack*): атакующая машина маскирует себя под участника связи и, находясь между корреспондентами, перехватывает данные их обоих.
- В каждой системе есть небольшая группа доверяющих друг другу участников. Злонамеренные законные пользователи могут сеять смуту, предоставляя другим лицам данные, для этого не предназначенные, или вводя данные, компрометирующие качество вашей компьютерной системы.

  Бороться с этим трудно. Приходится верить, что каждый пользователь ответственно подходит к тому уровню доступа к системе, который ему предоставлен. Если же пользователь не заслуживает доверия, вы не сможете исправить положение программным образом. Это показывает, что безопасность в такой же мере зависит от администрирования и политики, в какой от защищенности кода.
- Беспечные пользователи (или администраторы) могут сделать систему неоправданно открытой и уязвимой. Например:
  - Пользователи забывают разрегистрироваться; если нет таймаута для закрытия сеанса, кто угодно может позднее начать пользоваться вашей программой.
  - Многие злоумышленники применяют средства подбора паролей по словарям, которые выполняют многократные попытки зарегистрироваться в системе, пока один из паролей не сработает. Пользователи выбирают такие пароли, которые легко запомнить, но их легко и угадать. Уязвимой является любая система, которая допускает использование слабых, легко угадываемых паролей. Более надежные системы делают учетную запись пользователя временно недоступной, если несколько его попыток зарегистрироваться оказались безуспешными.
  - *Социальная инженерия* – умение получать важную информацию от людей, от находящихся в офисе предметов или даже из мусорного бачка – обычно действует гораздо проще и быстрее, чем проникновение в компьютерную систему. Человека обмануть проще, чем компьютер, и злоумышленники знают это.
  - В старых программах может обнаружиться много прорех. Многие поставщики ПО выпускают *извещения* о компрометации системы защиты и программные заплатки для них. Если администратор не следит за свежими сообщениями, система может оказаться уязвимой для атаки.

- Установка мягких прав доступа может открыть пользователям доступ к конфиденциальным частям системы – например, позволив им читать данные любых пользователей в ведомости выдачи зарплаты. Для исправления бывает достаточно изменить права доступа к файлам базы данных.
- Атаки вирусов (саморазмножающихся злонамеренных программ, обычно распространяемых через приложения к электронным письмам), трояны (скрытые злонамеренные участки в безобидных с виду программах) и шпионские программы (трояны, следящие за вашими действиями, тем, какие страницы вы посещаете, и пр.) заражают машины и могут вызывать самые разнообразные неприятности. Например, они могут с помощью регистраторов нажатий на клавиши получать пароли независимо от их сложности.
- Хранение данных «в открытом» (незашифрованном) виде – даже в оперативной памяти – небезопасно. Память не так безопасна, как кажется многим программистам; вирус или троян может просканировать память компьютера и вытащить из нее массу интересных вещей, которыми не преминет воспользоваться взломщик.

Риск увеличивается с ростом числа путей, которыми можно попасть в компьютер, количества методов ввода (доступ через Веб, командную строку или графические интерфейсы), отдельных средств ввода (различные окна, приглашения терминала, веб-формы или каналы XML) и числа пользователей (растут шансы на то, что кто-то вскроет пароль). Чем больше данных выводится, тем вероятнее проявление ошибок, имеющихся в коде вывода, приводящих к показу лишней информации.

Чем сложнее компьютерная система, тем более вероятно, что в ее системе безопасности есть пробелы. Следовательно, пишите как можно более простое программное обеспечение!

Наши оппоненты

Вероятно, трудно поверить в то, что кто-то не пожалеет сил и времени, чтобы попытаться взломать ваше приложение. Но такие люди есть. Они талантливы, целеустремленны и весьма терпеливы. Если вы хотите писать защищенные программы, следует знать своего противника. Тщательно разберитесь в том, что они делают, как это делают, какими инструментами пользуются и какие задачи перед собой ставят. Только тогда вы сможете выработать правильную стратегию.

Кто

Атакующий может оказаться обычным жуликом, талантливым взломщиком (cracker), *script kiddie* (презрительное название взломщиков, применяющих автоматические скрипты для взлома; они используют хорошо известные уязвимости, не вкладывая в них собственное творчество), бесчестным служащим, обманывающим свою

Как это по-научному...

В обсуждении проблем безопасности участвуют следующие важные термины:

Пробел (flaw)

Пробел в защите – это непреднамеренный дефект приложения. Это недостаток программы (см. «Терминология» на стр. 184). Не все пробелы представляют собой проблемы системы защиты.

Уязвимость (vulnerability)

Уязвимость возникает, когда из-за наличия пробела появляется возможность преодолеть защиту программы.

Эксплойт (exploit)

Это автоматизированное средство (или ручной метод), основанное на имеющейся в программе уязвимости и вызывающее ее непредусмотренное – и нарушающее безопасность – поведение. Не все уязвимости обнаружены или имеют эксплойты (по чистой случайности).

компанию, или бывшим служащим, мстящим за несправедливое увольнение с работы.

Взломщики (крэкеры, кракеры) хорошо подготовлены. Существует крэкерская субкультура, в которой они обмениваются информацией и простыми в эксплуатации инструментами для взлома. От того, что вы не знаете об этом, вы не станете невиннее и чище – просто окажетесь наивными и незащищенными против простейших атак.

Где

Благодаря широкому распространению сетей атака может прийти отовсюду – с любого континента и компьютера любого типа. Определить местонахождение злоумышленников, действующих через Интернет, очень трудно; обычно они умеют хорошо заметать следы. Часто сначала взламывают простые машины, чтобы воспользоваться ими как прикрытием для более дерзких атак.

Когда

Атака может произойти в любое время суток. Если вы на разных континентах с атакующим, то когда у одного из вас день, у другого ночь. Защищенные программы нужны вам в любое время суток, а не только в рабочее время.

Зачем

Потенциальных взломщиков так много, что и мотивы для атаки у них могут быть самыми разными. Цели могут быть преступными (нанести удар вашей компании по политическим мотивам или получить доступ к вашему банковскому счету) или хулиганскими (ка-

кой-нибудь шутник-соученик хочет поместить что-то забавное на вашем сайте). Возможно проявление любопытства (хакеру просто интересно узнать, как устроена ваша сеть, или попрактиковаться в своем искусстве взлома) или авантюризма (пользователь натыкается на данные, которые ему не полагается знать, и пытается извлечь из них какую-нибудь выгоду).

В мире, где почти все компьютеры объединены в одну сеть, обычно узнаешь о том, кто твой враг, уже после того, как он ударит. И даже тогда можно не узнать, кто это был; вашего искусства криминальной экспертизы может оказаться недостаточно, чтобы по еще теплой кучке цифровых останков восстановить картину происшедшего. Но, как юный пионер: *будь готов!* Не проходи мимо уязвимостей и не думай, что твоя машина никому не нужна; всегда кто-нибудь ей заинтересуется.

Не проходите мимо уязвимостей и не считайте себя непобедимым. Где-нибудь обязательно найдется тот, кто попытается применить эксплойт к вашему коду.

Крэкеры и хакеры

Эти два термина часто путают и используют неверным образом. Вот их правильное определение:

Крэкер

Тот, кто умышленно пользуется уязвимостями в компьютерных системах для получения несанкционированного доступа.

Хакер

Часто некорректно применяется для обозначения крэкера, но в действительности это тот, кто занимается хакерством – работой с кодом. Это имя в 70-х годах с гордостью носил определенный тип знатоков программирования. Хакер – это компьютерный специалист или энтузиаст. Употребляются еще два хакерских термина:

Герои

Хакеры «в белых шляпах» думают о том, к каким последствиям приводит их работа, осуждают действия крэкеров и неэтично ведущих себя пользователей компьютеров. Они считают, что трудятся на благо общества.

Разбойники

Это темные личности среди программистов, которые получают удовольствие от злоупотребления компьютерными системами. Это крэкеры, активно занимающиеся поиском путей непорядочного использования систем. Они не уважают прав других людей на собственность или личную жизнь.

Оправдания, оправдания

Как же атакующим удается так часто взламывать код? Они располагают оружием, которого у нас нет или о котором мы по недостатку образования ничего не знаем. Инструменты, знания, мастерство – все это работает на них. Однако они располагают еще одним основополагающим преимуществом – временем. В условиях промышленного производства программного продукта разработчики вынуждены вырабатывать максимальный объем кода, который только доступен их силам (или даже больше), и делать это в сжатые сроки, в противном случае последствия известны. Этот код должен удовлетворять определенным требованиям (функциональности, юзабилити, надежности и т. п.), а это оставляет крайне мало времени на заботу о «второстепенных» качествах, таких как защищенность. У атакующих таких проблем нет; у них достаточно времени, чтобы изучить все хитрости вашей системы, и они научились нападать с разных сторон.

Условия игры им благоприятствуют. Мы, разработчики программ, должны защитить все мыслимые места проникновения в систему; атакующий может выбрать самое слабое место и сосредоточить на нем свои усилия. Мы можем установить защиту только против известных эксплойтов; атакующий может, не жалея времени, отыскивать новые, неизвестные уязвимости. Мы должны постоянно быть настороже, ожидая нападения; атакующий может напасть, когда ему заблагорассудится. Мы обязаны писать хорошие и понятные программы, которые корректно взаимодействуют с окружающим миром; атакующий может поступать самым бессовестным образом.

Безопасность ПО ставит перед бедным, замученным программистом бесчисленное множество других, также важных проблем и задач. И что из этого следует? То, что мы *должны* их переиграть. Мы должны быть лучше информированы, лучше вооружены, лучше знать своего противника и лучше уметь писать код. Вопросы защищенности должны быть с самого начала учтены в проекте, им должно быть отведено должное место в процедуре разработки и в графиках работ.

Ощущение незащищенности

Задача программиста во всей этой сумятице – написать защищенный код, поэтому проведем обзор слабых мест в наших программах, чтобы определить, куда направить свои усилия. Существуют конкретные виды уязвимостей в коде – брешей, которыми может воспользоваться атакующий.

Опасный проект и архитектура

Это самая фундаментальная слабость, поэтому ее труднее всего исправить. Если вопросам безопасности не уделено должное внимание на

уровне архитектуры, то нарушения защиты будут преследовать вас всюду: передача незашифрованных данных в открытых сетях, хранение данных на легкодоступных носителях и пользование программными сервисами, пробелы в защите которых общеизвестны.

Вопросы безопасности нужно иметь в виду с самого начала разработки. Каждая компонента системы должна быть изучена с точки зрения наличия в ней незащищенных мест; безопасность компьютерной системы определяется надежностью ее самой слабой части, которой может оказаться даже не ваш код. Например, программа на Java не может быть более защищенной, чем JVM, на которой она выполняется.

Переполнение буфера

Большинство приложений обращено лицом к публике: они слушают открытый сетевой порт или обрабатывают данные, введенные через веб-броузер или графический интерфейс. Эти процедуры ввода данных – главное место, где происходит нарушение защиты.

В программах на C входные данные часто обрабатываются с помощью функции `sscanf`. Несмотря на то что эта функция принадлежит стандартной библиотеке C и регулярно встречается в C-коде, она совершенно беззастенчиво предлагает возможности для написания опасного кода.[1]

Можно встретить такой код:

```
void parse_user_input(const char *input)
{
    /* синтаксический анализ введенной строки input */
    int my_number;
    char my_string[100];
    sscanf(input, "%d %s", &my_number, my_string);
    ... теперь работа с ней ...
}
```

Вы заметили бросающуюся в глаза проблему? Некорректная входная строка – длина которой больше 100 символов – выйдет за границу буфера `my_string` и запишет произвольные данные по недопустимым адресам памяти.

Результат зависит от того, как используются испорченные адреса памяти. В некоторых случаях поведение программы никак не изменится, и тогда вам весьма повезет.[2] Иногда программа продолжает работать, но с малозаметными отличиями – их бывает трудно заметить и трудно найти причину. Иногда программа аварийно завершается, часто попутно

[1] Этот пример написан на C и характерен для C-кода, но область действия такого типа эксплойта далеко не ограничивается C.

[2] Если взглянуть с другой стороны, то вам очень не повезло. Вы не заметили дефект во время тестирования, поэтому он сохранится в окончательной версии, и там-то взломщик им и воспользуется.

вызывая сбой важных системных компонент. Но хуже всего, если проникшие в память данные окажутся в области кода, выполняемого ЦП. Организовать это не столь сложно, а в результате атакующий сможет выполнить на вашей машине произвольный код и при удаче получит к ней полный доступ.

Проще всего воспользоваться переполнением буфера, когда он располагается в стеке, как в описанном выше случае. Тогда появляется возможность перенацелить ЦП, заменив хранящийся в стеке адрес возврата из функции. Однако эксплойты могут быть созданы и для тех случаев, когда буфер размещается в куче.

Встроенные строки запросов

Данный тип атаки может быть применен для аварийного завершения программы, выполнения произвольного кода или несанкционированного получения данных. Как и в случае переполнении буфера, в основе атаки лежит неверный анализ входных данных, но при этом вместо прорыва за границы буфера используются действия, которые программа последовательно выполняет над непроверенными входными данными.

Классическим примером этого вида атак в C-программах является использование *форматной строки*. Обычно эксплуатируется функция `printf` или ее разновидности, и происходит это так:

```
void parse_user_input(const char *input)
{
     printf(input);
}
```

Злоумышленник подает на вход строку данных, содержащую маркеры формата `printf` (такие как `%s` и `%x`), и принуждает программу напечатать данные из стека или даже из адресов в памяти в зависимости от конкретного формата обращения к `printf`. Атакующий может также записать произвольные данные в память с помощью аналогичного приема (и маркера формата `%n`).

Решение проблемы отыскивается без труда. Запись оператора печати в виде `printf("%s", input)` гарантирует, что `input` не будет интерпретироваться в качестве форматной строки.

Есть много других случаев, когда встроенный запрос может быть использован злонамеренно. Например, можно скрыто ввести в приложение базы данных произвольный оператор SQL, который найдет в базе данные, нужные взломщику.

Другой вариант эксплойта для слабых веб-приложений известен как *кросс-сайтовый скриптинг* из-за способа прохождения атаки через систему: данные, введенные атакующим, пройдя через веб-приложение, воздействуют на веб-броузер жертвы. Специальный комментарий, помещенный атакующим в веб-систему обмена сообщениями, будет показан всеми броузерами, просматривающими данную страницу.

Если в сообщении есть скрытый код JavaScript, броузеры выполнят его незаметно для пользователей.

Условия гонки

Существует возможность эксплойта для систем, зависящих от скрытого порядка событий, с целью вызвать их нештатное поведение или аварийное завершение. Обычно это допускают системы со сложными моделями потоков или составленные из многочисленных взаимодействующих процессов.

Многопоточная программа может использовать общий пул памяти для своих рабочих потоков. Если не принять должных мер, то один поток может прочесть из буфера информацию, которую другой поток записал туда, но еще не собирался делать доступной остальным, например часть привилегированной транзакции или данные другого пользователя.

Эта проблема свойственна не только многопоточным приложениям. Рассмотрим следующий фрагмент кода C для UNIX. Его задача – записать некоторые данные в файл, а потом изменить права доступа к этому файлу.

```
fd = open("filename");            /* создать файл */
/* точка A (см. ниже) */
write(fd, some_data, data_size);  /* записать данные */
close(fd);                        /* закрыть файл */
chmod("filename", 0777);          /* назначить ему права доступа */
```

Существует ситуация гонки, которой может воспользоваться атакующий. Удалив файл в точке `A` и заменив его ссылкой на собственный файл, атакующий получит для своего файла специальные права доступа. Этим можно воспользоваться для дальнейшего проникновения в систему.

Целочисленное переполнение

Небрежное применение математических конструкций может привести к передаче программой управления неожиданными способами. Целочисленное переполнение происходит, когда тип переменной не позволяет представить результат арифметической операции. Применение беззнакового 8-разрядного целого типа (`uint8_t`) приводит к ошибке в следующем вычислении на C:

```
uint8_t a = 254 + 2;
```

Результатом будет 0, а не 256, как вы рассчитывали; 8 разрядов могут содержать числа, не превышающие 255. Атакующий может послать на вход очень большие числа, чтобы спровоцировать переполнение и вызвать непредвиденные результаты. Нетрудно видеть, что в итоге могут возникнуть серьезные проблемы; следующий код C может привести к переполнению кучи из-за целочисленного переполнения:

```
void parse_user_input(const char* input)
{
    uint8_t length = strlen(input) + 11; /* uint8_t может переполниться */
    char *copy = malloc(length);         /* этого может быть недостаточно */
    if (copy)
    {
        sprintf(copy, "Input is: %s", input);
        /* этот буфер может переполниться */
    }
}
```

Конечно, `uint8_t` едва ли будет выбран для переменной, содержащей длину строки, но точно такие же проблемы возникают с типами больших данных. При нормальной работе они маловероятны, но эксплойт на них построить можно.

Те же проблемы возникают при вычитании (тогда это называется *целочисленной потерей* (*integer underflow*), при использовании знаковых и беззнаковых чисел в присваивании, при некорректном приведении типов и при умножении или делении.

Дела защитные

Чем больше вы стремитесь к безопасности,
тем менее защищенным вы оказываетесь.

Брайан Трейси

Мы уже видели, что создание программного обеспечения напоминает строительство здания (см. «Действительно ли мы собираем программы?» на стр. 240 и главу 14). Мы должны научиться защищать свои программы точно так же, как мы защищаем дом, закрывая все окна и двери, нанимая сторожа и устанавливая дополнительные механизмы защиты (типа охранной сигнализации, электронных карточек доступа, значков с личными данными и т. п.). И тем не менее нужно постоянно поддерживать бдительность: несмотря на все хитрые замки, можно забыть закрыть дверь или включить охранную сигнализацию.

Стратегии защиты программного обеспечения применимы на разных уровнях:

***Установка* системы**

Конкретная конфигурация ОС, инфраструктура сети, номера версий всех работающих приложений имеют важное значение для состояния безопасности.

***Конструктивные* особенности программной системы**

Необходимо принять правильные конструктивные решения, например, относительно возможности для пользователя оставаться зарегистрированным в системе произвольно долгое время, способов связи между подсистемами, выбора протоколов.

***Реализация* программы**

В ней не должно быть дефектов. Наличие ошибок в коде может быть причиной уязвимости системы.

***Процедура* эксплуатации системы**

При неправильном использовании любая система может представлять собой угрозу. По возможности этому должно препятствовать правильное проектирование, но пользователям нужно объяснить, какие их действия могут привести к проблемам. Ведь сколько людей записывают свое имя и пароль на бумажке, которую кладут рядом с терминалом!

Всегда трудно создавать защищенные системы. При этом неизбежен компромисс между безопасностью и функциональностью. Чем более защищена система, тем менее удобно ею пользоваться. Самая защищенная система – та, которая ничего не вводит и не выводит; ее не атакуешь ни с какой стороны. Однако от такой системы мало пользы. В простейшей системе нет никакой авторизации и каждому позволено делать все, что угодно; но она совершенно не защищена. Необходимо выбирать нечто среднее. Выбор определяется природой приложения, уровнем его секретности и оценкой опасности подвергнуться нападению. Чтобы писать код, обеспечивающий достаточную степень защиты, необходимо ясно представлять, какие *требования безопасности* предъявляются к системе.

Подобно тому как вы предпринимаете некоторые меры для защиты здания, вы применяете некоторые технологии для защиты своего программного обеспечения от злоумышленников.

Технология установки системы

Каким бы хорошим ни было ваше приложение, но если система, в которой оно устанавливается, не защищена, ваша программа тоже уязвима. Даже самое надежное приложение должно работать в некоторой операционной среде: под конкретной ОС, на конкретном аппаратном устройстве, в сети и с определенной группой пользователей. Атакующий с таким же успехом может скомпрометировать компоненты среды, как и ваш код.

- Не запускайте на своем компьютере программы из ненадежных источников, потенциально опасные.

  Возникает вопрос: что может служить основанием для доверия или недоверия некой программе? Можно провести анализ исходного кода программного обеспечения, если он у вас есть, и убедиться в его корректности (если у вас есть такая склонность). Можно взять программное обеспечение, которым пользуются все, и считать, что вы застрахованы количеством пользователей. (Однако если в таком программном обеспечении будет обнаружена уязвимость, и вам, и всем остальным нужно обновить пакет.) Либо можно выбрать программу

у производителя с хорошей репутацией в надежде, что она послужит надежным показателем.

Запускайте на своем компьютере только программы, полученные из надежных источников. Установите четкую политику для определения надежности источников.

- Применяйте защитные технологии, например сетевые экраны и фильтры спама и вирусов. Не оставляйте потайных входов, которые могут быть обнаружены взломщиками.
- Будьте готовы к тому, что злоумышленники могут оказаться среди авторизованных пользователей, ведите регистрацию того, кто, что и когда делал в системе. Периодически делайте резервные копии всех данных, чтобы фальшивая их модификация не уничтожила все ваши прежние труды.
- Установите минимальное количество способов входа в систему, дайте каждому пользователю минимальный набор прав и сократите, если можете, число пользователей, допущенных к системе.
- Правильно настройте систему. В некоторых ОС по умолчанию устанавливается очень слабый уровень безопасности, просто зазывающий взломщиков войти в систему. Если у вас такая система, нужно научиться устанавливать на ней защиту в полном объеме.
- Установите *ловушку*: машину-приманку, которую взломщикам будет легче найти, чем ваши реальные системы. Если она будет выглядеть достаточно привлекательно, атакующие потратят свои силы на ее взлом, а ваши критически важные машины останутся вне их внимания. Если вы обнаружите компрометацию ловушки, постарайтесь дать отпор взломщику, прежде чем он доберется до ваших ценных данных.

Технология конструирования программного обеспечения

Необходимо начинать защиту программного обеспечения уже на этой ранней стадии. Если вы займетесь внесением в код модификаций, призванных обеспечить его защищенность, лишь на поздних стадиях цикла разработки, вас ждет неудача. Защита должна быть фундаментальной частью архитектуры и конструкции вашей системы.

Безопасность – важный аспект архитектуры любого программного продукта. Будет ошибкой не позаботиться о ней на ранних стадиях разработки.

Чем проще конструкция программного обеспечения, тем меньше в нем точек, доступных для атаки, и тем легче обеспечить его защиту. Естественно, что в более сложных конструкциях составные части взаимодействуют активнее, а потому в них больше мест, которые могут подвергнуться атаке взломщика. Если вы принадлежите к тем 99,9% программистов, которые не могут позволить себе запускать свои программы на

опечатанной машине, стоящей в бункере, находящемся в засекреченной точке пустыни, то вам следует позаботиться о том, чтобы сделать свою конструкцию возможно проще.

Проектируя код, подумайте над тем, как активно помешать кому бы то ни было воспользоваться вашим приложением непредусмотренным способом. Вот несколько полезных стратегий для достижения этого:

- Ограничьте в проекте количество точек ввода данных и направьте весь обмен данными через отдельную часть системы. В результате атакующий не сможет гулять по всему вашему коду, а будет ограничен одним (защищенным) узким местом. Он будет действовать лишь в некоем дальнем углу, а вы сможете сосредоточить в том месте свои усилия по защите.[1]
- Выполняйте все программы с минимально возможными правами доступа. Не запускайте программу от имени системного администратора, если это не является совершенно необходимым, а тогда примите *особые* меры предосторожности. Особенно важно это для программ UNIX с атрибутом setuid – их может запустить любой пользователь, но после запуска они получают в системе особые права.
- Избегайте функций, в которых нет реальной надобности. Вы сократите не только время разработки, но и шансы появления в программе ошибок – для них в программе останется меньше места. Чем ниже сложность кода, тем меньше шансы появления в нем опасных мест.
- Не основывайте свой код на опасных библиотеках. Опасной будет всякая библиотека, про которую вы не знаете, что она безопасна. Например, большинство библиотек GUI разработано без учета безопасности, поэтому не пользуйтесь ими в программе, выполняющейся с правами суперпользователя.

Проектируя программу, рассчитывайте только на известные, защищенные компоненты сторонних разработчиков.

- Если среда выполнения предоставляет средства обеспечения защиты, используйте их в своем коде. Например, среда .NET располагает инфраструктурой, которая позволяет, например, проверить, что вызывающий код подписан доверенным третьим лицом. Это не решает всех проблем (секретный ключ компании может оказаться скомпрометированным), и нужно уметь правильно применять такую технологию, но все же она способствует созданию защищенных программ.
- Не храните секретные данные. Если это все же необходимо, зашифруйте их, чтобы защитить от любопытствующих. При работе с сек-

[1] Разумеется, все не так просто. Переполнение буфера может произойти в любой части кода, и бдительность нужно проявлять постоянно. Все же большинство уязвимостей возникает в местах ввода в программу данных или поблизости от них.

ретными данными тщательно следите за тем, куда вы их помещаете; блокируйте страницы памяти с секретной информацией, чтобы администратор виртуальной памяти ОС не сбрасывал их на жесткий диск, откуда их сможет прочесть злоумышленник.

- Если пользователь вводит конфиденциальные данные, обращайтесь с ними осторожно. Не показывайте вводимые пароли.

Самая неудачная стратегия безопасности известна как *запутанность как защита* (*security through obscurity*), и она наиболее распространена. Ее цель – спрятать все детали конструкции и реализации, чтобы никто не смог разобраться, как код работает, а потому и придумать, как его взломать. При этом стараются сделать критически важные компьютерные системы как можно менее заметными в надежде, что атакующий их не обнаружит.

Это порочный путь. В один прекрасный день ваша система *будет* обнаружена, а через какое-то время *атакована*.

Не всегда такое решение принимается сознательно. Просто такой прием оказывается удобен, если вы вообще не занимались проблемами защиты во время проектирования системы – причем удобен до того момента, когда кто-нибудь *скомпрометирует* вашу систему. После этого начинается совсем другая история.

Готовьтесь к тому, что ваша система будет атакована, и проектируйте с учетом этого все ее части.

Технологии реализации кода

Так будет ли ваша программа защищена от взлома, если надежно спроектировать систему? К сожалению, это не так. Мы уже видели, как недостатки кода позволяют создавать эксплойты, дающие возможность повернуть выполнение программы непредвиденным образом.

Код, который мы пишем, – это передовая линия фронта, путь, которым атакующий чаще всего пытается воспользоваться для проникновения в систему и где разворачиваются основные сражения. Если система спроектирована плохо, то самый хороший код окажется уязвимым для атаки; но и на фундаменте тщательно продуманной архитектуры мы должны возвести крепкие защитные стены с помощью безопасного кода. Правильный код не всегда оказывается безопасным.

- Защитное программирование – основная технология разработки надежного кода. Его главный принцип – *не принимай ничего на веру* – как нельзя лучше соответствует задачам создания безопасного программного обеспечения. Паранойя – это благо, поскольку никогда нельзя надеяться на то, что пользователи станут взаимодействовать с вашей программой так, как вы от них ожидаете.

  Соблюдение простых правил защиты, таких как «проверяй *все* входные данные» (включая данные, введенные пользователем, ко-

манды запуска программ и переменные окружения) и «проверяй *все* вычисления», позволит избавиться от массы уязвимых мест в вашей программе.

- Осуществляйте *аудит безопасности*. Имеется в виду тщательное рецензирование исходного кода экспертами в области безопасности. При обычном тестировании вы редко обнаружите пробелы в защите; они проявляются при необычном сочетании условий, которое не придет в голову составителю тестов (например, переполнение буфера при вводе очень длинных последовательностей символов).
- С осторожностью порождайте дочерние процессы. Если атакующий сумеет перевести выполнение подзадачи в нужном ему направлении, он сможет получить контроль над любыми средствами. Пользуйтесь в коде C функцией `system`, только если не остается ничего другого.
- Тестируйте и отлаживайте программы нещадно. Давите ошибки со всей строгостью. Не пишите код, который допускает аварийное завершение; он может остановить всю систему.
- Заключите все операции в атомарные транзакции, чтобы атакующие не могли воспользоваться состоянием гонки в своих целях. Пример с `chmod` в разделе «Условия гонки» на стр. 305 можно поправить, применив `fchmod` к дескриптору открытого файла вместо `chmod` к имени файла: даже если атакующий заменит файл, вы точно знаете, какой файл нужно модифицировать.

Технологии процедуры

В основном здесь требуются тренировка и обучение, хотя желательно отобрать пользователей, которые не слишком беспомощны (если вы можете себе такое позволить).

Пользователей нужно научить приемам безопасной работы: не сообщать никому свой пароль, не устанавливать какое попало программное обеспечение вместе с критически важным и работать на машине только в соответствии с инструкциями. Однако ошибки бывают даже у самых старательных пользователей. При проектировании систем мы стремимся минимизировать риск таких ошибок и надеемся, что их последствия окажутся не слишком тяжелыми.

Резюме

Безопасность – это в своем роде смерть.

Теннесси Уильямс

Программирование – это война.

Безопасность – серьезная проблема в разработке современного программного обеспечения; невозможно спрятать голову в песок, чтобы не замечать ее. Страусы пишут плохой код. С брешами в системе без-

опасности можно бороться с помощью более тщательного проектирования и совершенствования архитектуры системы, а также лучшего информирования о существующих проблемах. Слишком велики риски, чтобы позволить себе не думать о защищенности систем.

Хорошие программисты...

- Разбираются в требованиях к безопасности в каждом проекте, над которым трудятся
- Инстинктивно пишут код, в котором нет стандартных уязвимых мест
- Учитывают требования безопасности при проектировании каждой системы, а не приступают к защите, когда продукт почти готов
- Располагают стратегией тестирования безопасности

Плохие программисты...

- Не занимаются проблемами защищенности, считая их малозначительными
- Считают себя экспертами в области безопасности (*настоящих экспертов* очень мало)
- Задумываются о недостаточной защищенности своих программ только при обнаружении уязвимостей или, еще хуже, в случае компрометации своего кода
- Думают о безопасности, только когда пишут код, игнорируя ее на уровне проектирования и архитектуры

См. также

Глава 1. Держим оборону

Защитное программирование – важная технология для написания безопасного кода.

Глава 8. Время испытаний

Необходимо строго тестировать наше программное обеспечение на предмет его безопасности.

Глава 13. Важность проектирования

Безопасность необходимо иметь в виду при проектировании каждого раздела кода.

Глава 14. Программная архитектура

Безопасность – одно из фундаментальных требований к архитектуре компьютерной системы. Ее проектированием нужно заниматься с самого начала.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 638.

Вопросы для размышления

1. Что такое «безопасная» программа?
2. Какие входные данные нужно проверять в безопасной программе? Какого типа проверка необходима?
3. Как можно защищаться против атак со стороны группы доверенных пользователей?
4. Где может произойти переполнение буфера, допускающее создание эксплойта? Какие функции особенно подвержены переполнению буфера?
5. Можно ли полностью исключить возможность переполнения буфера?
6. Как защитить память, используемую приложением?
7. Характерна ли для C и C++ принципиально более низкая защищенность, чем для других языков?

8. Учтен ли опыт C для проектирования C++ как более защищенного языка?
9. Как можно узнать, что ваша программа скомпрометирована?

Вопросы личного характера

1. Каковы требования к безопасности, предъявляемые в вашем текущем проекте? Каким образом были они установлены? Кто осведомлен о них? В каких документах они отражены?
2. Какой была самая страшная ошибка защиты в выпущенных вами приложениях?
3. Сколько бюллетеней по вопросам безопасности было выпущено по поводу вашего приложения?
4. Проводили ли вы *аудит безопасности* программы? Какие проблемы он выявил?
5. Кто, как вам кажется, вероятнее всего может попытаться атаковать вашу систему? В какой мере на это влияют:
   a. Ваша компания
   b. Тип пользователя
   c. Тип продукта
   d. Популярность продукта
   e. Конкуренция
   f. Платформа, на которой он работает
   g. Нахождение в сети и видимость системы широкой публике

III

Проектирование кода

В отличие от тонких вин, ваш код не становится нисколько лучше со временем. Если вначале он не больше кучки, произведенной собакой, то в конце он, несомненно, вырастет до размеров кучи, которую сделал слон.

В этом нет никакой тайны, и тем не менее фирмы-производители программного обеспечения продолжают выдавать творения слоновьих размеров, а потом страдают от результатов. Их продукты не обладают достаточными свойствами адаптируемости, расширяемости или изменяемости, чтобы обеспечить возможность соответствия будущим требованиям, да и разработка их проходит с трудом: они часто не укладываются во временные и финансовые рамки. Нас, программистов, это ударяет по самолюбию, но менеджеров это ударяет по кошельку.

Как быть? Одно решение есть – никогда не заниматься разработкой кода, но едва ли оно осуществимо. Другое решение – разрабатывать код с учетом представления о структуре системы в целом. Хороший код не возникает случайным образом; это продукт умелой ремесленной работы при значительных предварительных затратах труда на планирование и проектирование. Но помимо этого в разработке необходимо проявлять находчивость и сообразительность, чтобы справиться с проблемами и изменениями, с которыми вам неизбежно придется столкнуться по пути.

В этой части мы исследуем данный процесс. Мы рассмотрим:

Глава 13. Важность проектирования

Микропроектирование кода: рекомендации по конструктивным решениям низкого уровня для отдельных модулей кода.

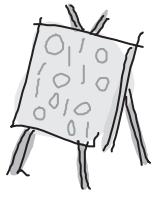
Глава 14. Программная архитектура

Проектирование систем на более крупном уровне – первый этап создания любых программных систем.

Глава 15. Программное обеспечение – эволюция или революция?

Взгляд на особенности развития программного продукта во времени и ряд практических рекомендаций по соединению новых разработок со старым базовым кодом.

Все это не факультативные добавки или благие пожелания. Это важные этапы в нашем деле, необходимые для выпуска качественного программного обеспечения. Можете опустить этот материал на свой страх и риск.



13

Важность проектирования

Как правильно проектировать программный продукт

В этой главе:

- Внутреннее устройство кода
- Что проектировать и зачем
- Каким должен быть хороший проект
- Инструменты и методологии проектирования

Верблюд – это лошадь, разработанная комитетом.

Сэр Алек Иссигонис

Иногда встречается код, глядя на который остается только вздохнуть.

Однажды мне нужно было написать драйвер устройства для встроенного продукта. Интерфейс драйвера с ОС был достаточно сложным. Интерфейс с устройством, с которым я работал, был тоже сложным. Поразмыслив, я разбил свой код на две части. В первой части была внутренняя библиотека для доступа к устройству, осуществлялась буферизация некоторых данных и предоставлялся простой API для доступа к этим буферизованным данным. Затем я написал второй, отдельный уровень, в котором реализовал капризный интерфейс драйвера ОС на языке моей внутренней библиотеки. Структура драйвера устройства показана на рис. 13.1.

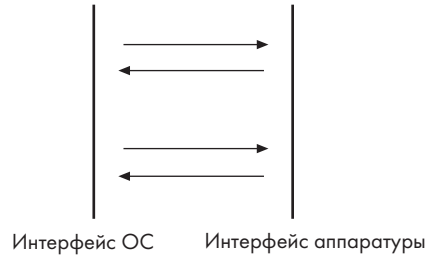


Рис. 13.1. *Разумная конструкция программы, сделанная Питом*

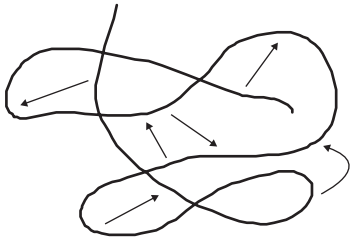


Рис. 13.2. *Как не надо проектировать программы*

Затем изготовитель аппаратуры прислал мне образец реализации драйвера того же самого устройства. Очевидно, что автор этого кода абсолютно его не продумал. Код представлял собой беспорядочную мешанину, в которой сложный интерфейс ОС совершенно непостижимым образом переплетался с аппаратной логикой. Его примерная структура приведена на рис. 13.2.

Дело не в том, что я хочу похвастаться (в большей мере, чем это заслужено). Смысл этой иллюстрации понятен. Первая конструкция лучше. В ней легче разобраться, потому что она очень проста, ее легче реализовать и в результате легче сопровождать.

Ч. Э. Р. Хоар писал: «Есть два пути конструирования программ. Первый путь – сделать конструкцию настолько простой, что в ней *очевидно* нет недостатков. Второй – сделать ее настолько сложной, что в ней нет *очевидных* недостатков. Первый путь намного труднее». (Hoare 81)

Одним из признаков зрелости программиста служит качество проектирования им кода. В данной главе мы посмотрим, чем отличается хороший проект, и изучим способы разработки высококачественных проектов программного обеспечения.

Программирование как конструкторская работа

Распространено мнение, что «проектирование» – это этап, который должен быть завершен перед началом написания кода. Его результатом является *спецификация* проекта в той или иной форме, которую сможет реализовать любой ремесленник.

На самом деле все совсем иначе. Программирование – действие, заключающееся в написании кода, – представляет собой *конструкторскую работу*.

Даже в самой подробной спецификации найдутся пробелы, иначе она сама и *будет* кодом – невозможно описать все мельчайшие подробности в проектной документации. Во время программирования проверяются первоначальные проектные решения и выполняется оставшаяся про-

ектная работа. При этом выявляются пробелы, несоответствия и ошибки и отыскиваются решения этих проблем. «Кое-кто из программистов не замечает, что во время программирования он занимается конструкторской работой, тем не менее, когда вы пишете код, вы всегда занимаетесь проектированием, прямо или косвенно» (Page Jones 96).

Программирование – это работа по проектированию. Это творческий и художественный акт, а не механическое написание кода.

При правильном процессе разработки это учитывают и не воздерживаются писать код, когда это целесообразно. Те, кто практикует *экстремальное программирование*, утверждают, что проект и есть код. (Beck 99) Проектирования как отдельного вида деятельности при этом нет; нет группы проектировщиков. Сами программисты постоянно совершенствуют и расширяют проект путем совершенствования и расширения кода. Это закреплено в их методе *проектирования на основе тестов*: прежде чем писать любой код, пишутся тесты как средство проверки правильности проектных решений. Это мудрая мысль.

Следует ли из этого, что необязательно думать перед тем, как начать кропать код? Ни в коей мере! Погрузившись в текстовый редактор, поздно планировать, что вы собираетесь написать. Все равно что отправиться на машине из Берлина в Рим и не подумать сначала, каким маршрутом ехать. Так вы очутитесь в Москве, еще не выяснив, в какой стороне север. По определению проект – это то, что делается *в самом начале*.

Думайте, прежде чем стучать по клавишам; составьте понятный проект. Иначе у вас получится не код, а хаос.

Что нужно проектировать?

Очевидно, программисты проектируют кодовые структуры. Но на разных стадиях разработки это может иметь разный смысл. На любой стадии проектирование заключается в разделении задачи на составные части и определении того, как работает каждая часть.

Эти уровни проектирования программного продукта таковы:

Архитектура системы

На этом этапе мы рассматриваем систему в целом, выделяем ее основные подсистемы и определяем средства связи между ними. Проект архитектуры имеет самое большое значение для производительности и характеристик системы в целом и меньше всего влияет на содержимое отдельных строчек кода. Это самый важный элемент проектирования, и о нем будет говориться в следующей главе. В настоящей главе нас интересует внутренняя конструкция кода, в которой участвуют последующие уровни проектирования.

Модули/компоненты

Подсистемы, входящие в архитектуру, обычно слишком велики для непосредственной реализации в коде, поэтому следующим шагом будет разбиение их на обозримые модули. Говоря о проектировании на уровне модулей, трудно добиться четкости. В некотором смысле такой вещи, как «модуль», реально не существует. *Модуль* может иметь разный смысл в зависимости от подхода к проектированию; это может быть логически обособленная часть кода, возможно, физический объект типа пакета Java, пространства имен C++/C# или многократно используемая библиотека. Модулем можно считать иерархию классов или даже отдельную выполняемую программу.

На этой стадии проектирования могут вырабатываться открытые интерфейсы. Последующая модификация их затруднительна, поскольку они образуют жесткие контракты между модулями кода и командами программистов, которые их пишут.

Классы и типы данных

После этого модули разбиваются на мелкие куски. Интерфейсы внутри модуля обычно проектируются менее формально и легче поддаются модификации. Возникает стремление выполнить это микропроектирование прямо с клавиатуры. От этого желания следует себя удержать, поскольку иначе появляется первый пришедший в голову код, а не тот, который лучше всего решает задачу.

Функции

Это, пожалуй, низший уровень в данной цепи, что не принижает его важности. Программа состоит из подпрограмм: если эти подпрограммы плохо сконструированы, страдает вся система. После точного определения, какие функции необходимы, мы проектируем их внутреннее устройство, задаем порядок передачи управления и выбираем алгоритмы.[1] Обычно это происходит в уме, а не в виде документированной процедуры, но старательное проектирование при этом необходимо.

Из-за чего весь этот шум?

Никто не станет утверждать, что проектировать нужно плохо, и тем не менее плохо спроектированного кода предостаточно. Проведя несколько лет на линии фронта, любой разработчик подтвердит вам это своими шрамами. (Закаленные в сражениях ветераны уже кивают головами и мысленно готовятся рассказать случаи из своего боевого прошлого.) Но почему так происходит?

---

[1] Важнейшие алгоритмы часто реализуются набором нескольких функций; они определяются на стадии проектирования модуля.

Небрежное проектирование может быть результатом неопытности программистов, но чаще оно обусловлено коммерческими интересами фирм-производителей программного обеспечения, которые не оставляют времени на разработку приличного проекта. Никто не желает слушать протесты несчастных кодировщиков. В реальном мире программирование подчинено задаче поставить программное обеспечение к назначенному сроку, каким бы ни было его качество. Ирония заключается в том, что практически всегда отсутствие хорошего проекта обходится дороже, чем стоило бы время, потраченное на его доработку. Как говорится, «нет времени, чтобы сделать работу, как надо, зато найдется время сделать ее дважды».

На самом деле *очень* важно правильно выполнить проектирование. Проект кода – это фундамент, на котором он будет построен. Если фундамент слаб, то и код будет нестабильным, ненадежным и малопригодным, то есть опасным. Плохое проектное основание приводит к появлению программного обеспечения, которое можно сравнить с падающей Пизанской башней. Хотя первое время ему удается выстоять под напором реальных условий жизни, но оно никогда не будет в полной мере соответствовать требованиям, и время с неизбежностью это покажет.

Код, который основан на прочном проекте:

- Проще написать (есть четко определенный план действий, и ясно, как собрать вместе отдельные части)
- Легко понять
- Проще исправлять (можно определить, в каком месте гнездится проблема)
- С меньшей вероятностью содержит ошибки (ошибки в программе не скрываются за таинственными проблемами проектирования)
- Более устойчив к изменению (проект поддерживает расширения и модификацию)

Хороший проект программного продукта

Любая задача программирования допускает несколько вариантов проекта кода. Вы должны выбрать из них один. Лучший. Или хотя бы достаточно хороший. Это не так просто...

- Как определить, что ваш проект жизнеспособен? Выработав безупречный план проведения атаки, вы уверенно приступаете к его реализации. Через какое-то время вы сталкиваетесь с неожиданной проблемой. Назад к чертежной доске!
- Как определить, что проект завершен? Это станет известно только тогда, когда вы действительно реализуете его и убедитесь в том, что он работает. Многие проблемы нельзя предвидеть загодя; нужно осуществить реализацию проекта и выяснить, работает ли она. Только опробовав некоторое решение, вы по-настоящему *начинаете*

понимать исходную задачу. Вооружившись накопленным опытом, вы можете теперь попробовать решить ее правильно.

- Как определить, что выбранное вами конструктивное решение является лучшим из существующих? Это можно сделать только после того, как вы опробуете все варианты. На практике это неосуществимо. Как тогда определить, что ваше решение *приемлемо*? Если предъявляются требования к производительности, то оценить свое решение вы сможете лишь тогда, когда система заработает.

Лучшие подходы к проектированию учитывают эти проблемы. Перечислим их:

Итеративность

Число неприятных неожиданностей сократится, если выполнить проектирование в небольшом объеме, реализовать этот проект, оценить его последствия и использовать в новом цикле проектирования. Такой поэтапный способ проектирования оказывается очень мощным.

Осторожность

Не пытайтесь спроектировать сразу все. Если что-то не ладится, причиной может быть целый ряд конструктивных решений. Ограничьте пространство для возможных сбоев, и вам станет легче продвигаться вперед. Маленькие, но твердые шажки увереннее приведут вас к успеху, чем большие и грубые.

Реализм

Директивный процесс проектирования срабатывает не всегда и не везде. Результат зависит от качества заданных требований, опыта разработчиков и строгости применения процедуры. При прагматичном подходе вы берете лучшее из всех методологий и признаете влияние такого фактора, как интуиция программистов; опыт имеет большое значение для формирования хорошего проекта.

Осведомленность

Вы должны полностью разбираться во всех требованиях и мотивах, касающихся решаемой вами задачи, как и представлять себе, какими важнейшими свойствами должно обладать правильное решение. В противном случае вы станете решать не ту задачу, которая перед вами стоит. Вам необходима эта информация, чтобы как можно раньше принимать правильные проектные решения, потому что некоторые из них впоследствии трудно изменить.

На ваш подход к проектированию неизбежно повлияет действующая общая методология разработки (см. их описание в разделе «Стили программирования» на стр. 530). Правильная *процедура* проектирования – это шаг в направлении создания хорошего проекта, но не гарантия его. Все равно все сводится к качеству принимаемых вами проектных решений. Выбор различных компромиссов ведет к различным проектам. Например, проект, ориентированный на скорость, будет отличаться от проекта, ориентированного на расширяемость. В конце концов, не бы-

вает правильных и неправильных проектов. В лучшем случае есть *хорошие* проекты и *плохие* проекты.

Хорошие проекты обладают рядом привлекательных свойств, отсутствие которых служит признаком плохого проекта. Об этих свойствах мы и поговорим.

Простота

Это главнейшая характеристика хорошо спроектированного кода. Если проект прост, его легко понять, в нем нет излишеств или пороков и его легко реализовать. Ему свойственны связность и последовательность.

Простой код мал по объему, насколько это возможно. Это требует определенного труда. Блез Паскаль писал: «Прошу извинить, что мое письмо получилось таким длинным, но у меня не было времени сделать его короче». Постарайтесь выяснить, каким самым *коротким* кодом можно обойтись, и не добавляйте ничего лишнего к нему. Запомните, что потом всегда можно будет дописать новые функции, но трудно будет удалить то, что обросло связями.

Лень *иногда* бывает оправданна. Делайте проект так, чтобы как можно больше работы отложить на последний момент, и сосредоточьтесь на непосредственных задачах.

Чем меньше, тем лучше. Вашей целью должен быть простой код, который при малом размере решает большие задачи.

Разработать простой проект не так легко. Для этого нужно время. Для всякой сколько-нибудь сложной программы окончательное решение получается в результате анализа огромного объема информации. Когда код хорошо спроектирован, кажется, что он и не мог быть иным, однако возможно, что его простота достигнута в результате напряженного умственного труда (и большого объема рефакторинга).

Сделать простую вещь сложно. Если структура кода кажется очевидной, не надо думать, что это далось без труда.

Есть масса способов сделать проект излишне сложным, например коряво разложив его на компоненты, бестолково заводя многочисленные потоки, выбирая неудачные алгоритмы и сложные схемы именования, создавая лишние или неподходящие зависимости между модулями.

Элегантность

Элегантность воплощает эстетические аспекты проектирования и часто сопутствует простоте. Она означает, что код лишен вычурности, не смущает изощренностью или чрезмерной сложностью. Хорошо спроектированный код обладает красотой структуры. Желательно, чтобы он обладал следующими свойствами:

Поиск компромисса

Проектирование программного обеспечения состоит в принятии решений – как разложить систему на составные части так, чтобы уравновешивались соперничающие силы, направленные в разные стороны. Окончательный проект определяется выбором тех или иных компромиссов.

Вот типичные примеры такого перетягивания канатов:

Расширяемость против простоты

Результатом проектирования, ориентированного на последующее расширение, будет множество точек интерфейсов, в которых возможно подключение нового кода, а также достаточно общая оснастка, которая допускает модификацию в соответствии с требованиями, которые могут возникнуть позднее. Стремление же к простоте исключает дополнительные уровни косвенности и излишнюю общность.

Эффективность против безопасности

Прирост производительности часто достигается за счет потери чистоты проекта – появления особых скрытых входов для важных операций или избытка связывания с целью сократить уровень косвенности доступа. Когда системы сильно оптимизированы, в них труднее разобраться, а модификации для них более опасны.

Однако не всегда эффективные проекты оказываются плохими; у многих хороших проектов производительность оказывается высока благодаря простоте.

Функциональность против объема работ

Когда проект начинается, обнаруживаются тысячи функций, которые в нем необходимы, и разумные основания, по которым они должны быть готовы на следующий день (или еще раньше). Однако, не располагая бесконечным количеством обезьянок и таким же числом ПК, вы никогда не сможете их все реализовать. Чем больше функций, тем дольше реализация.

Какие из них важнее, зависит от требований к проекту. Вот почему так важно определиться с ними заранее.

- Управление изящно перемещается по системе. Не бывает такого, чтобы одна операция прошла через все модули, преобразуя свои параметры между 16 разными представлениями, прежде чем исчезнуть.
- Все части дополняют одна другую, внося что-то индивидуальное и ценное.

- Проект не пестрит особыми случаями.
- Аналогичные вещи связаны между собой.
- Отсутствуют неприятные сюрпризы.
- Изменения затрагивают небольшую область: одно простое изменение в одном месте не требует модификации кода во многих других местах.

Хороший проект отличается соразмерностью и эстетичностью. Я не стану утверждать, что программирование – это искусство, хотя некоторые это убедительно доказывают. На элегантности и простоте держится большинство остальных характеристик в этом списке.

Модульность

Решая задачу разработки проекта, мы естественным образом делим его на части, которые называем *модулями*, или *компонентами*. Разбиение происходит на подсистемы, пакеты, классы и т. д. Каждая такая часть оказывается менее сложной, чем первоначальная проблема, но в совокупности они составляют целое решение. Первостепенное значение имеет качество такого разбиения.

Основными качествами модульности являются *связность* (*cohesion*) и *сцепление* (*coupling*). Мы стремимся к тому, чтобы у модулей были:

Сильная связность

Связность – это оценка того, в какой мере близкие функции собраны вместе и насколько хорошо части модуля действуют в совокупности. Связность – это то, что обеспечивает единство модуля.

Слабая связность модулей – признак неудачной декомпозиции. Каждый модуль должен иметь четко определенную задачу, а не выступать как мешок, в который сваливают не связанные друг с другом функции (типа распространенного, к сожалению, пространства имен `utils`; непонятно, о чем *думают* те, кто пишет такой код).

Слабое сцепление

Сцепление – это показатель взаимозависимости *между* модулями: количество входов и выходов из них. В простейших случаях модули мало связаны друг с другом и потому менее зависят один от другого. Очевидно, что нельзя полностью развязать модули, иначе они вообще не смогут взаимодействовать!

Модули соединяются между собой различными способами – прямыми или косвенными. Модуль может вызывать функции из других модулей или сам вызываться другими модулями. Он может использовать типы данных, определенные в другом модуле, или выделять какие-то данные для общего доступа (например, переменные или файлы). В хорошем проекте каналы связи создаются только там, где без них совершенно невозможно обойтись. Эти каналы связи во многом определяют конструкцию кода.

После того как модуль определен, его можно разрабатывать и тестировать независимо от других. Это преимущество модульности; оно позволяет распределить задание между несколькими программистами. Однако следует проявлять осторожность; закон Конвэя предупреждает, что структура программного продукта может отражать структуру команды разработчиков: «Если в разработке компилятора участвуют четыре команды, у вас получится четырехпроходный компилятор» (см. раздел «Организация и структура кода» на стр. 411). Следите за тем, чтобы декомпозиция была разумной и соответствовала проблеме, а не штатной структуре.

Проектируйте такие модули, которые внутренне связны и минимально соединены между собой. Декомпозиция должна отражать правильное разбиение задачи на части.

Хорошие интерфейсы

Модули помогают нам разбить задачу на части, разделить проблемы. В каждом модуле определяется *интерфейс* – открытый публике фасад, за которым скрывается внутренняя реализация. Этот набор допустимых операций часто называется *интерфейсом прикладного программирования* (*API*). Он образует единственный путь к функциям, предоставляемым модулем, и его качество определяет качество самого модуля, по крайней мере с внешней стороны.

Проведите черту, за которую никто не должен переступать: определите четкие API и интерфейсы.

Чтобы определить хороший интерфейс, действуйте следующим образом:

1. Определите клиента и его *потребности*.
2. Определите поставщика и его *возможности*.

   Успешно создать интерфейс между пользователем и реализацией можно лишь тогда, когда правильно определены обе стороны и выяснены их потребности. Если в этом есть ясность, то можно попытаться создать интерфейс, который удовлетворит пользователей и окажется практически реализуем.

   В плохом проекте операции осуществляются не там, где следует, в результате чего можно сломать голову, пытаясь проследить логику приложения, и трудно расширять проект. В итоге растет сцепление модулей и ухудшается их связность.

3. Вывести заключение о типе требуемого интерфейса.

   Будет ли это функция, класс, сетевой протокол или нечто другое? Это может определяться поставщиком функциональности, но интерфейс можно заключить в оболочку, которая будет представлять его иным способом. Например, создание объекта CORBA вокруг

библиотеки делает ее функциональность доступной в сети взаимодействующих компьютеров.

4. Определить сущность операции.

   Какую функциональность нужно предоставить *в действительности* – не сделать ли ее шире, чем требования данного конкретного клиента? В каждой функции может напрашиваться более полезная операция.

Есть ряд ключевых принципов (рис. 13.3), на основе которых можно обсуждать строение и качество интерфейсов:

Разделение

Интерфейс создает контактную точку, но в то же время и разделительную линию между клиентом и реализацией. Они могут общаться только заранее установленным и никаким иным образом.

Хорошо спроектированный код четко устанавливает *роли* и *ответственности*. Определение в системе главных действующих лиц и их обязанностей обеспечивает четкость и эффективность интерфейсов.

Хороший пример представляет мой дом. Его главный интерфейс – входная дверь. Дверь разделяет жителей и гостей и определяет ме-

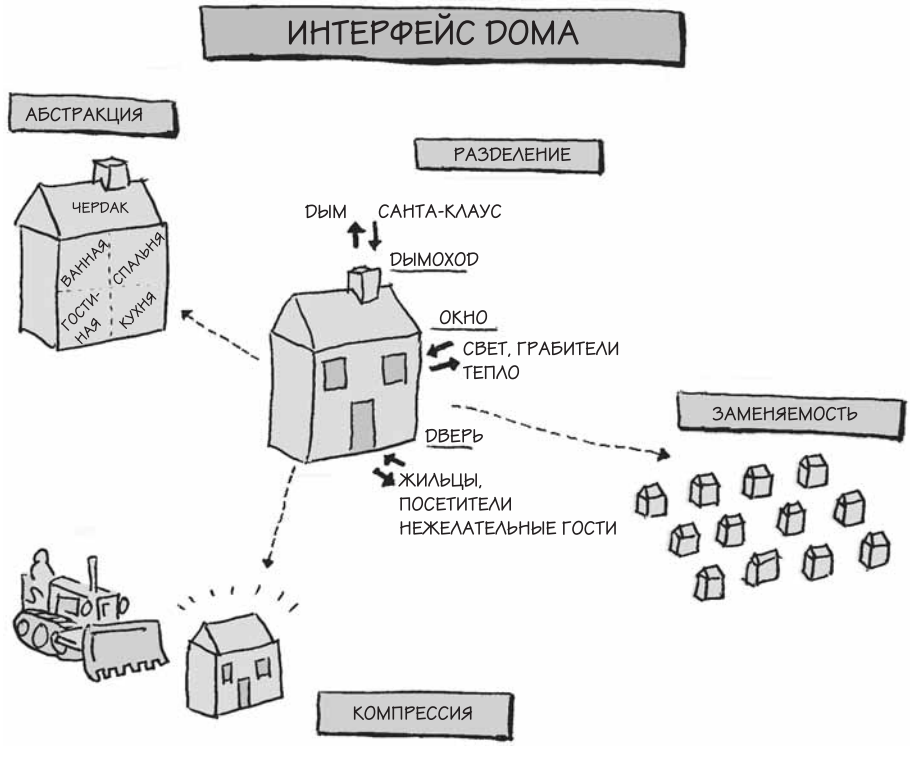


***Рис. 13.3.** Интерфейсы, которые предоставляет дом*

сто их встречи. Есть ряд других интерфейсов для других операций: окна, телефоны, дымовая труба и т. д.

Абстракция

Абстрагирование позволяет наблюдателю сосредоточиться на принятии важных решений, исключив из рассмотрения определенные детали. Реальность оказывается скрыта за более простым представлением, благодаря чему легче справиться со сложностью. В ОО-проектировании эта концепция имеет особую важность. При проектировании интерфейса абстракция достигается тщательным разделением того, что действительно важно для пользователя, и того, что лучше скрыть от него.

Если у вас есть ваза с фруктами, можно просто сказать: «*Съешь* фрукт, который сверху», а потом – «*Съешь* следующий», и нисколько не заботиться о том, что это за собой влечет, а именно: если это грейпфрут, его нужно очистить от кожуры, а если это ревень, то его нужно сварить и посыпать сахаром. Детали скрыты за абстрактным *съесть*; нам нужно, чтобы фрукт *был* съеден, и неинтересно, как это будет сделано.[1]

Абстракции могут образовывать иерархии. Мой дом может рассматриваться с различных *уровней абстракции* в зависимости от того, кто вы такой: строитель, физик-ядерщик или банковский служащий. Например, он может рассматриваться в виде:

- Совокупности помещений
- Структуры, состоящей из стен, потолков и полов
- Сооружения из кирпичей и деревянных деталей
- Совокупности молекул или даже атомов
- Закладной, требующей выплат

Компрессия

Имеется в виду способность интерфейса представлять большую операцию посредством чего-то более простого. Компрессия часто возникает в результате удачных абстракций, а плохие абстракции приводят к более пространному коду.

Заменяемость

Одну реализацию интерфейса можно заменить другой, если она удовлетворяет тому же контракту. Когда вы определяете в своей программе интерфейс сортировки, за ним может скрываться любой алгоритм: быстрой сортировки, пирамидальной или (не дай бог) пузырьковой. В любой момент вы можете заменить его, если только его поведение, видимое через интерфейс, останется тем же самым.

---

[1] Возможность скрыть различное физическое поведение за единой логической абстракцией называется *полиморфизмом*. Это понятие описывается в разделе «Полиморфизм» на стр. 533.

Виды интерфейсов

Вычислительная наука по большей части занимается определением интерфейсов и организацией с их помощью сложных вещей. Есть даже афоризм, гласящий, что «Любая проблема может быть решена с помощью дополнительного уровня косвенности», т. е. в результате сокрытия возникшей сложности за очередным интерфейсом. Интерфейсы бывают разных типов. Все они предъявляют своим клиентам некоторый публичный образ, а малоприглядные детали реализации прячут за этим фасадом.

Вот обычные виды создаваемых интерфейсов:

- Библиотеки
- Классы
- Функции
- Структуры данных (в особенности не совсем обычные с дополнительным поведением, например, семафоры)
- Интерфейсы ОС
- Протоколы (например, сетевые)

В иерархиях наследования классов любой объект может быть заменен своим надтипом.

Если вы хотите, чтобы я открыл вам дверь, вы звоните в дверной звонок. Раньше это был электрический выключатель, который проводами соединялся с механизмом звонка, но сейчас я потратился на модный беспроводной звонок. Для вас это не имеет никакого значения, вы даже можете не знать, что я сменил звонок; вы нажимаете на кнопку, и я выхожу.

Расширяемость

Хорошо спроектированный интерфейс позволяет при необходимости вставить в нужное место дополнительную функциональность. Опасность здесь состоит в том, чтобы не переусердствовать, пытаясь оставить возможность для любой будущей модификации.

Расширяемость может быть обеспечена с помощью вспомогательных программных структур: динамически загружаемых модулей, тщательно подобранных иерархий классов, в вершине которых лежат абстрактные интерфейсы, предоставление полезных функций обратного вызова и даже предельно логичной и податливой структуры кода.

Проектируйте с учетом последующего расширения, но не перестарайтесь, иначе вы напишете не программу, а ОС.

Хороший проектировщик старательно продумывает способы возможного расширения его программы в будущем. Если зацепки для будущего расширения разбрасывать по коду случайным образом, его качество может снизиться. Чтобы определить степень расширяемости, которую должен обеспечить проект, следует взвесить, какая функциональность требуется в данный момент, какую явно придется добавить в будущем и какие еще потребности могут возникнуть.

Избегайте дублирования

В хорошем проекте нет дублирования; повторяющихся частей быть не должно. Дублирование – враг элегантного и простого проекта. Ненужная избыточность в коде снижает надежность программы: если есть два схожих участка кода, незначительно различающиеся между собой, и в одном из них будет обнаружена и исправлена ошибка, легко пропустить необходимость исправить ту же ошибку в другом месте. Очевидно, что это снижает надежность кода.

Обычно дублирование возникает в результате *программирования в стиле «копирование-вставка»* – при копировании кода в редакторе. Более скрытые варианты возникают, когда программисты заново изобретают колесо в силу плохого представления о системе в целом.

- Если вы столкнетесь с явно сходно решаемыми задачами в разных участках кода, обобщите их в виде функции, вызываемой с разными параметрами. В результате, если возникнут проблемы, их нужно будет решать в одном месте. Если дать функции еще и содержательное имя, то дополнительным преимуществом станет более понятное назначение этого кода.
- Очень похожие классы указывают на то, что какие-то функции можно вынести в надкласс или что желательно добавить интерфейс, описывающий общее поведение.

Делай один раз. Делай хорошо. Избегай дублирования.

Переносимость

Хороший проект *необязательно* оказывается переносимым; все зависит от предъявляемых к коду требований. Есть много средств избежать зависимости от платформы, но портить код, когда переносимость не нужна, неправильно. В хорошем проекте переносимость обеспечивается *в соответствии* с предъявляемыми требованиями.

Знакомая история: ваш код изначально не предполагалось выполнять в другой среде, поэтому переносимость не ставилась в качестве задачи при проектировании. Однако во время разработки неожиданно потребовалось перейти на новую платформу. Проще было переделать имеющуюся программу, чем писать ее заново. Код не был готов к возможно-

сти переноса, а времени для рефакторинга или перепроектирования с поддержкой кросс-платформенности не было. Результат? Запутанный код, структура которого безнадежно искалечена и пестрит конструкциями типа `#ifdef NEW_PLATFORM`. Это результат не плохого программирования, а плохой философии.

Проявляйте осторожность при выборе структуры для разделов своего кода, которые зависят от ОС или аппаратной части. В будущем вы только выиграете от этого, а на эффективность или ясность кода это может не повлиять (в некоторых случаях понятность даже улучшается). Задуматься о таких проблемах стоит пораньше, потому что переделка предшествующих решений обходится дорого.

Стандартный прием – создание уровня абстрагирования от платформы (которым может быть тонкий слой поверх нескольких интерфейсных функций ОС). На каждой платформе этот слой может быть реализован по-своему.

Решайте вопросы переносимости кода на этапе проектирования, а не путем переделки готового кода.

Идиоматичность

В хорошем проекте, естественно, применяются лучшие практические приемы, включающие как методологию проектирования (см. раздел «Стили программирования» на стр. 530), так и идиомы языка реализации. Это позволяет другим программистам сразу понять структуру кода.

Если известен язык реализации (он может быть постоянным или определен в рамках проекта), нужно представлять себе, как правильно его применять. Например, в C++ есть такие идиомы, как *Resource Acquisition Is Initialization* (*RAII*, «Получение ресурса есть инициализация») и перегрузка операторов, которые могут оказать большое влияние на проектирование кода. Изучите их. Разберитесь в них. Применяйте их.

Документированность

Последнее по счету, но не по важности свойство хорошего проекта – качество его документации. Не заставляйте читателей самих догадываться о структуре. Особенно это важно для высших уровней проектирования. Документация должна быть короткой в силу простоты проекта.

С одной стороны, архитектурные решения документируются в спецификации. С другой – код отдельных функций самодокументируется. В промежутке, возможно, «грамотное программирование» составляет документацию по API.

Как проектировать код

Всегда проектируйте вещь, рассматривая ее в ближайшем контексте: стул – в комнате, комнату – в здании, здание – в окружающей среде, окружающую среду – в плане развития города.

Элиэль Сааринен

Как научиться правильно проектировать? Хорошими проектировщиками становятся или рождаются? Можно ли обучать проектированию и можно ли перенять это искусство? У программистов бывает природное чутье на хороший проект; так устроены их мозги. Они от природы эстетически развиты и способны в достаточной мере разобраться в проблеме, чтобы выносить здравые суждения. Тем не менее эффективному проектированию можно научиться.

Лично у меня от рождения не было особого таланта в гончарном ремесле. (И я не встречал людей, у которых он был.) Я и сейчас не добился в этом больших успехов, но одно время учился этому делу. Я понял механизм и могу делать горшки (почти самобытные). Вероятно, попрактиковавшись, я преуспел бы больше, но мастером-художником мне никогда не стать.

Точно так же никто не рождается с природными способностями к проектированию кода. Нужно учиться. Учиться методологиям проектирования и хорошим техническим приемам. Их цель – сделать проектирование воспроизводимым процессом, но они не заменят *мастерство*. Творческое мышление и способность к новаторским проектам передать гораздо сложнее; всегда будут талантливые проектировщики, которым это дается легко.

Хороший проект программного обеспечения эстетически привлекателен; для создания этих цифровых произведений искусства нужны мастерство, опыт и практика. Я не буду пытаться описать, как проектировать программы, подобно тому, как учат рисовать по клеточкам. Жаль: умей я разливать хорошие проекты по бутылкам, можно было бы стать миллионером. Чтобы научиться хорошо проектировать, нужно понять, что составляет хороший проект и каковы признаки плохого проекта. Затем практиковаться. Долго.

Помимо личных качеств существуют методы и инструменты проектирования, способные стать большим подспорьем для программиста. В завершение мы посмотрим, чем они смогут (или не смогут) нам помочь.

Методы и процедуры проектирования

Существует много методологий проектирования программного обеспечения. В одних подчеркивается система обозначений, в других – процедура. Систематический подход лучше, чем проектирование *как бог*

на душу положит; применяемый метод обычно определяется практикой и культурой компании. Я всегда стремлюсь не слишком увязнуть в каком-то процессе – старание удовлетворить ему во всех деталях снижает творческую активность.

Современные методы проектирования делятся на две основные группы в соответствии со своей базовой философией проектирования:

Структурное проектирование

Основная черта – *функциональная декомпозиция*, разбиение функциональности системы на ряд более простых операций. Подпрограммы составляют основу структуры; проект строится как иерархия подпрограмм. Для структурного проектирования характерен подход *разделяй и властвуй*, деление задачи на все более мелкие процедуры вплоть до момента, когда дальнейшая декомпозиция более невозможна.

Два основных подхода составляют *нисходящее* (*top-down*) и *восходящее* (*bottom-up*) проектирование.

- Как можно догадаться, при нисходящем подходе берется вся проблема в целом и разбивается на более мелкие функциональные области. Последние, в свою очередь, проектируются как независимые единицы, и так до тех пор, пока необходимость в дальнейшем делении не исчезнет.
- Напротив, при восходящем проектировании вначале проектируются наименьшие единицы функциональности – простые вещи, которые система обязана делать. Затем эти функции подключаются друг к другу, пока не образуют в совокупности решение проблемы.

На практике оба подхода применяются в паре, и процесс проектирования завершается, когда они встречаются между собой где-то посередине.

Объектно-ориентированное проектирование

В отличие от структурного проектирования, занятого представлением операций, которые должна выполнять система, ОО-проектирование сосредоточено на данных, обрабатываемых внутри системы. Модель программного обеспечения при этом состоит из ряда взаимодействующих между собой отдельных блоков, которые называются *объектами*.

При ОО-проектировании выделяются основные объекты предметной области и устанавливаются их характеристики. Задается поведение этих объектов, в том числе операции, которые они могут осуществлять, и взаимодействие с другими объектами. Эти объекты включаются в проект вместе со всеми необходимыми объектами области реализации. Проектирование завершается после определения поведения и взаимодействия всех объектов.

Объектно-ориентированное программирование было провозглашено спасительным средством для задачи проектирования программ, новой парадигмой, которая принесет всеобщее счастье, поэтому люди часто стесняются признаться, что не применяют ОО-проектирование. Но поднятый вокруг него шум в значительной мере оправ-

Шаблоны проектирования

Шаблоны (*patterns*) стали в последнее время модным словом в ОО-программировании. На их популярность оказала влияние книга Э. Гамма и др. «Design Patterns: Elements of Reusable Object-Oriented Software»[1] (Gamma et al. 94), авторов которой любовно назвали «бандой четырех» («Gang of Four» – откуда идет известное название «книга GoF»), а сами шаблоны проектирования являются переложением для программирования архитектурных разработок Кристофера Александера (Alexander 99).

Шаблоны образуют словарь опробованных проектных решений, и каждый шаблон описывает распознаваемую структуру сотрудничающих между собой объектов. Это не какие-то хитроумные проекты, а повторяющиеся, обнаруживающиеся в *реальном коде* шаблоны, которые показали свою работоспособность. *Языки шаблонов* собирают каталог шаблонов проектов, демонстрируя взаимосвязь и дополнительность. Каждый шаблон языка следует установленной форме, описывая *контекст*, *задачу* и ее *решение*. Эти данные позволяют должным образом применять шаблон в своих проектах.

В программной системе шаблоны всплывают на нескольких уровнях. Архитектурные шаблоны оказывают глубокое влияние на организацию системы. Шаблоны проектирования образуют промежуточный уровень сотрудничества программных компонент. Шаблоны уровня языка представляют собой специфические приемы кодирования, обычно называемые языковыми *идиомами*.

Названия шаблонов проектирования вошли в обиходную речь, что свидетельствует об их полезности. Можно слышать, как программисты уверенно обсуждают *адаптеры*, *наблюдатели*, *фабрики* и *синглтоны*.

О шаблонах проектирования следовало бы сказать гораздо больше, чем позволяет место. Это действительно полезная идея, на изучение которой стоит потратить некоторое время. Почитайте книгу «банды четырех» и другую литературу.

---

[1] Э. Гамма, Р. Хелм, Дж. Ральф, В. Джон «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб.: Питер, 2006.

дался, и с его помощью программные проекты смогли справляться со сложностью гораздо более крупных задач.

Более подробное описание методов и процедур проектирования см. в разделе «Стили программирования» на стр. 530.

Инструменты проектирования

Проект в конечном итоге находит свое выражение в коде, но часто оказывается полезным работать на более абстрактном уровне. Задача инструментов – помочь нам обдумывать проекты, создавать более эффективные проекты и сообщать об этих проектах программистам посредством документации, описывающей то, что мы *намерены* изготовить и что *уже* сделано.

В известном смысле инструментами служат методологии, но существует широкий круг других дополнительных средств.

Нотация

Хорошая картинка стоит многих слов. Многочисленные системы графических обозначений помогают выражать наши проекты наглядным образом. Большинство из них оказывается преходящей модой и быстро исчезает из виду, чтобы уступить место еще более привлекательному способу рисовать окошечки и соединять их линиями. В данное время самой популярной и проработанной системой обозначений является *UML* (*Unified Modeling Language – унифицированный язык моделирования*). Она позволяет моделировать и документировать практически все объекты, создаваемые в процессе разработки программного обеспечения. В процессе развития ее полнота стала такова, что возможности визуализации более не ограничиваются программным обеспечением; она применяется для моделирования аппаратных средств, бизнес-процессов и даже организационных структур.

Система обозначений позволяет описывать, обдумывать и обсуждать проекты программного обеспечения. Она служит двум целям:

- Быстро набросать проект «на манжетах» и выразить свои мысли на доске.
- Формально описать проект.

В последнем случае вычерчивание диаграмм должно происходить автоматически с помощью специального графического средства. Иначе диаграммы будет сложно обновлять, и по мере разработки кода они начнут устаревать. Свое время нужно тратить более эффективно, чем на вычерчивание прямоугольников и линий.

Я предпочитаю не увязать глубоко в формальном применении этой нотации, а пользоваться ею как средством изложения важных элементов проекта. Мне достаточно того, что я могу описать свои идеи, а вникать в смысл всяких звездочек и пунктирных линий в отдельных типах диаграмм мне недосуг.

Шаблоны проектирования

Мощное средство проектирования, предоставляющее словарь проверенных приемов проектирования и показывающее, как применять их на практике. Во врезке «Шаблоны проектирования» на стр. 334 о них рассказывается более подробно.

Блок-схемы

Специальный тип графических обозначений для визуализации алгоритмов. Они полезны для получения представления высокого уровня, но менее точны, чем код, и тоже нуждаются в синхронизации с кодом по ходу его изменения. По этой причине их применение ограничено.

Псевдокод

Псевдокод позволяет делать наброски реализации функций. Это прелюбопытное изобретение в области проектирования – некий гибридный язык, средний между естественным языком и языком программирования. Его преимущество – независимость от синтаксиса и семантики какого-либо конкретного языка. Вы можете сосредоточиться на задаче, отвлекшись от механизмов языка, и включать для ясности произвольное количество описательного текста.

Однако эти преимущества не столь высоки, если учесть недостатки. Псевдокод необходимо перевести на язык реализации. Можно сразу начать писать на этом языке и сэкономить немного труда. Если псевдокод применяется для документирования проекта, нужно поддерживать его соответствие коду.

PDL (*Program Design Language – язык проектирования программ*) – еще более абсурдная вещь; это формализованный псевдокод. Возможно, кому-то когда-то он помог. Интересно было бы посмотреть на компилятор этого псевдокода.

Проектирование в виде кода

Это удобный неформальный подход к проектированию кода. На начальных этапах проектирования вы пишете код со всеми API и интерфейсами нижнего уровня, но без реализации: одни заглушки, возвращающие вероятные значения, с комментариями, в которых описано, что нужно сделать. Когда проект достигает достаточно высокого уровня разработанности, оказывается, что большая часть кода системы уже написана.

Однако это достоинство носит сомнительный характер, поскольку может ограничить гибкость проекта. При внесении в проект изменений приходится перерабатывать соответствующий объем кода заглушек.

CASE-средства

CASE-инструменты (*computer-aided software engineering – автоматизированные средства разработки программ*) применяются в течение всего процесса проектирования или на отдельных его этапах для автоматизации рутинных процедур и управления рабочим процессом. Большинство из них способно генерировать код (неоднозначного качества) по вашим диаграммам. Некоторые могут даже обновлять диаграммы после модификации кода; это известно как *двусторонняя разработка* (*round-trip engineering* или *round-tripping*). Ряд CASE-средств поддерживает коллективную работу, предоставляя возможность группе программистов трудиться над одним крупным проектом.

Стоит упомянуть о такой разновидности CASE-средств, как инструменты *RAD* (*Rapid Application Development*) – среды для быстрой разработки приложений. Обычно они полезны в своих конкретных областях (как правило, при разработке простых приложений с пользовательским интерфейсом), но не столь эффективны как универсальные модели проектирования программного обеспечения.

Прагматически относитесь к инструментам и методологиям проектирования: применяйте их тогда, когда это действительно полезно, но не становитесь их рабами.

Резюме

Невероятная сложность может обернуться
невероятной простотой.

Уинстон Черчилль

Хороший код правильно спроектирован. Он обладает некой эстетической привлекательностью, которая располагает к нему. Прежде чем писать код, нужно составить его проект, иначе в итоге получится малоприятная путаница. Принимайте во внимание такие аспекты, как понятная структура, возможность расширения в будущем, корректные интерфейсы, уместная абстракция и переносимость на другие платформы. Стремитесь к простоте и элегантности.

Проектирование требует значительного мастерства. Лучшие проекты создаются опытными и умелыми руками. В конечном счете, чтобы получить хороший проект, нужен хороший проектировщик. Среднего уровня программисты не создают выдающиеся проекты.

Хорошие программисты...

- Стремятся привести в порядок все, к чему прикасаются
- Рассматривают программирование как творческий процесс и вносят в свою работу элементы искусства
- Сначала обдумывают структуру кода, а потом пишут его
- Испытывают потребность привести в порядок и переструктурировать запутанный код, прежде чем развивать его дальше
- Постоянно изучают другие проекты, обобщая опыт успехов и неудач

Плохие программисты...

- Закручивают и закручивают код в тугой узел, пока им это удается, а потом недовольны результатом
- Не замечают недостатков проекта и не испытывают неприятных ощущений, работая с бестолковым кодом
- Не стесняются быстро накропать код и убежать, предоставив разгребать завалы кому-то другому
- Не ценят внутреннюю структуру кода, над которым работают; топчут ее самым безжалостным образом

См. также

Глава 8. Время испытаний

Описывает, как *подготовить код для тестирования* – облегчить проверку того, что код работает правильно.

Глава 14. Программная архитектура

Высший уровень проектирования программного обеспечения называется *архитектурой ПО*. Ему свойственны свои специфические проблемы, о которых рассказывается в данной главе.

Глава 19. Спецификации

Проект программного обеспечения часто отражен в спецификациях.

Глава 22. Рецепт программы

Проектирование – часть общего процесса разработки программного обеспечения.

Глава 23. За гранью возможного

Тип создаваемой системы неизбежно отражается на внутреннем устройстве программного обеспечения.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 643.

Вопросы для размышления

1. Как масштаб всей задачи влияет на проект программного обеспечения и работу по его созданию?
2. Что лучше – хорошо документированный плохой проект или хороший проект, но документированный плохо?
3. Как оценить качество проекта по фрагменту кода? Можно ли количественно отразить его простоту, элегантность, модульность и т. п.?
4. Может ли проектирование быть коллективной деятельностью? Насколько важны навыки групповой работы при разработке хорошего проекта?
5. Есть ли зависимость между проектом и наиболее предпочтительной для него методологией?
6. Каким образом можно определить, является ли данный проект сильно связным или слабо сцепленным?
7. Если вам приходилось решать аналогичную задачу проектирования в прошлом, поможет ли это определить, насколько сложна проблема в *данном* случае?
8. Допустимы ли эксперименты в проектировании?

Вопросы личного характера

1. Обратитесь к прошлому и вспомните, как вы учились проектировать код. Смогли бы вы передать полученные вами знания абсолютному новичку в этом деле?
2. Какой опыт у вас есть в использовании конкретных методологий проектирования? Удачный или неудачный? Каким в результате получился код? Какой выбор мог оказаться более удачным?

3. Считаете ли вы, что нужно строго придерживаться выбранной методологии?
4. Какой код из тех, что вам встречались, был спроектирован лучше всего? Хуже всего?
5. Язык программирования является, по существу, инструментом для реализации вашего проекта, а не святыней, о которой можно спорить. Насколько важным является знание идиом языка?
6. Считаете ли вы программирование *технической дисциплиной*, *ремеслом* или *искусством*?

14

Программная архитектура

Фундамент проектирования программ

В этой главе:

- Что такое программная архитектура?
- В чем отличие программной архитектуры и проекта?
- Свойства хорошей архитектуры
- Обзор основных архитектурных стилей

Архитектура – это искусство чем-нибудь заполнить свободное пространство.

Филип Джонсон

Пойдите в город. Встаньте в центре. Оглядитесь. Если вы не выбирали место специально, вокруг окажется множество зданий разных годов постройки и стилей. Одни из них удачно вписываются в свое окружение. Другие выглядят совершенно не на своем месте. Некоторые удовлетворяют эстетическим потребностям и имеют хорошие пропорции. Другие совершенно уродливы. Одни простоят еще 100 лет. Другие – нет.

Архитекторы, проектировавшие эти здания, учли многочисленные факторы, прежде чем взяться за карандаш и бумагу. Во время проектирования они тщательно и методично следили за тем, чтобы проект здания был реализуем, и пытались найти равновесие между соперничающими силами:

Подпольное движение

Я включился в работу над проектом, в котором уже было написано много недокументированного кода. Проект развивался без плана и цели, без архитектора, руководящего процессом строительства. Естественно, он превратился в малоприятное зрелище. Настал момент разобраться, как все это *действительно* работает, и был сделан чертеж архитектуры системы. При этом обнаружилось столько различных компонент (многие в значительной мере избыточные), ненужных взаимосвязей и различных методов обмена данными, что диаграмма представляла собой невообразимое переплетение линий различного цвета (цвет тоже нес информацию) – как будто паука окунули в несколько банок с разными красками, после чего он сплел в помещении какую-то психоделическую паутину.

Потом меня осенило. Да мы же начертили схему лондонской подземки! Сходство было таким сильным, что вызывало ужас – практически непонятное постороннему, со многими маршрутами, ведущими в одну точку, и тем не менее значительное упрощение действительности. Странствующему коммивояжеру такая система доставила бы немало хлопот.

Отсутствие архитектурного видения явно отразилось на программном обеспечении. С ним было трудно работать и разбираться, отдельные функции были разорваны на части, раскиданные по разным модулям. Программа дошла до того состояния, когда лучшее решение – выкинуть ее.

В сооружении программ архитектура играет такую же важную роль, как и в сооружении зданий.

требованиями пользователей, строительными технологиями, возможностью ремонта, эстетикой и т. д.

Для изготовления программ не используются кирпичи и цементный раствор, но и тут надо удовлетворить аналогичному набору требований, поэтому необходимо все тщательно обдумывать. Люди начали строить дома гораздо раньше, чем писать программы, и это дает себя знать. Мы все еще пытаемся разобраться, что считать хорошей программной архитектурой.

В данном небольшом экскурсе в область программной архитектуры мы изучим некоторые стандартные архитектурные шаблоны и рассмотрим, чем программная архитектура в действительности является, а чем – нет, и для чего она применяется.

Что такое программная архитектура?

Может быть, это еще один термин, который с большей натяжкой продолжает метафору *строительства* зданий (см. раздел «Действительно ли бы собираем программы?» на стр. 240)? Возможно, но это действительно полезная концепция. Программную архитектуру иногда еще называют *проектированием верхнего уровня*. Термин *архитектура* лучше ассоциируется с идеями, заложенными в этой процедуре.

План программы

Как архитектор разрабатывает план здания, так и программный архитектор разрабатывает план программного продукта. Однако если архитектурный чертеж представляет собой строго проработанный план, учитывающий все важные детали, то программная архитектура – это определение системы на самом высоком уровне, обзор, в котором избегается излишняя детализация. Это макро, а не микроуровень.

С такой высоты все подробности реализации не видны; мы наблюдаем только важнейшие элементы внутренней структуры программы и основные характеристики ее поведения. Архитектурное представление решает следующие задачи:

- Определяет основные программные модули (или компоненты, или библиотеки; в данном случае их можно называть как угодно).
- Определяет, какие компоненты должны быть связаны между собой.
- Способствует выявлению и определению характера всех основных интерфейсов системы, проясняя *роли* и *ответственности* различных подсистем.

Эти данные позволяют судить о системе в целом, не вникая в работу отдельных ее частей. Архитектура определяет композицию, согласно которой затем осуществляется разработка. Она помогает распределить задачу между отдельными группами разработчиков и оценить различные стратегии реализации.

Архитектура дает не только картину составных частей системы, но и представление о будущем развитии системы. В большой команде разработка продукта проходит более четко, если есть ясное единое видение того, как должно модифицироваться программное обеспечение, что должно содержаться в каждом модуле и как модули должны быть связаны между собой.

Архитектура оказывает самое большое влияние на проектирование и дальнейшее развитие программной системы. По этой причине важно правильно определить ее на самых ранних стадиях разработки.

Будучи ранней стадией деятельности, разработка архитектуры впервые открывает перед нами возможность установить соответствие между

предметной областью (практической задачей, которую нужно решить) и *областью решений*. Не всегда удается установить простое взаимно однозначное соответствие между объектами и действиями из этих двух областей, поэтому архитектура показывает, как одна из них может быть выражена на языке другой.

Конкретные проблемы, решаемые архитектурой, могут разниться в зависимости от проекта. На этом этапе не важно, какой будет целевая платформа; архитектуру можно реализовать на разных машинах с помощью разных языков и технологий. Тем не менее:

- В некоторых проектах бывает необходимо указать конкретные аппаратные компоненты, особенно в случае встроенного программного обеспечения.
- Для распределенных систем количество машин и процессоров, а также распределение нагрузки между ними могут составлять задачу архитектуры. Следует рассмотреть минимальную и среднюю конфигурации.
- Архитектура может описывать конкретные алгоритмы или структуры данных, если они определяют общую конструкцию (хотя это маловероятно).

Всегда приходится балансировать. Чем больше информации заложено на архитектурном уровне, тем меньше места для маневра остается при последующем проектировании или на этапе реализации.

Точки зрения

В физической архитектуре обычно делают несколько разных чертежей или рисунков одного и того же здания: физическую схему, схему электропроводки, схему водоснабжения и канализации и т. д. Точно так же при разработке архитектуры программного продукта создается несколько ее представлений. Обычно рассматривают четыре представления.

Концептуальное представление

Иногда его еще называют *логическим*. Оно показывает основные части системы и их взаимосвязи.

Представление реализации

Это представление с точки зрения конкретно реализуемых модулей, которое может отличаться от чисто концептуальной модели.

Представление процесса

Оно должно показать динамику структуры с точки зрения задач, процессов и связи; наиболее полезно, когда разработка ведется с высокой степенью параллелизма.

Представление развертывания

Это представление применяется для размещения задач в физических узлах распределенной системы. Например, можно распределить

Цена вопроса

Программная архитектура имеет далеко идущие последствия, касающиеся не только начальной структуры кода, но и глубоко затрагивающие производственный процесс. Архитектура накладывает долгий отпечаток как на технологию, так и на практическую область. От архитектуры зависит, как будет развиваться код и как команды разработчиков станут совместно расширять его. Трехуровневая архитектурная модель приведет к формированию трех команд разработчиков, каждая из которых будет работать над своей частью задачи. Не исключено, что управленческий состав также разделится на три части, и возникнет три линии ответственности. От того, какое проектное решение кто-то примет вначале, будет зависеть, на каком рабочем месте вы окажетесь.

Поскольку архитектура определяет, в какой мере программный продукт будет открыт для модификаций и сможет ли основной код быть дополнен функциями, которые удовлетворят будущие потребности, постольку она определяет коммерческий успех вашей фирмы. Плохая архитектура не просто создает неудобства – она может подорвать ваше материальное благополучие. Дело нешуточное.

Самым непосредственным образом архитектура отражается на нас, программистах – от нее зависит, насколько интересной будет наша работа. Кто же захочет потеть, чтобы добавить мелкую функцию, которую при правильном проектировании можно было бы реализовать в две секунды! На концептуальном уровне следите, чтобы архитектура поддерживала то, что требуется от нее по вашему мнению, а не по представлениям архитекторов.

функциональность между сервером базы данных и пулом шлюзов веб-интерфейсов.

Не нужно браться за все перечисленное сразу. Конкретные представления возникают по мере дальнейшей разработки. Главным результатом начального архитектурного этапа должно быть *концептуальное представление*; на нем мы и сосредоточим свое внимание.

Где и когда этим заниматься?

Архитектура описывается в одном из важных документов, носящем какое-нибудь образное название, например *спецификация архитектуры*. В этом документе рассказывается о структуре системы и доказывается, что она удовлетворяет системным требованиям, в том числе таким существенным, как стратегия достижения показателей эффективности и отказоустойчивости.

Чья это работа?

Как мы видели, архитектура программного обеспечения затрагивает *всех участников* проекта, а не только программистов. С другой стороны, выработка архитектуры осуществляется значительно меньшей по числу группой людей. На нее падает большая ответственность.

Разработчик архитектуры называется *программным архитектором*. Это внушительный титул, отчасти спорный, как и «инженер». «Настоящие» архитекторы должны пройти обучение, сертификацию и достичь определенного профессионального мастерства, чтобы только называться архитекторами. В мире программирования такие требования не предъявляются.

Программные архитекторы принадлежат к числу инициаторов проекта, участвуя в нем с самого начала разработки. По мере того как разработка проекта продвигается вперед, к нему присоединяются программисты, осуществляющие реализацию выработанной архитектуры.

Однако в мелких проектах, в меньшей мере требующих специального опыта в области архитектуры, ее разрабатывают сами программисты. Обходятся без артиллерии главного калибра. Будьте готовы принять участие в архитектурных разработках.

Храните описание системной архитектуры в таком месте, где оно доступно всем, кому может потребоваться: программистам, ответственным за сопровождение и установку, менеджерам, а возможно, и клиентам.

Архитектура является начальным проектом системы. Поэтому она являет собой *первый* шаг в разработке после утверждения технических требований. Необходимо заранее выработать спецификации, потому что выработка архитектуры оказывается первой возможностью рассмотреть и проверить проектные решения, которые окажут самое существенное влияние на проект. Здесь могут выявиться слабые места и потенциальные проблемы. Если отвергнуть неудачное решение еще на этом этапе, можно сэкономить много времени, сил и денег. Менять основание системы, когда на нем будет построен большой объем кода, значительно дороже.

Архитектурная работа – это форма проектирования, отличная от стадии проектирования модулей и низкоуровневого проектирования кода, хотя они частично перекрываются. Последующая работа по детальному проектированию может подсказать изменения, которые нужно внести в архитектуру системы. Это естественно и нормально.

Для чего она применяется?

Архитектура – начальный проект системы. Но ее действие простирается шире. Мы пользуемся системной архитектурой с целью:

Проверки правильности

Архитектура – это наша первая возможность подтвердить правильность того, что мы собираемся строить. С ее помощью можно мысленно проверить, что система будет удовлетворять всем требованиям. Можно проверить, что такую систему действительно удастся построить. Мы можем гарантировать внутреннюю согласованность и увязку проекта, отсутствие специальных случаев или необоснованных ухищрений. Плохая работа на верхнем уровне проекта приводит к еще более опасным хакам на нижних уровнях.

Архитектура помогает избежать дублирования в работе, напрасной траты сил и избыточности. С ее помощью мы проверяем, не пропущено ли что-нибудь в нашей стратегии, все ли необходимые части включены. Мы гарантируем, что не возникнет несоответствий, когда отдельные части будут сведены вместе.

Распространения информации

С помощью спецификации архитектуры мы сообщаем информацию о проекте всем заинтересованным сторонам. В их число могут входить проектировщики системы, кодировщики, сопровождающие, тестеры, клиенты и администраторы. Это основное средство, позволяющее разобраться в системе, и важная составляющая документации, которая должна *постоянно* поддерживаться в актуальном состоянии по мере внесения изменений.

Спецификация архитектуры – важное средство для информирования о состоянии вашей системы. Следите за тем, чтобы она соответствовала состоянию программного обеспечения.

Архитектура передает видение системы, отображая предметную область на область решений. Она должна ясно определять, каким образом к ней могут подключаться будущие расширения, способствуя поддержанию *концептуальной целостности* системы (Brooks 95). Неявным образом она обеспечивает ряд соглашений и содержит элементы стиля. Понятно, например, что не следует добавлять компоненту, в которой обмен данными происходит на базе сокетов, если во всем остальном проекте используется инфраструктура CORBA.

Архитектура предоставляет естественный путь к следующему уровню проектирования, не будучи при этом излишне ограничительной.

Выявления различий

Архитектура помогает нам принимать решения. Например, она отражает решения в пользу самостоятельной разработки или покупки готовых компонент, определяет, нужна ли база данных и уста-

навливает стратегию обработки ошибок. Она выявляет проблематичные области и области особого риска для проекта, способствуя выработке мер по минимизации этого риска. Главная задача архитектора-строителя – добиться, чтобы проектируемое им здание не рухнуло после завершения строительства, если будут соблюдены все оговоренные условия (даже в каких-то неожиданных ситуациях). Так и мы должны обеспечить запас прочности для нашей программной структуры. Здание не должно рассыпаться, если возникнет слабый ветерок или некоторая дополнительная нагрузка.

Такая общая точка зрения на систему необходима нам, чтобы выбрать разумные компромиссы, гарантируя при этом соответствие проекта предъявляемым к нему требованиям. Такие важные задачи лучше решать в самом начале, а не пытаться с запозданием вносить изменения, когда близится конец разработки.

Все проектные решения для программного продукта принимайте в контексте архитектуры. Следите за тем, чтобы не отклоняться от системного видения и стратегии. Не делайте мелких ни к чему не относящихся добавлений.

Компоненты и соединения

Архитектура занята в основном компонентами и соединениями. Она определяет количество и тип тех и других.

Компоненты

Архитектура хранит информацию обо всех *компонентах*, что бы ни значили они в ее контексте. Компонентами могут быть объект, процесс, библиотека, база данных или продукт стороннего разработчика. Каждая из системных компонент определяется как понятный и логичный блок. Каждая решает лишь одну задачу и делает это хорошо. Никакие компоненты не содержат некой штуковины, если для этой штуковины есть конкретный модуль.

Не занимаясь проблемами реализации компонент, архитектура описывает все предоставляемые продуктом функции, а возможно, и важнейшие внешние интерфейсы. Она определяет *область видимости* компонент: что *они* могут видеть, а что нет, и кто *их* может видеть, а кто нет. Как будет показано далее, разные архитектурные стили устанавливают разные правила видимости.

Соединения

Архитектура устанавливает все соединения между компонентами и описывает их свойства. Соединение может представлять собой обычный вызов функции или поток данных в канале. Оно может быть обработчиком событий или передачей сообщений через какой-либо механизм ОС или сети. Соединения бывают *синхронными* (блокирующими вызвавшего до завершения обработки запроса) или *асинхронными* (сразу воз-

вращающими управление вызвавшему и использующими какой-то механизм последующей передачи ему ответа). Это имеет значение, поскольку влияет на порядок передачи управления внутри системы.

Иногда соединение бывает косвенным (и потому скрытым). Например, компоненты могут располагать общими ресурсами и обмениваться данными с их помощью – подобно размещению сообщений на доске объявлений. Вот примеры совместно используемых каналов связи: зависимые компоненты, общая область памяти или такая простая вещь, как содержимое файла.

Архитекторы и маркетологи

Архитектура оказывается неадекватной, если она не удовлетворяет требованиям к продукту по начальному развертыванию или развитию в будущем; качество проектирования не ограничивается совершенством технического исполнения. Технические задачи должны решаться попутно с задачами управления продуктом и вопросами маркетинга.

Нет смысла разрабатывать продукт, который никому не нужен; это будет пустой тратой времени. Но если не рассматривать с технической точки зрения требования рынка, можно упустить важные возможности для бизнеса. Отдел маркетинга определяет основные задачи бизнеса, в том числе стратегию продаж (взимать разовую плату или применять модель лицензирования/выставления счетов?), позиционирование продукта на рынке (будет ли это продукт верхней линейки, нашпигованный функциями и дорогостоящий, или дешевый массовый товар?) и значимость уникального бренда, под которым выпускается система.

В некоторых случаях открытая на показ хорошая архитектура может стать выигрышным моментом и создать значительное конкурентное преимущество. На других рынках внутренняя структура системы может иметь меньшее значение, но все же наличие архитектуры, готовой принять требования, которые могут возникнуть в будущем у клиентов, имеет важное значение для создания и сохранения сильной позиции на рынке.

Технические архитекторы должны тесно сотрудничать с маркетологами, чтобы представлять себе место нового продукта в общей стратегии компании и мнение клиентов о том, какое решение они сочли бы действительно выдающимся. Архитектура программного продукта должна учитывать проблемы маркетинга – юзабилити, надежность, обновляемость и расширяемость. Каждый из этих факторов оказывает реальное влияние на проектирование программного продукта. Одна лишь поддержка различных способов оплаты может сильно повлиять на прибыльность проекта –

обеспечение мощной поддержки регистрации активности системы дает возможность применения схемы оплаты отдельных операций, которая может увеличить прибыльность продукта. Однако при этом может потребоваться предусмотреть в архитектуре дополнительные меры безопасности и защиты от мошенничества.

Требования маркетинга должны учитываться в технической архитектуре. И наоборот, технические соображения оказывают влияние на маркетинговую стратегию. По-настоящему выдающаяся архитектура возникает тогда, когда техническое и стратегическое видение соединяются между собой, чтобы создать продукт, опережающий своих конкурентов.

Какими качествами должна обладать архитектура?

Ключ к созданию хорошей архитектуры – *простота*. Задача в том, чтобы правильно выбрать несколько модулей и разумно организовать между ними связь. Архитектура также должна быть *понятной*, что часто означает наглядность представления. Всем известно, что *лучше один раз увидеть, чем сто раз услышать*.

Хорошая архитектура системы должна быть простой. Достаточно бывает описания, состоящего из одного абзаца и одной простой диаграммы.

В правильно спроектированной системе должно быть не слишком мало и не слишком много компонент. Понятно, что все зависит от объема задачи. Если программа маленькая, ее архитектура может уместиться (и даже быть разработана) на одном листке – несколько модулей и простых связей между ними. Для крупной системы потребуется больше труда и больше листков.

Если компоненты *многочисленны* и мелки, то с такой архитектурой трудно работать. Это происходит, когда архитектор слишком углубляется в детали. Если компонент *слишком мало*, это означает, что каждому модулю поручен слишком большой объем работ; структура становится нечеткой, сложной для сопровождения и расширения. Следует стремиться к золотой середине.

Архитектура не определяет, как устроен каждый модуль внутри – для этого существует проектирование модулей. Нужно добиваться, чтобы каждый модуль как можно меньше знал об остальных составных частях системы. Наша цель – слабое связывание и высокая связность (см. раздел «Модульность» на стр. 325) на этом и на всех других уровнях проектирования.

Архитектура определяет важнейшие компоненты системы и их взаимодействие. Она не описывает их устройство.

В спецификации архитектуры приводятся принятые конструктивные решения и дается обоснование, почему тот или иной подход получил предпочтение перед возможными альтернативами. Нет необходимости подробно рассматривать эти альтернативы – достаточно обосновать сделанный выбор и показать, что он был вполне обдуманным. В спецификации должна быть правильно определена главная задача системы; например, *высокая производительность* и *легкость расширения* конфликтуют между собой и требуют различных архитектурных решений.

Хорошая архитектура оставляет пространство для маневра; она позволяет изменить свое решение. В ней может быть определено, что компоненты сторонних разработчиков будут заключены в оболочки абстрактных интерфейсов, а потому не составит труда заменить одну их версию другой. Можно предложить технологии, позволяющие выбирать между различными реализациями во время развертывания. По мере того как проект набирает силу, становится ясным, какие варианты реализации нужно выбрать, что не всегда очевидно на ранних стадиях. Удачная архитектура должна быть гибкой, чтобы быстро менять решения во время начальной неопределенности. Архитектура – первая точка, где происходит уравновешивание противодействующих сил; она показывает, какие уступки мы делаем в пользу одного качества и в ущерб другому.

Хорошая архитектура оставляет пространство для маневра, расширения, модификации. Но ее общность не переходит разумных границ.

Архитектура должна быть ясной и недвусмысленной. Предпочтительнее хорошо знакомые архитектурные стили и известные структуры (подробнее о них см. в следующем разделе). Архитектура должна быть простой для понимания и легкой для работы.

Как всякий хороший проект, хорошая архитектура должна обладать определенной эстетической привлекательностью, вызывающей к ней *доверие*.

Архитектурные стили

Форму в архитектуре определяет функция.

Луис Генри Салливен

Подобно тому как огромный готический собор, причудливая викторианская часовня, внушительный квартал высотных зданий и общественный туалет 70-х представляют собой различные архитектурные стили, существует ряд признанных стилей в архитектуре программного обеспечения, в соответствии с которыми может строиться система.

Выбор стиля может быть обусловлен многими причинами, не всегда вескими. Например, он может осуществляться исходя из прочных технологических обоснований, прежнего опыта работы архитектора и даже соображений моды. Каждой архитектуре свойственен определенный набор свойств:

- Устойчивость к изменениям в представлении данных, в алгоритмах и необходимых функциях
- Способ разделения и соединения модулей
- Удобопонятность
- Приспосабливаемость к требованиям производительности
- Возможность повторного использования компонент

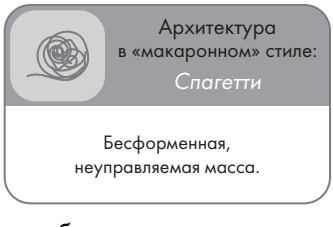
На практике в рамках одной системы встречается смесь разных стилей. В одних частях системы обработка данных может осуществляться с помощью каналов и фильтров, в других частях может применяться компонентная архитектура.

Разберитесь с основными архитектурными стилями и оцените их достоинства и недостатки. Это поможет вам более благожелательно относиться к программным продуктам, с которыми вы столкнетесь, и правильно проектировать системы.

В последующих разделах описываются некоторые популярные архитектурные стили и сравниваются с неким «макаронным» стилем.

Без архитектуры

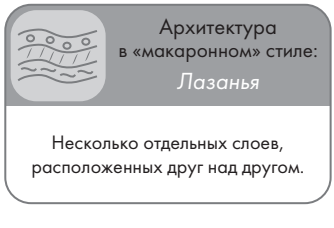
Архитектура у системы есть всегда, но, как в моем проекте «Лондонская подземка», она может быть *незапланированной*. Очень скоро такое положение дел оказывается ярмом для команды разработчиков. В получаемом при этом программном продукте царит полная неразбериха.



Определить архитектуру необходимо, если вы собираетесь создать хороший программный продукт. Без планирования архитектуры вы заранее обрекаете свою разработку на гибель.

Многоуровневая архитектура

Это, вероятно, наиболее употребительный стиль в концептуальных представлениях. Он описывает систему в виде иерархических уровней с помощью конструктивных элементов. Разобраться в такой модели очень легко; даже неспециалист быстро схватит ее идею.



Каждая компонента представлена одним блоком в стопке. Расположение их в стопке показывает, что где находится, как связаны между собой компоненты и каковы возможности одних компонент «видеть» другие. Блоки могут размещаться бок о бок на одном уровне и иметь высоту, охватывающую два уровня. Известным примером служит семиуровневая эталонная модель OSI для систем сетевой связи (ISO 84). Интереснее семиуровневая модель бисквита «Гудлифс», показанная на рис. 14.1.

На нижнем уровне стека располагается аппаратный интерфейс, если система действительно взаимодействует с физическими устройствами. В других случаях этот уровень остается за базовыми сервисами, например ОС или промежуточными технологиями типа CORBA. Самый верхний уровень, вероятно, займет интерфейс, с которым взаимодействует пользователь. Подымаясь вверх по стеку, мы удаляемся от аппаратного уровня, успешно изолируясь от него промежуточными слоями, подобно крыше дома, которую мало волнуют магматические процессы в ядре Земли.

В любой момент можно выкинуть находящиеся внизу уровни и вставить новую реализацию низлежащего уровня – система продолжит работу, как прежде. Это важный момент: он означает, что вы можете выполнять один и тот же код C++ на любой платформе, поддерживающей вашу среду C++. Можно перейти на другую аппаратную платформу, не меняя код приложения, – уровень ОС (например) переварит все технические отличия. Удобно.

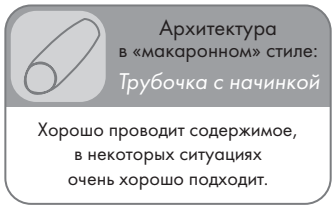
Верхние уровни пользуются открытыми интерфейсами уровней, находящихся непосредственно под ними. Смогут ли они пользоваться открытыми интерфейсами еще более глубоко расположенных уровней, зависит от того, как вы определите уровни. Иногда диаграмму подрабатывают так, чтобы она отражала такую возможность, как в случае

Рис. 14.1. *Семиуровневая эталонная модель бисквита «Гудлифс»*

блока хереса в стеке бисквита. Могут ли взаимодействовать между собой компоненты одного и того же уровня, тоже строго не определено. Определенно нельзя пользоваться тем, что находится на более высоком уровне; если нарушено это правило, значит, у вас не многоуровневая архитектура, а бессмысленная диаграмма, изображенная в виде стека.

Как можно видеть, большинство диаграмм уровней носит неформальный характер. Относительный размер и положение прямоугольников свидетельствуют о важности тех или иных компонент, и этого обычно достаточно для общего обзора. Соединения между компонентами явно не показаны, а методы связи несущественны. (Однако они могут быть важнейшим архитектурным фактором для эффективности системы – не станете же вы пересылать гигабайты данных через последовательный порт RS232.)

Архитектура с каналами и фильтрами

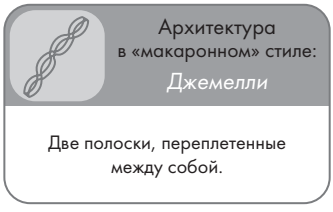


Эта архитектура моделирует логический поток данных через систему. Она реализуется в виде цепочки последовательно действующих модулей, каждый из которых читает некоторые данные, обрабатывает их и выдает дальше наружу. В начале цепочки находится генератор данных (это может быть интерфейс пользователя или некое аппаратное устройство). В конце находится устройство приема данных (например, дисплей компьютера или журнальный файл). Это старая игра в испорченный телефон, только теперь в цифровом виде. Данные проходят по каналу, встречая по дороге различные фильтры. Преобразования обычно носят нарастающий характер; каждый фильтр осуществляет одну простую операцию и не стремится сохранять свое состояние.

Архитектура каналов и фильтров требует четкого определения структуры данных, которые проходят через фильтры; неявно возникают накладные расходы в связи с повторяющимся кодированием выходных данных для передачи по каналу и обратным анализом их на входе каждого фильтра. По этим причинам поток данных обычно организован очень просто – в формате простого текста.

В такой архитектуре легко добавлять новые функции путем включения в канал нового фильтра. Главный ее недостаток – обработка ошибок. Когда в приемнике данных обнаруживается проблема, трудно узнать, в каком месте канала лежит ее источник. Передавать коды ошибок по цепочке затруднительно – их нужно дополнительно кодировать и сложно организовать единообразную обработку в нескольких отдельных модулях. Для ошибок фильтры могут использовать отдельный канал (например, `stderr`), но сообщения об ошибках могут легко перемешаться.

Архитектура клиент/сервер



Будучи типичной сетевой архитектурой, модель клиент/сервер делит функциональность на две главные части: *клиент* и *сервер*. Она отличается от прежнего стиля проектирования работы в сети с *мэйнфреймом* тем, как делится работа между участниками. «Клиент» мэйнфрейма – это тупой терминал, возможности которого немногим шире, чем перехват и отправка нажатий на клавиши, а также вывод некоторых данных на дисплей.

Клиенты в архитектуре клиент/сервер богаче функциями, более интеллектуальны и обычно могут представлять данные в графическом и интерактивном стиле. Рассмотрим подробнее роли этих двух элементов:

Сервер

Сервер предоставляет клиентам ряд четко определенных сервисов. Обычно это мощный компьютер, специально предназначенный для предоставления конкретной функциональности или управления ресурсами (общими файлами, принтерами, базой данных или пулом вычислительных мощностей).

Сервер ждет запросы от клиентов и отвечает на них. Количество одновременно обрабатываемых соединений с клиентами может быть произвольным или подчиняться некоторой ограничительной схеме.

Клиент

Клиент пользуется сервисами сервера. Он посылает запросы и обрабатывает полученные результаты. Клиенты могут быть выделенными терминалами, выполняющими только одну функцию, либо выполнять много функций (например, «клиентское» приложение может выполняться на стандартном ПК, с которого можно также читать веб-страницы и электронную почту).

Услугами одного и того же сервера могут пользоваться клиенты разных типов, посылая запросы из одного и того же набора, но разными способами. Один клиент может иметь веб-интерфейс, другой – интерфейс GUI, а третий действовать из командной строки.

По понятным причинам метод клиент/сервер иногда называют *двухуровневой* архитектурой. Он очень распространен в сфере разработки программного обеспечения. Встречаются разные способы связи между клиентом и сервером. Проще всего использовать стандартные сетевые протоколы, но можно также применять удаленный вызов процедур (RPC), удаленные запросы к базам данных SQL или даже собственные специфичные для приложения протоколы.

Есть разные способы разделения труда между этими двумя компонентами. Основная логика приложения (так называемая *бизнес-логика*)

может выполняться как клиентом, так и сервером, в зависимости от того, насколько интеллектуальным и специализированным предполагается клиент. Чем больше логики спускается клиенту, тем менее гибкой становится конструкция – клиентам приходится реализовывать идентичные функции, что снижает преимущества, достигаемые за счет центрального сервера. Обычно задача клиента в том, чтобы обеспечить удобный интерфейс между человеком и функциями, предоставляемыми сервером.

Типы интерфейсов

Основным принципом конструирования программного обеспечения является *модульность* – создание систем из заменяемых компонент. Это похоже на конструктор «Лего». При правильной организации ничто не должно помешать вам вынуть голубой брусок и заменить его более приятным красным. Если бруски одинаковой формы и размера, с одинаковым типом соединения, их можно вставить в одно и то же место, и они будут выполнять одинаковые функции.

Как это реализуется в программном обеспечении? Мы определяем *интерфейсы*; это наши точки соединения и границы компонент. Они устанавливают размер и форму каждой компоненты (по крайней мере, как они видятся снаружи) и определяют, что нужно сделать для равноценной замены. Вот основные типы интерфейсов:

API

Интерфейсы прикладного программирования (*API*) определяются как совокупности функций в физически скомпонованном приложении. Чтобы заменить компоненту, реализующую определенный API, необходимо заново реализовать все функции и снова собрать код.

Иеарархии классов

Можно спроектировать абстрактный класс «интерфейса» (в Java и C# вы фактически определяете интерфейс). После этого можно создать сколько угодно производных классов, которые будут реализовывать этот интерфейс.

Компонентные технологии

Такие технологии, как COM и CORBA, позволяют программе определять нужную реализацию компоненты на этапе выполнения. Обычно интерфейсы определяются с помощью абстрактного *языка определения интерфейсов IDL* (*Interface Definition Language*). Прелесть такого метода в том, что компоненты можно написать на любом языке. Необходимы поддержка со стороны промежуточного ПО или ОС.

Форматы данных

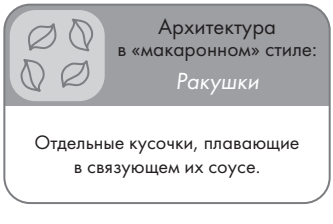
Эти форматы могут образовывать точки соединения в проектах, ориентированных на перемещение данных, а не потока управления. Любую компоненту в цепочке данных можно заменить аналогичной, работающей с теми же типами данных.

Как явствует из сказанного, архитектура – да, по большей части, и проектирование программного обеспечения – занимается разработкой подходящих интерфейсов. Каждая из этих технологий интерфейсов соответствует определенному архитектурному стилю. Выбирайте механизм интерфейсов, который дополняет архитектуру.

Иногда встречается расширение данной двухуровневой схемы, когда вводится еще один (*промежуточный*) уровень. Этот уровень специально создается для реализации бизнес-логики, отделяя ее как от клиентского приложения (которое теперь определенно ограничивается интерфейсом), так и от хранилища данных на сервере. Это *трехуровневая* архитектура.

Архитектура клиент/сервер отличается от *одноранговой* (*peer-to-peer*) архитектуры, в которой никакие сетевые узлы не обладают особыми возможностями или значением по сравнению с остальными. Одноранговые архитектуры труднее развертывать, но они отличаются повышенной устойчивостью к отказам. Архитектура клиент/сервер оказывается парализованной, когда недоступен сервер (вследствие сбоя программы или планового обслуживания): ни один клиент не сможет работать, пока сервер не задышит снова. По этой причине системы, организованные по принципу клиент/сервер, обычно требуют наличия администратора, который обеспечивает их бесперебойную работу.

Компонентная архитектура



В этой архитектуре управление децентрализовано, и вместо единой монолитной структуры она разбита на ряд отдельных взаимодействующих между собой *компонент*. Это объектно-ориентированный подход, который не требует обязательной реализации на каком-либо из ОО-языков. Открытые интерфейсы всех компонент обычно определяются на *языке определения интерфейсов* (*IDL*) и отделены от какой-либо реализации, хотя в некоторых компонентных технологиях (например, встроенной поддержке компонент .NET) могут определяться в самом коде реализации.

Архитектура, основанная на компонентах, возникла из соблазна быстро собирать приложения из готовых деталей, возможно, даже втыкая их на ходу. Единого мнения по поводу того, насколько успешным оказался такой подход, все еще нет. Не все компоненты имеют возможность многократного использования (оно достигается тяжелыми усилиями), и не всегда легко найти компоненту, которая делает то, что вам требуется. Проще всего, если речь идет об интерфейсе пользователя, для которого существуют популярные структуры и устоявшиеся рынки.

Стержнем компонентной архитектуры является инфраструктура коммуникаций, или *промежуточное* программное обеспечение, благодаря которому компоненты могут подключаться, оповещать о своем существовании и объявлять о предоставляемых ими услугах. Связи между компонентами не фиксируются жестко, а устанавливаются в результате поиска этой информации через механизм промежуточного слоя. В число обычных компонентных платформ входят CORBA, JavaBeans и COM; у каждой из них есть свои достоинства и слабые стороны.

Компонента[1], по существу, является отдельным блоком реализации. Она предоставляет один или несколько специфических открытых интерфейсов IDL. Интерфейс определяет порядок взаимодействия клиентов с компонентой. Тайные входы отсутствуют. Клиента должно интересовать, как работать с экземпляром этого интерфейса, а не как реализована компонента.

Каждая компонента – это отдельный независимый блок кода. За видимым интерфейсом находятся логика реализации (это может быть бизнес-логика или функция пользовательского интерфейса и т. п.), а также некоторые данные, которые могут быть локальными или открытыми (например, если компонента хранит файлы или базу данных). Компоненты не должны слишком много знать друг о друге. Если они тесно связаны между собой, то такая архитектура представляет собой просто запутанную монолитную систему.

Компонентная архитектура может быть развернута в сети с размещением компонент на разных машинах, но с таким же успехом ее можно установить на одной-единственной машине. Все может зависеть от типа применяемого промежуточного программного обеспечения.

---

[1] Мы уже говорили о компонентах как о модулях – эфемерных единицах реализации. Но здесь приводится новое определение этого слова, весьма специфичное для области, основанной на компонентах архитектуры. К сожалению, эти термины несут в себе несколько значений.

Каркасы

Архитектура в «макаронном» стиле: *Готовые равиоли*

По большей части работа уже сделана за вас. Остается сварить и подать на стол.

Вместо того чтобы разрабатывать новую архитектуру для конкретного проекта, может оказаться удобным воспользоваться *каркасом* (структурой) существующего *приложения* и доработать его. Каркас (framework) – это расширяемая библиотека кода (обычно группа взаимодействующих классов), которая образует пригодное для повторного использования проектное решение для конкретной предметной области. Большая часть работы в каркасе уже сделана за вас, а оставшаяся – похожа на вписывание недостающего в пустые поля. Различные каркасы отвечают разным архитектурным моделям; воспользовавшись каркасом, вы принимаете его архитектурный стиль.

Каркасы отличаются от обычных библиотек способом взаимодействия с вашим кодом. Используя библиотеки, вы явным образом обращаетесь к компонентам в своем собственном потоке управления. С каркасом все наоборот: он несет ответственность за структуру и поток управления. Он по мере необходимости посылает вызовы в добавленный вами код.

Близко к готовым каркасам лежат *паттерны* архитектурных проектов. Не будучи самостоятельным архитектурным стилем, паттерны представляют собой малоразмерные архитектурные шаблоны. Это микроархитектуры для нескольких взаимодействующих компонент, вобравшие в себя повторяющуюся структуру обмена данными. Архитектурные паттерны описывают стандартные структуры компонент на уровне архитектурного проектирования, разъясняя, как они удовлетворяют требованиям данного контекста. Паттерны – это набор лучших приемов проектирования, описанных в вездесущей книге «банды четырех» (Gamma et al. 94) и многочисленных последующих публикациях (см. врезку «Шаблоны проектирования» на стр. 334).

Резюме

Римский архитектор Витрувий сделал бессмертное высказывание по поводу того, что составляет хороший архитектурный проект: прочность (*firmitas*), полезность (*utilitas*) и красота (*venustas*). Это вполне справедливо в отношении архитектуры программного обеспечения. В отсутствие четко определенной архитектуры с четкими коммуникациями программному проекту будет не хватать связной внутренней структуры. Он станет хрупким, нестабильным и уродливым. В конечном итоге в нем возникнут разрушительные напряжения.

После всех этих разговоров о еде я проголодался. Иду строить семиуровневый эталонный бисквит...

Хорошие программисты...

- Понимают архитектуру своего программного продукта и пишут код в ее рамках
- В каждом проектном сценарии могут подобрать подходящую архитектуру
- Создают простые архитектуры, обладающие красотой и изяществом – ценят эстетические достоинства программного проекта
- Описывают архитектуру системы в документе, который постоянно обновляется
- Сообщают о проблемах структуры системным архитекторам, пытаясь улучшить проект

Плохие программисты...

- Пишут код, не обращая внимания на общие архитектурные представления, что приводит к порочному коду и неинтегрируемым компонентам
- Начинают писать код, не выполнив проектирования на верхнем уровне и игнорируя альтернативные архитектуры
- Хранят данные об архитектуре у себя в голове или в недопустимо устаревших спецификациях
- Смиряются с неадекватными архитектурами, добавляя новый плохо спроектированный код и тем самым усугубляя лежащие в основе проблемы; они не понимают, что их ждут более крупные неприятности

См. также

Глава 12. Комплекс незащищенности

Вопросы безопасности должны учитываться в архитектуре системы.

Глава 13. Важность проектирования

Проектирование кода – необходимый элемент создания кода.

Глава 15. Программное обеспечение – эволюция или революция?

Жизнь вашего программного продукта начинается с архитектуры, но это не единственное, что определяет его разработку.

Глава 22. Рецепт программы

Какое место архитектурный проект занимает в процессе разработки программного продукта.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 646.

Вопросы для размышления

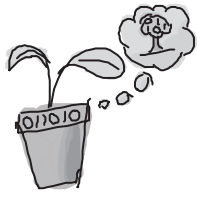
1. Объясните, где кончается *архитектура* и начинается *проектирование программного продукта*.

2. Как плохая архитектура может сказаться на качестве системы? Есть ли такие части, на которые недостатки архитектуры не оказывают влияния?
3. Трудно ли исправить выявившиеся недостатки архитектуры?
4. В какой мере архитектура оказывает влияние на:
   a. Конфигурацию системы
   b. Ведение журналов
   c. Обработку ошибок
   d. Безопасность
5. Какой опыт или подготовка необходимы для того, чтобы называться *программным архитектором*?
6. Должна ли стратегия рыночного поведения оказывать влияние на архитектуру? Если да, то каким образом? Если нет, то почему?
7. Какими особенностями должна обладать архитектура, рассчитанная на *расширение*? Рассчитанная на максимальную *производительность*? Какое влияние эти проектные цели оказывают на систему и в каких отношениях они находятся между собой?

Вопросы личного характера

1. Насколько широк диапазон архитектурных стилей, которые вы применяете в своей работе? В каком из них у вас больше опыта и как это влияет на программы, которые вы пишете?
2. Каков ваш личный опыт работы с архитектурами, оказавшимися успешными или неудачными? Какие особенности сделали их правильным выбором или помехой в работе?

3. Предложите каждому разработчику в вашем текущем проекте изобразить архитектуру системы – самостоятельно (без обсуждения с коллегами) и без обращения к системной документации или коду. Сравните их схемы. Что вас больше всего поражает в результатах их труда (исключая чисто художественные достоинства)?
4. Есть ли в вашем текущем проекте общедоступное описание архитектуры? Давно ли оно обновлялось? Какими видами представления вы пользуетесь? Если вам потребуется рассказать о системе новому сотруднику или будущему клиенту, какие данные вам реально потребовались бы в документе?
5. Сравните архитектуры своей системы и ваших конкурентов. Благодаря каким особенностям своей архитектуры вы рассчитываете добиться успеха своего проекта?



15

Программное обеспечение – эволюция или революция?

Как развивается код?

В этой главе:

- Как код развивается со временем
- Гниение программного обеспечения – как происходит разложение
- Как справляться с рисками, связанными со старением кода

Не знаю, будет ли лучше, если все изменится; но чтобы стало лучше, всё должно измениться.

Г. Лихтенберг

Если бы программы могли развиваться, как растения... Посадишь семечко идеи в какую-нибудь плодородную программистскую почву, польешь водичкой и ждешь! Заботливо ухаживаешь за ними: вносишь удобрения, заботишься об освещении и отгоняешь птиц. В какой-то момент появляется росток кода, и когда программа достаточно подрастет, можно показать ее миру. Если нужны новые функции, продолжаешь поливать его и удобрять почву, и оно развивается дальше. Ствол становится все прочнее и может поддерживать новые ветви, а программа сохраняет равновесие. Если она растет в нежелательном направлении, можно состричь ненужные ветви, и программа снова выпрямится.

К сожалению, в действительности все не так. Далеко не так.

Программное обеспечение – живой объект. Оно не наделено сознанием или свойствами растений, но у него есть своя жизнь: момент зачатия, неуклонное развитие и окончательное достижение зрелости. Затем его выпускают в большой мир, где оно должно зарабатывать себе на жизнь

Еще несколько метафор для построения программ

Мы уже рассматривали метафору здания и обсуждали, как она применима к процессу *построения* программного обеспечения (см. раздел «Действительно ли мы собираем программы?» на стр. 240). В этой главе я приведу еще несколько метафор. Они бросают новый взгляд на наши методы программирования:

Выращивание программного продукта

Относится к тому, как мы *расширяем* имеющееся программное обеспечение – обычно путем добавления новых функций. Исправление ошибок – это не рост, а уход за больными частями кода.

Код *растет*, когда мы вносим в него дополнения, но программирование не вполне аналогично выращиванию растений – у нас гораздо больше возможностей управлять и влиять на рост кода, чем на развитие растений. Код растет, скорее как жемчужина в устрице, путем медленного, но постоянного наращивания небольших дополнительных участков.

Эволюция программного продукта

Другая распространенная метафора строительства – это *эволюция* программного обеспечения. Мы начинаем с простейшего одноклеточного организма кода и наблюдаем, как он постепенно превращается в крупный и более сложный. Это процесс наращивания; программное обеспечение проходит через ряд стадий эволюции. Однако, в сравнении с биологической эволюцией, есть ряд важных различий:

- Мы намеренно совершаем изменения; программное обеспечение не может развиваться самостоятельно.
- Мы не пользуемся *естественным отбором* при выборе лучшей конструкции. У нас нет ни времени, ни желания разрабатывать многочисленные варианты одной и той же программы.

У нас есть возможность итеративно совершенствовать свой код, частично подражая эволюционному развитию. Мы можем воспользоваться опытом, полученным при разработке предыдущих версий продукта, чтобы приспособить код к его среде обитания и обеспечить ему более долгую жизнь.

и может заслужить уважение и восхищение. Оно может расти дальше, возможно, достигнув среднего возраста и утратив юношескую привлекательность. Со временем оно изнашивается и стареет и в конечном счете уходит на покой, попадая в цифровой отстойник, где может спокойно умереть.

В этой главе мы посмотрим, как культивируют программы, особенно после начального цикла разработки. Программы нуждаются в продуманном уходе и редко получают те заботу и внимание, которых заслуживают. Что можно сделать, чтобы предотвратить медленное расползание рака по коду, ведущее к его преждевременной смерти?

Чтобы ответить на этот вопрос, вернемся назад. Мы рассмотрим признаки неправильного развития кода, изучим, как мы выращиваем свой код, и определим некоторые стратегии разработки более здорового программного обеспечения.

Гниение программного обеспечения

Если ты молод, то растешь. Если ты созрел, то гниешь.

Рэй Крок

С хорошим кодом происходят плохие вещи. Каким бы удачным ни было начало, какими бы достойными ни были ваши намерения, каким бы ясным ни был ваш проект и какой бы четкой ни была реализация в первой версии, время будет разрушать и корежить ваш шедевр. Всегда помните про способность кода накапливать бородавки и шрамы на протяжении своей жизни.

Существует заблуждение, будто бы программное обеспечение развивается только на начальных стадиях своего существования. Самой длительной всегда оказывается стадия *сопровождения*[1] программного продукта. На нее уходит самая значительная часть всего труда – даже если этот труд не сконцентрирован так компактно, как на этапах начального проектирования и разработки. Б. У. Боэм (B.W. Boehm), уважаемый в области вычислительных наук ученый, подсчитал, что от 40 до 80% суммарного времени разработки приходится на сопровождение (Boehm 76).

Никогда не следует ожидать, что программный продукт застынет после выпуска. Сколь бы тщательно вы ни тестировали его, всегда обнаружатся нелепые ошибки, которые нужно исправлять. Клиенты требуют новых функций. Требования меняются прямо в ходе работы программистов. Допущения, которые были приняты во время разработки, в действительности оказываются ошибочными и требуют поправок.

[1] Имеется в виду работа, выполняемая после начальной поставки продукта, которая не считается основной новой версией.

В конечном итоге, больше кода оказывается написанным *после* того, как проект был признан завершенным.

На начальных стадиях разработки код полностью находится в ваших руках, и вы можете делать с ним что угодно в рамках отведенного вам времени. После выпуска продукта вы более ограничены в своих возможностях, и эти ограничения целесообразны, поскольку:

- Модификации должны быть минимальными, чтобы уменьшить их влияние на тщательно отлаженный основной код.
- API выпущенных и проданных продуктов уже в работе у клиентов, поэтому их труднее модифицировать.
- Пользователи привыкли к интерфейсу, и его нельзя менять беспричинно.

Ограничения могут также носить психологический характер и основываться на предубеждениях (в том числе ошибочных) разработчиков:

- Этот код всегда работал *так*, поэтому мы не можем его изменить.
- Пересматривать архитектуру на данной стадии слишком обременительно.
- Не стоит тратить время и деньги на правильную модификацию; продукту все равно осталось жить недолго.

Ограничение может даже заключаться в обычном непонимании – программист, осуществляющий сопровождение, может не понять, какую модель имел в виду автор оригинального кода, а потому неправильно ее модифицировать.

Есть тонкая граница между сопровождением имеющегося продукта и разработкой очередной его версии. Где она проходит – спорный вопрос. Но в любом случае модифицируется оригинальный базовый код – его автором или кем-то другим. Вот тут и начинается деградация кода. Модифицируется код или нет – *все равно плохо*; в любом случае код деградирует.

Если вообще не притрагиваться к коду, не делать своевременных исправлений и модификаций, программа деградирует. В худшем случае она вообще перестанет работать, если будет заменена ОС или сделанные допущения устареют. Ярким примером служит проблема 2000 года.[1] Либо программа морально устареет по мере того, как конкурирующие продукты станут предлагать больше функций и приобретут большую популярность. Нетронутый код медленно деградирует.

Если вводить расширения и исправлять ошибки, код все равно может деградировать. Исправляя одну ошибку, программист часто в качест-

[1] Многими старыми программами не предполагалось пользоваться после 2000 года, поэтому программисты не боялись указывать год двумя цифрами – 76 вместо 1976. Как только цифры перевалили за 00, все вычисления с датами пошли наперекосяк.

ве побочного эффекта вносит несколько новых ошибок. Брукс считает, что 40% исправлений приводят к новым ошибкам (Brooks 95). «Застольная песнь программиста» (напеваемая на мотив «99 Bottles of Beer on the Wall»), написанная неизвестным менестрелем, замечательно подытоживает этот результат:

> 99 маленьких багов в коде, 99 багов в коде,
> Исправь один баг, снова скомпилируй, 101 маленький баг в коде.
> (Повторять до BUGS == 0)

Даже не имеющие ошибок модификации могут быть источником неприятностей. Сделанные на скорую руку исправления громоздятся одно поверх другого, забивая все новые гвозди в гроб с первоначальным проектом и все больше затрудняя сопровождение. Здесь уместна аналогия с растениями: когда наверху вырастают тяжелые ветви, а ствол никак не укреплен, весь базовый код теряет устойчивость. В конечном счете он неизбежно переворачивается. Здоровые растения так не растут, и от такого кода нельзя ждать ничего хорошего.

Помните о том, как легко деградирует модифицируемый код. Избегайте модификаций, после которых система оказывается в худшем состоянии.

Может быть, это слишком пессимистично? Разве нельзя сохранить стабильность кода, если действовать осторожно? Вероятно так, но в современных условиях промышленного производства программ должные предосторожности не предпринимаются. Это вопрос культуры. Исправления должны осуществляться быстро и дешево. Программами часто пользуются дольше, чем предполагалось вначале. Многие заплатки на скорую руку сохраняются гораздо дольше, чем было запланировано для них.

Тревожные симптомы

Постоянно анализируйте код в поисках признаков деградации. Следите за появлением тревожных признаков: деградация начинается с любых изменений, приводящих к потере ясности или усложнению системы. Неоправданная сложность скрывается под разными масками.

Вот что должно восприниматься как мигающие красные огни и вой клаксонов:

- Код перегружен многочисленными крупными классами и запутанными функциями.
- Имена функций загадочны или обманчивы. У функций проявляются неожиданные побочные эффекты, не предусмотренные их именами.
- Отсутствует структура: непонятно, где искать нужную функцию.
- Есть дублирование: оказывается, что отдельные фрагменты кода занимаются одним и тем же.

- Высока степень связанности: сложные соединения и взаимозависимости модулей приводят к тому, что небольшие изменения в одном месте вызывают широкие отголоски во всем коде – иногда в модулях, которые, как казалось, независимы. (См. раздел «Модульность» на стр. 325).
- Проходящие через систему данные многократно преобразуются из одного представления в другое (например, отображаемые данные передаются между `std::string`, `char*`, Unicode, UTF-8 и обратно).
- API становятся расплывчатыми; некогда четкие интерфейсы стали слишком широкими по охвату в результате непродуманного добавления новых функций.
- API резко меняются от одной версии кода к другой.
- Элементы закрытой реализации просачиваются в открытые API ради изготовления быстрых заплаток.
- Код замусорен обходными путями: признак лечения симптомов, а не болезней. Под ними скрываются реальные проблемы. По краям системы громоздятся обходные пути, а проблемы притаились в центре.
- Есть функции с громадными списками параметров. Многие из них не используют эти параметры, а передают во вспомогательные функции.
- Попадается код, об усовершенствовании которого даже страшно подумать. Неизвестно, каков будет результат – улучшение работы, скрытая поломка или нечто худшее.
- Новые функции добавляются без соответствующей документации; имеющаяся документация устарела.
- При компиляции кода выводится множество предупредительных сообщений.
- Встречаются комментарии, гласящие *здесь ничего не трогать*...

Многие из этих видов пороков кода особенно заметны и могут быть определены при беглом осмотре кода или с помощью специальных инструментов. Однако есть класс более тонких, невидимых типов деградации, которые обычно проявляются на более высоком уровне, чем синтаксическая грязь. Модификации, которые отклоняются от первоначальной архитектуры кода или незаметно обходят принятые в программе соглашения, гораздо труднее обнаружить – для этого нужно глубоко погрузиться в систему.

Научитесь обнаруживать испорченный код. Изучите его признаки и обращайтесь с разложившимся кодом с крайней осторожностью.

Почему мы так беспокоимся о коде? Ответ простой: из-за *сложности*. Программа представляет собой огромный набор данных, организованных на разных уровнях: архитектуры, конструкции ее компонент, интерфейсов, реализации в коде и т. п. Во всем этом нужно разобраться,

чтобы начать работать над проектом. Когда установлены жесткие сроки окончания работы, не остается времени, чтобы разбираться в том, как работают те или иные несколько строчек кода, не говоря уже об определении их места в общей картине. В настоящее время мы не умеем справляться с этой огромной сложностью.

Как развивается код?

Никакие разработки не следуют до конца классической модели: зафиксировать окончательно все технические требования, полностью разработать проект, полностью написать код, выполнить интеграцию, протестировать и выпустить готовый продукт. В существующем базовом коде приходится делать неожиданные модификации. Каким-то образом прилепляются новые части. Это расширяющийся цикл разработки для достижения постоянно меняющихся целей.

Развитие кода происходит посредством одного из следующих механизмов, примерно упорядоченных по степени их отвратительности:

Удача

Это самый ужасный способ создания кода, притом слишком часто встречающийся. Код, который развивается благодаря удаче, не имеет никакого проекта. Он модифицирован без мыслей в голове. Его структура обусловлена случаем, и просто чудо, что он вообще работает.

Даже если изначально ваш код был тщательно спроектирован, модификации во время сопровождения могут следовать подходу «авось повезет». Такие сделанные наудачу поправки могут скрыть истинную проблему и затруднить ее реальное решение в дальнейшем.

Дополнение

Нам нужно добавить новую функцию. Если делать все по правилам, надо разобрать интерфейсы между несколькими основными модулями и пересмотреть большой объем кода. Времени на это нет, да если бы и было, задача могла оказаться слишком сложной. Мы просто приделаем снаружи еще один кусок кода. Прицепим к одному из имеющихся модулей – или сразу к нескольким – и будем общаться с ними через специальный тайный интерфейс. Что-то работающее мы получим очень быстро.

Это чудовищный клудж. И производительность будет ужасной. И у модулей больше не будет четких функций и обязанностей. Изящная конструкция перестанет существовать, и последующее сопровождение превратится в кошмар. Но зато мы быстро смастерим эту версию, а времени на то, чтобы делать все, как надо, у нас все равно нет.

Может быть, когда-нибудь мы вернемся и сделаем все по правилам...

Переписывание

Если вы понимаете, что код, с которым вы работаете, просто ужасен – непонятный, хрупкий и нерасширяемый, – его необходимо написать заново. Опыт показывает, что написать заново код иногда бывает быстрее и надежнее, чем ковыряться в той путанице, которая вам досталась. Однако переписывают код редко. Для этого необходимы храбрость и видение.

Риск возрастает, если приходится переписывать сразу много кода. Переписать заново весь проект – совсем не то, что заново написать код вместо плохой функции или класса. При хорошей модульности и разделении задач вам не нужно переписывать всю систему – только модуль, над которым вы работаете, сохранив его интерфейс. Если интерфейс плох или необходимость переписывания вызвана слабой модульностью системы, объем работы значительно возрастает.

Рефакторинг

Близкий родственник переписывания кода. Если код в основном в порядке, но отдельные части требуют переработки, можно выполнить *рефакторинг* этих неудовлетворительных фрагментов. Рефакторинг – это процедура внесения в тело кода небольших изменений с целью улучшения его внутренней структуры без изменения внеш-

Делимся пополам

Программный продукт достиг важного перепутья. У имеющегося базового кода фактически не было будущего – его действительно нужно было написать заново. Руководство в конце концов согласилось с этим, и был составлен план. Разработчики были поделены на две бригады. Одни продолжали колдовать над существующим кодом, пытаясь немного продлить его жизнь. Остальные должны были начать писать все приложение с чистого листа.

Одна из задач была привлекательной: разрабатывать совершенно новый проект с интересными задачами реализации и возможностью работать с новым, чистым базовым кодом. Другая задача была неблагодарной: заделывать течи в тонущем корабле, пока не будет готов новый роскошный лайнер (и тогда все прежние труды пойдут на свалку). В какой из двух команд хотелось бы оказаться лично вам?

Неудивительно, что в командах возникли чувства негодования, разочарования и соперничества. Многие программисты, которым было поручено работать со старым приложением, попросили перевести их на другой проект или ушли из компании в поисках лучшей жизни. Работа над старым базовым кодом считалась второсортной, потому что это был второсортный проект.

него поведения. Конструкция кода в результате усовершенствуется, что облегчает работу с кодом в будущем. Задача этой процедуры – не повысить эффективность кода, а усовершенствовать его конструкцию. В отличие от такой радикальной меры, как переписывание кода заново, рефакторинг представляет собой лишь ряд мягких массажных процедур для имеющегося кода.

Рефакторинг – причудливое название для ряда конкретных типов модификации кода. Мартин Фаулер формализовал их, описав несколько небольших и понятных усовершенствований кода (Fowler 99).[1]

Теория хаоса

Очевидно, что код определяется своим проектом, но большую роль играет также то, в какой организации он создавался и каков его жизненный путь. Довольно давно я принимал участие в проекте, где код пользовательского интерфейса вызывал особое отвращение. Он работал (как правило), но был совершенно непостижим – плотный комок запутанной логики, в котором невозможно было различить какой-нибудь архитектуры, и порядок выполнения представлял собой лабиринт. Такое состояние кода имело свое объяснение, если учесть историю его создания.

Код был изначально написан на скорую руку как простой интерфейс пользователя для единственного клиента, с минимумом спецификаций. Он успешно справлялся со своей задачей. К сожалению, на этом дело не кончилось.

Продукт был продан второму покупателю, который пожелал, чтобы внешний вид был изменен. Приладили новый скин (внешний вид). После этого продукт был продан новому покупателю, в другой стране. Приделали локализацию – с новым скином. Затем продукт был продан еще одному клиенту, который пожелал иметь в интерфейсе дополнительные функции – приклепали и их тоже. Так продолжалось еще долгое время. Последний интерфейс не имеет никакого сходства с прежним, а сопровождать его стало невозможно: все дополнения были заплатками на скорую руку, поскольку вся система всегда считалась временной.

Если бы в первоначальный проект были включены все нынешние функции, код сохранил бы аккуратность и логичность. Однако для этого потребовался бы слишком большой объем первоначальных работ, и компания никогда не взялась бы за этот проект. Жаль бедных программистов, которым приходится работать при таких обстоятельствах.

[1] Мартин Фаулер «Рефакторинг: улучшение существующего кода». – Пер. с англ. – СПб: Символ-Плюс, 2002.

Проект «на вырост»

Часто бывает понятно, каким образом код станет расширяться в будущем; например, некоторые функции просто отложены в расчете включить их в следующую версию. Можно специально спроектировать систему так, чтобы в будущем было проще вносить в нее дополнения. Как правило, это не слишком осложняет проектирование.

Даже если вы не знаете, какие функции будут добавляться в проект, тщательное проектирование оставляет пространство для будущего роста. Расширяемая система предоставляет специальные точки, в которых можно подключать новые функции. Однако имейте в виду, что это не погоня за ветром,[1] когда вы пытаетесь предсказать будущее, не имея понятия о том, как будет развиваться система. Расширяемость достигается за счет роста сложности системы. Если вы правильно угадаете, где эта сложность понадобится, ваша взяла; если догадка окажется неправильной, вы сделаете неоправданно сложную систему. Так можно *переусердствовать в проектировании*, что особенно часто случается, когда проектированием занимается комитет.

Существует теоретическое направление, в которое входит *экстремальное программирование*, требующее разрабатывать самый простой проект, который должен быть применим в любой ситуации. Это может противоречить духу «проектирования на вырост» (в зависимости от того, насколько податлив окажется первоначальный простой проект). Определение того, в какой мере ваш проект должен быть ориентирован на рост, является сложной, но важной задачей.

Вера в невозможное

Возможно, мы потому так часто сталкиваемся с плохим кодом и многочисленными небрежными заплатками, что существует ошибочное мнение, будто правильный подход к работе отнимает больше времени. Это мнение будет опровергнуто, если учесть сокращение времени отладки и простоту последующих модификаций. Можно быстро среагировать на сообщение об ошибке, внеся исправление на скорую руку, но это не лучшее решение. Настоящий мастер ответственно относится к модификации кода.

Менеджеры в компаниях часто рассчитывают на то, что ошибка будет быстро исправлена. Не так сложно убедить менеджера в том, что тяжелая бетонная плита недолго продержится на верхушке хрупкого шеста. Труднее заставить его встать под ней внизу. И гораздо труднее донести ту же мысль, если дело касается программного обеспечения. До менеджеров она просто не доходит. По их представлениям программисты – это волшебники, занимающиеся черной магией и обладающие

[1] Экклезиаст 2:11

безграничными возможностями. Достаточно объяснить им, что нужно сделать, назначить срок, и все будет в порядке – пусть они хоть ночи напролет программируют.

С нашими способностями и преданностью делу мы иногда оправдываем их ожидания. На самом деле мы делаем только хуже, потому что у руководства возникает уверенность, что такая тактика всегда срабатывает. Более того, когда она не срабатывает, вина возлагается на нас. Увы, настает момент, когда поспешно подправленную программу уже невозможно расширять далее, и она вот-вот рухнет, обретя окончательный покой. Руководству это вряд ли понравится.[1]

Развивать код легче, если в компании принято разрабатывать программное обеспечение путем небольших приращений (см. разделы «Итеративность» на стр. 322 и «Итеративная и инкрементная разработка» на стр. 545). На этом пути эволюционирование оказывается частью стратегии проектирования и переработка кода для интеграции изменений ожидается с самого начала. Альтернатива, когда монолитное здание кода требуется атаковать маленькой мотыгой и результат ожидается через 20 секунд, безрассудна, но не столь уж редка.

Как с этим бороться?

Боже, дай мне благоразумие спокойно принять то,
что я не могу изменить, мужество, чтобы изменить то, что могу,
и мудрость, чтобы отличить одно от другого.

Рейнхольд Нибур

Теперь, определив некоторые проблемы, связанные с развитием базового кода, мы встаем перед вопросом, как с ними справляться. Какую стратегию выбрать, чтобы избежать путаницы?

Первое и главное – признать существование проблемы. Очень многие программисты латают код, не задумываясь о качестве результата. Если им удается в кратчайшие сроки успокоить возмущенных пользователей, их не волнует, в каком состоянии после них остается код. В следующий раз им будет заниматься кто-то другой.

Добросовестно относитесь к написанию кода. Хорошие программисты думают о том, как будет выглядеть их код через несколько лет, а не о том, сколько усилий им понадобится приложить сегодня.

[1] Разумеется, это грубое обобщение, но оно не столь далеко от истины. Многие менеджеры когда-то были программистами и понимают, в чем конфликт. Хороший менеджер выслушает возражения программиста. Хороший программист заставит своего босса выслушать себя. Но слишком часто не происходит ни того, ни другого, а страдает программное обеспечение.

Как писать новый код

Прежде чем заняться *существующим* кодом, опишем некоторые подходы к тому, как писать новый код, чтобы облегчить его сопровождение в будущем:

- Обратите внимание на взаимосвязь модулей и сократите ее, насколько это возможно. Старайтесь, чтобы не было центрального модуля, от которого зависят все остальные; его модификация коснется всех прочих модулей в системе.
- Модульность и сокрытие информации (см. раздел «Модульность» на стр. 325) – это краеугольные камни современного программирования. Ограничьте все предполагаемые изменения какой-то небольшой частью системы, что сделает ее более устойчивой к модификациям.
- Расширяемость и способность к деформациям должны быть заложены в проекте, но, как говорилось, не за счет усложнения системы. Современные парадигмы на базе компонент/объектов предоставляют больше возможностей для повторного использования и расширяемости. Они обеспечивают взаимодействие между модулями через четкие интерфейсы. Однако если эти интерфейсы не обеспечивают возможностей расширения, код не сможет развиваться. Тщательно прорабатывайте интерфейсы своей системы.
- Пишите код аккуратно и ясно, чтобы в нем можно было легко разобраться, снабжайте его хорошей документацией и четко определенными API с понятными именами. Для документирования интерфейсов можно пользоваться инструментами грамотного программирования.
- *KISS* (*Keep It Simple, Stupid* – будь проще!). Не злоупотребляйте сложностью и техничностью. Оптимизируйте алгоритм, только если вы *знаете*, что он не обеспечивает нужной производительности, а не потому, что вы увидели способ сделать код более быстрым. Простота, как правило, более предпочтительна, чем эффективность, и несомненно, что она облегчает последующее сопровождение.

Пишите новый код с учетом возможной потребности модифицировать его в будущем. Делайте его понятным, расширяемым и простым.

Сопровождение существующего кода

Для сопровождения хорошего кода требуется иной план действий, чем для сопровождения плохого. В первом случае необходимо тщательно заботиться о сохранении целостности проекта и не отклоняться от его стиля при внесении изменений. Во втором случае постарайтесь не вносить в код еще большего беспорядка и, если удастся, попутно улучшайте код. Если вы не можете переписать раздражающий вас код заново, немного рефакторинга окажет существенную пользу.

Прежде чем менять любой код, нужно рассмотреть ряд организационных задач:

- Назначьте *приоритет* среди необходимых изменений. Взвесьте важность каждой задачи и ее сложность, после чего решите, чем заняться в первую очередь. Не окажут ли первые модификации влияния на последующий ход работы?
- Изменяйте только то, что необходимо. *Если что-то работает, не трогай это*. Не нужно «улучшать» фрагменты кода только потому, что вам этого захотелось – изменяйте код только там, где это действительно необходимо. Сделайте рефакторинг плохого кода, если вам нужно с ним работать. Все остальное обходите стороной.
- Следите за количеством одновременно вносимых изменений. *Самостоятельное* проведение нескольких параллельных модификаций – либо большое искусство, либо глупость; скорее, последнее. Работайте постепенно и осторожно.

  Если над кодом работает одновременно несколько человек, следите за окружающей обстановкой. Когда одновременно проводится несколько модификаций, между ними могут возникать странные конфликты. При методичной модификации, выполняемой одним разработчиком, лучше всего видно, где меняется код и где требуется наибольшая осторожность. Несколько одновременно проводимых модификаций могут ослабить код непонятным и незаметным ни для кого образом.

Соблюдайте осторожность при модификации. Будьте в курсе того, не занимается ли кто-то модификацией близлежащего кода.

- В рецензировании нуждаются не только новый код во время первоначальной разработки, но и последующие модификации. Организуйте официальное обсуждение и постарайтесь, чтобы в нем участвовали автор оригинального кода и рецензенты. Небольшими расширениями кода очень легко внести скрытые новые ошибки; при обсуждении многие из них бывают обнаружены.

Проводите обсуждение опасных изменений, особенно при подготовке к выпуску новой версии. Даже простейшие модификации могут нарушить работу другого кода.

Получив доступ к исходному коду, как следует с ним обращаться? Вот несколько практических рекомендаций:

- Чтобы правильно модифицировать код, соберите сведения о нем. Прежде чем переделывать файл или модуль, выясните:
  - Какое место он занимает в системе в целом.
  - Какие у него есть связи (т. е. на какие компоненты могут оказать влияние ваши изменения).

- Какие допущения были сделаны при создании документа (хорошо, чтобы они оказались отражены в спецификации кода).
- История уже сделанных модификаций.

Оцените качество кода. Сделать это очень просто, и вы сразу получите представление о том, насколько трудно вам будет с ним работать. Возможно, вам будут полезны средства для визуализации кода и получения метрик качества; они выявят места, где могут таиться скрытые дефекты. Сопоставьте всю относящуюся к делу документацию.

- Займите правильную позицию – не следует поддаваться соблазну сделать *еще одну скоропалительную заплатку*. Не пренебрегайте кодом в расчете, что в скором времени он будет выброшен или переписан. Ошибаетесь.

  Все время помните о тревожных симптомах, перечисленных на стр. 367. Если в результате вашей модификации код приблизится к одному из описанных там состояний, займитесь его рефакторингом и устраните проблему. Ответственность за возникновение таких проблем лежит на вас.

  Возможно, потребуется внести какие-то изменения в проект. Не бойтесь распустить швы и произвести некоторые хирургические операции, если это необходимо. Иногда модификация может потребовать от вас больших расходов (в смысле времени и труда), но ваши инвестиции в дальнейшем окупятся: в будущем работать с кодом станет значительно проще. Может показаться, что для старого кода это экономически невыгодно. К сожалению, именно старый код часто служит источником доходов и может еще долгое время не выходить в тираж. Если вы видите, что над каким-то разделом кода вам не раз придется поработать в будущем, постарайтесь сделать так, чтобы его структура поддерживала возможные расширения.

Не курочьте код бездумно. Остановитесь и поразмыслите, чем вы занимаетесь.

- Постарайтесь, чтобы добавленный код не создавал новых взаимозависимостей. Лишние связи делают код сложнее и затрудняют последующие модификации.
- При сопровождении любого кода соблюдайте стиль программирования, свойственный исходным файлам, с которыми вы работаете, даже если он вам не нравится или в вашей конторе принят другой стиль. С файлом кода, разные части которого выдержаны в различных форматах, трудно разбираться и работать. Можно иногда облагородить представление, если это в какой-то мере оправдано, но помните, что в результате будет затруднено автоматическое нахождение различий между версиями кода. Обновляйте комментарии в коде, с которым работаете (см. раздел «Сопровождение и бессодержательные комментарии» на стр. 128).

- С помощью комплекта тестов проверьте, что вы не нарушили работу кода. Полное регрессивное тестирование – единственный способ быть спокойным за сделанные вами модификации.

  Обеспечьте наличие адекватного набора тестов и регулярно запускайте его.

Тщательно тестируйте все сделанные вами модификации, какими простыми они бы ни были. Очень легко пропустить глупую ошибку.

- Если вы исправляете ошибку, убедитесь, что вам понятна ее причина. Напишите контрольный пример, в котором она проявляется. Он покажет, что вы понимаете причину, и проверит, что вы действительно исправили ошибку. Добавьте его в набор регрессивных тестов.

  Если ошибка успешно исправлена, поищите, нет ли во всем коде аналогичных дефектов. Этот часто пропускаемый шаг может оказаться очень важным: многие проблемы носят групповой характер, и гораздо проще расправиться с ними одним ударом, чем потихоньку урезать по мере проявления каждой из них.
- Если исправление оказалось неудачным, скорее устраните его следы. Не захламляйте код.

Настоящий мастер должен всегда избегать соблазна сделать скороспелую заплатку. Стремитесь к тому, чтобы выполнять модификации взвешенно и обдуманно. К сожалению, мы живем не в башне из слоновой кости и иногда компромиссы неизбежны; решение задачи идеальным способом не всегда коммерчески оправдано.

Этим объясняется такое большое количество хрупкого, рыхлого и рискованного кода. Но этим объясняется и то, что программы вообще существуют. Если бы не коммерческий интерес продать программный продукт, программисты только тем бы и занимались, что бесконечно переписывали свой код, добиваясь совершенства. Фирмы, в которых они работают, давно бы обанкротились.

Тем не менее, осуществляя прагматичные (но тошнотворные) модификации, планируйте исправить их в будущем. Поставьте себе в график работ задачу привести все в порядок.

Резюме

Перемены всегда приятны.

Аристотель

Я бы не согласился с Аристотелем. Перемены могут оказаться сущим мучением. Изменения в коде нужно проводить с большой осторожностью. Тогда программа будет развиваться в лучшую сторону, а не деградировать в неустойчивое бесформенное образование.

Необходимо правильно сопровождать и расширять приложение, сохраняя организацию кода и осуществляя не вызывающие отторжения модификации. Сопровождение – нелегкая задача. Переписывание, перепроектирование и рефакторинг могут потребовать больших затрат времени.

См. также

Глава 17. Вместе мы – сила

Разработка и сопровождение программного обеспечения – коллективный труд. Динамика, существующая в команде, неизбежно оказывает влияние на конечный вид вашего продукта.

Глава 18. Защита исходного кода

История разработки вашего кода отражена в *системе контроля версий*.

Глава 22. Рецепт программы

Жизненный цикл разработки программного продукта: описание процедуры создания и развития программ.

Хорошие программисты...

- Пишут программы, пригодные для сопровождения, – с ясной структурой и логичным планом
- Замечают, когда код плох, и готовы к работе с ним
- Прежде чем переделывать код, стараются разобраться в нем и в замысле автора
- Заботятся о качестве своего кода, отказываются делать грубые заплатки

Плохие программисты...

- Пишут сложный код, не думая о тех, кто будет его сопровождать
- *Избегают* сопровождения старого кода, предпочитая игнорировать проблемы, а не решать их
- Предпочитают латать наскоро, а не обдумывать правильное решение
- Засоряют код заплатками на скорую руку, предпочитают самые короткие решения
- Сосредотачивают свое внимание не там, где нужно, исправляя код, который, по сути, этого не требует

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 652.

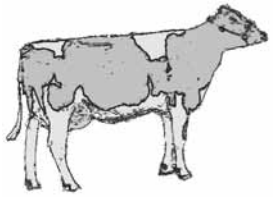
Вопросы для размышления

1. Какая метафора лучше всего отражает развитие программного продукта?

2. Если принять для разработки программного продукта метафору человеческой жизни, о чем говорилось во вступлении, каким реальным событиям соответствуют следующие этапы:
   - зачатие
   - рождение
   - рост
   - созревание
   - выход в мир
   - средний возраст
   - усталость
   - выход на пенсию
   - смерть
3. Есть ли предел продолжительности жизни программного продукта – как долго можно разрабатывать и совершенствовать программу, прежде чем начать работу заново?
4. Соответствует ли объем кода степени зрелости программы?
5. Важно ли обеспечить *обратную совместимость* при сопровождении кода?
6. В каком случае деградация кода происходит быстрее – при его модификации или сохранении в неизменном состоянии?

Вопросы личного характера

1. Чем вам больше приходится заниматься – написанием нового кода или модификацией имеющегося?
   a. Если это новый код, то пишется ли он для вновь создаваемых систем или для расширений существующих?
   b. Зависит ли от этого, *как* вы пишете код? Каким образом?
2. Есть ли у вас опыт работы с уже готовым базовым кодом? Если да:
   a. Повлияло ли это на ваш нынешний уровень мастерства? Чему вы в результате научились?
   b. Каким был этот код по большей части – плохим или хорошим? Каким образом вы это оценивали?
3. Случалось ли вам вносить изменения, ухудшавшие качество кода? Почему?
4. Через сколько ревизий прошел код вашего текущего проекта?
   a. В какой мере менялась функциональность от одной версии к другой? Как менялся код?
   b. Осуществлялось ли развитие *наудачу*, *по плану* или каким-то промежуточным образом? В чем это проявляется сейчас?
5. Какие меры принимаются в вашей команде, чтобы модификацией кода не занималось одновременно несколько программистов?



IV

Стадо программистов?

Скучные ряды перегородок. Начинающая компания. Тоска нереальных графиков, плохого менеджмента и ужасных программ. Искусственное освещение и дрянной кофе.

Добро пожаловать на фабрику программ.

Некоторые программисты фрилансерствуют, легко переходя из одного офиса в другой. Другие пишут дома код open source для удовольствия. Но большинство заключено в мало вдохновляющую обстановку фабрик программ, отбывая там свой срок за дело, которое по-прежнему очень любят.

Странные мы люди: антиобщественные по природе, предпочитающие компанию компилятора и веб-броузера. Однако чтобы создавать программные шедевры, мы вынуждены работать коллективно, вопреки своим природным инстинктам. Как будет показано, качество ваших программ определяется качеством ваших программистов и сотрудничества между ними. Без надежной тактики поведения в реальном мире вы погибли.

В этой части изучается влияние культуры и динамики на ваш код.

Глава 16. Кодеры

Важные умения и личные качества сильных программистов.

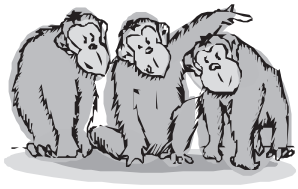
Глава 17. Вместе мы — сила

Как наладить эффективную и продуктивную работу в команде.

Глава 18. Защита исходного кода

Управление исходным кодом, доступ к которому имеет много программистов: как избежать катастрофы и сердечного приступа.

Так каким общим именем можно назвать группу программистов? Это явно не *рой*: далеко не та подвижность и редко такая организованность. Это не *прайд*: ни такой свирепости как у львов, ни шансов создать что-то достойное гордости. Ответ (по крайней мере, для C-программистов) ясен: это *стадо* (*brace*) программистов.



16

Кодеры[1]

Воспитание правильного отношения и подхода к программированию

В этой главе:

- Различные типы программистов
- Определение вашего естественного способа программирования
- Качества, присущие эффективным программистам
- Как успешно работать в коллективе

Мы – всего лишь несколько продвинутая порода обезьян на мелкой планете довольно заурядной звезды. Но мы можем понять, как устроена Вселенная. Это делает нас весьма уникальными.

Стивен Хокин

Популярная загадка: сколько нужно программистов, чтобы заменить электрическую лампочку? Варианты ответа:

1. Ни одного. Лампочка цела, просто сработала функция экономии электроэнергии.

---

[1] В оригинале «Code Monkeys» – международное сленговое выражение. Его значение зависит от контекста, но, как правило, это пренебрежительное название «нетворческого» программиста либо ироничная самоуничижительная самоидентификация вполне «творческого». Автором используется в разных смыслах. – *Примеч. перев.*

2. Один, но ему потребуется целая ночь и огромное количество пиццы и кофе.
3. Двадцать. Один будет решать задачу, а остальные девятнадцать отыскивать ошибки в том, что он напортачит.[1]

Каким должен быть правильный ответ? Он может быть любым из перечисленных в зависимости от того, кто берется за работу. Все программисты работают по-разному, и у каждого будет свой подход к решению одной и той же задачи. Всегда есть *несколько способов сделать это*[2], и различие в складе ума разных программистов приводит их к весьма различным решениям.

На протяжении всей этой книги мы отмечаем, какие установки характерны для хорошего программиста. В этой главе мы специально займемся этим вопросом: исследуем хорошие и плохие качества программистов и определим, какие из них наиболее важны для того, чтобы успешно программировать. Речь идет не только о подходе к написанию кода, но и о взаимоотношениях с другими программистами. Выводы относительно того, кто оказывается лучшими программистами, будут довольно неожиданными.

Мартышкин труд

Обитатели фабрики программного обеспечения – странное собрание чудаков и слабо пригодных к общественной жизни людей, *программистов* (*code monkeys*). Все серьезные программные системы создаются горсткой таких людей, у каждого из которых свой уровень мастерства и склад ума, но все они работают над достижением общей цели.

То, как мы работаем вместе и какой код пишем, неизбежно зависит от нашего отношения к работе, равно как от уровня технической компетентности. Если бы все из нас были прилежными, практичными и трудолюбивыми гениями, то наши программы стали бы гораздо лучше – их разработка укладывалась бы в сроки и бюджет и в них не было бы ошибок. Но мы, к сожалению, несовершенны, и это показывает наш код.

Чтобы выработать стратегию, действенную в таких условиях, я покажу вам галерею стереотипов программистов. Они созданы на основании наблюдений за людьми, которых я встречал на фабриках программного обеспечения. Разумеется, это довольно произвольный перечень, и вы вполне можете вспомнить программистов, которые не укладываются ни в одну из перечисленных категорий или относятся сразу к нескольким из них.

---

[1] Думаете, это шутка? У меня есть знакомая, которая меняла лампочки дважды в своей жизни. В первый раз весь ковер был усыпан битым стеклом. Во втором пришлось вызывать электрика, который заменил весь патрон.

[2] Мантра программистов, пишущих на Perl.

Несмотря на это, такая наглая классификация высвечивает важные факты и подсказывает нам пути для самосовершенствования. Мы рассмотрим следующие вопросы:

- Какие мотивы движут различными типами программистов
- Как работать с каждым из этих типов
- Как любой программист может усовершенствовать себя
- Чему можно у каждого поучиться

Читая описание каждого типа, задайтесь вопросами:

- Относитесь ли вы к данному типу программистов? Насколько близко описание соответствует вашему стилю программирования? Какие выводы вы можете сделать, чтобы улучшить свой способ кодирования?
- Многие ли из ваших знакомых относятся к данному типу? Приходится ли вам тесно работать вместе с ними? Нет ли способов улучшить ваши отношения?

Нетерпеливый

Начнем с этого парня, поскольку в нем[1], вероятно, воплощаются черты большинства программистов, которые прочтут эту книгу. Нетерпеливый быстр и летуч; он мыслит кодом. Импульсивный, прирожденный программист, он начинает писать код сразу, как только идея оформилась у него в голове. Он не остановится, чтобы подумать. Поэтому, несмотря на очень высокое техническое мастерство, его подлинный потенциал никогда не раскрывается в коде, который он пишет.

Нетерпеливый часто пытается применять новые функции или идиомы, потому что это модно. Его стремление попробовать новые трюки приводит к тому, что он применяет эти технологии даже там, где они непригодны.

Достоинства

«Нетерпеливые» продуктивны – в смысле количества кода. Они пишут уйму кода. Они любят изучать новинки и с энтузиазмом – даже со страстью – относятся к программированию. Нетерпеливый любит свою работу и искренне стремится писать хороший код.

Недостатки

В силу своего неумеренного энтузиазма Нетерпеливый не может ждать и бросается к редактору кода, вместо того чтобы сначала подумать. Он пишет много кода, но пишет так быстро, что в нем много ошибок, поэтому он тратит *уйму времени* на отладку. Немного

[1] Я буду говорить обо всех кодировщиках как о лицах мужского пола только для того, чтобы было легче читать текст.

предварительной умственной работы избавило бы его от многочисленных глупых ошибок и многих часов работы по их исправлению.

К несчастью, отладчик из него скверный. Так же бесшабашно, как при написании кода, он бросается в отладку. Методичность ему несвойственна, поэтому он теряет массу времени, занимаясь поиском ошибок в тупиках.

Он плохо умеет планировать время. Он еще может как-то оценить время, если все идет гладко, но в жизни ничего не происходит по плану, поэтому он никогда не укладывается в срок.

Как быть, если вы один из них

Не теряйте энтузиазма; это одно из лучших качеств программиста. Если радость вашей жизни в том, чтобы видеть, как программа работает, и восхищаться красотой кода, ищите практические пути для достижения этого. Хорошая идея – разрабатывать тесты модулей одновременно с кодом. Но обычно все сводится к простому совету: *остановись и подумай*. Не спеши. Выработай личные правила, которые тебе помогут, вплоть до того, что напиши *ДУМАЙ* на бумажке и приклей ее себе на монитор!

Как работать с такими людьми

Когда они работают хорошо, это один из лучших типов программистов для сотрудничества. Фокус в том, чтобы направить их энергию на создание полезного кода вместо бестолковой суеты. С ними прекрасно работать в паре.

Спрашивайте у Нетерпеливого, что он собирается делать сегодня и какие у него планы на будущее. Проявляйте заинтересованность в его проектах; это подстегнет его к серьезным размышлениям о них! Если вы зависите от результатов его работы, попросите его заранее предоставлять предварительные версии, а также разработанные им тесты.

Нетерпеливые выигрывают при надлежащем руководстве ими, способствующем дисциплинированности. Следите, чтобы его время было тщательно расписано в графике (писать план вместо него не нужно).

Кодер (Code Monkey)

Если вам понадобится набрать очень много программистов, скорее всего, вы выберете их из этой категории. (Хотя я бы не рекомендовал это делать; вы будете выбирать их *о-очень* долго!)

Кодер пишет крепкий, но лишенный вдохновения код. Получив задание, он честно трудится над ним и готов получить следующее. Поскольку они выполняют черную работу, их часто – возможно, несправедливо – называют «*пехотой*» (*grunt programmers*).

Кодеры непритязательны. Они боятся требовать для себя интересную работу, и потому им достаются малопривлекательные проекты. Они находят для себя нишу в качестве программистов сопровождения, поддерживая устаревший код, в то время как пионеры пишут ему замечательную замену.

Молодой Кодер может учиться и развиваться – при наличии времени и наставника, – но ему дают задания, не связанные с риском. Если Кодер пожилой, он, вероятно, закоснел и выйдет на пенсию все тем же кодером. Испытывать сожалений он при этом не будет.

Достоинства

Дайте ему задание, и он его выполнит – с приемлемым качеством и в приемлемые сроки. Кодер надежен, и обычно на него можно рассчитывать, если потребуются дополнительные усилия в кризисной ситуации.

В отличие от нетерпеливых, Кодеры хорошо рассчитывают время. Они методичны и дотошны.

Недостатки

Кодеры старательны и методичны, но они не *проявляют воображения*. У них нет изобретательности и интуиции. Кодер будет слепо следовать принятому проекту, не думая о возможных проблемах. Поскольку они не отвечают за проект, то не берут на себя ответственность за возникающие проблемы и обычно не проявляют инициативы в их изучении и решении.

Научить кодера чему-то новому трудно; ему это неинтересно.

Как быть, если вы один из них

Есть ли у вас желание осваивать новое и брать на себя дополнительные обязанности? Если да, то крепите свое мастерство, работая над персональными проектами. Берите книги и изучайте новые технологии.

Добивайтесь дополнительных заданий и постарайтесь включиться в проектные работы. Проявите инициативу в своей текущей работе – постарайтесь предвидеть критические точки и выработайте план, как их избежать.

Как работать с такими людьми

Не смотрите на Кодера свысока, даже если у вас выше техническое мастерство и вы выполняете более ответственную работу. Ободрите его – похвалите код и покажите полезные для работы приемы. Пишите продуманный код, чтобы максимально облегчить работу программиста сопровождения (т. е. сопровождающего кодера).

Гуру

Это легендарный таинственный гений: маг программирования. Гуру часто бывает спокойным и скромным, даже немного странным.[1] Он пишет прекрасный код, но плохо умеет общаться с простыми смертными.

Гуру поручают работу над фундаментальными вещами, такими как общие структуры, архитектура, ядро и т. п. Он пользуется заслуженным уважением (а иногда вызывает ужас) у своих коллег.

Всеведущий Гуру все знает и все видит. Он включается в любые технические дискуссии и высказывает свое мнение эксперта.

Достоинства

Гуру – опытные чародеи. Они знакомы со всеми современными технологиями и знают, какие из старых приемов лучше. (Все новейшие технологии Гуру сами и изобрели.) У них масса опыта, и они пишут зрелый код, который легко сопровождать.

Хороший Гуру – замечательный наставник; у него можно многому научиться.

Недостатки

Гуру редко умеют общаться. Они не всегда косноязычны, но думают настолько быстро и на таком уровне, что простым смертным за ними не угнаться. В разговоре с Гуру чувствуешь себя глупым, смущенным или то и другое вместе.

Чем слабее навыки коммуникации у Гуру, тем хуже из него наставник. Гуру не могут понять, почему другие чего-то не знают или думают не с такой скоростью, как они.

Как быть, если вы один из них

Попытайтесь спуститься с небес на землю и жить в реальном мире. Не рассчитывайте, что все будут столь же сообразительными и думать так же, как вы. Требуется немалое мастерство, чтобы давать простые и понятные объяснения. Нужно тренироваться.

Как работать с такими людьми

Если вы встретили Гуру, учитесь у него. Впитывайте все – не только технические сведения. Чтобы зарекомендовать себя как Гуру, необходим определенный темперамент и склад личности – знания, а не высокомерие. Учтите это.

---

[1] Во всяком случае более странным, чем «нормальные» программисты. Пожалуй, подойдет слово *эксцентричный*.

Псевдогуру

Псевдогуру *считает* себя гением. Он ошибается. Он говорит с видом знатока, но несет чушь.

Это, пожалуй, самый опасный тип программиста: Псевдогуру трудно распознать, пока не обнаружится нанесенный им вред. Менеджеры считают, что он – гений, потому что он говорит правдоподобно и самоуверенно.

Обычно Псевдогуру ведут себя достаточно нескромно по сравнению с гуру. В них больше хвастовства и самодовольства. Он сам назначает себя авторитетом. (Напротив, гуру признаются в качестве экспертов своими коллегами.)

Достоинства

Можно предположить, что у Псевдогуру достоинств нет, но его сильная сторона – это вера в себя. Очень важно верить в свои силы и быть уверенным в том, что ты пишешь код высокого качества. И все же...

Недостатки

Главный недостаток Псевдогуру – его уверенность в себе. Он переоценивает свои способности, и его решения ставят ваш проект под угрозу. Он – большая обуза.

Вы будете вспоминать его и после того, как он покинет вас. У вас на руках останутся его плохие проекты и заумный код.

Как быть, если вы один из них

Не медля, честно оцените свои способности. Не расхваливайте себя сверх меры. Честолюбие – вещь хорошая, в отличие от претензий выглядеть тем, кем вы не являетесь. Возможно, вы делаете это неумышленно, поэтому постарайтесь быть объективным в отношении своих возможностей. Пусть вас больше заботит качество ваших программ, а не стремление выглядеть важным или умным.

Как работать с такими людьми

Будьте крайне осторожны.

Распознав Псевдогуру, вы уже сделали полдела. Больше всего урона он может нанести, пока вы его не вычислили. Бдительно следите за Псевдогуру: нужно фильтровать мусор из того, что он произносит, бороться с его дефектными проектами и проверять его негодный код.

Высокомерный гений

Это тонкая, но важная разновидность Гуру. Потрясающий программист, он работает быстро, эффективно и пишет код высокого качества. Не будучи вполне Гуру, он все же выделяется на общем фоне. *Но* поскольку он хорошо знает о своем умении, то ведет себя заносчиво, снисходительно и унижающе.

Этот Гений неизлечимо болен стремлением спорить, потому что обычно он прав и всегда должен добиться, чтобы его правильная точка зрения победила чужие неправильные мнения. Он привык к этому. Самое неприятное то, что обычно он оказывается прав, поэтому в споре с ним вы все равно проиграете. Если же правы вы, он будет говорить до тех пор, пока спор не перейдет на предмет, относительно которого будет прав *он*.

Достоинства

Гений обладает высоким техническим мастерством. Он может осуществлять сильное техническое руководство и катализировать команду, если все с ним согласятся.

Переходя на личности

Приводимая классификация программистов не слишком научна. Психологи разработали более формальные классификации личности – официальные способы объявить вас уродом. Они не ограничиваются сферой разработки программного обеспечения, но дают ценное представление о поведении программистов.

Наиболее популярен, видимо, *опросник Майерс-Бриггс* (Briggs 80). Он раскладывает личность по четырем осям: экстраверт (E) или интроверт (I); доверяющий только фактам (S) или интуитивный (N); думающий (T) или импульсивный (F); ориентированный на результат (J) или процесс (P). Результат этой классификации оказывается четырехбуквенным индикатором; для программистов наиболее характерен индикатор ISTJ.

Роли в команде по Белбину – классификация *характеров по принципу склонности вести себя, участвовать в работе и относиться к коллегам определенным образом* (Belbin 81). Ее назначение – оценить врожденное общественное поведение человека и его способность устанавливать взаимоотношения, определить, как они способствуют или затрудняют успешную работу команды. Классификация показывает, как тип личности влияет на умение работать в команде. Белбин выделяет в поведении девять конкретных ролей: три типа людей действий, три типа социально-направленных и три типа интеллектуальных людей. С учетом этого можно подбирать эффективные команды из людей с взаимодополняющими свойствами; если бы все программисты оказались координаторами, никакого конечного продукта вообще не получилось бы.

Эти классификации личностей не имеют взаимнооднозначного соответствия с теми, которые предложены мной. Кроме того, в них явно отсутствуют приматы.

Недостатки

Гений не любит, когда доказывают его неправоту, и считает, что он *всегда* должен быть прав. Он не может не выступать как авторитет; Гений знает все обо всем. Он никогда не признается, что *не знает чего-то*, ибо это будет для него полным унижением.

Как быть, если вы один из них

Не все достигают божественного статуса, но есть множество хороших программистов, достойных уважения. Это нужно сознавать. Проявляйте скромность и уважайте мнение других людей.

Присматривайтесь к людям, у которых может быть более обоснованная точка зрения, и учитесь у них. Не прикидывайтесь и не пытайтесь скрыть свое незнание – честно признайте, что вы знаете, а что нет.

Как работать с такими людьми

Оказывайте уважение Гению, как и другим программистам, с которыми он работает. Избегайте неконструктивных ссор с ним. Но отстаивайте свою позицию – доказывайте правоту своих мнений и взглядов. Пусть вас не пугает общение с ним. Обсуждая технические вопросы с Гением, вы совершенствуетесь как программист; просто научитесь не поддаваться эмоциям. Если вы уверены в своей правоте, найдите себе союзников, которые помогут вам в споре.

Будьте осторожны и сами не станьте высокомерным или любителем поспорить.

Ковбой

Ковбой – плохой программист, активно избегающий тяжелого труда. Он не упустит короткого пути, если обнаружит его. Некоторые ошибочно относят ковбоев к *хакерам*. Но он не хакер в классическом понимании этого слова. *Хакер* – это термин, которым гики с гордостью описывают рискового кодировщика.[1]

Ковбой бросается сразу писать код, потратив минимум усилий на поиск решения проблемы. Ему безразлично, если это не лучшее решение, если оно нарушает структуру кода или не сможет удовлетворить требованиям, которые появятся в будущем.

Ковбой стремится покончить с текущей задачей и перейти к следующей. Если он читал что-то о технологиях, то назовет это «ускоренным программированием». На самом деле, это просто лень.

[1] Неграмотные люди извратили этот термин и неправильно употребляют его вместо *крэкера* – взломщика компьютерных систем. См. «Крэкеры и хакеры» на стр. 301.

Достоинства

Код ковбоя *работает*, хотя не слишком красив. Ковбои любят узнавать про новинки, но редко пользуются ими (слишком хлопотно).

Недостатки

Вы потратите уйму времени, подчищая за Ковбоем. То, что он оставляет после себя, малоприятно. Код Ковбоя всегда требует дальнейшего исправления, переработки и рефакторинга. Он освоил узкий набор технологий и не обладает настоящим техническим мастерством.

Как быть, если вы один из них

Постарайтесь стать *хакером* в истинном смысле этого слова. Покажите самолюбие в работе и тратьте на нее больше времени. Признайте свои недостатки и попытайтесь исправиться.

Как работать с такими людьми

Лучше не связываться с ковбоями; если его код вообще чего-то стоит, вы замучаетесь его переделывать! При этом они не какие-то вредители, просто с ленцой. Устройте рецензирование его кода. Найдите ему напарника. (Ковбой может сработаться с Нетерпеливым; если хотите посмотреть, как полетят пух и перья, сведите его с Плановиком.)

Плановик

Плановик столько времени обдумывает свою работу, что проект завершится раньше, чем он что-нибудь напишет.

Разумеется, предварительное планирование и крепкий проект необходимы, но этот тип залезает в кокон и прекращает всякое общение с внешним миром, пока не завершит начатое. Тем временем обстановка вокруг уже изменилась.

Озабоченный своей подготовкой, Плановик много читает и изучает. Он помешан на процессе; он знает все о «правильных технологиях разработки», но плохо справляется со сроками и доведением чего-либо до конца. (В конечном счете «помешанные на процессе» становятся менеджерами среднего звена, а потом их увольняют.)

Достоинства

Они *все-таки* проектируют. И они думают. Они не выдают безмозглого кода.

Недостатки

Когда Плановик приступает к работе, возникает серьезная опасность *переусердствовать с проектированием*. Он часто создает очень сложные системы. Плановики – главная причина *аналитического паралича*, когда разработка злоупотребляет методами и мо-

делями в ущерб прототипам и решениям. Плановик любит составлять бесчисленные документы и часто созывать совещания.

Он занимается обдумыванием столько, что на работу не остается времени. Он многое знает, но от теории до практики у него далеко.

Как быть, если вы один из них

Тщательные предварительные проекты необходимы, но попробуйте привлечь инкрементную разработку и создание прототипов как методов верификации проектов. Иногда нельзя останавливаться на каком-то проекте, не начав его фактической реализации. Только она покажет существующие проблемы.

Постарайтесь соблюсти более правильные пропорции между планированием и действиями. Утешьтесь тем, что потратить слишком много времени на проектирование лучше, чем написать плохой код – исправить последний гораздо труднее.

Как работать с такими людьми

Заранее установите для Плановика все контрольные точки и окончательные сроки. Добавьте контрольную точку «*проект завершен*». Плановик будет рад, что его задача признана важной, и это поощрит его вовремя завершить свое проектирование. Обычно этого достаточно, чтобы побудить Плановика действовать.

Избегайте совещаний с Плановиком. Вы убьете час на обсуждение с ним повестки дня.

Ветеран

Программист в возрасте старой закалки. Можете посидеть и послушать его рассказы о добрых старых временах, когда он использовал перфокарты, а памяти машин не хватало, чтобы сложить два целых числа.

Ветеран либо счастлив, что до сих пор может заниматься любимым делом, либо с горечью вспоминает, сколько раз его обошли с продвижением. Он все уже видел, на все знает ответы и не собирается изучать новомодные штучки (он вам скажет, что ничего нового и нет – лишь старые идеи в другой упаковке). Изучать новые языки у него нет желания: «C++ мне не нужен. Спасибо, мне вполне хватает ассемблера».

Ветеран не любит дураков. Он несколько сварлив, и его легко привести в раздражение.

Достоинства

Он программирует много лет и накопил много опыта и мудрости. У ветерана зрелый подход к программированию. Он знает, что такое хорошая программа и что такое плохая и как избежать распространенных ошибок.

Недостатки

Ветеран неохотно осваивает новые технологии. Он устал от модных идей, которые много обещают и мало дают в результате, несколько медлителен и может сопротивляться переменам.

Годы работы под неумелым руководством ослабили его упорство в работе. Он был свидетелем бесчисленных жестких сроков и неразумных администраторов.

Как быть, если вы один из них

Не будьте слишком категоричны в отношении молодых, более увлеченных программистов. Ведь когда-то вы были таким же, и *ваш* код был не так уж плох, не правда ли?

Как работать с такими людьми

«*Вам, молодым, все слишком легко достается.*» Не ходите обедать вместе с Ветераном, иначе вы узнаете, как ему удалось выжить на этой фабрике в течение стольких лет. Не спорьте с ним по пустякам. Оказывайте ему уважение, но относитесь как к коллеге, а не как к божеству.

Разберитесь в мотивации Ветерана. Возможно, он занимается программированием, потому что любит это дело, а может быть, его карьера зашла в тупик.

Фанатик

Фанатик – это новообращенный, которому промыли мозги, слепо верящий в то, что все создаваемое в *Большой Компании* превосходно. Девочки-подростки поклоняются рок-звездам; у программистов – свои идолы. В своем энтузиазме фанатик берет на себя роль бесплатного технологического евангелиста. Он будет пытаться включить изделия Большой Компании во все полученные задания.

Фанатик настолько предан Большой Компании, что исключает любые другие подходы и редко знает об альтернативах. Все, что не блещет совершенством в текущем продукте Большой Компании, будет обязательно исправлено в очередной версии, до которой мы *должны* обновиться немедленно.[1]

Достоинства

Он знает продукты Большой Компании вдоль и поперек и может делать на их базе действительно хорошие проекты. Эта технология обеспечивает ему продуктивность, но не всегда максимальную – другие, незнакомые ему подходы могут быть более эффективными.

---

[1] Среди фанатов могут быть не только «слепые» приверженцы продавцов программного обеспечения. Фанатик может быть сторонником open source или тосковать по устаревшему программному пакету.

Недостатки

Он фанатик, а потому необъективен и непрактичен. Могут существовать лучшие решения, не использующие продукты Большой Компании, но он их не увидит. Но более всего раздражает его непрерывное восхваление Большой Компании.

Как быть, если вы один из них

Никто не требует от вас отказаться от своей возлюбленной Большой Компании. Полезно разбираться в ее технологиях и знать, как их применять. Но не нужно быть слепым приверженцем технологии. Воспользуйтесь другими подходами и образом мышления. Не следует относиться к ним с превосходством и предубеждением.

Как работать с такими людьми

Не вступайте с Фанатиком в философские споры. Не пытайтесь рассказывать ему о достоинствах вашей любимой технологии – он не станет вас слушать. Будьте настороже: один разговор с этим человеком, и вы превратитесь в Фанатика. Его болезнь заразна.

В целом Фанатики безобидны (а со стороны на них даже приятно смотреть), если только вы не находитесь на важном этапе проектирования. В такой момент нужна ясная и непредвзятая точка зрения на предметную область и тщательная оценка всех способов реализации. Учтите, что он может быть прав.

Столкнувшись с неразумными аргументами, противопоставьте им хорошо подготовленную, точную и подробную информацию о достоинствах своего способа и недостатках его подхода.

Монокультурный программист

Это типичный гик – человек, который живет и дышит технологией. В этом вся его жизнь; наверное, она даже снится ему по ночам.

Монокультурный программист обладает очень ограниченным кругозором. Он берет работу на дом и возвращается с готовой написанной системой, в которой исправлены основные ошибки, и планом, как реализовать оставшуюся часть проекта. Вы и позавтракать не успеете, как у него все будет готово.

Достоинства

Монокультурный программист целеустремлен и решителен. Он костьми ляжет, но добьется, чтобы проект заработал. Он не жалеет своих сил и действительно приносит пользу, когда поджимают сроки.

Недостатки

Он рассчитывает, что все должны быть такими же одержимыми и целеустремленными, как он сам, и может неодобрительно относиться к тем, кто не проявляет таких качеств. Главная его опасность

состоит в узкой направленности взгляда, потому что он слишком погружается в задачу.

Как быть, если вы один из них

Займитесь коллекционированием или чем-то еще, чтобы переключиться на другое. *Одна работа и никаких игрушек – очень скучный мальчик*. Но, может быть, вам это безразлично.

Как работать с такими людьми

С такими ребятами работать замечательно. Они полны энтузиазма, которым заражают других, и с их участием проект движется быстро. Но не давайте им слишком много воли. При первой возможности Монокультурный программист и вашу работу сделает тоже! Это может показаться заманчивым, но в итоге вам придется сопровождать чужой код. Оно того не стоит.

Пусть вас не волнует, что у них нет никакой личной жизни, и не чувствуйте себя обязанным тоже дни и ночи сидеть над проектом; иногда самое полезное для проекта – это расслабиться вечерком.

Лодырь

Бездельник, уклоняющийся от работы. Это не сразу заметно, потому что он научился создавать видимость перегруженности задачами. Его «проект» – это игра в солитер, его «исследование» – это разглядывание спортивных машин в Интернете, его «реализация» – занятие личными делами. Лодырь активно увиливает от всех поручений. (О, я слишком *занят*, чтобы взяться за *это*.)

Более скрытый Лодырь занимается только тем, что ему нравится или что считает нужным, а не тем, чем должен. Несмотря на постоянную занятость, он никогда не выполняет задание вовремя.

Лодырь умеет отдыхать. Он слишком часто посещает вечеринки, и его вполне можно застать спящим под рабочим столом. Питается он в основном кофе, но в обед вы найдете его в баре.

Возможно, этот парень выгорел на работе; слишком много проваленных проектов убили в нем желание работать.

Достоинства

По крайней мере, он умеет развлекаться.

Недостатки

Лодырь – явная обуза. Доказать, что он бездельничает, трудно – с некоторыми сложными задачами приходится долго разбираться. Может быть, он и не лодырь; ему может не хватать мастерства, чтобы быстро решить задачу.

Как быть, если вы один из них

Работайте над своей этикой и начинайте понемногу трудиться. Или научитесь заглушать голос совести.

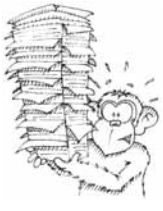
Как работать с такими людьми

Лучше не жаловаться на Лодыря; у вас своих недостатков хватает. Ему в свое время воздастся.

Примите меры, чтобы можно было доказать эффективность вашей работы и вину Лодыря за задержки. Полезно систематически вести журнал проделанной работы. Обычно Лодыря можно заставить трудиться, четко обозначив ему контрольные сроки. Не вздумайте делать за него его работу, даже в критическом положении. В следующий раз он только того и будет ждать.

Постарайтесь и сами не «выгореть»; работу нужно перемежать с развлечениями. Может быть, иногда стоит сходить вместе с ним в бар.

Руководитель поневоле



Это административная классика: разработчик, которого назначили руководить командой, потому что продвигать его по технической линии было некуда.

Всем очевидно, что в этой роли он чувствует себя неуютно. Он не обладает нужными для нее качествами, но старается быть на уровне. Он – программист и хочет программировать. По характеру он не организатор и не умеет руководить людьми, к тому же он необщителен.

Как правило, из программистов получаются на редкость неудачные руководители. Людей, которые действительно отлично умеют руководить командами программистов, мало; тут требуется особое сочетание как технического, так и организаторского мастерства.

Руководитель поневоле обычно обладает мягким характером и нерешительностью – иначе его и нельзя было бы убедить взять на себя эту работу. На него давят с двух сторон: и программисты, и администрация. И все шишки за отставание от графика и низкое качество продукта падают на него. Его лицо выглядит все более запуганным, пока он окончательно не «выгорит».

Достоинства

Руководитель поневоле искренне сочувствует тяжелой доле программистов – он из них вышел и к ним хотел бы вернуться. Часто он проявляет излишнюю готовность взять на себя ответственность за несвоевременный выпуск продукта, чтобы защитить программистов от гнева руководства. Как он не умеет поручить другим работу, так не умеет и распределить ответственность.

Недостатки

Когда руководитель команды пытается писать код, это ужасно. У него нет времени для того, чтобы тщательно проектировать, писать и тестировать. Он наивно рассчитывает, что может полный день программировать, параллельно выполняя обязанности руководителя. Ему не удается все это совместить, поэтому он все дольше задерживается в офисе, чтобы все успеть. Он не может правильно организовать работу, правильно доложить руководству и правильно управлять членами своей команды.

Как быть, если вы один из них

Учитесь. И как можно быстрее.

Если вас не устраивает эта роль, добейтесь, чтобы вас перевели на другую работу. Это не признание своего поражения; нет смысла растрачивать себя, пытаясь заниматься тем, что ты не любишь и не умеешь делать. Не у каждого есть талант или страсть руководить. Займитесь тем делом, для которого у вас есть нужные качества и которое вам нравится.

Если вы любите решать непосильные задачи, попробуйте навести порядок в практике продвижения по должностям в своей компании. Добейтесь признания того, что руководящая должность не должна быть следующей ступенькой после старшего разработчика. Очень немногие программисты становятся удовлетворительными менеджерами; у них мозги устроены по-другому.

Как работать с такими людьми

Будьте благожелательны и постарайтесь чем можно помочь руководителю команды. Вовремя докладывайте ему и старайтесь соблюдать график работ. Если чувствуете, что можете не уложиться в срок, заранее сообщите об этом руководителю, чтобы он смог принять какие-то меры.

Вы

Соблюдая правила вежливости, умолчим об этом любопытном создании. Увы, есть люди, помочь которым невозможно...

Идеальный программист

Из всего этого маловразумительного текста должно быть ясно, что мы представляем собой довольно странную породу. Какой из описанных типов программистов должен отвечать нашим устремлениям? Какая смесь этих типов образует *Идеального Программиста*?

К сожалению, в реальности идеальных программистов не существует – лишь в легендах. Поэтому данный вопрос носит академический характер, но поиск ответа на него укажет нам некоторые цели, к которым следует стремиться.

Легендарный Идеальный Программист – это:

Политик

Он должен быть дипломатом и мудро относиться к огрехам этих странных кодеров и многочисленных прочих обитателей программного производства – менеджеров, тестеров, вспомогательного персонала, клиентов, пользователей и т. д.

Коммуникатор

Он хорошо срабатывается с людьми. Не ограничивается своей частью проекта и готов взяться за другую задачу, если это идет на общую пользу. Он хороший собеседник – умеет слушать других и высказаться сам.

Художник

Умеет разрабатывать элегантные решения и ценить эстетическую сторону хорошо исполненной реализации.

Технический гений

Пишет надежный, промышленного качества код. Его технические знания разнообразны, и он умеет их применять в нужном месте.

Перечитывая этот список, вы понимаете, к чему мы должны стремиться.

И что из этого следует?

Не меняются только мудрецы и глупцы.

Конфуций

Занятно было поглядеть на клетки, в которых сидят эти «code monkeys», и посмеяться над ними. Но какой из этого всего вывод? Если не сделать его, значит, все было лишь развлечением, и после всего прочитанного вы и дальше станете совершать те же глупости, что и раньше.

Чтобы стать лучше как программист, нужно измениться. Меняться тяжело – это противоречит нашей природе. Как говорится, может ли леопард изменить пятна свои? Если бы мог, это был бы уже не леопард. Возможно, в этом разгадка. Побольше бы было среди нас антилоп гну или носорогов.

Поразмыслите над вопросами, приведенными ниже. Полезно будет записать свои ответы в форму «План действий», помещенную в конце этой главы.

1. К какому типу программистов вы себя относите? Если быть честным, то любой из нас найдет в себе какую-то часть каждого из них. Укажите один-два типа, которые лучше всего вас описывают.
2. Какие конкретные достоинства и недостатки у вас есть?
3. Взгляните снова на описания и определите, какие черты вы могли бы в себе изменить. Каким образом можно преодолеть свои дурные черты? Как можно было бы выигрышно воспользоваться хорошими?

Выясните, к какому типу программистов вы относитесь. Определите, как можно выгодно использовать свои хорошие качества и компенсировать плохие.

Для глупцов

Какие выводы нужно сделать из описаний каждого типа программистов, чтобы помочь нам определить необходимые изменения? У всех есть личные недостатки, но в приводимой классификации перечислены некоторые положительные черты и ряд областей, в которых обычно следует совершенствоваться. Чтобы стать хорошим программистом, нужно выработать в себе следующие качества:

Умение работать в команде

Научитесь эффективно сотрудничать. Попробуйте разобраться, какими особенностями характера обладает каждый из ваших коллег, и учитывайте их в общении с ними.

Честность и скромность

Реалистично оценивайте свои способности: нужно знать свои сильные и слабые стороны. Не прикидывайтесь более способным, чем вы есть на самом деле. Выработайте в себе готовность помогать людям и эффективно работать с ними.

Постоянное самосовершенствование

Какими бы обширными ни были ваши знания и опыт, как бы хорош ни был ваш код, *всегда* есть то, чего вы не знаете или не умеете, и есть неправильные позиции, которые следует поменять. Как сказал Конфуций: «Настоящее знание заключается в понимании размеров своего невежества». Согласитесь с тем, что у вас есть недостатки. Хороший программист постоянно занят самосовершенствованием.

Внимательность

Привыкните всегда думать, прежде чем что-то делать. Причиной глупых ошибок является невнимательность. Думайте головой. *Прежде чем* написать любой фрагмент кода, обдумайте свои действия. Перечитывайте написанное, даже если это небольшая модификация.

Увлеченность

Старайтесь поддерживать в себе такой же энтузиазм, как у Нетерпеливого программиста. Если вы любите приобретать новые навыки, больше читайте и практикуйтесь. Если вам периодически требуется делать перерыв в работе, запланируйте себе хороший отдых! Если вы получаете удовольствие от решения новых проблем, найдите себе такую работу, которая будет вас возбуждать.

Если вы станете степенным и скучным, ваша социальная позиция ослабнет, а качество кода снизится.

Резюме

Darwinian Man, though well-behaved,
At best is only a monkey shaved!

(Дарвиновский человек, со всем его благонравием –
всего лишь побрившаяся обезьяна!)

У. Гильберт

Программисты – животные общественные (несмотря на их неумение общаться). Их общежитие вызвано необходимостью: невозможно создавать замечательные и большие программные системы, иначе как командами тесно сотрудничающих программистов, входящими в более крупные социальные структуры (будь то отдел, компания или культура открытого программного обеспечения).

У каждого из этих программистов есть свои слабости и причуды. Их социальная установка оказывает влияние на качество программирования, определяя методы кодирования и отношения с товарищами по команде.

Если вы желаете стать выдающимся программистом, воспитывайте в себе правильную, позитивную общественную позицию.

Хорошие программисты...

- Являются политиками, коммуникаторами, художниками и специалистами
- Работают в команде, честны и скромны, постоянно совершенствуются и увлеченно работают

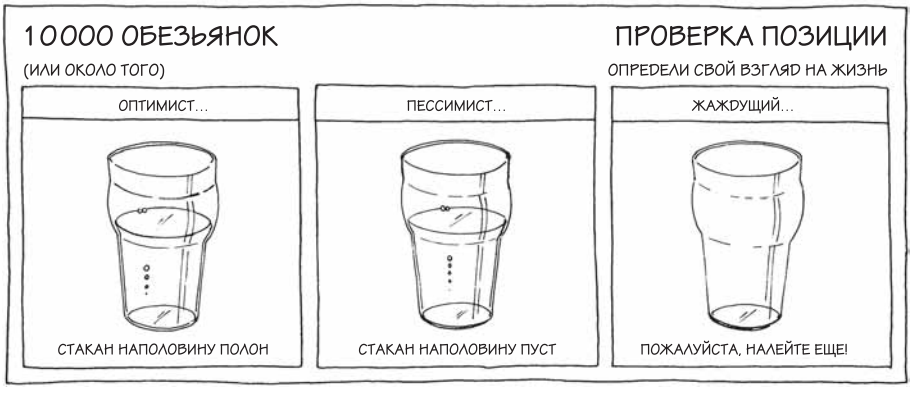
Плохие программисты...

- Не заинтересованы в том, чтобы писать хороший код
- Плохо работают в команде
- Пытаются выглядеть лучше, чем они есть на самом деле
- Стагнируют – не стремятся к самосовершенствованию

См. также

Глава 17. Вместе мы – сила

Более глубоко исследуется динамика команды.



План действий

Заполните анкету «План действий» и попробуйте определить, как применить на практике то, что вы узнали.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 657.

Вопросы для размышления

1. Сколько *нужно* программистов, чтобы заменить электрическую лампочку?
2. Что лучше – энтузиазм, но слабое мастерство (при некоторой компетентности), или выдающийся талант, но отсутствие мотивации?
   a. У кого код окажется лучше?
   b. Кто окажется лучшим программистом (это разные вещи)?Что больше влияет на результирующий код: техническая компетентность или социальная позиция?
3. Мы пишем разные программы в зависимости от «унаследованного» кода. Чем различается написание следующих типов кода?
   a. «Несерьезная» программа
   b. Абсолютно новая система
   c. Расширение существующей системы
   d. Сопровождение старого кода
4. Если программирование – это искусство, то каким должно быть соотношение обдумывания и планирования с одной стороны и интуиции и инстинктивности – с другой? Чем руководствуетесь вы – инстинктом или планом?

Кодеры — План действий

Внимательно заполните эту форму.
Подробности см. в описаниях типов программистов.

Я принадлежу к типу...

Отметьте тот тип, который больше всего вам соответствует.

Можете пометить второй тип, если вам кажется, что одной категории недостаточно. Если возникнет желание пометить более двух типов, я советую вам обратиться к хорошему психиатру.

- ☐ Нетерпеливый
- ☐ Кодер
- ☐ Гуру
- ☐ Псевдогуру
- ☐ Высокомерный гений
- ☐ Ковбой
- ☐ Плановик
- ☐ Ветеран
- ☐ Фанатик
- ☐ Монокультурный
- ☐ Лодырь
- ☐ Руководитель поневоле

Мои достоинства...

Перечислите то, что считаете своими лучшими качествами, умениями и способностями. Сравните с описанием своего типа программиста.

- •
- •
- •
- •

Мои недостатки...

Перечислите то, что считаете своими худшими качествами, умениями и способностями. Сравните с описанием своего типа программиста.

- •
- •
- •
- •

Я могу усовершенствоваться, если...

Как извлечь пользу из своих достоинств и компенсировать недостатки?

- •
- •
- •
- •

Я работаю с...

Вспомните программистов, с которыми ближе всего соприкасаетесь на работе. К каким типам они относятся?

Пометьте все типы. Подумайте, как лучше наладить с ними сотрудничество. Поможет ли установление их типов организовать более эффективное взаимодействие?

- ☐ Нетерпеливый
- ☐ Кодер
- ☐ Гуру
- ☐ Псевдогуру
- ☐ Высокомерный гений
- ☐ Ковбой
- ☐ Плановик
- ☐ Ветеран
- ☐ Фанатик
- ☐ Монокультурный
- ☐ Лодырь
- ☐ Руководитель поневоле

Наша команда может работать лучше, если...

Что могло бы помочь вашей команде лучше писать программное обеспечение? Можно ли принять какие-то конкретные меры?

- •
- •
- •
- •

Вопросы личного характера

1. Если вы еще не заполнили анкету, приведенную выше, сделайте это сейчас. Вы должны определить действия для совершенствования и начать их выполнять!
2. Вот интересная игра, которую вы можете организовать в своей команде, чтобы помочь каждому программисту выработать свой естественный подход к кодированию.

Команды

Если вас много, разбейтесь на группы по три-пять программистов.

Задача

Вашей команде программистов поручена разработка описанного ниже нового продукта. Спроектируйте систему в течение отведенного времени. Объясните, на какие компоненты вы ее разделили и как распределили работу среди членов команды.

Код пока писать не нужно (хотя можно присудить дополнительные очки за демонстрацию действующего прототипа!). Не увлекайтесь перфекционизмом (на это нет времени); просто начните проектировать нечто работоспособное.

Система

В силу обширных сокращений, проведенных в NASA, нет никого, кроме вашей команды, кому можно поручить написание программного обеспечения для управления очередным зондом, который высадится на Марс. ПО должно обеспечить:

- Перемещение по поверхности
- Фотографирование
- Получение метеорологических данных
- Связь с центром управления на Земле
- Быть очень надежным

Сроки выполнения

Это самое смешное. Вам дается всего пять минут. Конечно, это совершенно нереально, но хорошая метафора для графиков наших проектов. (Посмотрите, какое будет отставание...)

Подведение итогов

Посмотрите, хорошо ли сработались между собой люди. Какие команды достигли наибольшего успеха? Какие оказались неудачными? Почему? Как разные люди взялись за задание? Важен не результат выполнения задания, а то, как люди пытались ее решить.

Ответы на такого рода вопросы ясно покажут, какому типу программистов лучше всего соответствует каждый член команды.

17

Вместе мы – сила

Совместная работа и отдельные программисты

В этой главе:

- Совместная разработка программ
- Типы команд разработчиков
- Советы и рекомендации по налаживанию эффективной совместной работы

Важнейшей составляющей формулы успеха является умение ладить с людьми.

Теодор Рузвельт

В какой-нибудь из субботних вечеров, запасшись попкорном и выпивкой, вы усаживаетесь посмотреть фильм. Возможно, вы уговорили каких-то доверчивых людей, не имеющих отношения к компьютерам, посмотреть его вместе с вами. Вы ведь не стали их предупреждать, что это будет «Матрица»?

Фильм, который вы смотрите, стал результатом огромной работы специальных коллективов людей, совместно работавших над созданием последней картины. Вы можете не знать об этом, но в производство фильма было вложено много-много человеко-часов (человеко-дней и «мифических» человеко-месяцев).

Правда, смотря некоторые фильмы, начинаешь сомневаться – стоило ли им браться за дело.

Сравните эту координацию многих усилий с тем, как пишется программное обеспечение. Если вы попытаетесь создать фильм в одиночку, результат окажется жалким. Никто не сможет сделать фильм сам – по крайней мере, сколько-нибудь стоящий. Чтобы вы смогли увидеть по телевизору законченный, отредактированный фильм, нужны усилия многих людей – в области маркетинга, производства, дистрибуции, продаж и т. п. Возможно, вам удалось бы создать «профессиональный» программный пакет самостоятельно, но для этого потребовалось бы невероятно много времени. И кто бы дал вам такой контракт, учитывая коммерческие риски?

В большинстве областей хорошие продукты являются результатом хорошей совместной работы многих людей. Разработка программного обеспечения не составляет исключения. Работа в команде является необходимым условием успешности проекта. Неэффективная команда быстро затормозит всю работу над программным проектом, и его дальнейшее продвижение будет осуществляться благодаря героическим усилиям нескольких увлеченных людей, преодолевающих огромные трудности. Хороший инженер-программист – это больше, чем просто программист. Возможно, вы сумеете вычислить «пи» с огромной точностью, написав пять строчек кода или около того. Отлично. Но необходимо уметь еще многое другое, в том числе работать в команде.

Умение работать в команде – важное качество высококвалифицированного разработчика программного обеспечения.

В этой главе мы рассмотрим работу в команде применительно к нам, программистам. Мы разберем, из чего состоит хорошая групповая работа и как сделать наши команды более эффективными.

Команды – общий взгляд

Известно много типов команд, занимающихся разработкой программных продуктов. На одном конце спектра находятся команды со строгими официальными правилами (ношением костюмов на работе), жесткими структурами и регламентированным рабочим процессом, а на другом – стиль, свойственный движению open source, когда представить свой код могут все желающие, а в продукт он включается в зависимости от его достоинств.

Оба метода работы знают как крупные успехи, так и крупные провалы. В качестве успешных разработок для обоих лагерей можно привести IBM OS/360 и ядро Linux соответственно. Легендарной неудачей стал запуск Ariane 5; эта ракета-носитель, совместно разрабатываемая европейскими странами, взорвалась при первом запуске из-за того, что две команды разработчиков неправильно поняли формально определенный интерфейс. В области открытого программного обеспечения любопытным провалом стала Mozilla; когда Netscape сделала откры-

тым свой код, предполагалось, что он начнет быстро разрабатываться дальше и совершенствоваться. Последующие годы развития Mozilla принесли большое разочарование в сравнении с другими проектами open source.

Разработчик программного обеспечения обычно участвует в командах разного уровня, обладающих различной динамикой и требующих разной степени участия. Рассмотрим такую ситуацию:

- Вы создаете отдельный программный компонент, входящий в более крупный проект. Его разработку вы ведете самостоятельно или в составе группы программистов; это первая команда.
- Компонент должен войти в более общий продукт. Все, кто участвует в его создании (включая разработчиков аппаратуры, программистов, тестеров и нетехнический состав, такой как администрация и служба маркетинга) образуют вторую команду.
- Кроме того, вы работаете в компании, которая может быть занята несколькими проектами одновременно; вот и третья команда.

На практике в любой достаточно крупной софтверной компании уровней групповой работы намного больше. Они представлены на рис. 17.1 вместе с примерами различного рода существующей динамики. Межкомандная динамика, при которой взаимодействуют отдельные команды, выдвигает наиболее сложные задачи групповой работы: со-

***Рис. 17.1.** Уровни групповой работы*

трудничество внутри компании весьма страдает из-за политиканства и ошибок управления. Несмотря на то что компания, по существу, представляет собой одну команду, она нередко бывает расколота соперничающими подразделениями и группами. Это создает не самую лучшую атмосферу для эффективной разработки продукта.

Программисты непосредственно участвуют в совместной деятельности малого масштаба: в работе команд разработчиков. Тут они располагают наибольшими контролем и влиянием. Это уровень, на котором они несут ответственность, имеют право принимать решения по проектированию и реализации и отчитываться о проделанной работе. В меньшей степени программисты ответственны за действия команд более высокого уровня, но групповая работа «верхнего уровня» оказывает на них не меньшее влияние, чем групповая работа «нижнего уровня», хотя это не столь очевидно.

Численность команды разработчиков, а также ее место в организационной цепочке определяют динамику и характер совместной деятельности по созданию программного продукта. Можно возложить всю ответственность за архитектуру, проектирование и реализацию на единственного инженера. В очень маленьких организациях на него также могут быть возложены сбор требований и выполнение всего плана тестирования.

Как только разработчиков становится несколько, характер задачи программирования меняется. Речь уже идет не только об умении программировать; понадобятся навыки социальных контактов, координации и общения. И теперь уже на качество создаваемого программного продукта будет влиять ваше умение работать в команде – в лучшую или худшую сторону.

На результирующий код влияют как отношения внутри команды, так и внешние контакты. Следите за тем, как они отражаются на вашей работе.

Организация команды

Структура команды разработчиков определяется методами руководства и распределением обязанностей среди ее членов. Эти два фактора, как и следовало ожидать, определяют объем кода и размер тех частей, над которыми вы работаете. Отсюда видно, что код, который мы создаем, зависит от организации наших команд.

Методы управления

Проект может управляться на равноправной основе, когда ни один кодировщик не считается более влиятельным, чем остальные, или под руководством суперпрограммиста/менеджера. Команду программистов можно рассматривать как часть технологического потока: получая проект от предшествующей команды, она изготавливает код в соответ-

ствии со спецификациями.[1] Компетентные инженеры-программисты получают больше самостоятельности и ответственности.

Задачи могут распределяться с помощью долгосрочных планов, рассчитанных на многие месяцы (и часто становящихся устаревшими и неточными), либо путем оперативного назначения очередного задания разработчику, как только он завершает предыдущее. Программисты могут работать самостоятельно над отдельными частями системы либо совместно *в парах*, объединяя ответственность и знания.

Разделение ответственности

Основная задача заключается в разделении каждого направления разработки между программистами:

- При *вертикальной организации команды* вы набираете команду из универсалов, способных выполнять широкий ряд функций. Каждому дается определенный участок работы, и он реализует его от начала и до конца – начиная с выбора архитектуры и проектирования, с последующими реализацией и интеграцией, вплоть до тестирования и создания документации.

  Основное преимущество такого подхода состоит в том, что разработчики получают широкий диапазон навыков и большой опыт во всей программной системе. Когда есть основной разработчик для каждой функции, ее конструкция и реализация теснее связаны между собой. Однако универсалы стоят дорого, и их труднее найти. Они компетентны не во всех областях, и потому отдельные задачи решают дольше. Различные функции могут оказаться хуже согласован-

Все встали парами!

Программирование парами – это метод совместной разработки программного обеспечения, особенно модный среди поклонников ускоренного программирования (agile development). Утверждают, что оно приводит к росту эффективности: код создается быстрее и с меньшим количеством ошибок.

Два разработчика пишут код вместе: в одно и то же время, *за одним терминалом*. Пока один (водитель) пишет, другой (штурман) обдумывает сделанное и осуществляет контроль, устраняя замеченные ошибки до того, как они проявятся в действии. Периодически роли в паре меняются. Штурман видит ошибки, которые допускает водитель, сосредоточенный на вводимом коде, и устраняет опасные последствия.

[1] Здесь управление предполагает наличие заменяемого товара – рядовых программистов. См. «Кодер (Code Monkey)» на стр. 386.

Вдвоем можно придумать много способов решить любую задачу, поэтому вероятность того, что выбранное решение окажется лучшим, увеличивается. Из-за такого необычно тесного сотрудничества программирование в паре лучше всего подходит для способных программистов с позитивной общественной позицией.

Согласно отчетам после тренировки два программиста в решении любой задачи показывают совместную продуктивность, которая более чем вдвое превышает продуктивность каждого. В журнале «The Economist» были опубликованы следующие данные: «Лори Уильямс из Университета Юты в Солт-Лейк-Сити показал, что пара программистов лишь на 15% медленнее, чем два независимо работающих программиста, но ошибок у нее на 15% меньше. Поскольку тестирование и отладка часто во много раз более трудоемки, чем первичное программирование, этот результат впечатляет». (Economist 01)

У программирования парами много достоинств. Оно способствует передаче знаний и помогает обучению, концентрирует внимание (менее вероятно, что вы будете предаваться мечтательности, долго болтать по телефону или отключитесь), поднимает дисциплину и сокращает отвлечения (когда два человека заняты совместным делом, вы вряд ли станете им мешать; другое дело – одинокий программист, бессмысленно уставившийся в пространство). Этот метод действует как оперативный механизм предварительного контроля – моментальной проверки кода – и способствует повышению качества кода. Это социальный процесс: при правильном выборе напарников он способствует укреплению морального духа (хотя он может быть ужасен, если эти двое вызывают друг у друга раздражение). Программисты ближе знакомятся между собой и лучше понимают, как работать вместе. Программирование парами укрепляет коллективные права на код, содействует распространению правильной культуры кодирования, придает выразительность процессу разработки.

ными между собой, так как их реализуют разные разработчики. Заказчику приходится работать с большим числом людей, поскольку нет единой контактной точки – отдельно с каждым разработчиком необходимо обсуждать требования и утверждать проект.

Для эффективности такой команды нужно определить общие стандарты и нормы, а также хорошо наладить связь, чтобы избежать дублирования работы. Надо сразу договориться относительно общей архитектуры, иначе получится хаотическая и беспорядочная система.

- При *горизонтальной организации*, напротив, команда набирается из специалистов, между которыми распределяются все задачи разработки так, чтобы каждый в нужное время мог проявить соответст-

вующие способности. Поскольку каждой стороной разработки (определением требований, проектированием, кодированием и т. д.) занимается специалист, качество результата должно быть выше.

Результаты во многом противоположны тем, которые достигаются при вертикальной организации: отдельные участки работы оказываются лучше связанными между собой, но возникает опасность, что сами они будут менее прочными, поскольку над ним работало много людей. Внешнее взаимодействие команды (с клиентами или другими подразделениями компании) осуществляет небольшое число специалистов. Это облегчает работу самой команды и ее внешних контактов.

Необходимо обеспечить хорошую координацию специалистов и их наблюдение за каждым рабочим заданием до конца, иначе их польза будет ограничена. Когда в каждой процедуре разработки участвует много людей, управлять командой становится труднее; организация рабочего процесса усложняется. Такая структура требует хорошей связи, установленных процессов и плавных передач между разработчиками.

Нельзя утверждать, что какой-то один из видов организации является «правильным». Выбор наиболее подходящего типа зависит от участников команды, ее численности и характера выполняемой работы. Наиболее практичным является, видимо, некий промежуточный вариант.

Организация и структура кода

Организация команды неизбежно оказывает влияние на создаваемый ею код. В профессиональном фольклоре это воплощено в *законе Конвэя*. В краткой формулировке он гласит: «Если компилятор будут писать четыре группы разработчиков, получится четырехпроходный компилятор». Код всегда отражает структуру и динамику пишущих его команд. Основные программные компоненты разделены по командам, и связь между компонентами отражает взаимодействие между командами. Если группы тесно сотрудничают, связь между компонентами оказывается простой и четко определенной. Когда команды действуют разрозненно, их код взаимодействует плохо.

Естественно стремиться создать четко определенные интерфейсы для продукта труда каждой бригады, чтобы облегчить взаимодействие с этой бригадой. Это полезно даже тогда, когда правильнее было бы добраться до какой-то внутренней части другой компоненты. В этом отношении команды могут определять произвольное разбиение; несмотря на наши добрые намерения, проектные решения обусловливаются составом команд.

Разумеется, это не отменяет инкапсуляции и абстракции, но их применение в проекте должно быть разумно обосновано. Во всяком случае код, который вы создаете, должен определять состав и организацию команд.

Организуйте свою команду в соответствии с кодом, который собираетесь построить, а не наоборот – код в соответствии с имеющейся командой.

Инструменты для групповой работы

Существуют базовые инструменты, помогающие организовать действенную команду разработчиков. Они облегчают сотрудничество и превращают совместную разработку из хаоса в отлаженный механизм. Сами по себе они не превратят вашу группу в отряд командос, но это арсенал, которым необходимо располагать каждой классной команде, – необходимая предпосылка эффективного совместного труда разработчиков.

Управление версиями

Сосредоточием усилий команды разработчиков является исходный код – для него и предназначается эта система. Управление версиями помогает навести порядок в том, кто, чем и когда занимается, предоставляет окончательную версию самого свежего кода, позволяет управлять модификациями, делать откат ошибочных вариантов и гарантировать получение каждым разработчиком обновленных версий исходного кода. Эта система одинаково необходима для работы команды, состоящей из сотни людей, и для проекта с единственным разработчиком.

База данных ошибок

Мы уже видели, как она помогает в разработке (см. «Система контроля ошибок» на стр. 205), но следует отметить ее полезность для взаимодействия между командами: она находится в центре между тестированием и разработкой. С ее помощью организуются тестирование и исправление ошибок, устанавливаются приоритеты ошибок, назначаются ответственные за решение проблем и осуществляется контроль за открытыми позициями в списке ошибок программного продукта. Она позволяет узнать, какими ошибками в данный момент занимаются разработчики, а какими – тестеры.

Средства автоматизации групповой работы

Команда нуждается в эффективной инфраструктуре связи, особенно если она распределена географически. Централизованная система, содержащая календарь, адресный справочник и средство планирования совещаний, составляет цифровую основу администрирования.

Также необходим механизм для совместного доступа и разработки документов. Помощь работе группы могут оказать «вики» (средства документации, базирующиеся на веб) и внутренние телеконференции (почтовые дискуссионные форумы с постоянным хранением данных).

Методология

Важно установить ясную и всеми воспринятую методологию разработки, иначе работа будет носить хаотичный и произвольный характер. Один разработчик будет отдавать свой код для окончательной версии, в то время как другой будет отказываться сделать это до проведения тщательного тестирования и отладки. Один разработчик застопорит весь процесс, пока не получит сложной детальной спецификации, а другой будет сразу писать прототип. И не такие проблемы являлись поводом для религиозных войн.

Методология определяет детали процесса разработки, устанавливает ответственность за разного рода работу и порядок ее передачи. В результате каждый разработчик знает, что он должен делать и как ему участвовать в работе команды. Выбор методологии должен основываться на численности команды, типе создаваемого кода и способностях, опыте и динамике людей. Это обсуждается в главе 22.

План проекта

Чтобы выполнить какую-либо работу предсказуемым и своевременным образом, необходима определенная организация. Ее обеспечивает план проекта, конкретизирующий работу каждого участника на протяжении разработки. План приносит пользу только тогда, когда основан на здравых оценках и своевременно корректируется.

Программисты славятся неумением оценивать сроки, а менеджеры – неумением планировать. Нельзя поддаваться давлению работать в соответствии с нереалистичным планом проекта. Это действительно сложная проблема, которую мы разберем в главе 21.

Болезни, которым подвержены команды

Оправиться после неудачи иногда бывает легче,
чем развить успех.

Майкл Эйзнер

Даже при наличии хороших программистов и замечательной организации команда может оказаться неработоспособной. Есть много причин, по которым команды оказываются неспособными решать свои задачи, и мы можем выделить категории обреченных на провал команд так же, как выделили стереотипы программистов, и попытаться сделать из этого выводы.

Ниже приводятся некоторые классические типы неудачных команд. Для каждого из них мы опишем:

- Специфический путь к провалу
- Симптомы (по которым вы сможете узнать, что вас ожидает)
- Как вытащить команду, попавшую в эту колею

- Как остаться успешным программистом в условиях неудачной команды (иногда *вопреки* команде)[1]

Будем надеяться, что вы не найдете свою команду в приводимом списке.

Вавилонская башня

Как и в случае библейских строителей, «вавилонская» команда страдает обширным нарушением коммуникативного процесса. Если программисты теряют возможность общения, разработка обречена – если что-то получается, то, скорее, по счастливому случаю, а не преднамеренно.

При неэффективном общении делаются некорректные допущения. Часть работы уходит в песок, чреватые ошибками ситуации игнорируются, о сбоях забывают, программисты выполняют одну работу дважды, интерфейсами пользуются неверно. На мелкие задержки никто не обращает внимания, в результате они оборачиваются колоссальным отставанием, потому что никто не следит за соблюдением графика.

Строители Вавилонской башни раскололись из-за множества языков, на которых они говорили.[2] Однако интернациональные проекты редко страдают от вавилонского синдрома – когда требуется преодолевать языковые барьеры, люди проявляют стремление к налаживанию хорошего общения.

Разработчиков могут разделять не только языки, на которых они говорят. Разница в подготовке, методологии, языках программирования и даже в характерах может приводить к непониманию друг друга членами команды. Мелкое недоразумение, если его не преодолеть, может в итоге вырасти; обиды и разочарования накапливаются. При худшем исходе члены «вавилонской» команды перестают разговаривать друг с другом, и каждый программист сидит в своем углу и занимается своими делами.

Эта проблема может возникнуть как в самой команде, так и между сотрудничающими командами. Внешний вавилонский синдром возни-

---

[1] Я не утверждаю, что эти стратегии решат общую проблему команды; это просто первые пришедшие в голову способы сделать нынешнюю работу с минимальным риском проблем.

[2] Книга бытия 11:1–9.

кает, когда разработчики отказываются общаться с тестерами или команда менеджеров не контактирует с разработчиками.

Признаки опасности

Свидетельством того, что ваша команда движется в сторону Вавилона, является нежелание одного разработчика обращаться к другому по какому-либо вопросу, поскольку он чувствует, что это бесполезно. Признаком нездоровья служит отсутствие подробных спецификаций и нечеткие контракты в коде. Почта посылается в слишком малом или слишком большом объеме. Когда почты слишком много, значит, все кричат и никто не слушает: ни у кого нет времени, чтобы разобраться с потоком информации.

Для дороги в Вавилон характерно, что совещания команды не созываются и никто толком не знает, в каком состоянии проект. Возьмите любого разработчика, и он не ответит вам, в правильном ли направлении движется разработка.

Выход из ситуации

Начните говорить с людьми. Не нужно сдерживать себя! Вскоре все начнут делать то же самое.

Отношения в вавилонской команде трудно восстанавливать, если в ней укоренилось разложение, поскольку моральный дух достиг самой низкой отметки, апатия стала всеобщей и никто не верит в возможность перемен. Самая эффективная стратегия – попытаться поднять боевой дух команды, сплотить разработчиков между собой. Проведите какое-нибудь общественное мероприятие, чтобы встряхнуть их, – хотя бы просто совместный выезд на природу с выпивкой. Купите на обед пиццу и разделите ее на всех.

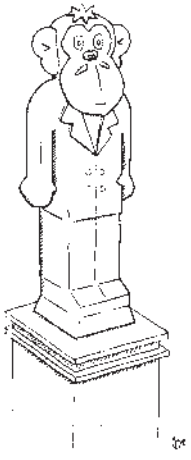
Затем выработайте какую-либо стратегию, чтобы заставить людей общаться друг с другом. Создайте небольшие фокус-группы для рассмотрения новых функций. Назначьте двух человек ответственными за участок проектирования. Введите программирование парами.

Стратегия успеха

Чтобы написать хороший код при наличии таких проблем, необходимо соблюдать строгую дисциплину. Прежде чем приступить к пакету работ, строго определите его. Если необходимо, сами напишите спецификацию и разошлите по почте всем, кого она касается, чтобы получить их одобрение (установите крайний срок подачи комментариев и сообщите, что отсутствие отзывов будет расцениваться как согласие). После этого успешность будет определяться выполнением согласованной спецификации.

Полностью заморозьте все внешние интерфейсы кода, чтобы не было никаких сомнений относительно ваших расчетов или того, что другие могут ждать от вашего кода.

Диктатура



Это настоящее *представление с одним действующим лицом* – команда, руководимая сильной личностью, нередко являющейся и высококвалифицированным программистом. Остальные программисты должны только поддакивать (даже если не хотят этого делать) и выполнять указания Диктатора, не обсуждая их.

Некоторые команды прекрасно работают в таком стиле – удачно подобранный благожелательный диктатор и команда, относящаяся к нему с уважением. Проблемы возникают, когда личность Диктатора не отвечает занимаемому им положению или его техническая подготовка недостаточна (см. раздел «Псевдогуру» на стр. 389). Если его самомнение встает поперек пути, команда в опасности: возникнет противостояние, которое заведет в тупик.

Когда такая команда формируется умышленно, она представляет собой иерархию с определенными линиями подчинения. Фредерик Брукс нашел сходство такой структуры с *хирургической бригадой* (Brooks 95). В хирургической бригаде наиболее высококвалифицированный технический специалист, главный хирург,[1] занимает самое верхнее положение: он выступает в роли кодера, а не администратора. Он выполняет основной массив работы и несет полную ответственность за возможные неприятности (если пациент умрет). Ему помогает специально подобранная бригада. В ее состав входит хирург-интерн, который решает более мелкие, менее рискованные задачи, помогает главному хирургу и учится у него. В команду также включают соответствующие эквиваленты анестезиологов, сестер и, возможно, других младших хирургов (например, чтобы наложить швы).

С такого рода командой связаны две опасности. Первая возникает, когда на диктатора оказывается внешнее давление с тем, чтобы он в большей мере занимался административными задачами; его техническая специализация практически гарантирует отсутствие навыков управления. Он станет уделять меньше внимания программному обеспечению, и проект рухнет. Вторая опасность – это самопровозглашенный диктатор, которого не признает команда. Рабочий процесс встанет, потому что команда не имеет соответствующей структуры и морально не готова поддерживать его лидерство.

Признаки опасности

Структура такого типа развивается медленно и незаметно, по мере того как будущий диктатор медленно меняет направленность своей

[1] Обычно это специалист-технолог по классификации групповых ролей Белбина.

рабочей функции и усиливает свою власть. Приближение диктатуры вы можете заметить, если обнаружите, что часто произносите такие фразы:

- *Я не могу этого сделать, не посоветовавшись с...*
- *Нет, ...будет недоволен, если мы поступим таким образом.*
- *Но ... говорит, что сначала мы должны сделать...*

Выход из ситуации

Если ваш Диктатор недостоин роли руководителя группы, нужно исправлять положение. В противном случае ваша команда зачахнет под игом тирании. Либо обсудите с ним эти проблемы (честно говоря, это вряд ли принесет успех – люди с трудом меняются, особенно если у них повышенное самомнение), либо сместите его, объяснив проблему менеджеру.

Освободившись от самовластия, вы должны либо изменить структуру команды, либо обзавестись новым царем. Главные хирурги на дороге не валяются, поэтому, вероятно, лучше будет изменить организацию команды.

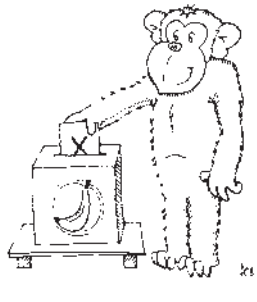
Стратегия успеха

В условиях Диктатуры (функционирующей нормально или ненормально) определите свой уровень полномочий и ответственности. Посоветуйтесь по этому поводу с тем, чье мнение действительно имеет вес – с менеджером или руководителем команды.

Однако после утверждения для себя справедливой роли в разработке вы (и остальные программисты) все равно должны прислушиваться к Диктатору и работать вместе с ним, даже если вам не нравится его нынешняя позиция. В противном случае у вас не получится совместной работы и вы не сможете писать код, который будет стыковаться. Проект должен быть единым, иначе программа не будет работать.

Не следует проявлять неуважение или грубость в отношении Диктатора; это лишь снизит моральный дух команды и обозлит вас.

Демократия



Как говорится, *все люди созданы равными*, и здесь это доведено до конца. Это команда равных – программисты близкого уровня мастерства и совместимых характеров, организованные не иерархическим образом. Это необычная для корпоративного мира структура, где принято, чтобы кто-то был боссом. Идея самоорганизующейся команды может показаться еретической. Однако доказано, что такая модель может быть действенной. Некоторые демократические команды периодически выбирают лидера из числа своих членов исходя из того, чьи навыки наиболее востребованы на данном этапе проекта.

Часто явного лидера нет, и все решения принимаются по общему согласию. Такой схеме, как правило, следуют разработки open source.

Мы иногда забываем вторую часть поговорки: «*Все люди созданы равными, но потом отдаляются друг от друга*». Такая культура команды может быть действенной только при особом подборе участников. Опасность для команды, основанной на таком похвальном принципе, заключается в том, что по мере ее роста или при уходе одного из участников (того, кто стимулирует принятие группой решений) могут возникнуть проблемы. Команда может утратить целеустремленность, не будучи в состоянии достичь согласия ни по какому вопросу и вовремя получать результаты. В пределе команда бесконечно обсуждает один и тот же вопрос, созерцает свой пупок и ничего не может добиться.

Проводя бесконечные совещания и одни и те же дискуссии, команда подвергается опасности *аналитического паралича*: сосредоточиться на процессе, а не на осуществлении проекта. Как и при реальной демократии, настоящие дела могут потонуть в море политиканства.

Можно нечаянно прийти к Демократии, если руководитель команды неэффективен и не способен принимать решения. Такой неудачный лидер постепенно теряет возможность управлять, сам того не замечая. Разочарованная команда начинает коллективно брать на себя его роль, требуя принятия решений и выбирая направление разработки.

Кризис команды проходит особенно тяжело, когда она устроена по демократическому принципу, даже если такая организация выбрана намеренно. Если межличностные разногласия не позволяют выбрать нужного в текущей ситуации лидера, необходимо передать руководство проектом лидеру, взятому со стороны.

Признаки опасности

Больная Демократия видна за версту: скорость, с которой принимаются решения, катастрофически падает. Если руководитель и существует, то все делается помимо него, чтобы не сталкиваться с его нерешительностью. Руководителем он остается только номинально; никто не признает его власти или способности чего-то добиться.

В отсутствие руководства никто ни за что не отвечает; непонятно, кто должен обеспечить выполнение задачи, поэтому ничего не делается. Проходят недели, а спецификация по-прежнему не закончена, и никакого прогресса не видно.

В условиях разгула демократии необходимость принятия малейшего решения переводит команду в режим комитета, который работает несколько дней. Или принимается решение типа «Пусть будет *так*, пока мы не решим иначе». «Но да будет слово ваше: да, да; нет, нет; а что сверх этого, то от лукавого»,[1] иначе вы будете вечно

[1] Матфей 5:37. Если же вы вавилонский строитель, то ваше «да» может быть *Oui*, а ваше «нет» – *Nein*!

перелопачивать старый код, как только кто-то решит, что нужно сделать иначе.

Можно также обратить внимание на отчуждение молодых программистов из-за невозможности стать лидерами.

Выход из ситуации

Цель демократии – устранить специфическое препятствие: принятие решений боссом, который не всегда для этого достаточно компетентен (особенно если он не разбирается в технике). В неправильно функционирующей демократии нет процедуры принятия решений, и решения не принимаются ни на каком уровне. Чтобы вернуться к здоровой демократии, обеспечьте возможность свободного перемещения руководства в команде и легкость замены лидера. Не пытайтесь организовывать демократию, пока у вас не будет достаточно потенциальных лидеров.

Как и в случае любого хромающего проекта, постарайтесь сделать проблемы видными всем – и разработчикам, и менеджерам. Пусть будет понятно, кто несет ответственность за проблему – тем более, если это не вы!

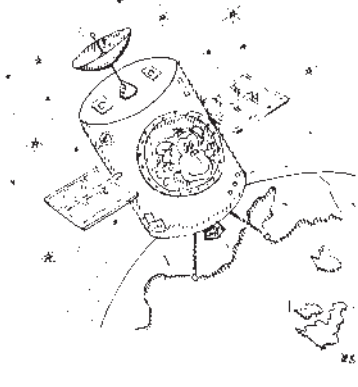
Можно попытаться поправить нерешительную Демократию, продемонстрировав сильную волю; не миритесь с ухудшением положения. Возможно, вас назовут смутьяном, но в конце концов ваше умение достичь результата будет оценено. Однако остерегайтесь сами превратиться в полудиктатора.

Стратегия успеха

Ради собственного блага избегайте людей, не способных решать простейшие вопросы.

Постарайтесь, чтобы вам выделили четко определенную часть проекта и поставили реалистичные сроки. Такой якорь удержит вас во время приливов и отливов нерешительного руководства.

Станция-спутник



Команда-сателлит, отколовшаяся от основной команды разработчиков, создает условия для проявления специфических неприятностей и ловушек. Трудно работать сплоченным коллективом, когда часть его физически отделена – как отрубленная рука.

Сателлитом может оказаться целый местный отдел или часть вашей команды, размещенная по другому адресу. *Работа на дому* представляет собой частный случай, когда сателлит состоит из одного человека.

Довольно часто высшее руководство находится в головном офисе, расположенном в другом месте, но это не вызывает проблем, поскольку его участие в повседневной программистской работе невелико. Если же часть команды разработчиков располагается на расстоянии долгих миль, то для успеха проекта нужно принять определенные меры. Здесь вам нужно проявить осмотрительность – разделенные команды без специальных мер совместно не работают.

Программирование требует тесного взаимодействия команд, потому что отдельные части создаваемого нами кода должны тесно взаимодействовать. Все, что угрожает взаимодействию между людьми, также угрожает нашему коду. Команды-сателлиты создают следующие опасности:

- Физически разделенные команды лишены возможности неформального общения, которое случается возле кофеварки. Они не могут просто и динамично сотрудничать. Уровень общего постижения и группового понимания кода становится ниже.
- Разработка теряет связность. Местная практика и культура разработки на каждой территории будут различаться (пусть даже незначительно). Отсутствие единообразия в методологиях затрудняет передачу работы.
- Поскольку вы не очень хорошо знакомы с людьми из команды-сателлита, то неизбежно относитесь к ним с меньшим доверием и теплотой. Возникает разделение *мы и они*.
- Старая поговорка гласит: «С глаз долой – из сердца вон». Не встречаясь регулярно с коллегами из команды-сателлита, вы забываете про них, не знаете, как у них идут дела, и не задумываетесь над тем, какое значение для них имеет ваша работа (технически или процедурно).
- Наличие сателлитов элементарно затрудняет общение. Приходится выяснять графики работы других программистов: когда у них совещание или отпуск.
- Проекты, разнесенные по разным странам, вызывают проблему часовых поясов. Промежуток времени, в течение которого команды могут общаться, сокращается.

Признаки опасности

Когда команды разделены географически, тут все ясно, но разделение может происходить и в пределах одного офиса. Рассадив разработчиков по разным комнатам, вы создаете искусственные барьеры, которые могут помешать сотрудничеству.

Остерегайтесь и разобщения между подразделениями. Оно тоже может нанести ущерб. Например, команды тестеров часто располагаются отдельно от разработчиков, иногда в другом здании или секции. Это никуда не годится; важное взаимодействие между двумя командами затруднено, а в результате страдает процесс контроля качества.

Выход из ситуации

Команду-сателлит не следует считать обреченной – просто ее наличие требует более тщательного наблюдения и управления. Проблемы не являются непреодолимыми, но они неприятны – по возможности избегайте их возникновения.

Правильным решением будет собрать в начале проекта всех участников команды для личного общения. Это способствует формированию согласия, доверия и взаимопонимания. Еще лучше проводить встречи регулярно. Обеспечьте еду и напитки для проведения встречи; это раскрепощает людей и создает атмосферу для лучшего общения.

Распределите работу так, чтобы требовалось меньше сотрудничества и координации между сателлитом и основной командой. Тогда проблемы со связью будут оказывать меньшее влияние.

Заранее определите интерфейсы между разработками разделенных участков, чтобы сократить проблемы взаимодействия между кодом. Но не допускайте, чтобы разделение на команды определяло структуру кода: его качество может от этого проиграть. Программирование связано с выбором практичных вариантов, поэтому выбор должен быть разумным.

Программное обеспечение для автоматизации групповой работы становится важным инструментом обеспечения эффективной связи с командой-сателлитом. Полезно воспользоваться средствами мгновенной передачи сообщений. И не стоит бояться пользоваться телефоном!

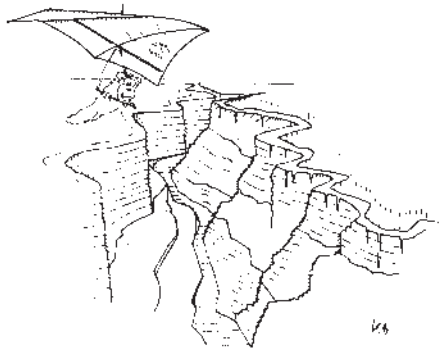
Стратегия успеха

Если часть ваших людей работает на другой площадке, постарайтесь лучше познакомиться с ними – в личном и профессиональном отношении. Это весьма важно. Вы будете знать, каковы их реакции и когда они настроены дружественно или саркастически. Постарайтесь дружелюбно относиться к программистам из группы-сателлита, иначе вас легко могут счесть сварливым идиотом, если по несчастью они будут вечно звонить вам в неудобное для вас время.

Выясните точно, кто именно работает на отдельной площадке. Запомните их имена, кто чем занимается и как с ними связаться. Совершенствуйте свои коммуникативные навыки. Не стесняйтесь связаться с кем-то, если это необходимо: задумайтесь, стали бы вы с ними разговаривать, если бы они находились рядом.

Большой Каньон

Это тип команды, составленной из программистов, чьи мастерство и опыт лежат на противоположных концах спектра. Существует явный разрыв в мастерстве; пропасть между старшими и младшими разработчиками столь велика, что команда разделилась на две различные



фракции. Почти в каждой команде типа «Большой Каньон» это явление носит как технический, так и общественный характер. Ничуть не лучше, если старшие разработчики сидят в одном анклаве, а младшие – в отдельном гетто.

Причины возникновения культуры Большого Каньона часто кроются в истории: проект начинают несколько классных разработчиков, которые должны быстро определить архитектуру и написать код, подтверждающий правильность концепции. Естественно, они сидят рядом и образуют эффективное и сплоченное подразделение. По ходу дела возникает потребность в большем количестве программистов, и появляются молодые работники. Они садятся на свободные места по краям и получают небольшие задачи, решая которые должны изучить структуру системы.

В отсутствие должного контроля старшие разработчики усваивают начальственную позу и свысока смотрят на младших разработчиков. Они поручают последним небольшие и скучные участки работы, чтобы самим продолжить заниматься интересными и важными задачами. Старшие считают, что передача младшим всех своих замыслов отняла бы недопустимо много времени, и в этом есть доля правды. Таким образом, молодежь может предполагать, что не получит ответственные и интересные задания. В результате растет ее разочарование.

Младшие программисты хотят изучать свою профессию, обладают энтузиазмом юности и страстью к программированию. У старших программистов совсем другое (более пресыщенное?) мировоззрение с надеждами получить административные или более высокие технические должности. Различие в мотивациях движет фракции в разных направлениях.

Признаки опасности

Внимательно наблюдайте за тем, как расширяется команда. Следите за демографическими данными и распределением работы между участниками. Контролируйте социальную динамику своей команды; в больных командах возникают группировки.

Выход из ситуации

Проблема Большого Каньона в разнородности команды; выделяются фракции, которые не смешиваются между собой. Рецепт прост: разработайте планы, способствующие перемешиванию людей. Например:

- Рассадите людей так, чтобы фракции перемежались. Конечно, это отнимет время от разработки, но день, потраченный на пере-

движение столов, может обратиться неделями продуктивного труда.

- Устраивайте совещания, способствующие распространению информации.
- Организуйте программирование парами, сведя вместе опытных и молодых программистов. Пусть младший сядет за руль, а старший будет штурманом. Это будет дисциплинировать старшего и расширять знания младшего.
- Составьте план обучения молодых разработчиков. Это подчеркнет разницу в мастерстве, но будет способствовать сближению фракций.
- Пересмотрите названия должностей всех разработчиков – не поощряют ли они опасную и вредную иерархичность?

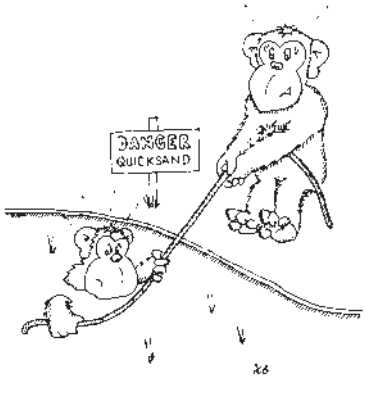
Стратегия успеха

Относитесь ко всем как к равноправным коллегам.

- Если вы старший программист, согласитесь с тем, что молодым необходимо учиться. Вы и сами когда-то были новичком и мало понимали в жизни. Не гребите под себя все интересные задачи. Старайтесь, чтобы другие тоже могли брать на себя ответственность.
- Если вы младший программист, просите, чтобы вам давали более сложные задачи. Старайтесь учиться. Выполняйте свои задания как можно лучше; это будет свидетельствовать о вашей готовности к большей ответственности.

Зыбучие пески

Достаточно одного человека, одной ложки дегтя, одной отвязавшейся пушки, чтобы выбить из колеи всю команду. Нужно несколько хороших программистов, чтобы создать хорошую команду, но нужен лишь один плохой, чтобы испортить команду. Завязшая в зыбучих песках команда неожиданно для себя столкнулась с негодным человеком. Все может произойти незаметно: все могли не обратить внимания на проблему, а виновный не имел намерения причинить вред.



Попасть в зыбучие пески можно по ряду причин:

- В команде появился технически *некомпетентный программист* (возможно, это Ковбой, описанный на стр. 391). Его не так просто сразу выявить, и никто не заметит, что он пишет плохой код. Но мина

замедленного действия заложена, и позже проект застопорится, пока его безобразие не будет вычищено и заменено.

- Какой-то *моральный урод* потихоньку разрушает боевой дух всей команды, вытягивая из нее энтузиазм и бодрость. Через несколько недель никто не может собраться с силами, чтобы написать хоть какой-то код, и хочется броситься вниз с ближайшего моста.
- *Горе-менеджер* делает все прямо противоположное тому, что должен делать хороший менеджер: постоянно меняет свои решения, меняет приоритеты задач, сдвигает сроки и обещает клиентам невозможное. Члены команды не чувствуют под собой почвы, потому что она постоянно уходит у них из-под ног.
- Некий программист умудряется вопреки теории относительности *искажать время*, так что идущие около него процессы замедляются. Все, с чем он сталкивается, требует невероятно долгого времени. Решения стопорятся в ожидании его ответа, его код никогда не готов, и совещания вечно открываются с опозданием по его вине. Уважительная причина всегда найдется – он, скорее всего, был занят другими, очень важными делами, – но он накапливает множество невыполненных вовремя задач и у него никогда не доходят до них руки. В конце концов, другим программистам это надоедает и они обходятся без него.

В команде Зыбучих песков недостатки одного участника могут быстро подорвать общую продуктивность. Особенно это опасно, когда винов-

Горе-менеджер в деле

Разработчики образовали хорошую команду. Они работали с удовольствием. И работали напряженно. К несчастью, их руководство оказалось посредственным (если не хуже).

Однажды утром руководитель (у которого, судя по его виду, плохо начался день) созвал совещание и стал выражать недовольство тем, что разработчики не понимают «реалий окружающего мира», бездельничают и никогда не соблюдают установленных им (безумных) сроков (он уже продал продукт, которого нет в природе). Он отметил, что иногда люди отсутствуют на работе в положенные часы и что отныне все должны быть на местах. Иначе будут последствия.

Результат был замечательным.

В этот день программисты не работали. Вообще. Потом они решили работать строго в установленное время – никаких неоплаченных сверхурочных. Я полагаю, что этот менеджер одним махом снизил производительность и моральный дух команды не менее чем вдвое.

ник занимает достаточно высокое положение. Чем больше на нем ответственности, тем тяжелее последствия.

Признаки опасности

Искать нужно того, кто не вписывается в команду. Это тот, на кого все жалуются,[1] или программист, который всегда работает в одиночестве (потому что его все избегают).

Выход из ситуации

Самая радикальная мера и, пожалуй, самая простая – избавиться от Зыбучего песка. Но сначала нужно узнать, кто это, что иногда бывает очень трудно. Менеджеры боятся претензий в несправедливости при увольнении и не хотят от кого-то избавляться лишь потому, что несколько человек не могут с ним поладить. Нужны какие-то вопиющие проявления некомпетентности, чтобы это случилось.

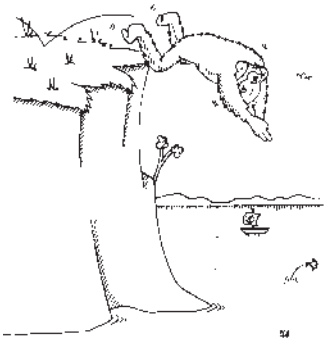
Поэтому приходится искать способы сократить вызываемый им хаос или лучше интегрировать его в команду.

Стратегия успеха

Самое главное: сам не будь Зыбучим песком![2]

Ну, а если это не вы сами, постарайтесь как можно лучше обезопасить себя от его влияния. Поменьше общайтесь с ним, чтобы не поднималось кровяное давление. Не ставьте себя в большую зависимость от его кода и ни в коей мере не полагайтесь на получение от него каких-либо данных. Не втягивайтесь в его дурные привычки и не реагируйте слишком бурно, действуя прямо противоположным образом; это еще более усугубит ситуацию.

Лемминги



Подобно кучке милых пушистых зверьков, способных в безрассудном порыве прыгнуть с ближайшей скалы, эта команда горит страстным желанием выполнить любое данное ей поручение. Даже если оно фальшивое.

Команда состоит из очень доверчивых и преданных участников. Они компетентны технически, но не видят дальше полученных ими инструкций. Их энтузиазм и стремление похвальны, но при отсутствии среди них человека с воображением – который задает вопросы и пытается понять, что дейст-

[1] Они будут жаловаться за его спиной – и это тоже втягивает команду в зыбучие пески. Никто прямо не скажет о проблеме. Никто не хочет раскачивать лодку. Никто не хочет с ним связываться.

[2] Лука 6:42

вительно нужно, не ограничиваясь спецификациями, – команда пребывает в постоянной опасности изготовить продукт, который был заказан, но не тот, который требовался.

Команды Леммингов особенно уязвимы в условиях становления компаний. Неприятности начинаются, когда руководство говорит: «Быстренько напишите этот код; потом его переделаем». «Потом» не будет никогда; вместо этого лемминги слышат: «Компании нужен новый код, и как можно скорее, так что набросайте еще вот это». В скором времени культура команды начинает плясать под чужую дудку. Постепенно работа становится все более сложной, задачи ставятся титанические, а качество кода неуклонно падает.

В итоге вместо команды остается куча переломанных костей у подножия 20-метровой скалы. Игра закончена.

Признаки опасности

Если вам не нравится спецификация, над которой вы сейчас работаете, может быть, вы попали в команду леммингов. Без веры в свой проект вы становитесь просто наемным кодировщиком. Если вы слышите пустые обещания и заняты бессмысленной работой, если никто не обсуждает план и не указывает на его недостатки, вы явно очутились в стране леммингов. Надеемся, вам у нас понравится.

Выход из ситуации

Оцените, чем занята ваша команда в настоящее время. Продолжайте работать, но сравните требования клиента с тем продуктом, который намечается. Будет ли код, над которым вы работаете, удовлетворять действительные потребности? Или это недальновидное решение, которое долго не удержится в вашей базе кода или которому не предстоит долго жить?

Стратегия успеха

Критически относитесь к порученной вам работе. Разберитесь в ее мотивации. Защищайте принципы хорошего программирования и никогда не верьте, что кто-то даст вам потом время на исправление кода, если только это не записано в плане, которому вы доверяете.

Личное мастерство и качества, необходимые для работы в команде

Просто удивительно, как много можно сделать,
когда всем безразлично, кому достанутся лавры.

Роберт Йейтс

Конечно, не все такие команды обречены. Попробуем разобраться, что можно извлечь из этой ситуации и как ее поправить. В оставшейся части главы мы рассмотрим приемы, которые помогут улучшить рабо-

ту вашей команды программистов и избежать отмеченных ловушек. Хотя инструменты и технологии действительно помогают повысить производительность, но наибольший прирост лежит в сфере человеческого фактора и отношения людей к своей работе.

Каждая команда программистов состоит из отдельных личностей. Чтобы повысить производительность труда своей команды, не нужно далеко ходить – разберитесь в *своем* отношении к команде и проекту, над которым вы совместно трудитесь. Не все из нас менеджеры, но это та область, на которую мы действительно можем оказывать влияние.

Отличный программист должен уметь отлично играть в команде. Существует ряд не связанных с техникой навыков, качеств и позиций, которые должен выработать в себе эффективный член команды, и лишь после этого можно оценивать, насколько уверенно он владеет языком программирования или умеет проектировать.

Общение

Групповая работа невозможна без общения. Отдельные части не могут дружно работать без общения. Цель и видение не могут разделяться всеми, если нет общения. Проекты действительно терпят неудачу в результате отсутствия налаженного общения.

Общение внутри команды происходит разными путями: в беседах между отдельными инженерами, с помощью телефона, совещаний, письменных спецификаций, почтовой переписки и мгновенных сообщений. Иногда даже с помощью картинок! У каждого носителя своя динамика применения и свои особенности, определяющие его пригодность для того или иного вида обсуждения.

Наиболее эффективно общение, когда в нем участвуют (хотя бы в качестве наблюдателя) все заинтересованные стороны. Оно должно в достаточной мере входить в детали и не требовать слишком больших затрат времени или труда. Для общения нужно выбрать подходящее средство; например, проектные решения должны быть отражены в виде письменных спецификаций, а не устных соглашений.

Мы уже отмечали, что сам код является формой общения. Программист *должен* уметь хорошо общаться. Это подразумевает возможность как получения, так и передачи информации, а именно способность:

- Писать точно выраженные спецификации, ясно излагать мысли и быть кратким.
- Правильно читать и понимать спецификации, внимательно слушать и понимать сказанное.

Помимо общения внутри одной команды нужно учитывать общение *между* разными командами. Классическим примером неправильного общения служат отношения, складывающиеся во многих компаниях между службой маркетинга и инженерами. Если маркетинг не консультируется с инженерами относительно существующих технических

возможностей, то он начинает продавать продукты, которые компания не в состоянии изготовить. Эта проблема заключена в порочный круг: после первого такого происшествия и получения печального опыта эти две команды могут прекратить общаться (затаив обиду). В результате та же история будет повторяться многократно.

Наличие каналов для эффективного общения жизненно необходимо для нормальной работы команды. Такие каналы необходимо организовать и поддерживать. Хороший программист должен уметь хорошо общаться.

Скромность

Это важное качество, часто отсутствующее у людей нашей профессии.

Скромные программисты стремятся внести свой труд на благо команды. Они не бездельничают, вынуждая других выполнять всю работу. Они не считают себя единственными способными людьми, которые могут сделать что-то ценное.

Нельзя загрести под себя все хорошие задачи; просто невозможно, чтобы один человек делал все. Вы должны стремиться дать возможность участия в работе другим членам команды – даже если это задача, которой вы сами *хотели* бы заняться.

Необходимо прислушиваться к другим людям и ценить их мнение. Ваша точка зрения не единственная, как и предлагаемое вами решение. Вовсе не факт, что только вы знаете лучшее решение всякой проблемы. Слушайте других, уважайте их, цените их работу и учитесь у них.

Разрешение конфликтов

Нужно быть реалистами: есть люди, которые непременно «заводят» друг друга. В таких случаях нужно занимать рассудительную и ответственную позицию и учиться избегать (или разрешать) конфликтные ситуации. Конфликты и враждебность резко снижают эффективность команд.

Однако управляемый конфликт может оказаться важным фактором успеха в работе вашей команды. Коллеги по работе, которые стимулируют и провоцируют друг друга, могут создавать отличные проекты. Разногласия могут оказывать облагораживающее влияние и обеспечивать проверку состоятельности идей. Когда вы знаете, что ваша работа подвергнется критическому анализу, это мобилизует ваше внимание.

Необходимо следить, чтобы такого рода конфликты были конструктивными и оставались на профессиональном уровне, *не переходя* на личности.

Обучение и приспособляемость

Вы должны постоянно приобретать новые технические навыки, но при этом также учиться работать в составе команды. Этот дар не дает-

ся с рождения. Вновь созданная команда должна учиться работать вместе, ее участники должны изучить характер друг друга, сильные и слабые стороны коллег и возможности применения их на общее благо (подробнее см. в разделе «Рост команды» на стр. 437).

Эмерсон писал: «Каждый человек, которого я встречаю, в чем-то превосходит меня». Посмотрите, чему вы можете научиться у своих коллег. Перенимайте у них знания, сделайте для себя уроки из их внеш-

Потеря связи

Есть много способов общения в нашем тесно взаимосвязанном мире, и необходимо научиться эффективно их применять на пользу сотрудничества в команде. Главное – знать присущие им динамику, этикет и отдельные преимущества.

Телефон

Его лучше всего использовать для связи, когда ответ нужен вам срочно, но звонок отвлекает человека от его занятия. Поэтому звонки, не носящие срочного характера, доставляют неудобства. Мобильные телефоны значительно облегчили возможность связи, что имеет как достоинства, так и недостатки.

Когда вы только слышите собеседника, вам недоступны выражение его лица или язык телодвижений. При разговоре по телефону легко ошибиться и неправильно истолковать смысл, придя к неправильным умозаключениям.

Технари часто остерегаются использовать телефон. Не нужно бояться: для срочной связи он незаменим.

Электронная почта

Это асинхронное и низкоскоростное средство связи. Можно указать степень срочности, но доставка почты никогда не происходит мгновенно. Это богатый возможностями носитель, с помощью которого можно быстро переслать приложение или ответить в удобное для вас время. Его часто применяют для циркулярной рассылки меморандумов многочисленным получателям. Почтовый архив представляет собой довольно надежное средство регистрации общения. Электронная почта является очень мощным механизмом связи.

Нужно научиться пользоваться почтой как инструментом, не становясь ее рабом. Не стоит немедленно открывать каждое новое полученное сообщение; ваша работа будет слишком часто прерываться, и пострадает производительность труда. Выработайте для себя график чтения почты и придерживайтесь его.

Мгновенный обмен сообщениями (Instant messaging)

Быстрое средство разговорного характера, требующее большего внимания, чем электронная почта, но допускающее больше возможностей для игнорирования, чем телефон. Интересный и полезный промежуточный вариант.

Письменный отчет

Письменный отчет является менее разговорной и более долговременной формой общения, чем электронная почта. Письменные отчеты и спецификации являются официальными документами (см. главу 19). Они требуют больше времени для подготовки, вследствие чего их трудно ошибочно истолковать. Письменные отчеты подвергаются всеобщему изучению и обсуждению, вследствие чего носят более обязательный характер.

Совещания

Несмотря на все чудеса современной техники, им трудно соперничать со старым проверенным общением глаз к глазу, при котором можно быстро и эффективно решать проблемы. Программисты очень часто избегают личного общения (этот вид по природе своей не общественный!), но совещания занимают важное место в нашей групповой работе. Подробнее мы поговорим о них во врезке «Это судьба» на стр. 435.

него вида и поведения. Научитесь общаться с ними. Отмечайте высказанную ими критику в любой области – от формального рецензирования кода до мнений, высказанных походя в разговоре.

Приспособляемость тесно связана с учебой. Если в команде есть потребность в знаниях, которыми не обладает ни один из ее членов, и нет возможности привлечь внешние ресурсы, необходимо найти решение. Легко приспособляющиеся программисты могут быстро освоить требуемую область и помочь команде.

Знание пределов своих возможностей

Если вам поручена работа, сделать которую вы не в состоянии, или обнаружилось, что вы не можете ее завершить, нужно *как можно раньше* сообщить об этом своему руководителю. Иначе вы не сможете выполнить свою часть проекта и пострадает вся команда.

Многим кажется, что признание своей неспособности – признак слабости. Это не так. Лучше признать ограниченность своих возможностей, чем подвести всю команду. Хороший менеджер найдет возможность оказать вам помощь, а попутно вы приобретете новые навыки, которых прежде у вас не было.

Принципы групповой работы

Ниже приводятся основные заповеди командной работы, которые, будучи надежно усвоены командой, изменят ваш стиль работы. Они переносят центр внимания с отдельных личностей на программное обеспечение и его коллективную разработку. Запомните: для того чтобы эти принципы принесли пользу вашей команде, вы должны решительно осуществить соответствующие перемены, а не просто согласиться с ними и продолжать кодировать по-старому.

Коллективное владение кодом

Многие программисты защищают границы своей работы. Это естественно: программирование – очень личное и творческое действие. Мы гордимся, когда удается создать элегантную модель, и не хотим, чтобы кто-то трогал ее и калечил наш шедевр. Это было бы святотатством.

Но для эффективной работы в команде необходимо спрятать свое самолюбие, входя на территорию фабрики по производству программного обеспечения. Не нужно жаловаться, что кто-то ковырялся в вашем коде. Это групповой проект, и код принадлежит не вам, а всей команде. Если не принять такой позиции, то вместо успешной программной системы каждый программист будет создавать собственную империю.

Никто из программистов не является владельцем никакой части кода. Каждый из участников имеет доступ ко всему коду и может модифицировать его по мере надобности.

Когда принимается такая культура, команда защищает себя от мелких царьков, правящих своими островками кода. Если некто написал код, на который никому нельзя посмотреть, то что будет, когда этот человек уйдет из проекта? Потеря специалиста в какой-то области создаст серьезные трудности в работе команды.

Не нужно питать к своему коду родительских чувств, охранять его и лелеять. Вместо этого должно быть здоровое отношение к интересам команды. Чувство *собственника* можно попробовать заменить чувством *управляющего*. Управляющий не владеет тем, что находится под его попечением; они назначаются, чтобы блюсти *интересы* владельца. Основная обязанность управляющего участком кода – это содержание его в исправности, уход и охрана границ. Обычно все изменения осуществляет управляющий, но их проведение можно доверить некоторым участникам команды с его итоговой проверкой. Это конструктивное отношение к своему коду, которое должно пойти на пользу команде.

Уважайте чужой код

Даже в просвещенной среде разработчиков, где отсутствует личная собственность на код, нужно уважительно относиться к коду, написанному другими лицами. Не нужно ничего в нем менять произволь-

но. Это особенно важно, если с кодом работают в данное время. Нельзя модифицировать код, с которым работает другой программист; это вносит невероятное смятение.

Уважение к чужому коду подразумевает, что вы сохраняете его стиль представления и текущие проектные решения. Не допускайте беспричинных и необоснованных модификаций. Придерживайтесь установленного способа обработки ошибок. Комментируйте свои модификации соответствующим образом.

Не делайте мелких поправок, которые при проверке кода будет неприятно видеть. Они появляются, когда нужно срочно заставить код работать, и маленькое исправление в каком-то месте дает возможность скомпилировать ваш код. Если вы потом забудете заменить заплатку должной модификацией, это приведет к снижению качества чужого кода. Даже в мелких поправках нужно проявлять уважение.

Нормы кодирования

Для того чтобы в условиях коллективной разработки создавать приемлемый код, ваша команда должна соблюдать ряд норм кодирования. Они определяют стандарт, которого должен придерживаться программист при написании кода, что гарантирует достижение системой определенного минимального уровня качества.

Не следует возбуждать споры по поводу расположения кода (хотя лучше, если все будут придерживаться одинакового стиля). Однако по поводу стандарта и механизма документирования кода, использования языка и его идиом, процедуры создания интерфейсов и архитектуры должно существовать полное единогласие.

Команды, обходящиеся без таких норм, на самом деле ими располагают – только в виде неписаных соглашений. Проблема таких неявных норм заключается в том, что новые члены команды пишут код в стиле, расходящемся с общепринятым, пока не усвоят существующую в команде культуру.

Определите, что считать успехом

Чтобы иметь ощущение прогресса и успешности совместной работы, участники команды должны иметь перед глазами четкий набор задач и целей. Одних контрольных точек в плане проекта недостаточно, хотя они *могут* способствовать мотивации: определите побольше контрольных точек в виде краткосрочных задач и отмечайте их прохождение.

Нужно определить критерий успеха, чтобы команда знала, как он выглядит и как его достичь. Что можно считать успехом для вашего текущего проекта? Своевременный выпуск продукта, достижение заданного уровня качества,[1] удовлетворенность клиента, получение опреде-

[1] А как его измерить?

ленного дохода или снижение количества ошибок до заданного? Установите приоритеты для этих факторов, и пусть программисты знают, чем должна мотивироваться их работа. Они станут действовать совсем по-другому.

Установите ответственность

У всех эффективных команд есть четкая структура и ясные обязанности. Это не обязательно означает наличие строгой иерархии и нескольких уровней управления. Структура команды должна быть ясной и понятной. Должно быть очевидным:

- За кем остается последнее слово при принятии важных решений? Кто следит за бюджетом, кто принимает решения о найме/увольнении, кто определяет приоритетность задач, кто утверждает проекты, готовность кода к выпуску, составляет графики и т. п.? Эти функции не обязательно выполняются внутри команды, но команда должна знать о существовании людей, которые их выполняют.
- Где нужно остановиться и чья голова полетит, если становится ясно, что проект провалился?
- Каковы *обязанности* членов команды и перед кем они *отчитываются*? За что они персонально отвечают, чего от них ждут и кому они подчиняются?

Избегайте истощения

Команда не должна получать нереальные задания. Проверьте, что проект, за который вы принимаетесь, реально осуществим – ничто так не обескураживает, как понимание того, что провал неизбежен.

Следите за распределением работы между программистами. Старайтесь не поручать всю трудную или рискованную работу одним и тем же немногим людям. Это частая ошибка, особенно если команда выросла несколько обособленных специалистов. Если они выдохнутся в результате чрезмерного труда или беспокойства по поводу возможных последствий сделанных ошибок, то поставят под угрозу проект и деморализуют команду.

Поздравьте команду, если она работает результативно и с полной отдачей сил. Сделайте это публично. Не забывайте хвалить и поощрять членов команды. Немного поддержки и энтузиазма оказывают удивительно бодрящее действие.

Давайте людям разные задания; не заставляйте никого заниматься одними и теми же задачами, пока им это не наскучит и они не потеряют интерес. Дайте каждому возможность учиться и овладевать новыми навыками. «Перемена работы – лучший вид отдыха». Даже если нет никакой возможности снизить темпы работы, некоторое разнообразие поможет избежать усталости.

Жизненный цикл команды

Собраться вместе – это лишь начало, не разбежаться врозь – прогресс, а работать вместе – это успех.

Генри Форд

Рассмотрим наши команды программистов с точки зрения того, как протекает их жизнь. Команды не возникают из воздуха и не существуют вечно.

Успешные команды развиваются и работают целенаправленно; их возникновение нельзя приписывать случаю.

В жизненном цикле команды есть четыре стадии: создание, рост, работа и закрытие. На каждой стадии цель действий своя. Иногда происходит многократное повторение этих стадий в разном порядке, но каждая команда проходит через все стадии. Более мелкие команды в составе основной команды разработки проекта проходят через аналогичный процесс; это рекурсивная модель. Ниже мы рассмотрим детали, относящиеся к каждой из этих стадий.

Создание команды

Вырисовывается новый проект. Для него нужна команда разработчиков. *На старт. Приготовились. Марш!* Назначается руководитель, и он обязан собрать команду. Участников можно забрать из других команд или специально нанять для этого проекта. Откуда бы ни пришли люди, они должны вместе составить эффективную команду – от этого зависит успех проекта (и сохранение своего места руководителем).

С этого все начинается. Ряд членов образует ядро команды. На этом этапе не ведется серьезная работа, и команда еще толком не составилась. При формировании команды нужно решить ряд важных вопросов:

- Необходимо определить место команды в организационной структуре. С какими другими командами она будет поддерживать отношения? Определите каналы связи между ними, чтобы было ясно, каким образом происходит обмен данными между подразделениями и с кем нужно поддерживать контакт.

  Тщательно продумайте это и постарайтесь минимизировать связь между разными командами, чтобы не усложнять работу. На этом этапе вы можете способствовать успеху команды, устранив лишние бюрократические препоны.

- Чтобы быть эффективной, команда должна включать в себя компетентных, талантливых участников, которые могут образовать отдельное высокопроизводительное подразделение. Их знания и опыт должны перекрывать все важнейшие области, которые *могут* понадобиться, иначе разработка будет останавливаться, пока не найдет-

ся нужный человек. Планируйте расширение команды по мере надобности и решите, когда нужно будет начать поиск новых людей.

- Выберите подходящую модель команды и озвучьте ее, иначе команда примет случайную структуру и хаотическую рабочую практику. Структура команды должна обеспечивать минимум администра-

Это судьба

У попавших на фабрику программистов быстро вырабатывается отвращение к совещаниям, потому как их проводится бесчисленное множество и все они ужасны. Совещания поглощают уйму ценного времени, которое можно было бы использовать для программирования и тем самым предотвратить провал проекта. Одни и те же несколько пунктов обсуждаются до бесконечности, пока всех не распустят, а потом все забывают сказанное, и на следующем совещании все повторяется сначала.

Чтобы команда работала эффективно, нужно *научиться* проводить эффективные совещания. Это не столь сложно; необходимо только немного планирования и дисциплины. Вот руководство из семи пунктов для проведения эффективных совещаний: правила боя. Ответственность возлагается на того, кто созывает совещание.

1. Проводить совещания важно и необходимо. Не стесняйтесь созвать совещание, когда в этом возникает необходимость. Однако обойдитесь без него, если неформальный разговор в коридоре может решить проблемы быстрее и без труда.
2. Заранее объявите о совещании – не за несколько часов, а за несколько дней. Правильно определите участников; их должно быть не слишком мало (что толку, если те, кто принимает решения, будут отсутствовать) и не слишком много (толку не выйдет, поскольку каждый будет стремиться, чтобы его выслушали).
3. Выберите разумное время проведения совещания. Не раннее утро, когда половина участников еще не вполне проснулась, и не конец дня, когда все устали и хотят домой.
4. Установите время окончания и заранее его объявите. Соблюдайте назначенный регламент. Тогда участники будут знать, сколько времени у них еще останется на работу. Если вы не укладываетесь в срок, оставьте вопрос для очередного совещания.
5. Сообщите каждому участнику, по какому поводу созывается совещание и почему он приглашен. Разошлите повестку дня. Каждый, кому требуется подготовиться, должен знать, чего от него ждут.

6. Все должны быть оповещены о месте проведения совещания. Проследите, чтобы помещение было должным образом подготовлено: доска, компьютер, достаточно стульев (звучит глупо, но об этом часто забывают).
7. Распределите роли до начала совещания. По меньшей мере, должны быть назначены:

   Председательствующий

   Этот человек ведет совещание, направляет дискуссию, чтобы она не уходила от темы, и следит за соблюдением распорядка. Его задача – вовремя завершить совещание с принятием соответствующего постановления (например, о созыве нового совещания).

   Секретарь

   Ведет протокол совещания, потом оформляет его и распространяет среди заинтересованных лиц (их круг может быть шире, чем участники совещания).

   Лица, принимающие решения

   За ними окончательное слово по каждому вопросу. Когда нет полномочного лица, дискуссии продолжаются бесконечно без какого-либо результата.

Уясните цель совещания. Совещания бывают либо с целью распространения информации (бычно это аудитория поневоле), либо с целью разрешения конфликтов (выработать решение насущной проблемы). Чтобы совещание стало полезным, все должны понимать его цель и действовать соответствующим образом. Личные проблемы могут быстро увести информационное совещание в произвольном направлении; председательствующий должен следить за этим и пресекать подобные попытки.

тивных расходов и внутренних каналов связи, чтобы достичь как можно большей гибкости.

Начальная цель при формировании команды – создать нечто большее, чем просто группу людей. Вам требуется не очередное сборище или клуб, а тесно сплоченный рабочий коллектив, члены которого заинтересованы в достижении одной общей цели.

Не собирайте команду вместе, пока не разберетесь *точно*, какова ее задача. Если попросить людей начать работу, не дав им настоящего задания и оставив в ожидании дальнейших указаний, они усвоят на будущее привычку тянуть дело и никогда не будут полностью использовать свой потенциал. Если команда не может сразу начать работать, не стоит пока собирать ее вместе.

Рост команды

После формирования команды в составе основных сотрудников, проект начинает набирать скорость. Команда должна расширяться, чтобы справляться с растущей нагрузкой. Этот процесс имеет несколько сторон: с одной стороны, команда должна увеличиваться численно, с другой – должны расти ее опыт и видение. Команда должна развиваться как *внутри*, так и *снаружи*.

Внутреннее развитие команды

Во время совместной работы члены команды знакомятся друг с другом на личном и профессиональном уровнях. В команде устанавливаются система работы и культура кодирования. Поначалу этим процессом нужно тонко управлять, чтобы культура была здоровой и отвечала структуре и задачам команды. Это *кристаллизация* команды по Тому Демарко – объединение отдельных членов в сплоченную команду. (DeMarco 99)

На этой стадии уравниваются личные и групповые цели, определяются индивидуальные роли и отношения. Складывающаяся в это время обстановка в команде задает тон всему проекту, поэтому следите, чтобы не появились скептицизм или дурные настроения.

Если еще нет инфраструктуры команды, она создается по мере набора темпа работы. Развертываются такие средства, как система управления версиями и система автоматизации групповой работы. Составляются спецификации проекта, уточняются задачи и определяется объем работ.

Внешнее развитие команды

Внешний рост характеризуется включением в команду новых членов. Это то, что можно видеть снаружи. Ниже перечисляются роли, существующие в команде на пике ее численности. Каждой роли необязательно должна соответствовать отдельная должность – все зависит от общей численности команды. В небольшой команде один человек может исполнять несколько функций в течение полного рабочего дня или частично. В больших проектах некоторые функции могут выполняться отдельными подразделениями.

Аналитик

Связующее звено между командой программистов и заказчиком. Аналитик (или *специалист в предметной области*) исследует проблемы клиента и разбирается в них достаточным образом, чтобы написать спецификацию, которую могут реализовать разработчики.

Архитектор

Авторитет в деле проектирования верхнего уровня, который разрабатывает структуру системы в соответствии с требованиями аналитика.

Администратор базы данных

Проектирует и разворачивает структуру базы данных для обеспечения проекта.

Проектировщик

Работает на более низком, чем архитектор, уровне, проектируя компоненты системы. Часто эту работу выполняют программисты.

Программист

Разумеется, это главный человек в команде!

Менеджер проекта

Несет общую ответственность за проект, принимая важнейшие решения. Менеджер находит компромисс между противоречивыми требованиями к проекту (такими как бюджет, срок завершения, требования, набор функций и качество программного обеспечения).

Администратор проекта

Помогает менеджеру, решает практические проблемы функционирования команды.

Инженер по контролю за качеством

Разрабатывает планы QA и добивается соответствия качества выпускаемого кода принятому стандарту.

Специалист по подготовке пользователей

Составляет руководства по работе с программным продуктом, следит за точностью маркетинга, составляет графики обучения и т. п.

Специалист по поставке продукта

Другими словами, *инженер по выпуску*. Организует упаковку, изготовление, дистрибуцию и установку конечного продукта.

Инженер по сопровождению

Осуществляет поддержку продукта, установленного у пользователей.

Успешный проект не должен охватывать все эти функции. Когда ощущается потребность в какой-либо из этих ролей, но до того как она становится острой, подыскиваются соответствующие специалисты. Их выбор требует понимания администрацией характера кандидата, его профессионального мастерства и качеств, которые требуются для выполнения данной функции. После того как команда сформировалась, новых членов следует включать в нее с учетом сложившейся рабочей практики и дополнительности к уже имеющимся работникам.

Групповая работа

Это стадия, когда команда полностью укомплектована. Вращаются все шестеренки, и процесс создания программного продукта неумолимо движется вперед.

В этой фазе команда проходит большую часть своего существования, добиваясь целей проекта. Для этого общая большая задача разбивается на серию мелких. Каждый член команды получает свой пакет работ и синхронизирует свою деятельность (скажем, с помощью проведения совещаний или личного общения). Результаты труда всех членов команды интегрируются по мере готовности. Постепенно программный продукт обретает форму.

Работа происходит согласно заранее установленной процедуре разработки, но команда должна приспосабливаться к возможным переменам: решению непредвиденных проблем, изменениям в своем составе или проклятому «синдрому плавающих требований». По ходу дела каждый член команды должен выявлять возникшие проблемы и риски и справляться с ними.

Команда должна войти в колею разработки – найти подходящий темп работы и достигать промежуточных целей. Однако колея не должна превратиться в рутину. Не бойтесь, если возникнет необходимость перетряхнуть рабочую практику, чтобы команда не впадала в самодовольство и лень или чтобы противодействовать неэффективным членам команды, которые представляют опасность для дальнейшего продвижения.

Роспуск команды

Со временем все, даже самые затянувшиеся, проекты подходят к концу. Конец может означать выпуск успешного продукта, которым доволен клиент. Возможен вариант, когда продукт не удался и разработка прекращена досрочно. В любом случае проект завершен, и команда снимается с него.

С самого начала разработки нужно иметь в виду четкую конечную точку. Ни одна команда не может быть вечной или планировать работать до бесконечности. Желание завершить работу служит мотивацией, и многие программисты не стали бы сильно напрягаться на работе, не будь они поставлены в жесткие временные рамки.

По этой причине любая команда должна планировать роспуск или преобразование в другой тип команды (например, по обслуживанию или сопровождению) по завершении проекта. Этот план должен быть рассчитан как на обычные условия, так и на случай аномального завершения.

Роспуск команды не происходит в одночасье. Проекты не прекращаются неожиданно; они постепенно сворачиваются. Обычно люди снимаются с проекта по мере того, как отпадает в них надобность. Никому не нужны праздно шатающиеся люди, только потребляющие ресурсы. При уходе каждого участника позаботьтесь о том, чтобы сохранились его важные знания или результаты труда. При распаде команды очень легко потерять информацию.

Не теряйте информацию при уходе людей из команды. Организуйте ее передачу и получите от них все важные данные, включая документацию по коду, инструментарий тестирования и инструкции по сопровождению.

Что происходит, когда команда добирается до окончания проекта? Можно предпринять одно из следующих действий:

- Перевести команду в режим поддержки для сопровождения продукта.
- Начать какую-то новую разработку (может быть, новую версию того же продукта).
- Инициировать посмертное вскрытие, если проект провалился.

Кадры решают все!

Вот несколько простых принципов управления командой разработчиков программного обеспечения. Без программистов не сделаешь программу, поэтому необходимо использовать методы, которые реализуют заложенные в людях возможности и помогут им в совместной работе. Даже если вы сейчас не занимаете руководящей должности, вам будет полезно узнать о них, чтобы оценить, как управляется ваша нынешняя команда и каково отношение к ее членам. То, что мы уже рассматривали, здесь сведено в краткие практические рекомендации.

Лучше меньше, да лучше

Большие команды требуют больше каналов связи и больше администрирования, в них больше потенциальных возможностей для провала и труднее обеспечить общее видение.

Распределяйте задачи соответственно способностям, а также мотивации

Не забывайте про Принцип Питера[1]: не нужно выдвигать отличных программистов на руководящие должности, которые они не могут или не хотят занимать.

Делайте инвестиции в людей

- Вы получите от них больше отдачи, если что-то вложите в них. Технологии быстро развиваются; дайте им возможность обновлять свои знания. Иначе они уйдут туда, где могут приобрести более ценный опыт.

---

[1] Принцип Лоуренса Дж. Питера гласит: «Хороших работников продвигают на высший уровень, соответствующий их компетентности, и поскольку они с ним справляются, их продвигают еще на одну ступеньку – на тот уровень, где они уже некомпетентны.

Не выращивайте экспертов

Опасно, если некий программист становится единственным экспертом в своей области. Из-за этого весь проект ставится под угрозу.[1] Есть люди, которые активно стараются стать незаменимыми программистами, других делают ими насильно, не разрешая заниматься ничем другим. Когда ваш эксперт захочет увидеть новые горизонты, он уйдет от вас. И как вы после этого будете сопровождать программный продукт?

Выбирайте людей по принципу дополнительности

Невозможно, чтобы команда состояла целиком из экспертов мирового масштаба. Точно так же нельзя, чтобы все члены были неопытными программистами. Необходимо здоровое сочетание разных навыков. Необходимо также подбирать людей по личным качествам, чтобы они хорошо сочетались и могли работать вместе.

Избавляйтесь от тех, кто мешает

Тех, кто не вписывается в команду, нужно удалять. Делать это нелегко, но гнилой член может испортить всю команду, а последствия промедления будут тяжелыми (см. «Зыбучие пески» на стр. 423). Не будьте сторонним наблюдателем хода событий и не надейтесь, что все само уладится. Решайте проблему.

Успех команды зависит от того, кто в нее входит. В успешной организации правильно выбирают сотрудников и полностью используют их возможности.

[1] У менеджеров проекта есть шутка по поводу количества грузовиков для проекта: сколько человек может быть сбито грузовиком без того, чтобы проект рухнул.

- Разделить команду на части для работы над отдельными проектами (или отпустить людей, если их контракт закончился).

Использовать команду повторно или распустить – сложный вопрос, который часто решается плохо. То, что команда успешно завершила один проект, не значит, что она будет столь же успешна в следующем. Для нового проекта может потребоваться иное сочетание профессиональных качеств или другой подход к разработке. Тем не менее разумно сохранить хорошую команду в целости. Хорошо интегрированные команды из компетентных специалистов с эффективной культурой работы встречаются редко. Разбрасываться ими не следует.

Если есть выбор, делать его нужно исходя из особенностей следующего проекта. Иногда такой выбор перед вами не стоит: в небольших организациях команда проекта – это все имеющиеся разработчики. У вас

просто нет возможности переставлять и подбирать людей, поэтому вы вынуждены работать с ними над следующим проектом.

Резюме

Важно понимать, что нужна команда, а команда должна получить по заслугам и в случае успеха, и в случае поражения. Найдется немало желающих приписать успех своим заслугам, но никто не хочет нести ответственность за поражение.

Филип Колдуэлл

Программистов по-настоящему заботит только то, как написать хороший код, поэтому имеет ли это все какое-то значение? Имеет: здоровье и структура вашей команды оказывают прямое влияние на здоровье и структуру вашего кода. Они неразрывно связаны. Программы пишутся людьми. Хорошо согласовываться друг с другом, обмениваться информацией и образовывать крепкую структуру должны не только программные компоненты, но и программисты, пишущие их.

Для хорошей групповой работы недостаточно четкого определения процесса или задания структуры. Основу хорошей групповой работы составляют хорошие люди. Как говорится, «целое больше, чем простая сумма его частей». Но это так, лишь если все части работают хорошо. Если какая-то часть с браком, страдает целое. Наши личные черты влияют на качество команды, а потому и на качество создаваемого ею кода. Чтобы создавать хороший код, нужно позаботиться о личных чертах. Знание своих врожденных социальных качеств поможет вам улучшить ваше мастерство программиста.

Профессиональный программист должен уметь работать в команде. Наряду с техническим мастерством нужно уметь создавать части, которые складываются в общую мозаику. А это означает умение общаться и работать с другими. Это означает понимание своей функции и надлежащее ее выполнение с привлечением всех своих способностей. Это означает сотрудничество с другими членами команды и главенство не личных, а командных интересов.

В последующих главах некоторые из этих тем сотрудничества найдут дальнейшую разработку: мы расскажем об управлении версиями, методологиях разработки и технологии оценки и планирования.

Хорошие программисты...

- Не ограждают свой код от других
- Берутся за любую задачу, если это нужно для системы
- Учатся и растут, делая общее дело; достигают личных целей, не пренебрегая общими

Плохие программисты...

- Пытаются воздвигнуть свою империю и стать незаменимыми
- Занимаются своими личными задачами и ищут самую привлекательную работу

Хорошие программисты...

- Умеют общаться: всегда выслушают других членов команды
- Скромны, делают общее дело и уважают коллег

Плохие программисты...

- Работают по личному плану в ущерб интересам команды
- Стремятся утвердить свое собственное мнение
- Считают, что команда должна их обслуживать и что они – украшение команды, дар божий сообществу программистов

См. также

Глава 16. Кодеры

Профессиональные и личные качества хороших программистов.

Глава 18. Защита исходного кода

Команды программистов совместно разрабатывают код, и без системы управления версиями это почти невозможно.

Глава 22. Рецепт программы

Методологии разработки: как команды взаимодействуют и совместно разрабатывают код.

План действий

Заполните анкету «План действий» и подумайте, как применить на практике то, что вы узнали.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 659.

Вопросы для размышления

1. Почему программы пишут командами? В чем преимущества относительно самостоятельной разработки?
2. Опишите симптомы хорошей и плохой групповой работы. Каковы необходимые условия хорошей работы и как проявляется плохая групповая работа?
3. Сравните групповую разработку программ и метафору строительства (см. раздел «Действительно ли мы *собираем* программы?» на стр. 240). Позволяет ли она лучше понять групповую работу?
4. Какие факторы – внешние или внутренние – больше всего угрожают эффективности команды разработчиков?
5. Как размер команды влияет на ее динамику?
6. Как защитить команду от проблем, создаваемых неопытными членами?

Вопросы личного характера

1. Что представляет собой команда, в которой вы работаете в данное время? Какому из описанных стереотипов она более соответствует?
   a. Создавалась ли она такой намеренно?
   b. Это здоровая команда?
   c. Требуются ли в ней перемены?

   Какие факторы, по вашему мнению, мешают хорошей групповой работе?

   Если вы еще не сделали этого, заполните «План действий». Разберитесь, как можно улучшить вашу команду, и начните перемены.
2. Вы умеете работать в команде? Что вы можете сделать, чтобы успешнее сотрудничать с коллегами и создавать лучшие программы?
3. Какие именно обязанности возлагаются на инженера-программиста в вашей команде?

Групповая работа План действий

Тщательно заполните форму. Отвечайте на вопросы честно.

Инфраструктура команды

Оцените использование в своей команде следующих инструментов. Поставьте + (если «да») и оценку от 1(*очень плохо*) до 5 (*очень хорошо*).

	У нас есть (да/нет)	Для этого есть администратор (да/нет)	Мы умеем пользоваться им (да/нет)	Применять легко (1–5)	Активно применяем (1–5)	Повышает эффективность работы * (1–5)
Система контроля версий						
База ошибок						
Автоматизация групповой работы						
Методология/процесс разработки						
План проекта						
Спецификации						

* Если какой-то пункт не дает вклада в эффективность, то почему?

Члены команды

Оцените каждое из следующих утверждений от 1 (*совершенно не согласен*), 3 (*безразлично*) до 5 (*полностью согласен*).

- ☐ У нас есть члены команды с большим диапазоном навыков
- ☐ Сменяемость кодировщиков низкая
- ☐ Обеспечивается необходимое обучение
- ☐ Все необходимые роли закрыты (см. раздел **«Рост команды»**) и
- ☐ Эти роли формально определены и признаны
- ☐ Все члены команды достаточно компетентны
- ☐ Мы не справились бы без любого члена команды
- ☐ Никто не перегружен работой
- ☐ В команде есть проблемы
  (1– большие проблемы, 5 – никаких проблем)
  – Какие проблемы?
  – Как их решить?

Структура команды и работа

Структура кода и структура команды

- ☐ Структура кода определяет структуру команды
- ☐ Структура команды определяет структуру кода
  (обратный счет: 1 – полностью согласен, 5 – не согласен)

Документация

- ☐ Документация доступна через систему контроля версий
- ☐ Мы записываем протоколы собраний и проектные решения

Рабочая практика

- ☐ У нас есть система наставничества
- ☐ Мы программируем парами
- ☐ Мы проводим инспекцию кода
- ☐ Мы проводим инспекцию документации
- ☐ У нас нет «культуры владения кодом»
- ☐ У нас есть руководство по кодированию

Управление

- ☐ Нами управляют хорошо
- ☐ Мои потребности ценят, как и успех команды

О плане

- ☐ Существует план разработки
  - ☐ Все знают, где он лежит
  - ☐ Он обновляется
  - ☐ Все знают очередной крайний срок
  - ☐ Крайние сроки реалистичны
- ☐ Мы знаем свои цели
- ☐ Мы знаем, как их достичь

Здоровье команды

- ☐ Команда мотивирована
- ☐ Команда растет (а надо?)
- ☐ Команда сокращается (а надо?)

Общение

- ☐ Общение в команде эффективно
- ☐ У нас хорошие совещания
- ☐ Я знаю, чем занимаются остальные
- ☐ Я знаю, кто отвечает за каждую область

Общая картина

Для каждой команды в организации оцените следующие утверждения от 1 (*совершенно не согласен*) до 5 (*полностью согласен*).

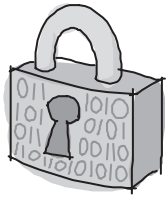
Добавьте к списку прочие команды, с которыми вы работаете

	Хорошая связь с	Хорошие отношения с	Эффективно работаем вместе	Знаем членов команды
Другие команды разработчиков				
Тестирование				
Маркетинг				
Менеджмент				
Заказчик				

Оценка

Теперь просмотрите данные вами ответы.

- Что говорят вам эти ответы о вашей команде?
- Какие оценки превалируют – высокие или низкие?
- Какими мерами можно поправить положение?



18

Защита исходного кода

Контроль над исходным кодом и контроль над собой

В этой главе:

- Охрана своего исходного кода
- Управление версиями исходного кода и управление конфигурацией
- Резервное копирование
- Лицензирование программного обеспечения

Совершенный человек, пребывая в покое, помнит об угрозе. Чувствуя себя защищенным, помнит о возможной катастрофе. Когда все спокойно, помнит о возможности беспорядков. Этот человек вне опасности, и его государства и все их кланы сохранятся.

Конфуций

Ни один ювелир, изготовив бриллиантовое ожерелье, не оставит его в незапертой мастерской, откуда его может украсть любой прохожий. Когда автомобильная компания выпускает на рынок новую модель, она не забывает о поддержке и обслуживании старых моделей. В обоих случаях это было бы профессиональным (и коммерческим) самоубийством, безрассудным отношением к результатам своего труда.

Код, который мы пишем, тоже стоит дорого: в него вложены наш труд и время, и он ценен не только в финансовом, но и в эмоциональном отношении. Мы должны охранять исход-

ный код, как любые другие ценности, и принять такую рабочую практику, чтобы не повредить, не подвергнуть опасности или не потерять его.

Код обладает ценностью. Относитесь к нему с уважением и заботой.

В этой главе мы возьмем на себя роль няни, телохранителя и сторожа, выработав важные методы защиты нашего кода. От кого или от чего мы его защищаем? С разной степенью драматичности мы боремся с такими опасностями, как:

- Мы сами и наши глупые ошибки
- Наши коллеги и их глупые ошибки
- Проблемы, присущие процессу коллективной разработки программного обеспечения
- Механические повреждения (взрывающиеся компьютеры и испаряющиеся жесткие диски)
- Воры, которые хотят воспользоваться программным обеспечением

Содержимое этой главы касается вашего здоровья, счастья и даже средств существования. Тот, кто зазевается, может поплатиться!

Наши обязанности

Будучи добросовестными ремесленниками, мы должны нести ответственность за свою работу. Мы должны не только писать код высокого качества, но и гарантировать для нашего продукта:

Надежную защиту

Недопустимо, чтобы код потерялся после трех месяцев работы над ним или чтобы секретная информация, которую он собой представляет, вышла за пределы компании.

Доступность

Те, кому полагается, могут легко модифицировать код. Его видят только они и никто из посторонних.

Воспроизводимость

Выпущенный код не должен быть потерян или уничтожен. Должна существовать возможность и через 10 лет собрать из него *точно* такое же приложение, даже если версии инструментов поменялись и язык программирования больше не поддерживается.

Сопровождаемость

Это означает не только использование хороших идиом программирования, но и возможность модификации кода всеми программистами команды. Могут ли несколько программистов одновременно работать над кодом или это приведет к катастрофе? Можно ли модифицировать старые продукты, одновременно разрабатывая новую версию?

Эти цели достигаются путем применения техники безопасной разработки. В данной главе мы не рассматриваем вопросы безопасности работающих приложений;[1] нас интересует технология разработки. Может показаться, что это скучные проблемы, мало связанные с процессом написания кода, но не следует недооценивать их важность. Мастерство в такой же мере относится к творческому процессу, в какой к его конечному результату.

Управление версиями исходного кода

Для того чтобы члены команды могли совместно разрабатывать код, они должны иметь возможность одновременно работать с базовым кодом. Это не так просто, как может показаться – нужно сделать так, чтобы не было конфликтов между одновременно осуществляемыми модификациями кода и никакие результаты работы не потерялись. Существуют низкотехнологичные способы совместной работы с кодом:

- Проще всего работать на одном компьютере и редактировать код по очереди. Двум программистам уместиться на одном стуле не так просто, поэтому конфликтов в коде в результате редактирования возникнуть не должно. Однако это существенная потеря производительности, поскольку в каждый данный момент только один человек может редактировать код.

  Можно поставить два стула и программировать в паре, рассчитывая на рост продуктивности (см. врезку «Все встали парами!» на стр. 409). Но если три, четыре или более программистов попытаются одновременно работать с одним и тем же кодом, этот способ не подойдет.

- В другом варианте можно организовать совместный доступ к коду через файловый сервер. Тогда разные разработчики смогут видеть исходные файлы и даже редактировать их параллельно. Однако это далеко не идеальное решение. Код находится в общем доступе, но он не защищен, потому что два человека могут работать над одним и тем же файлом одновременно. В результате, когда они оба нажмут кнопку Save, может произойти все, что угодно, в том числе потеря результатов работы. А что будет, если кто-нибудь отредактирует главный файл заголовков в тот момент, когда происходит сборка программы? Ответ: получится противоречивый исполняемый модуль, который станет аварийно завершаться или вести себя совершенно непредсказуемым образом.

По этой причине, когда из первичного цифрового бульона образовались команды программистов, они изобрели *систему контроля версий*, которая выступает в качестве центрального хранилища исходного кода, предоставляет доступ к нему и наводит порядок среди его одновременных модификаций. Но управление версиями необходимо, даже

[1] Об этом рассказывается в главе 12.

если вы работаете в одиночестве; как будет показано, централизованное хранилище кода – чрезвычайно полезная вещь.

Система контроля версий – важный инструмент разработки программного обеспечения. Она необходима для надежной работы команды.

Система контроля версий позволяет одному или более программистам работать с одним и тем же хранилищем исходного кода контролируемым образом, избегая проблем. Она позволяет каждому разработчику создавать (или получать, *check out*) личную копию общего хранилища кода и работать с ней изолированно. Эту копию называют также *песочницей*, поскольку сделанные локально изменения не могут выйти из нее и испортить работу другим программистам. Песочницу при необходимости можно обновить, внеся в нее изменения, сделанные другими пользователями, – обратившись к системе с просьбой о синхронизации. Когда модификация завершена, она *вносится* (*checked in*) в главное хранилище, и ее могут видеть другие разработчики.

Чтобы это осуществить, система контроля версий применяет одну из двух моделей доступа:

Жесткая блокировка

Ряд систем физически не разрешает пользователям редактировать один и тот же файл одновременно, используя механизм блокировки файлов. Изначально все файлы в песочнице имеют доступ только для чтения – их нельзя редактировать. Вы должны сообщить системе, что желаете отредактировать файл `foo.c`; он открывается для записи, и никто другой не сможет редактировать его, пока вы не запишете сделанные модификации или не вернете файл в неизменном виде.[1]

Оптимистическая блокировка

Более сложные системы разрешают пользователям редактировать одни и те же файлы одновременно. Стадия резервирования файлов отсутствует, и они всегда доступны для записи. Модификации объединяются при внесении их в систему. Объединение обычно осуществляется автоматически. Иногда возникают конфликты, и разработчик должен выполнить объединение вручную (обычно это нетрудно сделать). Такой способ называется *оптимистической блокировкой* (хотя, по сути, здесь нет блокировки вообще).

По поводу того, какой из подходов лучше, разгораются жаркие страсти. Одновременная модификация предпочтительнее, когда имеется широкая сеть разработчиков, взаимосвязанных через Интернет. Когда трудно управлять людьми, лучше не ставить большие препятствия; блокировка взятых для модификации файлов может сильно мешать работе.

---

[1] Поэтому считается дурной практикой блокировать файл на слишком длительное время – это может помешать работе других программистов. Это неотъемлемое ограничение такого метода доступа.

Рассказ бывалого человека

Плохое управление исходным кодом в сочетании с системой контроля версий может создать большие трудности в разработке. Здравые и простые с виду правила могут случайно удушить работу над продуктом.

В одном крупном проекте известной компании была принята политика жесткой блокировки – любая загрузка кода из хранилища для модификации становилась исключающей, не разрешая разработчикам одновременно модифицировать один и тот же файл. К несчастью, политика кодирования требовала, чтобы все перечисления `enum` помещались в один и тот же файл. Со временем этот файл разрастался все больше.

Нетрудно предсказать конечный результат: файл стал узким местом загрузки кода из хранилища. Разработчики постоянно болтались без дела в ожидании, когда освободится этот файл.

Контроль версий

Системы контроля за исходным кодом не просто хранят последнюю версию каждого файла. Хранилище записывает различие с предыдущей версией при каждом сохранении файла. Имея эту важную информацию, можно получить любую версию файла за всю историю разработки. Поэтому мы также пользуемся названием *системы контроля версий* (или *модификаций*). Это очень мощный инструмент: всякую модификацию можно целиком отменить – у вас в руках оказывается машина времени! Управление версиями в хранилище дает вам возможность:

- Отменять любые изменения, сделанные в любой момент.
- Следить за модификациями исходного кода, с которым вы работаете.
- Узнать, кто и когда изменял каждый файл (и даже выполнять сложные команды поиска, чтобы выяснить объем работы, проделанной над проектом отдельным разработчиком – полезно, когда разработка длится годами).
- Получить копию хранилища по состоянию на указанную дату.

Все хорошие системы управления исходным кодом позволяют создавать *текстовые метки* (или *теги*) и помечать ими конкретные версии группы файлов. С их помощью можно помечать важные состояния хранилища: пометить все файлы, которые составили конкретную версию продукта, и извлекать их в дальнейшем по этой метке. Это удобно, если вы работаете над версией 3, а важный клиент обнаруживает критическую ошибку в первой версии, и вам нужно немедленно получить ее код.

При каждом сохранении очередной версии в хранилище можно записывать в него метаданные – хотя бы текстовое описание совершенных модификаций. С помощью этих сообщений можно получить обзор разработки, просматривая журнал модификации файла. Более сложные инструменты позволяют добавлять к версиям файлов произвольные метаданные, например ссылки на сообщения об ошибках, вспомогательную документацию, данные тестирования и т. д.

Хорошие средства контроля за исходным кодом хранят версии не только файлов, но и каталогов. Это позволяет отслеживать модификации файловой структуры, включая создание, удаление, перемещение и переименование файлов. В одних системах модификации регистрируются отдельно для каждого файла; если вы загружаете сразу несколько файлов, для каждого из них ведется своя история версий. В других системах регистрируются *групповые модификации*: изменение нескольких файлов фиксируется как одна модификация. Это позволяет увидеть все файлы, измененные за один сеанс работы.

Мания контроля версий

Какие файлы нужно помещать в систему контроля исходного кода? Чтобы эффективно управлять вашим продуктом и его версиями, нужно разместить в одном хранилище *все* дерево его исходного кода, в том числе:

- Весь исходный код
- Среду для сборки
- Код для тестирования модулей и все средства для тестирования
- Любые другие файлы, необходимые для создания готового дистрибутива (графика, данные, настройки и пр.)

Цель состоит в том, чтобы все необходимое для сборки находилось в хранилище. Цепочки инструментов сборки и системы контроля исходного кода должно быть достаточно, чтобы сгенерировать готовый продукт, выполнив несколько простых действий (т. е. открыть хранилище и ввести команду make), не добавляя при этом никаких других файлов и ничего не редактируя. Если в дополнение к исходному дереву вам нужно что-то еще, значит, ваш продукт не находится под контролем версий.

Но за чем дело стало? Можно расширить этот список и сделать его окончательно полным:

- Поместить в систему контроля всю среду разработки. Заносите в нее все обновления инструментов сборки, и пусть они синхронизируются с каждым релизом вашего продукта.
- Присоедините сюда всю документацию: спецификации, сопроводительные записки к версиям, руководства и т. д.

Контроль доступа

Хранилище исходного кода может находиться на вашем компьютере или на другой машине, доступ к которой происходит по сети. Приняв надлежащие меры безопасности, можно организовать к нему доступ из любой точки света через Интернет, облегчив задачу координации разработчиков, находящихся в разных часовых поясах.

Средства контроля за исходным кодом также разграничивают права доступа к разным частям базового кода. Они позволяют установить правила видимости и права модификации. Обычно настройку системы контроля версий осуществляет *мастер сборки* (*билдмейстер*), который устанавливает эти права доступа и следит за порядком в хранилище. Если права администрирования дать всем разработчикам, это может привести к неприятностям, даже если они будут действовать с лучшими намерениями. Поэтому нужно всегда назначать специального администратора системы контроля версий.

Работа с хранилищем

Есть два способа разработки кода с применением системы контроля версий:

- При подходе *часто и понемногу* каждый файл загружается для проведения любой, пусть даже малой модификации. Поэтому в хранилище оказывается множество версий каждого файла. В результате легко проследить, какие изменения делались в ходе разработки, и показать все модификации файла за время его существования. Однако многочисленность версий файлов может создавать путаницу.
- Альтернативный подход (очевидно, *помногу и редко*) состоит в том, чтобы записывать в хранилище только важные модификации: версии, предназначенные для выпуска или содержащие отлаженные новые функции. В этом случае проще получить конкретную прежнюю версию кода, но значительно труднее проследить отдельные модификации, в результате которых она была получена.

Предпочтение следует отдать подходу «часто и понемногу». С помощью тегов можно пометить основные крупные версии, поэтому возможности другого подхода также оказываются доступными.

При сохранении модификаций кода нужно поддерживать дисциплину. Ваша работа сразу становится доступной другим разработчикам, поэтому нужно сначала тщательно протестировать код. Вас не похвалят, если по вашей вине намертво встанет вся команда. Во многих командах вводят наказание за такое антисоциальное поведение, чтобы впредь человек работал внимательнее. Никаких жестокостей – что-нибудь вроде общественного осмеяния в электронной почте или покупки пива на всех.

Внимательно относитесь к хранилищу. Следите, чтобы не записать в него неработающий код, который не позволит работать остальным разработчикам.

Пусть растут деревья

Одна из наиболее мощных возможностей, предоставляемых системами контроля за исходным кодом, – это *ветвление*: создание нескольких параллельных потоков разработки из файла или группы файлов. Ветвление находит ряд применений, в том числе:

- Одновременное добавление нескольких функций к базовому коду.
- Предоставление личного рабочего пространства разработчику для сохранения не вполне отлаженного кода, чтобы не нарушить работоспособность базового кода.
- Сопровождение старой версии продукта при одновременной работе над новой версией.

Допустим, что ваш продукт – графический редактор, и вы хотите добавить в него новые инструменты. Кроме того, вы хотите начать новый проект на базе тех же исходных файлов с целью перенести продукт в другую операционную систему. Обе задачи начинаются независимо, но в конце концов должны быть объединены. Это распространенная ситуация. Для каждой задачи вы создаете в хранилище *ветвь* и модифицируете код в ней, а не в основном разрабатываемом коде. Этим самым вы разделяете две задачи. Один разработчик создает новые инструменты, другой – занимается переносом. Один не мешает другому.

Как и следует из названия, ветви создают в хранилище древовидную структуру параллельных версий файлов. Всегда есть, по крайней мере, одна линия разработки кода, называемая *стволом* (по очевидным причинам). На рис. 18.1 показано, как на практике может происходить ветвление отдельного файла. Он был создан (*версия 1*) и первоначально разрабатывался в стволе (центральная колонка). В *версии 2* мы создаем ветку первой функции (для новых графических инструментов) и в ней производим сохранение ряда версий. Ни одна из них никак не влияет на ствол. Работа над стволом продолжается параллельно, и в *версии 3* создается вторая ветка для задачи портирования кода.

Работу, ведущуюся в ветке, можно *слить* с другой веткой или снова со стволом. Например, можно выполнить в ветке экспериментальную работу по исправлению ошибки, и если она окажется успешной, слить ее с основным кодом. Если разработка зашла в тупик, можно бросить ветку, не трогая при этом ствол. Очень удобно. В нашем примере первая ветвь на *версии 2.9* сливается с основной *версией 4*. В результате появляется основная *версия 5*. Позже вторая ветка также сливается с главной.

Даже если вам не нужно одновременно разрабатывать новые функции в своем базовом коде, ветвление можно с успехом применять в разработке одной линии. При такой системе основная версия остается стабильной: это всегда законченный отлаженный продукт – возможно, последняя выпущенная версия кода. Каждая функция разрабатывается в своей ветви функции или релиза, и сам продукт выпускается из этой ветки. Закончив разработку, мы сливаем ветку со стволом и снова

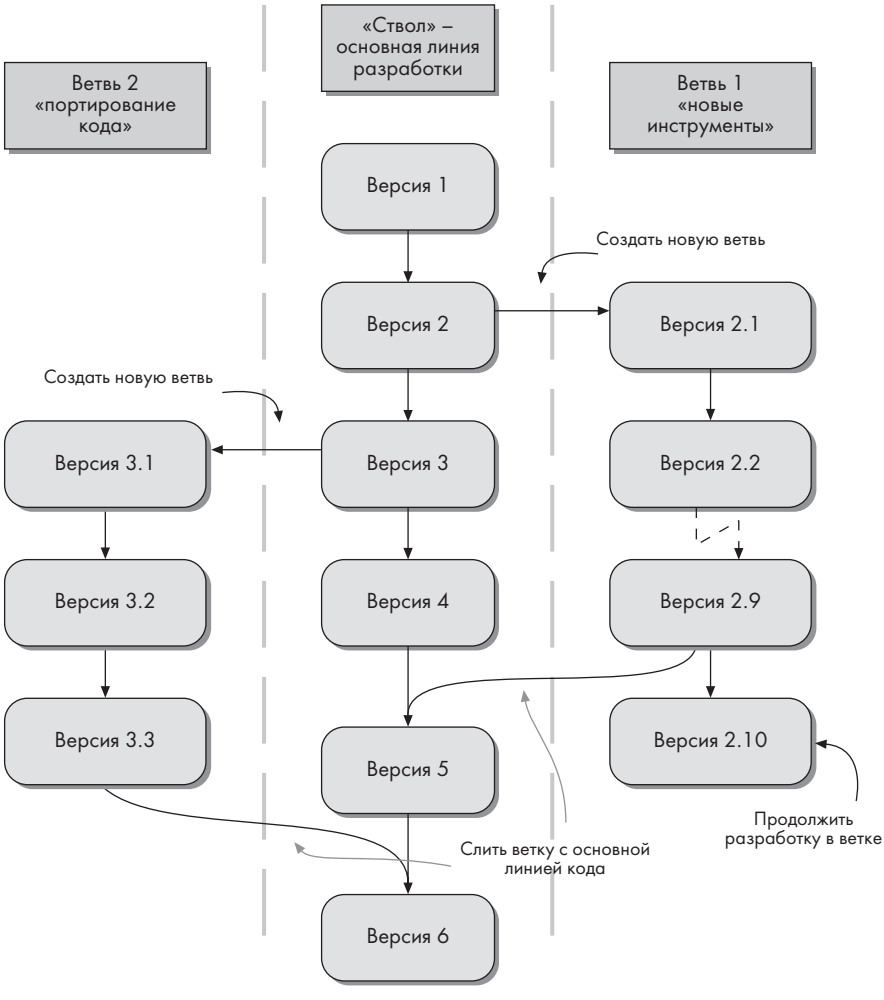


Рис. 18.1. *Ветвление проекта в системе контроля версий*

делаем ветку для новой функции. В итоге в основной линии нет неотлаженного кода, который мог бы нарушить ее целостность, а вся работа ведется в отдельной ветке, а не разбросана повсюду вместе с кучей других ведущихся разработок.

Краткая история систем контроля за исходным кодом

Существует много разных систем контроля за исходным кодом – как с открытыми, так и с коммерческими лицензиями. Часто выбор системы определяется существующей в компании традицией. («Мы всегда работали с... и хорошо ее знаем».) К сожалению, это не всегда означает, что выбран правильный или лучший инструмент. Во многих ком-

паниях пользуются устаревшими системами; затраты и сложности переноса больших объемов кода из одной системы в другую препятствуют переходу на новую систему.

Родоначальницей всех систем управления версиями является *SCCS* (*Source Code Control System*), разработанная в Bell Labs в 1972 году. Ее заменила *RCS* (*Revision Control System*). Самая распространенная система в мире программного обеспечения с открытым кодом – *CVS* (*Concurrent Versions System*), хотя она начала устаревать. CVS, основанная на базе RCS, дает возможность нескольким разработчикам работать над одним файлом одновременно. RCS реализует модель блокировки файлов (описанную в разделе «Жесткая блокировка» на стр. 450), тогда как CVS обеспечивает параллельную работу. Преемником CVS является система *Subversion*, в которой устранено большинство недостатков CVS.

Системы контроля за исходным кодом обладают тонкими функциональными различиями, но при этом большинство из них предоставляет оба интерфейса – командной строки и графический. Все они могут встраиваться в популярные IDE. Если вы не знаете, какой инструмент выбрать для своего проекта, попробуйте поработать с Subversion и одним из существующих для нее графических интерфейсов.

Управление конфигурацией

Управление конфигурацией программного обеспечения – тема, близкая к контролю за исходным кодом, но не следует одно путать с другим. Это отдельная большая тема.

Как мы видели, перед системой контроля за исходным кодом стоят следующие задачи:

- Обеспечить централизованное хранение кода
- Сохранить историю изменений, которым подверглись файлы
- Дать возможность разработчикам совместно трудиться, не создавая друг другу помех
- Дать возможность разработчикам решать задачи параллельно с последующим слиянием результатов

Основываясь на этом фундаменте, управление конфигурацией регулирует разработку программного продукта на протяжении всего времени существования проекта. Оно включает в себя контроль за исходным кодом плюс процедуру разработки. Формально управление конфигурацией ПО определяется как «Порядок установления конфигурации системы в дискретные моменты времени с целью систематического контроля за изменениями в этой конфигурации и сохранения целостности и возможности слежения за этой конфигурацией на протяжении всего времени существования системы» (Bersoff et al. 80). Оно контролирует *артефакты* проекта (то, что вы помещаете в систему контроля за исходным кодом) и его *процессы* разработки.

Некоторые системы контроля за исходным кодом предоставляют средства управления конфигурацией и могут интегрироваться с инструментами рабочего процесса проекта, например управлять отчетами об ошибках и запросами на модификацию, следить за ходом их развития и связывать с физическими изменениями в базовом коде.

Управление конфигурацией включает в себя:

- Определение всех отдельных программных компонент системы и артефактов, необходимых для их построения (это особенно полезно, когда один базовый код может быть сконфигурирован для генера-

Терминология и определения

Контроль за исходным кодом – наше главное оружие в борьбе за сохранность кода. Это важный инструмент, без которого профессиональный разработчик не может обойтись. Мы уже столкнулись с разными названиями этой системы. Они используются взаимозаменяемым образом, но каждое название вскрывает определенный аспект работы:

Контроль за исходным кодом (source control)

Это механизм *управления файлами с кодом*, который мы пишем. Он поддерживает файлы и структуру их каталогов, а также регулирует порядок одновременного доступа к коду и его модификации.

Управление версиями (version control)

Называемая также *контролем исправлений* (*revision control*) или модификаций (change control), это система контроля за исходным кодом, регистрирующая изменения, сделанные в файле. Она позволяет изучать, извлекать и сравнивать любые версии файла на протяжении всего времени работы с ним.

Обычно управление версиями наиболее эффективно работает с файлами в текстовом формате – в них легко отыскивать различия. Но можно хранить версии файлов и других типов: документы, графику и т. д. Исходные файлы этой книги хранятся в системе управления версиями, и я могу проследить историю их изменений.

Управление конфигурацией

Основано на управлении версиями и обеспечивает надежную среду для управления разработкой программного обеспечения и ведением необходимых процессов.

Распространены такие акронимы, как SCMS (система контроля за исходным кодом), VCS (система управления версиями) и RCS (система управления «ревизиями»).

ции нескольких вариантов продукта или версий для нескольких платформ).

- Управление релизами продукта и версиями компонент, участвующими в каждом релизе.
- Слежение за состоянием кода и его компонент и составление отчетов. В каком состоянии проект – *бета*-версия или *кандидат на выпуск*? (См. «Альфа, бета, гамма...» на стр. 197.)
- Управление формальными заявками на модификацию, слежение за их приоритетами и принятием для разработки; связь заявок с необходимой проектной работой, исследованиями, модификацией, тестированием и проверкой кода.
- Определение документации, относящейся к конкретным вариантам продукта, и среды для компиляции, необходимой в каждом случае.
- Проверка полноты и корректности программных компонент.

Как вы в данное время управляете конфигурацией своего базового кода?

Резервное копирование

Это обычный здравый смысл. Резервные копии – ваша страховка против случайного удаления файла, отказа вычислительной системы и, если копия хранится в другом месте, против потери данных в случае пожара в офисе. От простуды пока еще не лечат, но, возможно, какая-нибудь занимающаяся резервированием компания уже работает над этим.

Каждый знает, что необходимо регулярно делать резервные копии своих разработок. Но все мы люди; если мы знаем, как нужно разумно действовать, это еще не значит, что так мы и поступаем – есть более срочные (и интересные) дела. Задним умом все сильны: сидя среди дымящихся обломков компьютера без всякой надежды его починить и потеряв все данные, вы проклянете ту минуту, когда решили сыграть в солитер, вместо того чтобы сделать копию данных. Работу многих дней нужно повторять заново, и, даже если вы почти все помните, второй раз делать то же самое всегда менее интересно (и тяжело морально). Если к тому же близок срок сдачи работы, это чревато катастрофой.

Задумайтесь: есть ли резервная копия всего вашего кода? Меня ужас берет, когда я вижу, для какого огромного объема работы не выполнено резервное копирование. Люди подвергают себя бессмысленному риску.

Делайте резервное копирование своих данных. Разработайте стратегию восстановления, не дожидаясь, когда грянет катастрофа.

Вы должны разработать разумную процедуру резервного копирования. Не стоит вручную копировать файлы. В один прекрасный день вы забудете скопировать что-то важное, или допустите слишком большой

промежуток времени между копированиями, или скопируете не то, что нужно. Вспомните закон Мерфи (на стр. 31): *Если что-то может сломаться, оно сломается*. Правильным будет поместить все важные файлы в файловую систему, которая участвует в резервном копировании. Если я пользуюсь рабочей станцией, которая не участвует в резервном копировании, я сохраняю свой код на сетевом файл-сервере, для которого *делается* резервная копия, а не оставляю на ненадежном локальном диске.[1]

Резервное копирование приносит пользу, если оно:

- Осуществляется регулярно
- Контролируется и проверяется
- Данные легко восстанавливаются
- Осуществляется автоматически (автоматически запускается и выполняется без вмешательства оператора)

Важно, чтобы все хранилища исходного кода размещались на сервере, который подвергается резервному копированию. Иначе это все равно что положить ценности в сейф, но не закрыть его. На самом деле, запись модификаций в хранилище «часто и понемногу» уменьшает зависимость от резервного копирования персонального компьютера – большая часть проделанной работы есть в копируемом хранилище. Потеря файлов на вашей рабочей станции не будет иметь решающего значения для всего проекта.

Вывод следующий: ваш труд не защищен, если его нельзя восстановить в случае человеческой ошибки или отказа аппаратуры. Даже если это «всего лишь» код для личного использования, защитите его резервным копированием. Стоит немного потратиться на программу копирования, дополнительную внешнюю память и администрирование. Неприятности, связанные с потерей данных, обойдутся вам значительно дороже.

Выпуск исходного кода

Иногда требуется ослабить хватку, с которой вы вцепились в свой исходный код, и отпустить его в Большой мир. Например, вы можете продавать библиотеку, и вашим продуктом будет сам исходный код. Может быть, в условиях контракта оговорено, что вместе с исполняемым модулем вы должны предоставить исходный код. Даже если вы не собираетесь выпускать свой исходный код, он может быть продан новому владельцу или вы решите привлечь сторонних разработчиков

[1] Разумеется, за все приходится платить. Такой простой подход замедляет доступ к файлам, поскольку начинают играть роль задержки в сети и на файловом сервере. Но это неудобство (обычно малосущественное) можно пережить.

для реализации новой функции. Необходимо принять меры, чтобы обеспечить защиту и доступность кода и в таких ситуациях.

Какова степень связанной с этим опасности, зависит от характера вашего кода. Закрытый исходный код, написанный специально для внутреннего применения в продуктах компании, является тщательно охраняемой *интеллектуальной собственностью*, и обычно считается самоубийственным с коммерческой точки зрения выпускать его открыто, дав вашим конкурентам возможность воспользоваться им. Полной противоположностью является программное обеспечение с *бесплатным* или *свободно распространяемым* исходным кодом (*open source*), которое специально пишется так, чтобы каждый мог его посмотреть или модифицировать. В каждом случае характер выпускаемого продукта и варианты различны:

- Если вы выпускаете некий закрытый фирменный код, необходимо заключить с клиентом *соглашение о нераскрытии коммерческой тайны* (*NDA*). Это стандартный договор, который обязывает клиента не раскрывать код кому-либо и не использовать его иначе, чем это обусловлено в договоре. Договор имеет юридическую силу, и его главная цель – не давать покоя юристам компании, пока технические специалисты заняты главным делом – созданием замечательных программ.

  Если тот, кому вы предоставили код, собирается использовать его для получения прибыли, нужно заключить лицензионное соглашение, которое обеспечит вашу долю. На самом деле, это забота служб маркетинга или продаж, а простым программистам можно не беспокоиться по поводу этой грызни между компаниями.

- Разработчики открытого программного обеспечения должны выбрать тип лицензии, которая определит, что пользователь может делать с кодом и должен ли он раскрыть результаты своей работы, построенной на этом коде. Подробнее о лицензировании программного обеспечения сказано во врезке «Лицензии».

В любом случае нужно обеспечить презентабельный вид файлов исходного кода. Код *должен* быть полностью написан вами или у вас должны быть права распространения для тех частей, которые написаны кем-то другим. Это причина, по которой существует масса старого коммерческого кода, который нельзя сделать открытым: если компании не принадлежат все права на исходный код, его можно свободно распространять только после дорогостоящей модификации.

Чтобы обеспечить себе прочную юридическую основу, поместите в каждый файл исходного кода сведения об авторских правах, указав владельца (автора или компанию) и краткое описание лицензии, под которой он выпущен. Тогда тот, кому попадется этот код, будет знать, что это конфиденциальный материал. Подробнее об этом см. раздел «Комментарии в заголовке файла» на стр. 124.

Следите за тем, чтобы не сделать код доступным непреднамеренно: примите меры против реконструкции кода из выполняемого модуля, которая иногда оказывается возможной – особенно в случае компилируемых в байт-код языков типа Java и C#. Существуют инструменты для обработки байт-кода с целью воспрепятствовать его реконструкции.

Где я оставлю свой код...

Наконец, подумайте о месте для хранения своего исходного кода. Совершенно секретную информацию компании не следует оставлять в ноутбуке, лежащем в незапертой машине. Точно так же не стоит держать исходный код в общедоступной сети.

Храните в тайне свои пароли регистрации. У посторонних (или злонамеренных коллег) не должно быть возможности причинить вам ущерб, воспользовавшись не принадлежащими им правами доступа.

Лицензии

Лицензия на программное обеспечение определяет, какие права в отношении ПО есть у пользователя. Она касается как программ, распространяемых в двоичном виде, так и исходного кода, из которого они созданы. Большинство коммерческих лицензий запрещает копирование, модификацию, передачу, сдачу в аренду и запуск на нескольких машинах. Напротив, лицензии open source защищают ваше право копировать и распространять программное обеспечение по своему желанию.

Авторы программ выбирают лицензии исходя из своих задач и идеологии. На самом деле, автор может выпустить программный продукт под несколькими лицензиями, относящимися к разным схемам его применения и предполагающими разные цены и степень поддержки. Для исходного кода существует много типов лицензий, хотя лишь некоторые из них используются широко. Они различаются в следующих аспектах:

Допустимые способы применения

Можно ли использовать лицензированный код коммерческим образом или его можно применять только в бесплатно распространяемых программах? Проблема не столько в деньгах, сколько в праве включать вашу разработку в закрытый коммерческий продукт без вашего разрешения. Некоторые лицензии open source требуют, чтобы пользователь делал открытым любой код, построенный с помощью лицензированного продукта. Коммерческие лицензии обычно допускают любое применение, если вы его оплачиваете.

Условия модификации

Если вы сделали изменения в коде, должны ли вы их опубликовать? Или можно поставлять производные продукты без каких-либо обязательств? Некоторые лицензии open source называют «вирусными», потому что любые модификации должны выпускаться под той же лицензией open source, равно как любой поставляемый вами код, использующий этот продукт.

Коммерческие лицензии пишут юристы компаний, преследуя низкие цели – защитить коммерческие интересы этих компаний. Однако есть много бесплатных и open source лицензий. «Open source» (открытый исходный код) – это термин, придуманный в *Open Source Initiative* (*OSI*), организации, сертифицирующей программные лицензии. Открытости исходного кода недостаточно для квалификации продукта как open source. Для этого должны быть также предоставлены определенные права: свободно модифицировать и распространять далее сам код или любые его модификации, с тем ограничением, что такое право предоставляется всем и является безотзывным.

Open source вступает в противоречие с *бесплатным* программным обеспечением, как его понимает Free Software Foundation. FSF (который управляет проектом GNU) делает бóльший акцент на идеологии и рекламирует лицензии на программное обеспечение, которое *свободно*, как *свобода речи*, а не просто *бесплатно*, как *бесплатное пиво*, т. е. слово *free* используется в смысле французского слова *libre*. OSI допускает некоторые «бесплатные» лицензии, которые не внушают любви преданным сторонникам GNU. Знаменитые лицензии GNU – это Стандартная общественная лицензия (GNU *General Public License*, *GPL*) и Стандартная общественная лицензия ограниченного применения (GNU *Lesser General Public License*, *LGPL*). Последняя – ослабленная «библиотечная» лицензия, допускающая компоновку с закрытым кодом.

Резюме

Мы должны с уважением относиться к прошлому и с недоверием – к настоящему, если хотим обеспечить себе безопасность в будущем.

Жозеф Жубер

Важен не размер вашего кода, а то, что вы с ним делаете.

В этой главе мы рассмотрели различные действенные методы, гарантирующие, что мы берем на себя ответственность за создаваемый код

и разрабатываем его надежным и контролируемым образом. Это действительно важные вещи; несчастье, если оно случится в опасный момент, может обернуться катастрофой для вашего проекта. Важный базовый код необходимо защитить.

Система контроля за исходным кодом – важное оружие в борьбе за безопасную разработку кода. Она облегчает взаимодействие в команде, обеспечивает предсказуемость и надежность групповой разработки и ведет архив всей деятельности по разработке. Это средство, обеспечивающее безопасность разработки, и без него ваша жизнь была бы существенно хуже.

Хорошие программисты...

- Берут на себя ответственность за свою работу и умеют принимать меры для защиты разрабатываемого кода
- Осторожно пользуются системой контроля за исходным кодом, так что хранилище всегда находится в корректном и работоспособном состоянии
- Не заносят в хранилище неработающий код
- Внимательно применяют все инструменты, стремясь создавать понятный и доступный для сопровождения код

Плохие программисты...

- Ждут, когда разразится катастрофа, и лишь потом начинают думать о защите кода и доступе к нему
- Считают, что о защите и резервном копировании их данных позаботится кто-то другой
- Не обновляют документацию
- Не думают о том, в каком состоянии находится их код в хранилище – записывают неработающий код и оставляют за собой беспорядок, который должны убирать другие

См. также

Глава 7. Инструментарий программиста

Инструменты, способствующие эффективной разработке кода.

Глава 10. Код, который построил Джек

Доступность вашего кода оказывает влияние на легкость сборки – как новейшего варианта базового кода, так и старых версий, нуждающихся в переработке.

Глава 12. Комплекс незащищенности

Еще одна проблема безопасности – защищенность выполняющихся программ в отличие от защищенности процесса разработки.

Контрольные вопросы

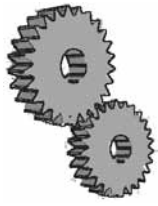
Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 665.

Вопросы для размышления

1. Как надежно передать свой исходный код другим людям?
2. Какая из двух моделей редактирования файлов в хранилище (блокировка загруженных файлов или их параллельная модификация) лучше?
3. Как различаются требования к системе управления версиями для двух команд разработчиков – территориально распределенной и компактно размещенной?
4. Чем нужно руководствоваться при выборе системы управления исходным кодом?
5. Как при групповой разработке разделить новейший код, над которым идет активная работа, и стабильную версию?

Вопросы личного характера

1. Приносит ли вашей команде пользу система контроля за исходным кодом?
2. Есть ли у вас резервная копия вашей текущей работы? Считается ли в вашей команде важным выполнять резервное копирование? Когда создаются резервные копии?
3. На каких машинах хранится ваш исходный код?



V

Часть процесса

Создание программного обеспечения высокого качества не ограничивается написанием хорошего кода. Очевидно, хороший код приносит пользу. Некоторую. Но этого мало. Хорошее программное обеспечение делается осмысленным образом; необходимы проектирование, предвидение и надежный план боя. Что собой представляет план боя, мы и узнаем в этой части. Однако прежде чем собирать войска, нужно подумать о том, чем они будут заниматься. Неплохо указать им всем одно и то же направление.

В этом разделе рассматриваются некоторые специфические части процесса разработки, дополнительные мероприятия, для которых нужно выделить время и которые помогут нам целенаправленно создавать отличный код. Будут рассмотрены следующие темы:

Глава 19. Спецификации

Как писать и читать спецификации для программного обеспечения. Правильный подход к регистрации того, чем вы собираетесь заниматься и уже сделанного. В этой главе будет показано, как сделать, чтобы спецификации облегчали вашу жизнь, а не действовали на нервы.

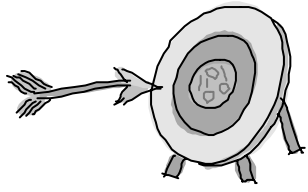
Глава 20. Рецензия на отстрел

Рассказ о проверке (рецензировании) кода – важной практике, гарантирующей написание высококачественного кода.

Глава 21. Какой длины веревочка?

Оценка сроков разработки программного обеспечения – важный элемент процесса планирования, который в программистском сообществе продолжает считаться черной магией. В этой главе будут опровергнуты некоторые мифы, связанные с оценкой сроков, и даны практические рекомендации по этому вопросу.

Безжалостное давление, оказываемое на нас промышленным производством программ, заставляет работать все быстрее и напряженнее. Единственный способ выжить – научиться работать с умом. Мы должны применять все эти приемы, и только тогда у нас появляется шанс на победу.



19

Спецификации

Составление спецификаций программного обеспечения

В этой главе:

- Зачем нужны спецификации
- Типы спецификаций
- Что в них содержится
- Почему их игнорируют?

Не было у меня таких тревог, которые не улеглись бы после чтения в течение часа.

Шарль Монтескье

Практически все, чем вы пользуетесь, имеет документацию. У вашего DVD-плеера есть инструкция по эксплуатации. У вашей машины есть руководство по обслуживанию. В контракте есть текст, набранный мелким шрифтом. У шоколадного торта есть рецепт. Есть книги и журналы, посвященные практически любым задачам, которые могут возникнуть. Если ваше программное обеспечение представляет собой какую-то ценность, у него тоже должна быть хорошая документация.[1]

Мы все знаем, что тщательно протестированное программное обеспечение, которое

[1] Разумеется, это не оправдание для плохого интерфейса; он все равно должен быть простым и интуитивно понятным.

мы передаем своим клиентам, должно сопровождаться документацией. Каким должен быть объем этой документации, вопрос спорный. Пользователь комплекта офисных программ, несомненно, считает, что поставляемой с ним документации недостаточно. В отсутствие руководства, описывающего механизм применения вашей программы в каком бы то ни было виде, возникает ошибочное предположение, что программа может делать больше, чем в действительности в нее заложено, или ее применяют для целей, которые не могли бы придти в голову ни одному здравомыслящему программисту.

Разработчики тоже могут легко допустить подобного рода ошибки во время кодирования. Документация необходима как для конечного программного продукта, так и для промежуточных стадий разработки. Такого рода документацию конечный пользователь (обычно) не видит. Это описание конструкции и изготовления программы. В этом и состоят *спецификации* программного продукта.

Профессиональному программисту необходимо умение составлять спецификации и работать с ними. Способность общаться на английском (или каком-то другом естественном языке) так же важна, как способность общаться с помощью кода.[1] Так же как употребление в пищу овощей и регулярные физические упражнения, спецификации «полезны для вас» и полезны для ваших программ. Тем не менее, как и в случае капусты и гимнастического зала, мы ими пренебрегаем, а потом горько жалеем о последствиях, когда получаем нездоровое, дряблое программное обеспечение.

Принято представлять себе спецификацию программы как громадную пачку бумаги с текстом, напечатанным мелким шрифтом, загадочными таблицами и бессмысленной терминологией. Перспектива отнюдь не вдохновляет: документ, работа с которым отнимет больше сил, чем код, который в нем описан. Разработчики живут в постоянном страхе, что их заставят работать со спецификациями.

Но совсем необязательно, чтобы все было именно так. При правильном применении спецификации облегчают процесс разработки. Они снижают риски разработки, повышают производительность и значительно облегчают жизнь. В этой главе мы посмотрим, какие виды спецификаций необходимы, что они должны содержать и почему практика так далеко расходится с идеалом.

---

[1] В самом деле, Дейкстра заметил однажды: «Помимо склонности к математике, очень важным качеством квалифицированного программиста является отличное владение родным языком».

Что же это такое, конкретно?

Приложи сердце твое к учению и уши твои – к умным словам.

Притчи 23:12

Спецификации – это официальные документы, участвующие в процессе разработки и предоставляющие внутреннюю документацию для программного обеспечения. Есть много видов спецификаций (мы их вскоре рассмотрим) для разного рода информации и разных аудиторий. Каждая спецификация относится к конкретной стадии процесса создания программного обеспечения – от замысла проекта до итогового поставляемого продукта. Мы пользуемся ими, чтобы точно зарегистрировать требования пользователя (или что он надеется получить, если это разные – что, как правило, случается – вещи), подробно описать архитектуру программного решения, интерфейс конкретного модуля кода, решения о конструкции и реализации части кода и т. д.

Спецификации помогают работать более осмысленно и создавать программы лучшего качества. Однако плохие спецификации приводят к противоположным результатам. Качество спецификаций, так же как и качество кода, имеет первостепенное значение. Хорошие спецификации и документация обычно воспринимаются как должное, тогда как плохие спецификации все начинают ненавидеть, воспринимая как жернов на шее проекта.

Процесс разработки программного обеспечения нуждается не просто в наличии спецификаций, но в их высоком качестве.

Спецификации представляют собой форму внутренних и внешних коммуникаций команды. Мы уже видели, что проекты гибнут при недостаточном общении. Поэтому мы должны использовать спецификации как средство связи – там, где это необходимо. (Проекты могут терпеть неудачу и потому, что слишком много времени тратится на составление документов, а на написание кода его не хватает!)

Значение спецификаций растет пропорционально размерам проекта. Это не значит, что в малых проектах они не столь важны, но в больших проектах и потери могут быть большими – в них занято много людей, и недостатки связи и координации оказывают большее отрицательное воздействие на результат процесса разработки программного продукта.

Спецификации – важный механизм связи между разработчиками программного обеспечения. Они служат для фиксирования сведений, которые не должны быть потеряны или забыты.

Составление спецификаций делает вашу информацию:

Лучше защищенной

Информация перестает храниться только в голове, где она может быть забыта или искажена. Когда все важные данные зафиксированы, уход людей из проекта представляет меньшую опасность для него: объем потерянной информации минимален, а пришедший на замену программист быстрее включается в работу.

Доскональные и полные спецификации снижают риск того, что разные люди будут исходить из разных предположений – классическая причина, по которой два отдельно разработанных модуля отказываются взаимодействовать в начале интеграции. Спецификации помогают избежать скрытых ошибок.

Доступной

Вся информация удобно хранится в известном месте. Новые люди, подключающиеся к проекту, могут разобраться в том, чем занимается каждая компонента и как они связаны между собой, просто прочтя документацию. Чтобы начать работать, им не нужно выспрашивать эту информацию у сотен разных людей.

Более точной

Когда вся информация собрана и зафиксирована, легче заметить проблемы, выявить упущенные в проекте части и обнаружить неудачные последствия или побочные эффекты. Проверить несколько отрывочных мыслей, приходящих в голову, бывает труднее.

Типы спецификаций

Спецификация каждого типа образует для программного продукта промежуточный шлюз – метод передачи данных между различными частями процесса разработки. Например, спецификацию API программной компоненты пишет группа людей, которые рассматривают ее функциональность и интерфейс. Программист действует согласно этой спецификации; она достаточно полная, чтобы реализовать весь код. Та же спецификация является контрактом, описывающим подробности того, как системный интегратор может включить ее в систему и как другие программисты могут ею пользоваться. Она описывает также предположительное поведение компоненты, так что тестеры могут проверить правильность ее работы.

Благодаря этому выход одной спецификации естественно перетекает в содержимое следующей, оставляя документальный след в фарватере быстро развивающегося программного продукта. Пример такого бумажного следа приведен на рис. 19.1. Можно видеть, как по мере продвижения проекта образуется естественная иерархия документов – у каждой подкомпоненты есть набор документов, аналогичный существующему для проекта в целом; ее разработка может рассматриваться как мини-проект.

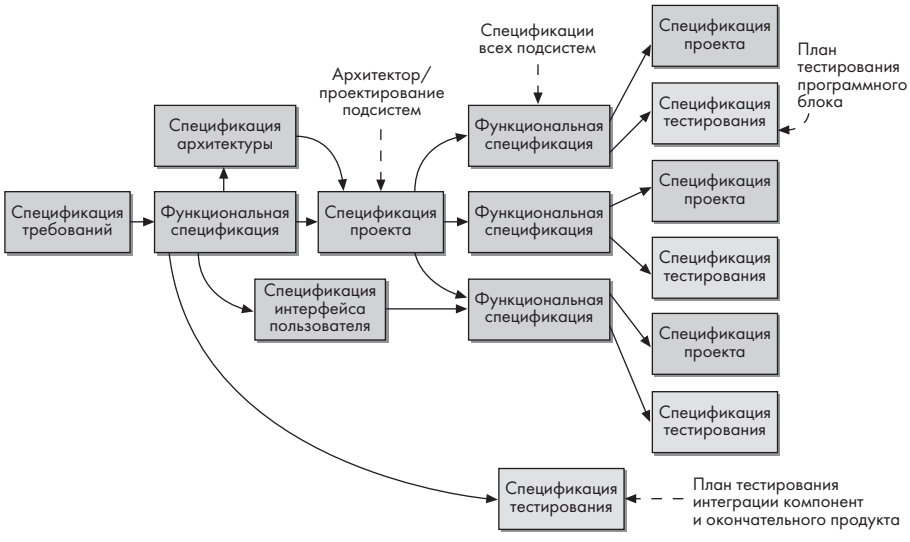


Рис. 19.1. *Типичное документальное свидетельство спецификаций*

Так как разработка программного обеспечения является итеративным процессом, этот поток информации не направлен в одну сторону (иначе вы попадаете в узкие рамки каскадной методологии – см. «Каскадная модель» на стр. 539). Если вы обнаруживаете, что не хватает какой-то информации, или нужно скорректировать конструкцию программы, в спецификацию должны быть внесены соответствующие изменения. Если изменение и сопровождение документации затруднены, это плохо отражается на разработке. Бюрократические процедуры стремятся удушить разработку хороших программ требованием, чтобы вся работа проводилась согласно Спецификации, даже если ей 10 лет и она полностью устарела. Хорошие программисты считают свои спецификации таким же податливым материалом, как их код.

Рассмотрим разные виды спецификаций программного обеспечения и как они могут помочь совершенствовать стиль кодирования. К сожалению, на практике эти документы известны под многими разными названиями. *Спецификацию требований* могут также назвать *спецификацией пользовательских требований* или *спецификацией функциональных ограничений*.

Спецификация требований

Если бы все остальные спецификации катастрофически исчезли в процессе разработки, этот документ следовало бы спасать в первую очередь. Он идет во главе парада проектов-победителей и является камнем преткновения многих неудачных проектов. В нем содержится жизненно важная информация. От него зависит ваше благополучие.

Требования к проекту всегда бывают вначале неясны; клиенты не могут *точно* объяснить, чего они хотят от своего программного обеспечения (они же не эксперты в компьютерах). Это вызывает разного рода проблемы, поэтому необходимо иметь один документ – *спецификацию требований*, в котором устанавливается, что должна делать программа и каковы приемлемые характеристики реализации. В ней очень подробно (или достаточно подробно, что уже хорошо) описывается, как должен вести себя код. Она должна охватывать понятным и недвусмысленным образом все важные, связанные с риском и ценные области поведения системы.

Обычно требования записываются в виде ряда пронумерованных предложений, каждое из которых содержит один реальный факт. Например:

> *1.3.5. Интерфейс пользователя должен представлять собой черный прямоугольник с текстом «Без паники», показанным красным шрифтом sans-serif кеглем 13 пунктов.*

Однозначная нумерация каждого требования позволяет легко ссылаться на него в будущем и помогает проследить связь конкретной конструкции или реализации с определенным требованием.

Мы должны рассмотреть:

Функциональные требования

Эти требования детализируют, что должна делать программа. Например: *Должна обрабатывать изображения в формате BMP и преобразовывать их в JPEG или GIF.*

Требования к производительности

Эти требования показывают, насколько быстрой должна быть программа и есть ли операции, для которых установлена предельная продолжительность. Например: *Пользователь должен получать результат каждой операции не позднее, чем через одну секунду, а все операции должны завершиться не позднее, чем через пять секунд.*

Требования к совместимости

Описывают прочие программные, аппаратные и внешние системы, с которыми должен взаимодействовать программный продукт. Например: *Должен поддерживать связь по HTTP и RS232 с сервером обновления.*

Требования к будущим функциям

Эти требования определяют, для какой функциональности должно быть подготовлено место, даже если она будет не сразу реализована. Например: *Должна быть предусмотрена возможность замены скинов интерфейса, чтобы пользователь мог выбрать более подходящий для него внешний вид.*

Эти требования делятся на две части. *Дискретные требования* носят двоичный характер. Взглянув на исходный код, легко проверить, удо-

влетворяет ли им программа: для каждой функции должен быть участок кода, который ее реализует. Можно написать конкретные тесты для проверки выполнения каждого дискретного требования.

Недискретные требования труднее уловимы. Рассматривая исходный код, вы не определите, удовлетворяет ли им программа. К числу таких требований относятся отказоустойчивость системы, время наработки на отказ сервера, среднее время между отказами, защищенность или масштабируемость. Такого рода требования могут быть очень важными и очень трудно проверяемыми.

Процедура создания спецификации требований в каждой компании своя и часто зависит от особенностей проекта и заказчика (насколько он умен и компетентен). Спецификация требований проверяется маркетинговым подразделением, фокусной группой будущего продукта или *бизнес-аналитиком*, задача которого разобраться в предметной области и оценить необходимую работу. Обычно в этом принимает участие заказчик или его представитель.

Заказчик должен одобрить и подписать спецификацию требований; она, в сущности, представляет собой контракт между разработчиком программного обеспечения и его клиентом. Изготовитель обязуется поставить продукт, функциональность которого отвечает спецификации, а клиент согласен оплатить его. Без согласованной спецификации клиент может отказаться от продукта по своей прихоти, и разработчики потратят впустую массу труда. Увы, это обычная проблема производства программ, с которой я неоднократно встречался, особенно часто возникающая в тех случаях, когда заказчик не является техническим специалистом и не представляет себе, каким должно быть хорошее программное решение. Когда требуемая программа наконец-то сделана, заказчик осознает, что он просил сделать не то, что ему в действительности нужно: *пишите все сначала*. Вы вернулись на первую клетку. Такие вещи случаются постоянно, поэтому спецификация требований – ваш страховой полис.

К сожалению, многие софтверные компании не занимаются сбором требований или уделяют ему недостаточно внимания. Важно согласовать требования заранее, до начала проектирования и уж во всяком случае до написания какого бы то ни было кода. Функциональные спецификации требований используются со следующими целями:

- Чтобы двигать проект в правильном направлении и с соблюдением сроков – путем исключения (или хотя бы ограничения) позднейшего добавления новых функций, которое может задержать поставку.
- Чтобы добиться большей удовлетворенности заказчика – путем определения того, на что он может рассчитывать.
- Чтобы уменьшить количество ошибок – ограничивая *ползучий функционализм*, мы препятствуем изменениям в коде, осуществляемым в последнюю минуту и приводящим к жутким ошибкам.

- Чтобы сохранить свое здоровье – когда отсутствуют спецификации требований, у разработчиков быстро уменьшается количество волос на голове.

В зависимости от принятой методологии либо еще до начала всякой разработки пишется единая цельная спецификация требований, либо она развивается параллельно с написанием кода. Разберитесь с тем, каким образом вы получаете требования от заказчика и как это влияет на разработку вами кода.

Требования к программному обеспечению нужно установить заранее, чтобы не возникало неоправдавшихся ожиданий, чтобы препятствовать расширению функциональности и снизить беспокойство разработчиков.

Следует также рассмотреть *требования разработчика*: что вам нужно иметь для успешной работы. Например, вам может понадобиться определенная внутренняя архитектура, чтобы обеспечить возможности расширения в будущем, а также система управления версиями (здесь двух мнений быть не может). Включение таких требований в спецификацию оправданно.

Функциональная спецификация

В *функциональной документации* – документе, вероятно, чаще всего используемом программистами, описывается внешнее поведение программного объекта. Эта спецификация создается на основе спецификации требований и должна ей удовлетворять. Обычно в одном проекте есть несколько функциональных спецификаций: одна для продукта в целом и отдельные спецификации для программных компонент.

Функциональная спецификация для программной компоненты содержит полное и однозначное описание ее открытого интерфейса. Оно сводится к перечислению всех методов или функций в API модуля и описанию того, что они делают и как ими пользоваться. Кроме того, описываются детали всех внешних структур и форматов данных, все зависимости от других компонент, рабочих пакетов или спецификаций.

Это не просто руководство пользователя по части программного обеспечения. Приводимых подробностей достаточно для создания компоненты по данной спецификации. Две команды могут прочесть этот документ и самостоятельно работать над реализацией. Несмотря на различие полученных реализаций, обе компоненты должны вести себя одинаково.

Это обстоятельство применяется на практике: на некоторых космических аппаратах NASA вместо одного компьютера работает пять; четыре компьютера реализуют спецификацию для некоторого расчета, выполняя независимо разработанные программы. Пятый компьютер усредняет результаты четырех расчетов (или приходит к заключению, что один из четырех дает результаты, совершенно не согласующиеся с остальными).

Если вы создаете программную компоненту без функциональной спецификации, начните писать ее сами. Покажите ее тем, кого она касается, чтобы заручиться их согласием в том, что написанного вами будет достаточно и они не будут разочарованы, когда вы закончите ее разработку.

Если ваша задача программирования недостаточно четко поставлена, не начинайте кодирование, не написав спецификации и не согласовав ее.

Спецификация системной архитектуры

Спецификация архитектуры описывает общий вид и структуру программного решения. В нее входят такие вещи, как:

- Физическое расположение компьютеров. (Будет ли это система клиент/сервер или однопользовательское настольное приложение?)
- Компонентный состав программного обеспечения. (Как оно разделяется? Какие части нужно написать, а какие можно купить?)
- Параллелизм. (Сколько потоков выполняется одновременно?)
- Хранение данных (включая структуру базы данных).
- Другие аспекты системной архитектуры (избыточность, каналы связи и пр.)

Необходимо подробно описать эти вещи до начала активной разработки. Архитектура влияет на последующие стадии разработки; ошибки или неясности в ней приводят к серьезным просчетам в дальнейшем. Конечно, ничто не устанавливается на века: если вы обнаружите дефект в спецификации архитектуры, его нужно исправить, какой бы объем работы ни был уже выполнен. Не нужно относиться к плохой спецификации архитектуры как камню на своей шее. Тем не менее необходимо заранее осуществлять адекватное проектирование архитектуры. Архитектура программного обеспечения подробно обсуждается в главе 14.

Спецификация интерфейса пользователя

В этом документе содержатся данные об интерфейсе пользователя: как он должен выглядеть и какие действия инициирует. Это способ представления пользователю возможностей системы. В спецификации может быть описан графический интерфейс, интерфейс, основанный на веб, система голосового телефонного меню, интерфейс доступа по азбуке Брейля, простой светодиодный экран.

Иногда то, что представляется пользователю, сильно отличается от реализации, скрытой за сверкающим фасадом. Приведем два примера:

- Система, активно использующая сетевые соединения, может быть развернута на одной машине и скрываться за единым интерфейсом пользователя.

- Имеющаяся функциональность может быть упрощена с целью облегчить использование или создать урезанную, более дешевую версию.

В спецификации пользовательского интерфейса описываются принятые в нем соглашения и метафоры и показывается, каким представляется пользователю взаимодействие функций. Она состоит из текстового описания с рисунками и снимками экрана. Часто в спецификации бывает показан сценарий интерфейса в работе – картинки состояний интерфейса с переходами между ними и что в каждом из них отображается. Приводятся все экраны, которые сможет увидеть пользователь, со всеми деталями (графикой, полями, списками, кнопками и их размещением на экране). Кроме того, фиксируется допустимое время реакции системы на каждое действие и поведение в случае возникновения ошибок (описание не будет полным, т. к. перечислять *все* возможные состояния ошибки можно практически бесконечно!)

Спецификация может содержать *прототип интерфейса* или быть основой для его создания. Прототипы делаются с разной степенью детализации и точности; все зависит от приложения и объема последующих тестирования и проверки. Проект интерфейса пользователя на данной стадии неизбежно оказывается неполным, но это первая возможность узнать, как будет выглядеть конечный продукт. Хотя прототипы помогают предвидеть поведение интерфейса, должным образом пересмотреть и поправить его можно только после интеграции системы.

Проектная спецификация

Проектная (или *техническая*) *спецификация* содержит описание внутреннего устройства компоненты. Она описывает, какой должна быть – или уже есть – функциональная спецификация. В проектной спецификации описываются все внутренние API, структуры данных и форматы. В ней должны быть описаны все важнейшие алгоритмы, пути выполнения и взаимодействие между потоками. Указывается, какой язык программирования будет использован и какие инструменты будут применяться для сборки кода. Все это важная информация для реализации и сопровождения кода.

Во многих тяжеловесных технологиях разработки требуется, чтобы до начала реализации была создана проектная спецификация; ее проверяют до начала кодирования, чтобы не завести работу в тупик. Однако в большинстве организаций этот документ пишут параллельно с кодом или позже.

Идея, казалось бы, хорошая, но большинство проектных спецификаций – лишь большие траты времени! Их нужно постоянно корректировать, чтобы они соответствовали коду, который описывают. Если не прилагать усилий, они быстро устаревают и становятся неточными или неполными – потенциальные ловушки для доверчивых читателей. Поэтому я предлагаю вам *не писать проектной документации!*

Но, прежде чем вы с облегчением вздохнете, послушайте следующий совет. Замените ее документом с той же информацией, но в котором легче поддерживать точность. *Средства грамотного программирования* (см. раздел «Практические методологии самодокументирования» на стр. 103) служат замечательным механизмом составления документации, которая генерируется из самого кода – нужно лишь добавить в него комментарии в особом формате. Такая документация может заменить обременительные проектные спецификации.

Пользуйтесь инструментами грамотного программирования для составления технической документации. Не пишите в текстовом редакторе документ, который быстро устаревает.

Необязательно располагать готовым кодом, чтобы применять инструменты грамотного программирования. Можно точно так же документировать будущую структуру кода. В результате вы сможете автоматически сгенерировать проектную документацию, создать прототип, служащий доказательством правильности концепции, и при желании развить его в готовый код.

Спецификации тестирования

Спецификация тестирования описывает стратегию тестирования определенного программного объекта. Она объясняет, как проверить соответствие реализации функциональным спецификациям, чтобы определить готовность программы к выпуску. Естественно, объем этой задачи зависит от того, что проверяется – отдельная компонента, подсистема, настольное приложение или встроенный продукт.

В спецификации тестирования перечисляются все тесты, которые нужно провести. Для каждого теста детально описывается *тестовый сценарий*: набор простых действий для запуска теста, критерии прохождения и среда выполнения. Сценарии могут представлять собой отдельные документы или быть включены в данную спецификацию.

Как мы видели в главе 8, многие тесты уровня кода могут сами выполняться *в коде* и запускаться автоматически в процессе разработки. Эти тесты стоят в стороне от тестов верхнего уровня, которые можно проводить только путем запуска программы в окончательном контексте, моделируя ввод данных оператором.

Если можно создать программные поблочные тесты для вашего продукта, это предпочтительнее, чем писать пространные тестовые спецификации. Тестовые спецификации на уровне кода в процессе развития системы быстро устаревают точно так же, как и проектные спецификации. Пользуйтесь кодом программного теста как документацией вашей стратегии тестирования – писать «литературный» тестовый код так же просто, как обычный «литературный» код. Автоматизированные циклы тестирования заставят вас обновлять тесты по мере изменения кода, иначе они просто не пройдут успешно!

Адвокат дьявола

Спецификации обходятся дорого: чтобы читать их или писать, нужны время и труд. Они требуют лишних усилий. Так ли уж *необходимы* все эти документы? Да, необходимы: чтобы писать программы высокого качества, нужно собрать всю эту информацию и разместить ее в таком месте, откуда ее при необходимости можно извлечь. Спецификации побуждают нас следовать правильной практике разработки: собирать требования, проектировать и составлять план тестирования – и мы видели, как они способствуют общению.

Ускоренные процедуры (см. раздел «Методологии ускоренной разработки» на стр. 547) уделяют значительно меньше внимания составлению спецификаций, но они не поощряют писать код как бог на душу положит. Поскольку спецификации не пишутся сами по себе, легко устаревают и требуют сопровождения, а у программистов и без того работы хватает, разумно писать только самые необходимые документы. Нужно всегда избегать долгих процедурных барьеров. *Но любую спецификацию, от которой вы отказались, нужно заменить равноценным объемом информации*. Не пренебрегайте спецификациями, если вы не заменили их документами равного качества, содержащими ту же информацию.

В экстремальном программировании не пишут длинные спецификации требований, но все требования фиксируются в эквивалентных *историях пользователя* (*user story*), хранящихся на *карточках историй*. Проектных спецификаций избегают: *код является собственной документацией*.

В ускоренной разработке также практикуется *проектирование, управляемое тестированием*, когда кодифицированные тесты выступают в качестве дополнительной документации по коду и его поведению. Такой полный и понятный набор тестов для блоков может заменить спецификацию тестирования для отдельных компонент, но редко подходит для проверки пригодности конечного продукта.

Что должны содержать спецификации?

Разные типы спецификаций весьма различаются по содержанию. Однако в любой спецификации в отношении информации должны соблюдаться следующие требования:

Корректность

Несмотря на очевидность, это крайне важное требование. Некорректная спецификация может стать причиной многих дней напрас-

ного труда. Необходимо ее постоянно обновлять, иначе она становится опасной: ее чтение оказывается потерянным временем, приводит к путанице и может стать причиной появления ошибок.

Если спецификация допускает неоднозначное толкование, это не спецификация. Два человека, прочтя ее, сделают разные выводы, что неизбежно приведет к печальным последствиям. Сделайте так, чтобы ваши спецификации точно передавали ваш замысел.

В тексте не должно быть противоречий. Если спецификация оказывается достаточно объемистой, соблюсти ее непротиворечивость становится затруднительно. Такая проблема особенно характерна для случаев, когда модификацию проводит сопровождающий (не автор кода) – легко, изменив информацию в одном месте, оставить в прежнем виде другие разделы, ссылающиеся на те же самые данные.

Спецификация должна быть составлена в соответствии с существующими стандартами (например, определениями языка и принятыми в фирме стандартами кодирования). Она должна следовать принятым в фирме стандартам/соглашениям для документов и использовать существующие шаблоны.

Понятность

Хорошую спецификацию приятно читать и легко понимать. Она понятна всякому, кто ее прочтет. Если она составлена настолько техническим языком, что разобраться в ней могут только инженеры, то прочие нетехнические службы (маркетинга или администрации) сочтут, что этот документ написан не для них, и не отнесутся к нему с должным вниманием. Возможные проблемы будут обнаружены слишком поздно.

Как и хороший код, лучшие спецификации пишутся с точки зрения читателя, а не писателя. Информация представляется так, чтобы быть понятной новичку, а не в удобном для автора виде. Блез Паскаль как-то извинялся: «Это письмо получилось у меня слишком долгим, потому что у меня не было времени его укоротить». Для хорошего стиля характерны краткость и очевидность главного смысла, не затененного многословием. Для этого требуются лишние время и труд, но оно того стоит, если в результате достигается простота изложения.

Не нужно думать, что спецификация должна содержать массу скучной прозы. Попробуйте применить приемы, сокращающие объем и облегчающие чтение. Маркированные и нумерованные списки, графики, заголовки и подзаголовки, таблицы и разумное использование пустого пространства позволяют разбить поток текста и помочь читателю составить мысленную карту материала.

Полнота

Спецификация должна быть самодостаточной и полной. Это не означает, что в ней должна содержаться *вся* мыслимая информация;

вполне допустимо ссылаться на относящиеся к делу документы, если эти ссылки точны (учитывайте в них версии документов) и позволяют читателю легко найти нужный документ.

Уровень детализации в спецификациях не должен опускаться до деталей реализации, иначе она станет слишком ограничительной или трудной для понимания. Люди склонны игнорировать сложные спецификации, а потому они оказываются невостребованными. Оставшись пылиться в углу, они только вводят в заблуждение тех читателей, которые не знают, что ими больше не руководствуются.

Проверяемость

Спецификация интерфейса программной компоненты приводит к появлению двух вещей: программной реализации и комплекта тестов для ее проверки. Поэтому содержание спецификации должно поддаваться проверке. На практике это сводится в основном к ее корректности, однозначности и полноте.

Модифицируемость

Ничто не должно устанавливаться навечно – ни код, ни документы. Если спецификация требует обновления (например, для исправления фактической ошибки), это не должно вызывать затруднений. Жесткая спецификация помогает вам обрести почву под ногами. Но если спецификация неверна, в этом нет никакого смысла. Документ должен быть доступен для редактирования (т. е. должен быть обеспечен доступ к источнику, а не просто экземпляру в формате PDF), а процедура выпуска и обновления не должна быть слишком обременительной.

Чтобы облегчить модификации, документ должен быть тщательно структурирован и иметь минимальный размер.

Самоописательность

В каждой спецификации должны присутствовать как минимум следующие части:

- *Форзац*, ясно показывающий название документа, подзаголовок, авторов, номер версии, дату последней модификации и статус доступа (например, закрытый документ компании, распространение на условиях NDA, открытый доступ).
- *Введение*, содержащее краткие сведения о задачах, области применения и предполагаемой аудитории.
- *Термины и определения*, необходимые для понимания содержимого. (Но не относитесь к читателю свысока: если документ предназначен инженерам-программистам, не нужно объяснять им, что такое RAM.)
- *Ссылки* на аналогичные или цитируемые документы.
- *Исторический* раздел, перечисляющий важные модификации и версии.

Прослеживаемость

Должна существовать процедура контроля за документами (типа системы управления исходным кодом) и центральное файловое хранилище документов. Каждая выпускаемая версия спецификации должна помещаться в хранилище и оставаться доступной, чтобы можно было узнать, с какой версией вы работали год назад; в один прекрасный день она вам снова понадобится. Можно воспользоваться системой управления версиями – этот инструмент годится для отслеживания версий файлов любого типа.

На форзаце документа имеется контрольная информация (номер версии, дата, автор и т. д.), которая позволяет убедиться, что у вас на руках самый свежий экземпляр.

Составляя спецификацию, обдумайте ее содержание. Выберите структуру и словарь, понятные аудитории, и убедитесь, что документ корректен, полон и содержит собственное описание.

Процесс составления спецификаций

То, что написано без усилий,
обычно читается без удовольствия.

Сэмюэл Джонсон

Теперь, когда мы знаем, какие типы спецификаций существуют и что они должны содержать, мы полностью вооружены. Пора что-нибудь написать! Процесс составления спецификации прост:

1. Выберите начальный шаблон документа. Он может быть определен установленной процедурой разработки проекта. Если шаблона нет, возьмите за основу существующую спецификацию.
2. Напишите документ. Действительно, это трудная часть. Что писать, зависит от типа спецификации.
3. Организуйте обсуждение документа. Привлеките всех, кому он может быть интересен.
4. После согласования (и, если требует процедура, официального подписания) присвойте документу номер версии, поместите в хранилище и разошлите соответствующим адресатам.
5. Если в дальнейшем возникнут проблемы, сделайте заявку на изменение спецификации и проверьте, что вам ясно, как модификация повлияет на объем работы по разработке. Если не ясно, то объем работ может совершенно незаметно удвоиться.

Написать этот список нетрудно – трудно выполнить. Можно сосредоточить усилия на пункте 2 и опустить остальные, чтобы облегчить себе жизнь. Но без прочих действий вы не сможете создать официальный, опознаваемый документ; впоследствии это может создать проблемы.

Создавая свой литературный шедевр, воспользуйтесь следующими рекомендациями. Первые из них относятся к авторству и вашим эстетическим пристрастиям:

- Обычно тексты лучше удаются, когда у каждого документа один автор. Координировать несколько авторов и согласовывать разные стили бывает трудно. Если вы пишете документацию для большой системы, разбейте спецификацию на части, и пусть над каждой частью работает один человек. Создайте общий документ, который свяжет вместе все части.

  Вопреки некоторым мнениям, нет ничего нескромного в том, что на обложке спецификации красуется одна фамилия. Должен же кто-то отвечать за этот документ своей репутацией – и когда автор достоин похвалы, и когда заслуживает порицания!

  Если вы существенно развили документ, написанный кем-то другим, не стесняйтесь добавить свое имя к списку авторов. Но никого не удаляйте из этого списка, если только не убрана написанная им часть.

- Автора нужно подбирать правильно. Служба маркетинга не напишет вам функциональные спецификации; она снабжает вас требованиями. Менеджеры не проектируют код; это делает разработчик, у которого есть соответствующие знания и умение. Автор должен уметь писать – это мастерство, которым можно овладеть, мускул, требующий упражнений.

- Для каждого документа должен быть определен владелец, который несет за него ответственность. Владелец может не быть автором первоначальной версии; это может быть технический руководитель или лицо, сопровождающее документ, если его автор более недоступен.

Вот несколько советов по написанию документа:

- Полезно иметь *хорошие конкретные* примеры спецификаций каждого вида. По ним автор может определить, чего от него ждут.

- Если спецификация носит предварительный характер, она должна быть помечена соответствующим образом и содержать предупреждение, что данный вариант не является окончательным. Это защитит людей от ошибочного восприятия спецификации как готовой, а вас – от их жалоб на качество (временно). Ведите список незавершенных разделов и открытых проблем внутри самого документа.

- Важно рецензировать документ; это способствует его корректности и правильному представлению. Рецензирование служит средством получить согласие других лиц с вашими решениями и тем самым придать авторитетность вашему документу. Особенно это необходимо для спецификаций, направляемых вне проекта – заказчику или в другие подразделения.

- Завершив разработку спецификации, не забывайте про нее. Она должна работать и своевременно обновляться. Функциональная

спецификация не кончается с завершением этапа проектирования. Требования претерпевают изменения, и мы получаем новые данные о функционировании системы. Все это нужно отражать в новых версиях спецификаций.

Языковые барьеры

Ненавижу определения.

Бенжамин Дизраэли

Очень внимательно пишите текст спецификации. По сравнению с кодом английский язык полон двусмысленностей и сложностей. По газетным заголовкам особенно хорошо видно, насколько неоднозначными бывают простые английские предложения: «Stolen painting found by tree», «Kids make nutritious snacks», «Red tape holds up new bridge» и «Hospitals are sued by 7 foot doctors».

Спецификации – официальные документы, они не должны быть многословными или написанными разговорным стилем; это скрыло бы за словами важные факты. У читателей, для которых английский не является родным языком, могут возникнуть трудности. Однако слишком сжатый документ тяжело воспринимается. Необходимо соблюдать меру, и рецензирование документа поможет выбрать правильный стиль письма.

Официальные документы пишутся от третьего лица, в настоящем времени. Очень важно тщательно подбирать слова. Полезное соглашение содержится в RFC 2119. Там определены следующие ключевые термины для спецификаций протоколов (они очень полезны и в спецификациях требований):

Must (Должен)

Слово *must* (или *shall*, или *is required to*) означает, что следующее определение является безусловным требованием спецификации.

Must not (Не должен)

Слова *must not* (или *shall not*) означают безусловное запрещение спецификации.

Should (Следует)

Пользуйтесь *should* (или *recommended*, рекомендуется) для обозначения необязательного требования – поведения, которое можно игнорировать, но только если понятны и учтены все возможные последствия.

Should not (Не следует)

Пользуйтесь *should not* (или *not recommended*, не рекомендуется) для описания определенного поведения, которого следует избегать, если только нет веских причин выбрать именно его – опять-таки взвесив все последствия.

May (Может)

Слово *may* (или *optional*) означает, что элемент действительно необязателен. Реализующий может по своему усмотрению поддерживать его или нет, но при этом одна реализация должна работать с другой, в которой сделан противоположный выбор.

Это слово часто следовало бы использовать там, где люди пишут *can*. *Can* часто ошибочно используют в спецификациях и стандартах; оно двусмысленно и в зависимости от интерпретации читателем может означать как *должен* (*must*), так и *может* (*may*).

Почему мы не пишем спецификации?

Ибо не понимаю, что делаю. Потому что
не то делаю, что хочу, а что ненавижу, то делаю.

К Римлянам 7:15

Хорошие спецификации примечательны тем, что в реальности их не найти. Мы знаем, что работать без них – нехорошо, поэтому шустрые разработчики пытаются замять их отсутствие и сделать вид, что нет никаких проблем. Довольно часто можно получить задание на программирование без надлежащих спецификаций требований или функциональности. (Это методическая проблема, которую нужно решать путем постоянных жалоб, просветительной работы, а при необходимости и превышения полномочий.)

Но столь же часто небрежные разработчики уклоняются от самостоятельного написания документов. В чем причина? Есть ряд отговорок, встречающихся постоянно. Разработчики не пишут спецификаций, потому что:

- Они не знают, что это необходимо.
- Они забывают.
- У них нет времени.
- Они сознательно решают не писать спецификации, считая, что могут обойтись без них («Да кто их вообще читает?»).

Ни одно из этих объяснений не выдерживает критики. Опытный разработчик, несомненно, не может предъявить первые два, если его работа предполагает поставку спецификации.

Программисты любят писать программы, а не длинные документы. Большинство программистов не слишком хорошо владеет пером; они пишут изящные программы, но ужасные тексты. Неудивительно, что они стремятся избежать составления спецификаций: это трудная, неинтересная работа, или они просто не любят ею заниматься. Часто они рассматривают ее как пустую трату времени на то, в чем нет необходимости. Или думают, что *сначала напишут код, а потом вернутся к документации*. Печальный опыт показывает, что этого не происходит.

Гнетущая мысль, что *никто никогда не прочтет их замечательную спецификацию*, отворачивает многих программистов от приложения умственных усилий для написания текста. Вероятно, они правы: возможно, что ни одна живая душа не прочтет их шедевра. Ну и что из того? Само по себе написание спецификации заставляет вас напрячь мозги – весьма важный результат. Разумеется, есть гуру, которые могут программировать на ходу и с отличными результатами. Но большинство программистов не может – согласны они с этим или нет. Мы должны проектировать. Тщательно. До написания кода. Результаты проектирования должны быть зафиксированы в документе. Может быть, этот документ никому кроме вас не попадется на глаза. Но если в один прекрасный день вы услышите свыше зов и отправитесь монашествовать в хорватский монастырь, то как другому программисту подхватить вашу работу? Эта спецификация вас переживет. Считайте ее своим наследием.

Отсутствие времени – единственный случай, который может оказаться вам неподвластен: иногда вам ставят задачу, и времени для написания хорошей спецификации действительно нет. Но если у вас нет времени на спецификацию, то и на написание правильного кода его тоже, скорее всего, будет недостаточно. Нужно различать, когда вы действуете правильно, а когда извергаете код, не придерживаясь никаких правил; в последнем случае результаты явно непригодны для включения в окончательную версию.

Экономия времени на спецификациях практически всегда оказывается ложной; спецификации помогают *сберечь время*, затрачиваемое иначе на передачу данных. Если вы пишете спецификацию, то должны лишь один раз объяснить, как работает программа. Если же вы пропустите этот этап, то не меньший объем информации вам все равно придется передать, но уже на другой основе: спустя долгое время и не всегда в удобных вам условиях. Такой обмен данными окажется значительно менее эффективным и фактически займет больше времени, поскольку вам придется излагать одно и то же снова и снова, слегка приспосабливая рассказ к конкретной аудитории.

Избегать составления спецификаций опасно и непрофессионально. Если не хватает времени для написания спецификаций, его, скорее всего, недостаточно и для написания кода.

Разумеется, мало кто пишет спецификации в домашних условиях для личных проектов. Это крайний случай, когда детальная спецификация не нужна. Всякий достаточно крупный проект (по числу исходных файлов, модулей, разработчиков или клиентов) реально требует составления спецификаций.

Резюме

Слова – самый сильный наркотик,
который использует человечество.

Редьярд Киплинг

Конечно, составление спецификаций не самое приятное занятие в жизни разработчика программного обеспечения, но это важная часть процедуры написания кода. Научитесь эффективно читать и писать их, записывайте нужную информацию в нужном месте, чтобы избавить себя в будущем от лишних затрат времени и неприятностей. Но не дайте поработить себя бумажной бюрократии.

Хорошие программисты...

- Понимают важность спецификаций и облегчают с их помощью свою жизнь
- Знают, насколько *детальной* должна быть документация
- Стремятся улучшить свои писательские навыки и ищут возможности их потренировать

Плохие программисты...

- Бросаются очертя голову в кодирование, не задумываясь над проектированием, документированием или рецензированием
- Не обдумывают текст, который пишут, и создают бессистемные, малопонятные спецификации
- Не любят писать документы, считая это скучным и бессмысленным занятием

См. также

Глава 4. Литературоведение

Самодокументирование кода – это надежная технология, помогающая избавиться от части документации по коду. С хорошим кодом настолько просто и интуитивно легко работать, что длинное руководство для него не нужно.

Глава 18. Защита исходного кода

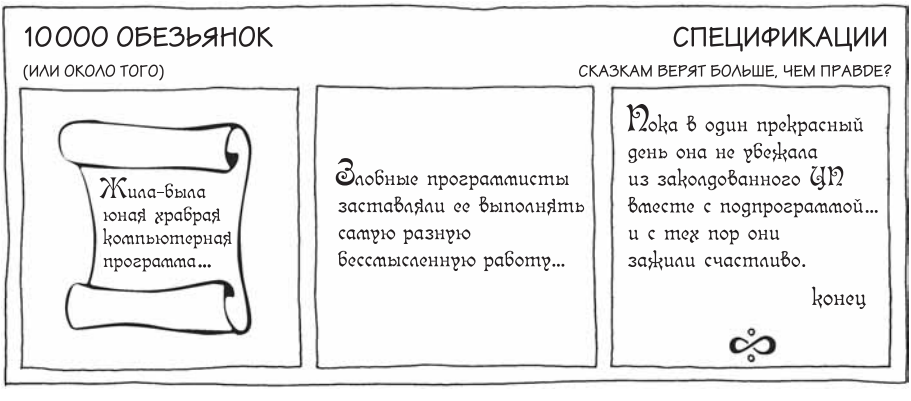
Рассмотрите возможность применения системы контроля версий и стратегии резервного копирования для ваших спецификаций – они столь же важны, как ваш код, и нуждаются в защите.

Глава 20. Рецензия на отстрел

Наряду с вашим кодом, все написанные вами документы следует подвергнуть рецензированию, чтобы обеспечить их корректность и высокое качество.

Глава 22. Рецепт программы

Спецификации – важная часть процесса разработки программного обеспечения и часто служат шлюзами между различными этапами разработки.



Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 671.

Вопросы для размышления

1. Что лучше – плохая спецификация или ее полное отсутствие?
2. Насколько детальной должна быть хорошая спецификация?
3. Нужно ли требовать, чтобы все документы в компании/проекте были выдержаны в одинаковом стиле?
4. Как нужно хранить документы? Например, стоит ли создавать для них указатель (по типу или проекту)?
5. Как организовать рецензирование спецификации?
6. Делает ли самодокументирование кода ненужными *все* спецификации? А какие-либо отдельные типы спецификаций?

7. Как можно организовать работу над документом, если авторов несколько?

Вопросы личного характера

1. Кто решает, что должно содержаться в ваших документах?
2. Рассмотрим ваш текущий проект. Есть ли у вас для него:
   a. Спецификация требований?
   b. Архитектурная спецификация?
   c. Проектная спецификация?
   d. Функциональная спецификация?
   e. Какая-нибудь еще спецификация?

   Являются ли они актуальными? Являются ли они полными? Знаете ли вы, где находятся самые свежие версии? Есть ли у вас доступ к старым версиям?
3. Есть ли у вас контроль версий ваших документов? Если да, то как он организован?

20

Рецензия на отстрел

Рецензирование кода

В этой главе:

- Что такое рецензирование кода?
- Что оно дает?
- Как его организовать?

Рецензенты имеют одно преимущество над самоубийцами: совершая самоубийство, вы отнимаете жизнь у себя; сочиняя рецензию – у других.

Джордж Бернард Шоу

Как стать хорошим столяром? Нужно пойти в ученики к столяру. Смотрите, как работает мастер, помогайте ему изо дня в день, постепенно беритесь за более трудные дела и слушайте его советы. Нельзя рассчитывать на то, что, не имея практического опыта, вы сразу сможете изготавливать качественные изделия.

В области кодирования так не бывает, хотя программирование – такое же ремесло, как инженерное дело (возможно, в большей степени). Хороший программист учится различать плохой и хороший код на собственном опыте, узнавая, что работает в реальном мире, а что нет. Нельзя научиться этому по книжкам, и лишь немногим выпадает удача

иметь учителя. Для большинства из нас *рецензирование кода* ближе всего подходит к этому идеалу.

Рецензирование кода (называемое также *инспекцией*, или *критическим анализом*) напоминает модель разработки open source – оно предоставляет возможность другим лицезреть ваш бесценный код, а вам – посмотреть, как работают другие. Оно облегчает обмен знаниями. Но главная задача рецензирования – повысить качество кода. С его помощью можно найти ошибки прежде, чем они повлекут за собой катастрофу.

У рецензирования кода есть еще одно скрытое достоинство: оно побуждает вас с большей ответственностью относиться к своему труду. Когда вы знаете, что ваш код увидит кто-то другой и будет его изучать, использовать, сопровождать и критиковать, подход к работе меняется. Реже возникает соблазн сделать поправки на скорую руку, на переделку которых потом никогда не будет времени. Чувство подотчетности, возникающее при практике рецензирования, повышает качество кодирования. Рецензирование способствует также установлению культуры «коллективного владения кодом», о котором говорилось на стр. 431.

Звучит соблазнительно, не правда ли? Что ж, заглянем на кухню и посмотрим, что там делается...

Что такое «рецензирование кода»?

Рецензирование помещает код под микроскоп, имея задачей подвергнуть его критике и проверке. Его цель не высмеять автора, а повысить качество программного обеспечения, создаваемого командой. Обычно в его процессе создается список исправлений, которые нужно сделать (его размер отражает уровень вашего мастерства!). Иногда отмечаются усовершенствования, заниматься которыми в данный момент не стоит; возьмите эти открытия на заметку, чтобы заняться ими в будущем.

Мы ищем ошибки и код, который можно исправить. Рецензирование кода выявляет проблемы разного уровня:

- Общая конструкция (проверяем выбор алгоритмов и внешних интерфейсов).
- Выражение конструкции в коде (разбиение на классы и функции).
- Код в отдельных семантических блоках (проверяем корректность каждого класса, функции и цикла, выбор подходящих идиом языка и практической реализации).
- Отдельные операторы кода (соответствие их стандартам, принятым в проекте, и лучшей практике).

Рецензии кода бывают:

Личные

Автор тщательно и методично изучает свою работу, проверяя ее качество. Это не то же самое, что перечесть код, напечатав его; самостоятельное рецензирование кода – более детальная и сложная задача.

Один на один

Вы показываете свой код для критического анализа другому программисту. Тот проверяет логику и смотрит, нет ли ошибок в указанных вами местах. Такие рецензии обычно неформальны и проходят под управлением автора. Код, таким образом, рассматривается под его углом зрения, со сделанными им допущениями, а не с более объективной точки зрения стороннего наблюдателя.

Официальные

Задействуя других программистов, вы привлекаете дополнительные знания и опыт и обеспечиваете свежий взгляд на сделанную работу. Широкое рецензирование труднее организовать, но оно может оказаться более успешным в искоренении проблем. При личной инспекции сложно выполнить столь глубокий анализ; автор слишком привыкает к своему коду и может не обратить внимания на недостатки.

Обычно такое рецензирование производится во время официального совещания, но оно может быть и виртуальным, с использованием средств связи.

Рецензирование каждого типа может осуществляться в разные моменты разработки. Проверка «один на один» может происходить ежедневно, в порядке проверки интеграции перед сохранением модификаций в исходном дереве. Официальное рецензирование вводится ближе к концу разработки в качестве финального аудита качества программного обеспечения.

Помимо очевидных достоинств корректного кода у рецензирования есть другие полезные побочные эффекты. Перекрестное опыление, происходящее при изучении программистами кода своих коллег, способствует единообразию стиля кодирования в проекте. Рецензирование также распространяет знание внутреннего устройства важных участков кода, поэтому уменьшается риск потери информации при уходе программистов из проекта (см. раздел «Роспуск команды» на стр. 439).

Рецензирование кода – отличное средство для поиска и устранения скрытых ошибок, повышения качества кода, усиления коллективной ответственности за код и распространения знаний.

Когда проводить рецензирование?

Если вас не критикуют, значит, вы ничего не делаете.

Дональд Рамсфельд

В идеале каждый кусочек кода должен быть тщательно прорецензирован, прежде чем попасть в окончательную версию. Согласно Институту технологии программирования Университета Карнеги-Меллона, тщательное рецензирование кода должно занимать не менее половины всего времени кодирования (с учетом личного рецензирования)

(Humphrey 98). Это больше, чем может себе позволить большинство реальных проектов.[1]

Создавая систему, выясните, нужно ли проводить рецензирование, и если да, то какого кода.

Нужно ли рецензировать

Мы знаем, что ошибок избежать невозможно и что ваш код наверняка содержит несколько классических ошибок. Очевидные ошибки вы быстро обнаружите, но гораздо больше тонких проблем сможет выявить только свежая пара глаз, непредвзято рассматривающих код. Самому автору трудно заметить недостатки, присущие его собственной работе, поскольку он слишком близок к коду и страдает от *когнитивного диссонанса*, как описано у (Weinberg 71). Если ваш код представляет какую-либо ценность (подсказка: да, представляет, иначе вы не стали бы его писать) и вас заботит его качество (подсказка: да, заботит, иначе вы позорите себя), вы *должны* подвергнуть его рецензированию.

Если вы не рецензируете код, резко возрастает вероятность пропустить ошибку в готовый продукт. Это может повлечь за собой неприятности, большой объем переделки и обновления продукта с выездом к заказчику, а иногда и финансовый крах вашей компании. Такие последствия не сопоставимы с затратами на рецензирование. Согласно Хамфри, «Студенты и инженеры обычно допускают от 1 до 3 ошибок в час при проектировании и от 5 до 8 при написании кода. Они устраняют лишь от 2 до 4 ошибок в час при тестировании, но обнаруживают от 6 до 12 ошибок в час во время рецензирования» (Humphrey 97).

Для того чтобы не проводить рецензирование, обычно ищут разные оправдания. Например, говорят, что «этот код слишком велик, чтобы полностью рецензировать его» или «он слишком сложен; никто в нем не разберется, и нет даже смысла пытаться его рецензировать». Если проект смог обеспечить достаточное количество человеко-часов, чтобы написать такую программу, найдется и достаточно времени для ее рецензирования. А если код слишком сложен, то тем более он нуждается в рецензировании! Возможно, он требует даже более радикальных мер. Грамотно написанный код раскладывается на самостоятельные секции, которые можно рецензировать отдельно.

Какой код рецензировать

В любом проекте быстро образуется масса исходного кода. В любых, кроме самых строгих, процедурах разработки просто нет времени, чтобы рецензировать код до последнего клочка. Как же выбрать те части, которые нужно рецензировать? Это не так просто.

---

[1] Более серьезную проблему представляет то, что они редко готовы вообще тратить время на рецензирование.

Нужно отобрать код, которому рецензирование больше всего пойдет на пользу. Это код, в качестве которого вы больше всего сомневаетесь

Альтернативы рецензированию

Существует ряд технологий разработки, которые, по мнению некоторых, делают излишними официальные рецензии. Перечислим их:

Программирование в паре

Программируя в паре (как описано на стр. 409), вы фактически осуществляете постоянное рецензирование. Две пары глаз лучше, чем одна, и обнаружат гораздо больше ошибок – прямо при вводе. Однако при рецензировании можно обнаружить еще больше недостатков за счет привлечения к нему людей, которые физически и эмоционально не связаны с данной реализацией.

Open source

Открытое и бесплатное распространение кода позволяет увидеть его всем желающим, оценить качество и исправить недостатки. Некоторые считают это вершиной рецензирования. Однако такая система не гарантирует, что кто-нибудь станет изучать ваш код. Только у действительно популярных открытых проектов есть активно поддерживаемая кодовая база. Сделав свой код общедоступным, вы не обязательно сразу получите преимущества рецензирования.

Поблочное тестирование

Это средство автоматической проверки того, что модификация не снизила степени корректности вашего кода (см. раздел «Руками не трогать!» на стр. 203). При этом ваш код может представлять собой «спагетти», но если он пройдет поблочное тестирование, никто этого не заметит. Если блочные тесты недостаточно строги, они могут пропустить ошибки.

Отсутствие рецензирования

Можно просто довериться программисту в расчете, что он все сделает правильно – в конце концов, это его работа. Если такой стратегии достаточно для успеха, то и тестировать код ни к чему. В добрый путь!

Ни одна из этих альтернатив в отдельности не может по-настоящему заменить рецензирования. Возможно, что их сочетание при наличии в команде какой-нибудь особенно эффективной культуры разработки уменьшает необходимость рецензирования, но мне такие команды еще не встречались.

или который важнее всего для правильного функционирования системы. Можно воспользоваться следующими стратегиями:

- Отобрать главные фрагменты кода основных компонент.
- Запустить профайлер, чтобы выяснить, где тратится больше всего процессорного времени, и подвергнуть рецензированию соответствующие участки кода.
- Запустить утилиту, анализирующую сложность, и отправить на рецензирование самый подозрительный код.
- Выбрать области, где уже обнаруживалось больше всего ошибок.
- Выбрать код, написанный программистами, которым вы не доверяете (месть в виде рецензии!).

Самый практичный подход – воспользоваться всеми перечисленными способами. Отберите наиболее подходящие фрагменты кода, основываясь на трезвой оценке своей команды, базового кода и текущих характеристик системы (производительности, подсчета ошибок и т. п.)

Тщательно отберите код для рецензирования. Если вы не можете прорецензировать весь код, сделайте обоснованный выбор. Не нужно отбирать наугад и терять драгоценное время.

Проведение рецензирования кода

То, что мы делаем постоянно, становится
легче не потому, что задачи становятся легче,
а потому что растет наше умение решать их.

Ральф Уолдо Эмерсон

Простого *проведения* рецензирования кода недостаточно. Само по себе это не решает всех проблем. Нужно обеспечить *правильное* проведение рецензирования. Этому мы и посвятим несколько следующих разделов.

Рецензирование на собраниях

Рецензирование чаще всего проводится в виде официального *совещания* (по крайней мере, когда процедура разработки предполагает строгий церемониал). Предлагаются жесткая повестка дня (чтобы выполнить все необходимые действия) и определенные границы (необязательно по времени, но указывается, какой код вы рецензируете, а какой нет, поскольку это не всегда очевидно).

Ниже приводится пример организации совещания.

Где?

Лучше всего проводить рецензирование в тихом помещении. Участников не должны беспокоить. Следует организовать кофе (или чай, если кому-то он больше нравится).

Полезным может оказаться наличие сети из лэптопов с редакторами кода, а также компьютера с проектором. Программисты старой школы предпочитают распечатывать код и делать пометки ручкой – когда отрываешься от экрана компьютера, можно увидеть новые ошибки. Все зависит от того, насколько вас заботят потребление электроэнергии и сохранение лесов.

Когда?

Очевидно, в удобное всем время. Здравый смысл подсказывает, что 5 часов вечера в пятницу не лучший выбор. Времени потребуется изрядно, поэтому позаботьтесь, чтобы вас никто не беспокоил и не отвлекал.

Если код очень большой, разбейте рецензирование на несколько встреч. Нельзя держать людей часами в замкнутом пространстве и рассчитывать, что качество их рецензий будет при этом высоким.

Роли и обязанности

Важнейший фактор успеха совещания по рецензированию – выбор участников. Каждому из них должна быть назначена определенная роль; в небольших группах один участник может исполнять несколько ролей. Роли следующие:

Автор

Очевидно, тот, кто написал код, должен участвовать в обсуждении, чтобы рассказать, какую работу он проделал, отвергнуть несправедливую или некорректную критику и выслушать правильные, конструктивные замечания (и принять их к руководству).

Рецензенты

Рецензентов нужно тщательно отобрать среди людей, у которых есть время и способности к рецензированию. Хорошо, если код входит в их сферу компетентности или каким-то образом их касается. Например, разработчика библиотеки можно пригласить на рецензирование программы, которая использует эту библиотеку, чтобы отметить случаи неправильного применения API.

Должно присутствовать достаточное число опытных программистов. Можно пригласить представителя QA или тестирующего подразделения (см. врезку «Контроль качества» на стр. 187), чтобы QA убедились в качестве программного продукта и процесса разработки.

Председатель

У каждого совещания должен быть председатель, иначе возникнет хаос (см. врезку «Это судьба» на стр. 435). Это лицо ведет совещание и направляет обсуждение. Он должен следить за тем, чтобы обсуждение было деловым и не уходило в сторону. Мелкие вопросы, не требующие обсуждения на совещании, должны им сниматься. Дай программистам волю, и они станут часами обсуждать мелкие технические детали в ущерб рецензированию остального кода.

Секретарь

Секретарь ведет протокол, записывает, какие вопросы были подняты, чтобы ничего не забыть по окончании рецензирования. Если есть контрольный список для рецензирования (см. пример на стр. 504), секретарь его заполняет. Председатель, ведущий совещание, не должен выполнять роль секретаря.

Предполагается, что до начала совещания все участники ознакомятся с кодом. Все должны прочесть сопутствующую документацию (спецификации и пр.)[1] и быть в курсе стандартов кодирования, принятых в проекте. Тот, кто организует совещание, должен указать на эти документы в объявлении о предстоящем мероприятии во избежание недоразумений.

Повестка дня

Чтобы организовать рецензирование кода:

- Автор сообщает, что его код готов к рецензированию.
- Председатель организует совещание (резервирует место, назначает время и подбирает рецензентов).
- Подготавливаются все необходимые ресурсы (компьютеры, проектор, распечатки и т. п.).
- О предстоящем совещании нужно объявить заранее, чтобы рецензенты могли подготовиться.
- После объявления о рецензировании автор не должен изменять код без крайней необходимости – это было бы нечестно по отношению к рецензентам.

Рецензирование проходит так:

- Председатель организует подготовку помещения, с тем чтобы вовремя начать совещание.
- Автор в течение нескольких минут (не более!) рассказывает о назначении кода и немного о его структуре. Это должно быть известно заранее, но поразительно, сколько недоразумений обнаруживается на этой первой стадии.
- Предлагается высказаться по структуре проекта. Имеется в виду структура реализации, а не уровень операторов кода. Это может касаться разделения функциональности на классы, распределения кода по файлам и стиля написания функций. (В достаточной ли мере соблюден защитный стиль и есть ли хорошие тесты?)
- Предлагается высказать общие замечания по коду. Они могут касаться неправильно выбранного стиля кодирования, неудачных шаблонов приложения и проекта или неправильных идиом языка.

[1] Естественно, вся сопровождающая документация должна быть тщательно прорецензирована заранее.

- Последовательно детально разбирается код, по строке или по блокам, и ищутся ошибки. То, на что следует обратить внимание, описывается ниже (раздел «Идеальный код» на стр. 501).
- Рассматривается несколько сценариев применения кода и изучается поток управления. Если есть полный набор для поблочного тестирования (а он должен быть), тогда в нем есть все сценарии, которые нужно рассмотреть. Он поможет рецензентам проследить все возможные пути выполнения.
- Секретарь отмечает, какие изменения необходимо произвести (записывает имя файла и номер строки).
- Проблемы, которые могут касаться более широкой области кода, регистрируются с целью дальнейшего исследования.
- После рецензирования обсуждается, какими должны быть последующие действия.

Возможные сценарии:

Okay

Отличный код, не требует никакой доработки.

Доработать и проверить

Требуется некоторая доработка кода, но созывать новое совещание нет необходимости. Председатель назначает ответственного проверяющего. После доработки проверяющий сверяет код с протоколом рецензирования.

Для всякой доработки устанавливается такой срок, чтобы ее суть и причины оставались свежи в памяти.

Переработать и снова рецензировать

Код требует значительной переработки, и сочтено необходимым провести новое рецензирование.

Помните, что цель совещания состоит в выявлении проблем, а не устранении их на месте. Некоторые проблемы могут потребовать поиска путей решения, и этой работой автор (или тот, кто модифицирует код) должен заняться после обсуждения кода.

При проведении рецензирования кода вам может оказаться полезен контрольный список, приведенный в конце главы.

Интеграционное рецензирование

Совещания по рецензированию кода представляют собой весьма регламентированный метод. Их проведение требует труда, но несомненно, что с их помощью обнаруживается много проблем, которые иначе прошли бы незамеченными.

Существуют менее напряженные процедуры рецензирования, предоставляющие большинство возможностей совещаний, но в более удобном виде. Наиболее эффективным из них может быть *интеграционное*

рецензирование, проводимое при включении нового кода в основную ветку разработки.

Его можно проводить:

- *Перед записью* нового фрагмента кода в систему контроля версий.
- *Сразу после записи* нового фрагмента кода в систему контроля версий.
- После слияния ветви разработки функции с главной ветвью.

В один из этих моментов рассматриваемый код помечается как подлежащий рецензированию и выбирается подходящий рецензент: кто-либо из ответственных за этот модуль (интегратор или сопровождающий[1]) или *дублер*, назначаемый для рецензирования работы «один на один».

Такой контролируемый ввод кода в систему часто реализуется с помощью программных средств, интегрированных с системой контроля за исходным кодом. Организовать его вручную довольно трудно, и обычно он представляет собой порядок сохранения кода в системе: вы не должны ничего вводить без предварительного рецензирования коллегами. Управлять таким методом достаточно трудно; ошибки могут проскочить, когда копирование кода в систему производится в спешке, в последнюю минуту.

Фактическое рецензирование в данном случае проходит гораздо менее формально, чем на описанных выше собраниях. Рецензент проверяет, нет ли в коде очевидных недостатков, тестирует его (возможно, с проверкой качества имеющихся тестов) и затем дает разрешение на включение в основную ветвь. Только после этого интегратор переносит проверенный код в дерево выпуска. В серьезных проектах или в ответственные моменты (например, перед важной контрольной точкой) процедура рецензирования может становиться более строгой и требовать дополнительных рецензентов и действий.

Поскольку рецензенту и автору не требуется встречаться лично (хотя это было бы желательно), можно считать такой вариант разновидностью виртуального рецензирования.

Пересмотрите свое отношение

Как хотите, чтобы с вами поступали люди,
так поступайте и вы с ними.

Лука 6:31

Рецензирование требует конструктивного подхода; без правильного отношения пользы не будет. Это касается обеих сторон – и автора, и рецензента.

---

[1] Сравните с модератором проекта open source, который сравнивает патчи, поступившие от разных разработчиков, и включает их в основное дерево кода, периодически выпуская обновленные версии.

Позиция автора

Многие стараются избежать рецензирования, опасаясь, что оно выставит на обозрение их недостатки. Не следует этого бояться. Благодаря рецензированию вашего кода вы можете узнать новые приемы. Нужно иметь достаточно скромности, чтобы признать свое несовершенство и принять критику. Ваш стиль кодирования улучшится, если вы сделаете выводы из предложенных модификаций.

Кем бы ни был написан код, он подлежит рецензированию и тщательному изучению со стороны коллег. Добивайтесь рецензирования своего кода.

Автор кода не должен любой ценой защищать его от критики. Естественно, что любая критика воспринимается как личная и как сомнение в ваших способностях. Когда рецензируют ваш код, нужно поубавить самомнение и гордость. Никто не может написать идеальный код: даже в коде самого потрясающего программиста найдутся досадные мелкие проблемы, которые будут отражены в рецензии.

Это *неэгоистичное программирование*, описанное Джеральдом Вайнбергом в книге 1971 года «The Psychology of Computer Programming» (Психология компьютерного программирования): не устаревающее описание критического отношения, благодаря которому рецензии оказываются действенными. (Weinberg 71) Те программисты, которые не боятся, что кто-то другой найдет в их коде ошибки, пишут в результате более надежные и правильные программы. Стремление к тому, чтобы вам помогли найти ошибки в вашей работе, существенно характеризует программиста-мастера.

Когда вас приглашают «на ковер», постарайтесь не заставлять других тратить напрасно свое время. Прежде чем представить код на рецензирование, проведите репетицию самостоятельно. Представьте себе, что излагаете свою работу другим. Вы удивитесь, какое количество мелких недостатков обнаружится, и это поможет вам более уверенно чувствовать себя при настоящем рецензировании. Не нужно предлагать сырой код и рассчитывать, что вместо вас кто-то другой найдет в нем ошибки.

Позиция рецензента

При рецензировании кода и его критике нужно проявлять деликатность. Комментарии должны быть конструктивными, а не преследующими обвинительные цели. Не нападайте на автора лично. Важно проявлять дипломатичность и такт. Адресуйте свои комментарии коду, а не его автору. Лучше сказать: «*Код всегда...*», а не «*Вы всегда...*».

Рецензирование кода – это *процедура с равными участниками*: старшинство здесь не имеет значения, и учитываются все точки зрения. Любопытно, что даже наименее опытному программисту найдется что сказать в рецензии на код. При этом рецензирование может пойти на пользу как автору, так и рецензенту.

Система в безумии

Рецензирование кода – повсеместно признанная технология, бытующая со времен перфокарт. Мы подробно рассмотрели две процедуры, но существует много тонких разновидностей рецензирования. Команды программистов выбирают тот механизм, который более подходит их членам и характеру работы. (Жаль те команды, которые вообще не проводят рецензирования.)

Вот еще два популярных метода рецензирования:

Проверки Фэгана

Это достойная процедура формального рецензирования, во многом совпадающая с описанной в данной главе и предложенная Майклом Фэганом в его книге «Defect Free Process» (Процесс устранения дефектов) (Fagan 76). Фэган подчеркивает важность умения осуществлять рецензирование и показывает, как можно его развить. Инспекции Фэгана выявляют проблемы как в продукте, так и в технологии его создания.

Дублирование

Это промежуточная точка между парным программированием и рецензированием кода. У каждого модуля есть *главный разработчик*, работающий над кодом. Назначается также *дублер* (*shadow developer*), периодически обсуждающий модуль с главным разработчиком. По ходу кристаллизации проекта дублер проверяет принятые решения. По ходу написания кода дублер контролирует достигнутые результаты и дает конструктивные советы.

При более формальной организации дублер получает право утверждать код к выпуску. Ни один модуль нельзя включать в окончательную версию, пока на это не будет получено согласие дублера.

Со временем у вас накопится богатый опыт рецензирования (особенно если вы будете рецензировать код для интеграции). Старайтесь, чтобы рецензирование не превратилось для вас в скучную обязанность – тогда оно станет для всех пустой тратой времени. Сохраните позитивный подход к рецензированию кода. В качестве рецензента старайтесь в каждом случае высказать что-либо полезное. Иногда это легко, а иногда очень трудно сказать что-либо интересное. Заставляя себя выступать с комментариями, вы избежите рутины и не превратитесь в человека, со всем согласного и ничего своего не вносящего в дело.

Успех рецензирования сильно зависит от позитивной позиции автора и рецензентов. Цель рецензирования – совместными усилиями улучшить код, а не назначить виновных или оправдать решения, принятые во время реализации.

Идеальный код

Когда же настанет совершенное, тогда то,
что отчасти, прекратится.

1 Коринфянам 13:10

Мы пока не рассматривали вопрос о том, какой код может пройти рецензирование, а какой может провалиться. В задачи данной главы не входит описание того, каким должен быть хороший код, – важные стороны высококачественного кода описаны в первых 15 главах книги. Когда мы ищем в коде недостатки и ошибки, постоянно возникает ряд вопросов. Рецензируемый код должен быть:

Свободен от ошибок

Ошибки – это наш враг и бич разработки хороших программ. Нам нужна уверенность в качестве результатов своего труда, и ошибки должны быть выявлены на возможно раннем этапе разработки. Чем раньше мы станем их искать, тем раньше обнаружим и исправим, и тем меньше убытков и неприятностей они нам причинят (см. врезку «Экономика ошибок» на стр. 216).

Корректен

Код должен отвечать всем действующим стандартам и требованиям. Проверьте правильность типов всех переменных (например, чтобы не возникло числового переполнения). Комментарии должны быть точными. Код должен соответствовать требованиям к памяти и производительности (особенно важно для встроенных платформ). Проверьте правильность параметров функций и обращений к библиотекам.

Код должен быть проверен на соответствие спецификациям требований и функциональности. Считается, что спецификации были составлены корректно, если же нет – задача значительно осложняется! Иногда комментарии в рецензии на код касаются спецификаций (например, если они не вполне понятны), но это не входит в задачи рецензирования – не отвлекайтесь на дискуссии относительно правильности спецификаций; секретарь должен зафиксировать проблему в протоколе, и рецензирование должно быть продолжено.

Полон

Код должен полностью реализовывать функциональную спецификацию. Он должен быть удовлетворительным образом интегрирован и отлажен и пройти все тесты. Комплект тестов должен быть полным.

Хорошо структурирован

Убедитесь, что реализация разумно спроектирована, код легко понять, дублирования и избыточности в коде нет. Проверьте, например, что нет бросающихся в глаза образцов *программирования путем копирования и вставки*.

Предсказуем

В коде не должно быть чрезмерной сложности и неожиданных сюрпризов. Код не должен модифицировать себя сам, не должен основываться на неопределенных начальных значениях и не должен вызывать подозрений в возможности бесконечных циклов или рекурсий.

Надежен

Код придерживается защитной стратегии. По возможности обеспечена защита против обнаруживаемых ошибок времени выполнения (деления на нуль, недопустимых численных значений и т. п.). Все данные ввода должны проверяться (как параметры функций, так и вход программы). Код обрабатывает все возникающие ошибки и исключения. Все необходимые сигналы перехватываются.

Контролирует ошибки

При доступе к массивам в стиле C осуществляется проверка границ. Отсутствуют другие аналогичные коварные ошибки доступа к данным. Если в коде несколько потоков, для предотвращения состояния гонки и взаимной блокировки правильно используются мьютексы. Проверяются *все* значения, возвращаемые системными/библиотечными вызовами.

Легок в сопровождении

Программист разумно применяет комментарии. Код хранится в системе управления версиями. Имеются необходимые данные о конфигурации. Форматирование кода отвечает внутрифирменным стандартам. Код успешно компилируется без вывода многочисленных предупредительных сообщений.

Если вы не знаете, как должен выглядеть хороший код, вы не сможете обоснованно судить о качестве работы других программистов.

За пределами рецензирования кода

Процесс рецензирования – важный элемент выпуска любого продукта высокого качества, поэтому его применение не ограничивается разработкой исходного кода. Аналогичная процедура рецензирования применяется при создании спецификаций, перечней требований и т. п.

Резюме

Проще критиковать, чем не ошибаться самому.

Бенджамен Дизраэли

Рецензирование кода – важная часть процедуры разработки программного обеспечения, которая помогает обеспечить высокое качество кода. Подобно тому как подмастерье осваивает ремесло благодаря передавае-

мым ему секретам, так и рецензирование кода способствует распространению знаний и обучению кодированию. Будучи в большей мере взаимодействием между равными, чем между учеником и мастером, рецензирование позволяет учиться как автору, так и рецензенту.

Пишите код в расчете на рецензирование. Помните, что читать его будете не только вы сами; другие люди должны иметь возможность его сопровождать. Автор всегда отвечает за качество своей работы. Хороший программист более заботится о создании отличного кода, чем об удовлетворении своего чувства гордости.

Хорошие программисты...

- Стремятся к рецензированию своего кода, будучи уверены в его качестве
- Прислушиваются к мнению других и извлекают из него уроки
- Способны деликатно и точно прокомментировать чужой код

Плохие программисты...

- Боятся рецензирования своего кода и чужих оценок
- Плохо воспринимают критику: занимают оборонительную позицию и легко обижаются
- Пользуются рецензированием для демонстрации своего превосходства над менее способными программистами; их комментарии излишне резки и неконструктивны

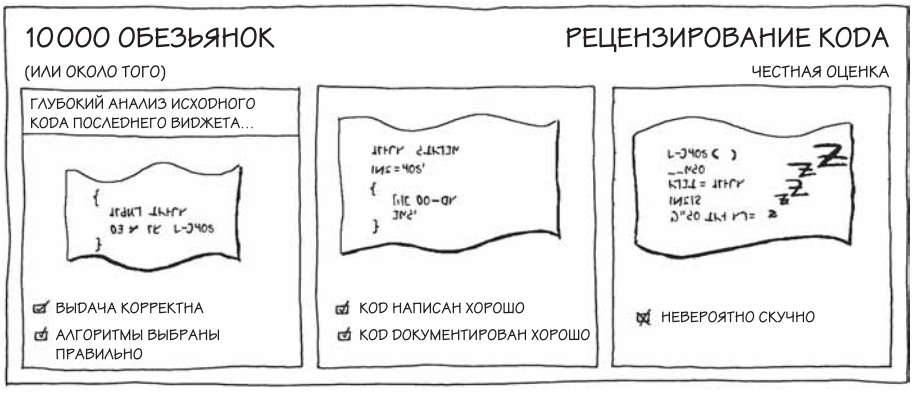
См. также

Главы с 1 по 15

Во всех начальных главах книги описываются разные аспекты правильно написанного кода.

Глава 9. Поиск ошибок

Описание типов ошибок, которые могут встретиться в коде.



Глава 19. Спецификации

При рецензировании код сравнивается со *спецификацией*. Спецификации тоже требуют тщательного рецензирования.

Контрольный список

Во многих процессах рецензирования участвует *контрольный список* – набор характеристик хорошего (удовлетворительного) кода, наличие которых нужно проверить. Если код не отвечает этим критериям, значит, он не прошел рецензирование. Списки различаются в деталях, объеме и предметной области.

На следующей странице представлен пример такого списка. Он может быть вам полезен при рецензировании. В отличие от существующих списков, в нем нет систематического перечисления всех проблем, которые могут возникнуть во всех языках; он просто помогает направить процесс рецензирования и определить, когда перейти к следующему этапу.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 675.

Вопросы для размышления

1. Почему программы пишут командами? В чем преимущества относительно самостоятельной разработки?
2. Зависит ли количество рецензентов от объема рецензируемого кода?
3. Какие инструменты полезны при рецензировании кода?
4. Когда нужно рецензировать код – до или после обработки утилитами проверки исходного кода?
5. Какая подготовка нужна перед совещанием по рецензированию?
6. Как отличить замечания рецензентов, которые нужно реализовать сразу, от тех, которые нужно взять на заметку для следующего проекта?
7. Как провести виртуальное совещание по рецензированию?
8. Насколько полезны неформальные рецензии кода?

Вопросы личного характера

1. Подвергается ли рецензированию код в вашем проекте? *Достаточно* ли проводится рецензирований?
2. Есть ли у вас программисты, чей код считается выше того, чтобы быть рецензируемым?
3. Какой процент вашего кода когда-либо проходил рецензирование?

Рецензирование кода Контрольный список

Эта форма поможет вам рецензировать код.

О коде

Имя модуля: _____
Версия: _____
Автор: _____

Кто рецензировал: _____
Дата: _____
Язык: _____
Количество файлов: _____

Автоматизированная проверка

☐ Код компилируется без ошибок
☐ Код компилируется без предупреждений
☐ Существуют тесты блоков
  ☐ Тестов достаточно (есть граничные случаи и т. п.)
  ☐ Код проходит тесты

☐ Код хранится в системе контроля версий
☐ Код прошел тестирование с помощью средств:

Название средства	Результаты
_____	_____
_____	_____
_____	_____

☐ Перейти к след. разделу ☐ Прекратить рецензирование

Проект

☐ Код полный (относительно спецификации)
☐ Выбор алгоритмов хороший
☐ Оптимизация необходима и уместна
☐ Отсутствие функций ясно помечено в коде

Общие замечания о структуре кода

☐ Код хорошо структурирован
☐ Есть проектная документация
  ☐ Код соответствует документации

☐ Перейти к след. разделу ☐ Прекратить рецензирование

Общие замечания о коде

Стиль
☐ Структура кода понятна
☐ Соответствует правилам проекта
☐ Есть хороший открытый API
☐ Имена выбраны хорошо

Защитное программирование
☐ Доступ к массивам защищен (C/C++)
☐ Типы выбраны правильно
☐ Все входные данные проверяются
☐ Особенности компилятора не используются

Общие комментарии
_____
_____
_____

Общие замечания о качестве кода

Обработка ошибок
☐ Ошибки регулярно проверяются
☐ Логика проверяется операторами assert
☐ Исключения проверяются
☐ Ошибки распространяются, не скрываются
☐ Утечки ресурсов нет

☐ Существует многопоточность
☐ Потоки безопасны
☐ Взаимоблокировки исключены

Структура
☐ Избыточности в коде нет
☐ Дублирования кода нет

☐ Перейти к след. разделу ☐ Прекратить рецензирование

Обзор на уровне команд

Заполните таблицу, оценив проблемы от 0 (косметические изменения) до 5 (нужно исправить) баллов

Файл	Строка	Проблема				Серьезность

Заключение

Вывод:
☐ Код хороший
☐ Доработать и проверить
☐ Доработать и снова рецензировать

Записать результат рецензирования

Работу выполнить к (дата): _____

Назначен проверяющий: _____

21

Какой длины веревочка?

Черная магия оценки продолжительности проекта

В этой главе:

- Зачем нужно оценивать время?
- Почему оценивать так трудно?
- Практические способы оценки
- Как уложиться в график

Я никогда не гадаю. Очень дурная привычка – действует пагубно на способность логически мыслить.

Шерлок Холмс (Сэр Артур Конан Дойл)

Так какой же длины веревочка? Или в нашем случае сколько веревочке виться? На этот вопрос столь же непросто ответить, и смысла в нем примерно столько же.

Эта глава посвящена *оценке времени разработки программного обеспечения* – важному умению профессионального программиста. Это один из мистических аспектов разработки, в котором больше догадок, чем науки, и эти оценки часто оказываются неверны. Это сложная, но важная часть процесса разработки, которую должен освоить каждый программист.

Правила промышленного производства программ неизбежно подчиняются экономике – денежным потокам. Оценки времени важны,

поскольку основную часть стоимости разработки программного обеспечения составляет оплата труда – программисты недешевы. Стоимость средств разработки и аппаратной части относительно незначительна. Чтобы создать программный продукт, мы должны знать, какой объем работы для этого потребуется, сколько людей необходимо привлечь и когда он будет готов и сможет начать приносить доход. В результате мы узнаем, во сколько обойдется его создание. Отдел маркетинга оценит, сколько смогут принести его продажи. Эти два предсказания сталкиваются в смертельной схватке: как урезать бюджет, чтобы сделать проект коммерчески выгодным?

Странная вещь – *планирование*; большинство программистов не очень в нем искушено. Но не волнуйтесь: на то у нас и существуют менеджеры. Но, если вы хотите добиться успеха, правила игры нужно знать. Создание коммерчески успешных программ требует большой прозорливости и тщательного планирования. Ну и, конечно, стальных нервов.

Чтобы создать план разработки, мы выполняем общее проектирование программной системы, разбиваем ее на компоненты и оцениваем для каждой из них время, необходимое для ее написания. Редко хватает времени, чтобы серьезно изучить и спроектировать каждую компоненту, поэтому оценки оказываются весьма грубыми. При выборе модели разработки (см. раздел «Процессы разработки» на стр. 536) мы составляем из оценок план, распределяем работу среди некоторого количества программистов и на основании этого делаем экономический расчет. Очевидно, что качество этого плана определяется качеством временных оценок. Слишком большая неточность при этом может привести к финансовому краху, поэтому оценки играют важную роль.

Когда нет плана, продукт разрабатывается наудачу, а не целенаправленно. Оценка времени входит в процедуру планирования проекта, но это не значит, что ею занимаются планировщики проекта! Никто кроме программистов, которым предстоит делать работу, не может предоставить информацию о необходимом времени. То есть никто кроме вас! Таковы реалии коммерческой стороны существования промышленного производства программ.

Выстрел в темноте

В любой организации, в любом проекте и в любой момент оценки сроков разработки программного обеспечения представляют собой лишь основанные на фактах догадки – иначе они и не были бы оценками. Гадание, конечно, вызывает сомнение в профессионализме, но ничего лучшего не существует: вы никогда не узнаете точно, сколько времени требует задача, пока она не будет завершена, а к тому моменту эта информация обычно оказывается бесполезной.[1]

[1] За исключением того, что у вас появляется опыт для будущих оценок.

Качество оценки определяется главным образом глубиной понимания решаемой задачи, то есть тем, насколько вы *действительно* понимаете ее, а не вашим *представлением* об этом. Кроме того, точность оценки зависит от имеющегося у вас для ее проведения времени и объема труда, который вы готовы вложить в реалистичное проектирование и технико-экономическое обоснование. Когда спецификация точная, оценку можно выполнить быстро; если она расплывчатая, оценивание будет очень долгим. Для разумной и обоснованной оценки может потребоваться создание нескольких прототипов, чтобы изучить варианты реализации – выбор одного из них может оказать радикальное влияние на сроки исполнения и степень соответствующего риска.

Если времени для детальной оценки недостаточно, нужно остановиться на худшем варианте продолжительности разработки. Чем меньше труда вы вложите в оценивание, тем менее надежной окажется полученная вами цифра и тем большим расхождение реальности с оценкой. Разработка может завершиться в течение половины вычисленного вами срока, занять его полностью или потребовать дополнительного времени. Мы управляем этим риском, оставляя в плане разработки время для *непредвиденных обстоятельств*. Каким должно быть это время? Это тоже из области догадок. Ниже мы рассмотрим этот вопрос.

Оценка длительности разработки программного продукта представляет собой основанную на фактах догадку. Для каждой оценки есть некоторая мера вашей уверенности в ней.

Хорошие оценки обоснованны и оправданны, а плохие делаются наугад. Это стандартная техническая проблема, требующая проницательности и гибкости в управлении. Так было из века в век.[1] Менеджеры и планировщики имеют дело с оценками для проекта в целом. Это *исключительно* трудное занятие. Мы рассмотрим только оценки для отдельных задач программирования. К счастью, это не исключительно, а *просто* трудно.

Почему трудно делать оценки?

Я живу в Кембридже, в Соединенном Королевстве; моя семья живет в Бристоле. Оценка времени разработки похожа на оценку времени, которое мне нужно, чтобы навестить своих родственников. Легко определить длительность поездки при попутном ветре и отсутствии пробок. Но если на дороге ведутся работы, или есть пробки, или сломается машина, или я поздно выеду, или я поеду в часы пик, оценка становится менее надежной. Если я предвижу какие-то проблемы, то могу назвать вероятный интервал времени своего прибытия. Я знаю, сколько займет поездка в самых благоприятных обстоятельствах; у меня есть представление

[1] Пример из Библии есть у Луки 14:28!

о худшем сценарии (иногда мои поездки бывают ужасны). Я могу оценить время прибытия как промежуточное между двумя крайними моментами. Однако я никогда не могу предвидеть все случаи – если сломается машина, я застряну. Мобильный телефон в такой ситуации полезен: если я опаздываю, можно позвонить домой и предупредить, чтобы там подогрели для меня ужин (и не отдавали его собаке).

Аналогично обстоят дела с разработкой программ. При планировании создания программного продукта можно предвидеть потенциальные проблемы, учесть зависимость от сторонних поставщиков и оставить резерв на непредвиденные случаи. Можно сделать оценку длительности решения отдельной задачи для лучшего случая и рассмотреть худший сценарий. Разумеется, от неточности оценки будет зависеть уже не ваш ужин в кругу семьи, а судьба проекта, а возможно, и всей компании.

Такого рода примеры показывают, почему оценка продолжительности разработки так трудна и так важна. Есть масса факторов, осложняющих эту задачу:

- Нужно учесть множество параметров. Они связаны со сложной природой задачи, последствиями выбранной конструкции кода и существующей программной средой, в которую он должен вписаться. Некоторые из этих параметров меняются ежедневно.
- В предъявляемые к системе требования постоянно вносятся коррективы, увеличивающие объем задачи. Во время технико-экономического обоснования проекта новые задачи и требования пользователей выявляются с невероятной скоростью. Это осложняет задачу оценки – удовлетворить всех очень трудно (как с этим справляться, см. в разделе «Спецификация требований» на стр. 471).
- Нельзя дать точной оценки, не зная всего объема предстоящей работы. Возможно, предстоит переделывать существующие библиотеки, функций которых недостаточно, или выполнить рефакторинг имеющегося кода, чтобы обеспечить возможность его безопасного расширения. Если вы вовремя этого не заметили, ваша оценка окажется заниженной.
- Редкий проект начинается на пустом месте. Необходимо изучить существующую систему, и только после этого можно оценивать продолжительность работы. Обычно времени, отведенного для оценки, недостаточно, чтобы сделать это должным образом.

Слабое звено

Непредвиденные проблемы могут совершенно спутать ваши карты. Не так давно мой редактор связей не справился с размером получающегося исполняемого модуля, и я не смог запустить свой код, пока не разобрался с компоновщиком. В результате время разработки более чем втрое превысило первоначальную оценку.

- Если вы таких задач раньше не решали, то оценить длительность будет труднее. Прежний опыт был бы полезен для оценки.
- Многие проекты имеют внешние зависимости, которые могут оказаться сущим наказанием. Вы можете оказаться зависимым от поставки операционной системы, небольшой, но важной библиотеки, внешней спецификации и даже от клиента. Возможности управлять такими внешними поставщиками у вас нет, и ваша оценка исходит из того, что они выполнят свои обязательства в срок. Такие зависимости увеличивают риск задержек и должны тщательно контролироваться.

Делать оценки трудно. Но это не снимает с нас ответственности. Мы обязаны учесть то, что действительно можно предвидеть – типа дорожных работ или плохой погоды. Необходимо найти правильное соотношение между пессимизмом, оптимизмом и (где-то посередине) реализмом.

Оценка времени разработки программ представляет собой действительно трудную задачу. Избегайте недооценки объема предстоящей работы. Учитите возможные последствия неправильной оценки.

История на этом не завершается. Трудно не только правильно сделать оценку. Не менее тяжело пережить последствия.

- Оценки включаются в договорные обязательства и служат основой для определения сроков поставки продукта заказчику. Будучи раз установленными, эти сроки трудно изменить, а нарушение их чревато большими убытками.
- Трудно работать согласно оценкам, выполненным кем-то посторонним – если вы не уложились в срок, означает ли это, что вы оказались не на высоте задачи или оценка была неверна?
- Во время разработки часто возникают новые задачи, которые нужно учесть и вставить в график, отодвинув все остальное на более поздние сроки. Аналогичным образом недостатки спецификации могут обнаружиться только после начала реальной разработки. Изменения в спецификации влияют на объем необходимой работы и, следовательно, на оценку сроков ее исполнения.
- *Всегда* возникают неожиданные проблемы. Мелкие проблемы можно компенсировать некоторым увеличением интенсивности работы, чтобы не отстать от графика. Можно ведь месяц недосыпать, правда? Но крупные проблемы влекут такой дополнительный объем работы, что график может оказаться сбит.
- Оценка возлагает на вас *дополнительную* ответственность: вы не только должны разработать код, причем высокого качества, хорошо спроектированный и доступный для сопровождения, но и поставить его в срок, который сами назначили. Несчастные программисты!

Под давлением

Там, где занимаются промышленным производством программ, не размышляют и испытывают большой соблазн делать оптимистические оценки. Особенно уязвимы программисты, впервые участвующие в этой игре. Руководство требует от них сжатых графиков, объясняя, как это важно для получения новых контрактов, объявления о новых релизах, поддержания внутренней политической стабильности и т. п. Это все понятная и объяснимая реальность; ни одна компания не живет в вакууме, и держатели акций желают есть икру и пить шампанское.

Но нажим идет не только сверху. Его оказывает личная гордость программиста. Технари любят давать оптимистические обещания; мы заинтересованные люди и гордимся тем, что создаем, и сроками, в которые это делаем. Соблазнительно думать: «Ну, это не займет много времени!». Но есть существенная разница между быстрым наброском кода или прототипом и законченным, готовым к выпуску продуктом. Наши графики должны основываться на реальности, а не на идеальных пожеланиях.

Все (включая вас) хотят сокращенных графиков. Не обманывайте себя относительно того, что реально можно сделать за выделенное вам время. Не давайте нереальных обещаний, когда требуется предоставить код, готовый для поставки.

Нужно сознавать, что на вас оказывают нажим, и правильно реагировать на него. Опасна и противоположная позиция. Очень просто стать пессимистом, растягивать задачу до бесконечности и делать бессмысленно завышенные оценки. Опасность завышения оценок в том, что проекты занимают все время, которое для них отведено! Если есть несколько лишних дней, вы всегда найдете кусок кода, который можно отшлифовать за это время.

В идеале срок окончания проекта устанавливается *после* проверки технической осуществимости, которая устанавливает, что проект осуществим за приемлемые сроки. Но на практике так бывает редко. Вместо этого вам назначают срок («поставка к Рождеству»), и вы должны думать, как в него уложиться. Если объем работы не позволяет уложиться в срок, нужно обсуждать пути решения задачи: убрать какие-то функции, привлечь дополнительных разработчиков, отдать часть работы на сторону или договориться о том, что реализация части функций откладывается до будущей версии. Иногда такое планирование превращается в занятие маркетингом и требует немалой изобретательности!

А кто сказал, что это просто?

Рассказ бывалого человека

Компания только что получила самый крупный и стратегически важный заказ за всю свою пятилетнюю историю. От него зависело все будущее. Отдел продаж очень стремился заключить контракт и согласился на сжатые сроки, установленные заказчиком: продукт должен был быть поставлен до конца года. Подписали контракт и похлопали друг друга по спине.

Но ни у кого не хватило времени (или ума), чтобы посоветоваться с техническими специалистами и убедиться в том, что проект технически осуществим. Он не был технически осуществим. Менеджеры пришли в состояние паники, но ввиду невозможности изменить срок или объем функций им было трудно найти выход. Инженеры выражали недовольство и размахивали своими планами, но им было сказано: постарайтесь уложиться. Они напряженно работали день за днем и вечером допоздна и вскоре выбились из сил. С каждой неделей они все больше отставали от безнадежно нереального графика.

Последними отчаянными усилиями они завершили код в срок, но споткнулись на непредвиденной аппаратной проблеме, которая задержала проект на два месяца. В плане не было резерва на непредвиденные обстоятельства, который позволил бы пережить эту катастрофу.

Проект провалился, программисты выдохлись, нервы перенапряглись, клиент остался недоволен. Вскоре после начала нового проекта большинство программистов ушло.

Практические способы оценки

Как нам справиться с нарастающим давлением на нас с целью превратить программистов в пророков? Точность оценки, как и многие другие навыки, приходит с опытом. Не нужно ждать старости, но если не работать в условиях риска срыва графика и не ставить себе целей, к которым нужно стремиться, то вы не улучшите свое мастерство в этом деле. Совершенство достигается путем тренировки.

На практике у нас едва ли бывают тренировочные проекты или возможность поэкспериментировать с оценками времени. Но где-то на пути от младшего программиста к гуру должны же вы этому научиться! Жаль, что не существует магической формулы или простого рецепта получения оценки. Но вы значительно повысите свою точность, если выполните следующие шаги:

1. Разбейте задачу на минимально возможные блоки, фактически осуществив первый подход к проектированию системы.

2. Добившись хорошего деления на подходящие по обозримости части, сделайте оценки времени разработки каждого блока в *человеко-часах* или *человеко-днях*.
3. Определив все отдельные графики, сложите вместе их длительности – готово: у вас есть мгновенная оценка продолжительности проекта.

Эта стратегия действует, поскольку полностью охватить и точно оценить ряд небольших задач проще, чем сделать то же для одной гигантской задачи. *Никогда* не следует делать оценки в более крупных единицах, чем человеко-дни: крупные единицы показывают, что вы не вполне вникли в задачу и не можете дать надежную оценку. Безжалостно разбивайте крупные задачи, пока не достигнете мелких блоков работы, доступных для оценки.

Оценки времени надо применять к небольшим задачам, объем работы которых легко понять. Единицей измерения должны быть человеко-часы или человеко-дни.

Конечно, разработка часто осуществляется несколькими людьми параллельно; разбив ее на мелкие понятные части, мы можем по-разному распределять задачи и определять возможность их одновременного выполнения, приближая срок общего завершения. Это уже задача планирования проекта.

Оставьте себе достаточно времени для проведения оценки. Необходимый для нее общий проект появляется не сразу; не думайте, что вам легко удастся угадать сроки. Вы просто будете обманывать себя, если станете брать оценки с потолка, не основывая их на прежнем опыте или умении проектировать системы.

Важно учесть *все* виды работ, которые потребуется провести для поставки программного продукта. Это означает, что выделения времени требуют следующие задачи:

- Тщательное и продуманное проектирование
- Необходимые исследования или создание прототипов
- Фактическое написание кода
- Отладка
- Создание тестов модулей
- Интеграционное тестирование
- Написание документации
- Исследования или обучение, которые могут потребоваться во время проекта

Из этого перечня следует, что на написание кода может быть отведено меньше времени в сравнении с другими второстепенными работами. Программирование состоит не только из написания кода; не забудьте учесть в своих оценках тестирование и создание документации. Это

важные виды работ. Без тестирования и документации вы выпустите код, который будет работать с ошибками, и никто не сможет его исправить, потому что неизвестно, как им пользоваться.

Не нужно рассчитывать *полное время* (путем учета отвлечений на другие проекты, чтения почты, серфинга в сети, перерывов на кофе и отправления естественных надобностей). Оно обязательно будет сильно отличаться от фактически потраченного на решение задачи. Задача может выполняться параллельно с другими или прерываться, чтобы обеспечить выполнение другого проекта. Мы учтем это в плане проекта (см. раздел «Игры с планами» на стр. 517).

Насколько консервативной должна быть оценка? К чему следует склоняться – к оптимизму или пессимизму? Правильный ответ: оценка должна быть реалистичной. Готовьтесь к возможным проблемам и учитывайте их, но не нужно придумывать 1000 причин, по которым простая задача может затянуться, и делать из них предлог для завышения оценки. Не завышайте оценки ради перестраховки или возможности полодырничать, раскладывая пасьянс. Оценки отдельных задач не должны учитывать все возможные неполадки. Управление риском нужно проводить на *уровне проекта*; планировщик составит на основании наших оценок разумный план с резервом на случай непредвиденных обстоятельств.

Чтобы сделать оценки более точными, учтите следующие важные обстоятельства:

- Чем более конкретным и определенным является проект, тем проще делать оценки. Вам дали хорошую спецификацию?

  Без спецификации нет отслеживаемости, и значительный объем работы в каждом пакете оказывается предположительным. Но два человека могут исходить из разных предположений относительно области, охваченной проектом, и ожидать на выходе разные результаты. Строгая спецификация решает эту проблему.

  Вовремя изготовить не ту систему, которая требовалась, нисколько не лучше, чем изготовить то, что нужно, но с опозданием. Если спецификации нет, напишите ее сами и добейтесь одобрения у ответственных лиц.[1] По крайней мере, нужно документировать все допущения, сделанные во время работы.

- Чем больше функций затребовано, тем труднее сделать оценку. Постарайтесь отказаться от всей ненужной работы. Отличный метод – поэтапная поставка продукта и оценка времени для каждого этапа.

  Сообщайте данные оценивания наверх. Тогда ответственные за проект смогут сопоставить важность каждой функции со сложностью ее реализации. В результате станет видно, какие мелкие затребованные функции удвоят время разработки.

[1] Конечно, для этого потребуется время, которое вы не учли в плане!

- Если проблема в целом вам не вполне ясна, ваша оценка может оказаться очень неточной. Потратьте время и выясните точно, что должен делать программный продукт. Если требуется дополнительное время для оценки, попросите об этом либо выразите свою уверенность во временных параметрах. Никогда не гадайте, рассчитывая на удачу – если не можете обосновать оценку, не делайте ее.
- Если решение задачи зависит от третьих лиц, оценка осложняется. Кто отвечает за то, чтобы добиваться от третьей стороны своевременной поставки? Может потребоваться учесть это обстоятельство при оценивании сроков разработки. Выясните срок поставки заказа третьей стороной и добавьте в оценку время на его интеграцию со своим кодом (никогда не рассчитывайте, что он сразу «встанет на место»). Подумайте, в какой мере можно доверять третьей стороне, и включите соответствующий резерв времени на непредвиденные обстоятельства.
- Разные люди выполняют одну и ту же задачу с разной скоростью. Это естественно; у каждого свой уровень мастерства, опыта, уверенности и степень отвлечения на другие дела (например, ранее начатые проекты или домашние проблемы). Нужно измерить скорость своей работы и хорошо понимать задачу, за которую вы беретесь. Оценка должна быть персональной.

Различайте, чью работу вы оцениваете по срокам: свою (над системой, которую хорошо понимаете) или чужую (того, кому может потребоваться изучение системы).

- Не поддавайтесь давлению свыше делать оптимистичные оценки. Не обещайте нереалистичных сроков в расчете на их выполнение с помощью сверхурочной работы. Имейте обоснованный ответ для менеджеров, требующих сократить сроки.
- Самое главное, *никогда* не планируйте заранее работать сверхурочно.

Простое средство уточнить свои оценки – это попросить помочь их провести. Если вы не понимаете задачу, найдите того, кто понимает, и попросите высказать свое мнение. В книге Джеймса Шуровьески (James Surowiecki) «The Wisdom of Crowds»[1] описано, как большие группы людей оказываются умнее немногочисленной элиты. Идя таким экстремальным путем, предложите всем разработчикам из своей команды дать грубые оценки всех задач в плане и потом усредните их. Такая оценка может оказаться близкой к истине!

Не занимайтесь оцениванием в одиночестве. Выясните мнение других людей, чтобы уточнить свои оценки.

[1] Джеймс Шуровьески «Мудрость толпы». – Пер. с англ. – Вильямс, 2007.

Игры с планами

Несколько не связанных между собой оценок времени никому не интересны. Их нужно объединить, создав нечто полезное: план проекта, с помощью которого можно управлять графиком разработки. В соответствии со своими отдельными оценками времени задачи размещаются на графике работ и распределяются между разработчиками. Выявляются зависимости между задачами и учитываются в плане (очевидно, что зависимые задачи нельзя начать ранее, чем будут завершены те, от которых они зависят). Окончательный результат представляет собой диаграмму, в которой по горизонтальной оси отложено время, а задачи размещаются параллельно, примерно как на рис. 21.1 (вариант классической *диаграммы Ганта*).

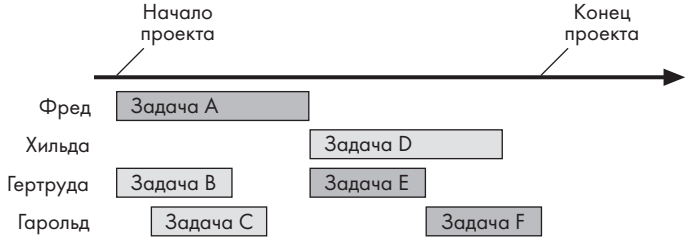


Рис. 21.1. *Диаграмма Ганта*

При составлении плана проекта происходит распределение заданий между разработчиками и составление графика работ. Но это сделать просто. Важная задача – *управление рисками*: создание надежного и разумного плана, учитывающего неопределенность и скрытые ловушки.

Надежные планы проектов:

Сокращают *критический путь*

Это одна цепочка примыкающих вплотную задач, проходящая от начала до конца проекта и показанная темными прямоугольниками на приведенной выше диаграмме. Задержка в любой из этих задач тормозит все зависящие от нее задачи и отодвигает окончательное завершение проекта.

Критический путь в плане есть всегда – по определению. Вот отчего седеют составители плана проекта! Наша цель – оптимальное взаимное расположение задач, обеспечивающее кратчайший (или связанный с наименьшим риском) критический путь.

Не злоупотребляют параллелизмом

Часто при планировании крупных проектов ошибочно полагают, что чем больше программистов бросить на задачу, тем быстрее она будет решена. Так бывает редко. При управлении большим количеством людей возникают существенные дополнительные расходы:

больше каналов общения, больше объем координации, больше точек возможного срыва. Эта тема обсуждается в классическом труде Брукса «Мифический человеко-месяц». (Brooks 95)

Нельзя чрезмерно распараллеливать план проекта, как не следует распараллеливать отдельных разработчиков. Если поручить одному разработчику выполнять два задания одновременно, нельзя ожидать, что это займет у него столько же времени, как при последовательном решении этих задач. Кажется очевидным, но на практике часто происходит иначе: вам могут предложить осуществлять поддержку старого проекта и одновременно начать работу над новым. Значительное время тратится на переключение с одной задачи на другую, что снижает суммарную эффективность. Если делать две вещи одну за другой, вам понадобится меньше времени (хотя это может не соответствовать бизнес-целям организации).

Не слишком длинные

Не очень хорошо, если план проекта слишком длинный. Одна мелкая проблема на критическом пути – и весь проект оказывается под угрозой.

В такой ситуации выгодно применять итеративную и инкрементную разработку (см. раздел «Итеративная и инкрементная разработка» на стр. 545), разбивая крупные графики разработки на более мелкие и менее рискованные итерации, которыми проще управлять. Это делает план более динамичным; фактически он воссоздается заново в каждый момент поставки. Такой подход является более безопасным и позволяет выявлять проблемы на ранних стадиях разработки, но он требует большего суммарного объема работы. Многим менеджерам это не нравится – им нравится иллюзия, создаваемая схемой «водопада», не допускающей отклонений.

В хороших планах не просто пригнаны вплотную отдельные оценки. В них учитываются реалии программного производства и предусматриваются средства снижения рисков. Для этого берутся во внимание следующие факторы:

Отпуска

Продолжительность отпусков, предоставляемых каждому разработчику, известна заранее и должна быть учтена в графике. Необходимо также учесть общественные праздники и рождественские каникулы во всех компаниях. В среднем на одну неделю работы программиста приходится полдня отпуска.

Загрузка

Чтобы быть реалистичным, план должен учитывать обычные отвлечения (совещания, учебу, болезни и т. д.). Как правило, для каждого разработчика это учитывается в плане в виде загруженности на 80%. Для тех, кто пользуется большим спросом, загруженность оказывается меньше. Нужно честно это учитывать, иначе «популярные»

программисты начинают отставать от графика, несмотря на то, что напряженно работают, и вскоре теряют веру в свои силы.

Непредвиденные обстоятельства

Это крупная проблема. Нужно учесть проблемы, которые едва видны на горизонте, и обеспечить зазор для непредвиденных неприятностей, которые могут помешать вам завершить работу в срок. Тут вступает в действие управление рисками.

Риском лучше всего управлять здесь, на уровне проекта, а не в оценках отдельных задач. В плане разработки можно оставить место для потенциальных проблем и принимать обоснованные решения, имея перед глазами процесс разработки в целом. Альтернативный выбор – пессимистические индивидуальные оценки, включенные в общий план, – приводит к завышенной продолжительности.

Возникает главный вопрос: сколько времени зарезервировать на непредвиденные обстоятельства? Можно просто умножить срок на три и назвать это учетом непредвиденных обстоятельств! Хороший путь – присвоить каждой задаче коэффициент надежности и, основываясь на нем, включить в план дополнительные псевдозадачи для наиболее рискованных работ как «аварийный запас». Назначить им длительность, пропорциональную продолжительности исходной задачи и коэффициенту надежности.

Интеграция

Задача не кончается с завершением написания кода и тестирования. Оставьте достаточно времени на объединение всех компонент и тестирование системы в целом. Потребуются отладка и решение проблем, которые обнаружатся только при стыковке компонент (например, недостаточная производительность или несоответствие интерфейсов).

Поддержка

Чем дольше работает человек в организации, тем чаще к нему могут обращаться с просьбами поддержать старые проекты, ответить на сообщение клиента об ошибке и т. д. Учтите эти обстоятельства при планировании загрузки программистов, отметив потребность в них других проектов.

Когда ключевых специалистов разрывают на части, проекты начинают потихоньку отставать.

Уборка

Выделите в конце своего плана время на приборку. В битве за выпуск программного продукта в срок часто срезают углы. Это называют также накопившейся *технической задолженностью*. Несмотря на все проповеди относительно правильного проектирования и хорошей практики кодирования, это неизбежное зло; это вполне прагматично. Однако нужно выделить время для уборки и приведения кода в опрятное состояние. В противном случае очередной круг раз-

работки будет основываться на кривом и неработоспособном коде. Если оставить эту проблему без внимания, растущая куча временных заплаток станет тяжелым грузом для ваших программистов.

Считайте эту работу частью *предыдущего* задания (хотя она выполняется после того, как прошел день выпуска продукта), а не как начало следующего. Отдайте свой долг тому проекту, который его создал.

Не считайте такую приборку необязательной данью; она представляет собой важную часть проекта. В сумасшедшей обстановке программного производства на факультативные задачи, помещенные в конец графика, просто никто не обращает внимания. Тщательно следите за этими задачами.

Составляйте графики работ так, чтобы оставалось время на приведение кода в порядок. Планируйте возврат технической задолженности.

Глубокое изучение составления планов проектов выходит за рамки данной книги; это большая и сложная задача. Но необходимо разбираться в ее базовых принципах. Нужно уметь разрабатывать программы в соответствии с планом и знать, что кладется в его основу, чтобы хорошо разбираться в том, что от вас требуется.

Существует много моделей планирования – формальных способов делать основанные на фактах догадки. *PERT* (*Program Evaluation and Review Technique – система планирования и руководства разработками*) – это классический метод планирования, разработанный в 1950-х годах в ВМФ США. Он напоминает мой расчет времени прибытия при поездке в Бристоль. Для каждой задачи делается три оценки в зависимости от вероятности поставки в срок: лучший сценарий, худший и наиболее вероятный. Это связывается с процедурой планирования, в которой определяется критический путь и вычисляется время завершения проекта в лучшем и худшем случаях. Чем больше разрыв между двумя оценками для задачи, тем больший риск с ней ассоциируется. В таком случае она может потребовать более тщательного руководства или выделения для своего решения более опытного сотрудника.

Конструктивная модель затрат Боэма (*COCOMO – Constructive Cost Model*) появилась в 1981 году и представляет собой модель оценки, основанную на анализе реальных программных проектов. На ее базе появилась модель *COCOMO II*, более точно отражающая природу современных программных проектов. (Boehm 81) *Проекты в контролируемых средах* (*Projects in Controlled Environments, PRINCE*) – это пример классической британской бюрократии, воплощенной в виде управления проектом; если бы можно было потребовать стояния в очередях, это обязательно было бы сделано![1] Эта модель охватывает весь жизненный

[1] Стояние в очередях – любимое времяпрепровождение британцев наряду с чаепитием и игрой в крикет.

цикл проекта – с начала и до конца. Процесс планирования PRINCE состоит из семи ступеней, включая проектирование плана, оценивание и составление графика и выполнение плана. На базе этой модели была развита новая модель с оригинальным названием *PRINCE2*.

Не отставай!

Как получается, что проект опаздывает на год?
... Это происходит по одному дню.

Фредерик П. Брукс

Когда работа отстает от графика и приближается срок сдачи продукта, программисты работают очень напряженно, но им не доверяют. О необходимости строгого тестирования напрочь забывают в безумном стремлении сдать что-то готовое в срок. Основной причиной этого цирка являются неверные оценки. Они способствуют созданию у менеджеров неправильных представлений о сложности задач разработки – откуда им было знать, что график составлен некорректно? Когда мы даем оценку срока, мы не должны ошибиться.[1]

При наличии реалистичных оценок длительности задач программирования есть несколько способов уложиться в график и избежать спешки последних минут:

- Приступая к новой задаче, проверьте, реально ли ее выполнение в заданные сроки – особенно если у вас не было возможности самому их оценить. Даже если оценку давали вы сами, проверьте ее заново. Не бросайтесь сломя голову к текстовому редактору в надежде, что выдержите срок; проверьте, что действительно сможете представить результат вовремя. Немного трезвой проверки в начале, и вы убережетесь от последующих мук и неприятностей.
- Поглядывайте на график – это важно. Постоянно следите, сколько времени вы потратили в сравнении с запланированным. Ведите учет времени и держите его под руками. Самостоятельно оценивайте всякие промежуточные задачи, не включенные в основной план, и ведите свой мини-проект. Если вы вовремя проходите свои личные контрольные точки, больше шансов не отстать от внешнего расписания. Пересматривайте свой список хотя бы раз в день.

---

[1] Забавно, что хорошие оценки тоже могут стать причиной этой проблемы. Демарко и Листер вспоминают реальный случай, когда руководитель проекта доложил о своей 100-процентной уверенности в своевременном и не выходящем за рамки бюджета завершении проекта (DeMarco 99). Менеджеры, опешившие от таких хороших новостей, решили перенести окончательный срок на более раннее время! Как бы ни был хорош инженер, всегда найдется еще лучший менеджер, который погубит весь его тяжелый труд!

Все зависит от планирования

Команда разработчиков все росла, и в помещении, где мы работали, стало *весьма* тесно. После долгих поисков было найдено новое место, и в пятницу нам сообщили, что с понедельника мы работаем в новом офисе. В течение выходных все компьютеры, серверы, кабели, роутеры, принтеры и т. д. должны были быть погружены в транспорт и перевезены на новое место. Нас уверили, что переезд пройдет без проблем и утром в понедельник все будет готово.

Утром в понедельник мы прибыли на новое место, и действительно все было установлено и отлично работало! Вся инфраструктура IT была в порядке. Серверы были подключены к сети и полностью функционировали. Все рабочие места были настроены. Огромная работа!

Тем не менее возникла небольшая проблема: не было стульев. Ни единого. Их как-то не учли в плане переезда, потеряли и нигде не могли найти. Стульев не было три дня.

И это называется планированием.

Если вы выясните, что не сможете уложиться в срок, как можно раньше сообщите об этом, чтобы можно было скорректировать план. Так же как я звоню заранее по телефону при поездке в Бристоль, лучше всего обнародовать этот факт как можно раньше. Если об опасности задержки становится известно заранее, можно принять плановые решения, минимизирующие влияние задержки.

На практике это случается очень редко. Когда над проектом работает пять программистов и они должны докладывать о своих успехах, никто не хочет первым сообщить, что не укладывается в график. Это называют *бесшабашным отношением к графику*. В результате все, казалось бы, идет хорошо, и вдруг обнаруживается существенное запаздывание проекта. Задержка каждый раз была на один день, но никто не хотел в ней признаваться. Разбейте этот порочный круг и дайте сигнал тревоги как можно раньше.

Постоянно сверяйте свои успехи с планом. Тогда не случится внезапно обнаруженного отставания.

- Делайте только то, что необходимо, и ничего лишнего. Очень интересно бывает добавить интересную новую функцию. Не делайте этого. Есть более важные – и запланированные – задачи. Предложите, чтобы важные дополнительные функции включили в план позднее, если в них нет *настоятельной* потребности в данный момент. Если я при поездке в Бристоль выберу окружной маршрут, я не смогу

приехать вовремя, даже если буду ехать среди красивых пейзажей, поэтому я рассудительно выбираю прямую дорогу.

- Тщательное проектирование с применением модульности обычно сокращает зависимость между компонентами и потому уменьшает пагубные последствия задержек и заторов в расписании. Согласуйте интерфейсы компонент на раннем этапе и сделайте заглушки вместо компонент, чтобы не задерживать разработку из-за неготовности других частей системы.
- Пишите добротный код с полным набором тестов для модулей. Для опытного мастера это должно быть очевидно! В результате вы резко снижаете время на отладку и сопровождение.

  Не забывайте оставить после кодирования время для составления документации и тщательного тестирования. Не допускайте, чтобы в последние несколько дней перед датой окончания требовалось в срочном темпе писать код. Необходимо оставить время для проверки работоспособности кода. В противном случае отладка выйдет за отведенные вам пределы времени. Если вам не хватает времени на проведение всей этой работы, сообщите об этом и добейтесь продления срока. Нельзя пропускать эту работу – отрицательные последствия скажутся позднее.
- Следите за тем, чтобы изменения в требованиях и спецификациях не выбили вас из графика. Если изменения неблагоприятны для сроков, немедленно об этом сообщите. Не принимайте молча все изменения в функциональности.
- Жестко относитесь к отвлечениям. Не принимайтесь за другие задачи, если они не учитываются в плане. Научитесь говорить *нет* старым проектам, новым заданиям из других отделов и вмешательствам по телефону и электронной почте.

  Берегитесь отвлечений извне, даже если они короткие и невинные с виду; они в состоянии понизить качество вашей работы. Нужно время, чтобы *войти в зону* – то продуктивное состояние, когда вы полностью сосредоточены на задаче и код слетает с кончиков ваших пальцев (так называемое состояние *потока*). Даже короткие отвлечения выводят вас из этого состояния, и требуется время, чтобы вернуться в него, когда вы снова обращаетесь к своей работе. Фактически отвлечения отнимают у вас втрое больше времени, чем их чистая длительность. (DeMarco 99)

  Развивайте такую культуру разработки, которая благоприятствует своевременному выполнению работы. С уважением относитесь к мыслительным процессам коллег. Проводите совещания только в случае реальной необходимости – не отвлекайте разработчиков на бессмысленные сборища, отнимающие время.
- Поддерживайте позитивный и оптимистичный подход. Если думать, что проект обречен, так оно и окажется на деле.

Резюме

Удача – это оценка бездельником успеха работяги.

Неизвестный автор

Оценка времени работы и планирование помогают нам выпускать коммерчески успешное программное обеспечение. Однако не существует точных методов оценки продолжительности разработки. Поэтому приходится довольствоваться *оценками*.

Стремитесь совершенствовать свои методы оценивания и следите за потенциальными проблемами, которые могут сорвать ваш тщательно разработанный план разработки. Научитесь работать согласно графику и определять, когда ваши графики нереальны.

Хорошие программисты...

- Делают хорошие оценки, учитывая все составляющие процесса разработки и основываясь на разумном разделении системы на компоненты
- Стремятся выпускать отлаженный код с полной документацией и надлежащей интеграцией в отведенных временных рамках
- Выявляют проблемы со сроками на раннем этапе, чтобы их можно было своевременно решить

Плохие программисты...

- Делают оптимистичные оценки, основываясь на догадках и внутреннем голосе
- Могут выдать код в установленные сроки, не успев отладить его и придать законченный вид
- Считают, что сообщить о возможности не уложиться в срок – признак слабости, и отчаянно пытаются догнать график, а когда это не удается, выглядят глупыми и измученными

См. также

Глава 13. Важность проектирования

Надежные оценки сроков исполнения могут основываться только на крепком начальном проекте кода.

Глава 19. Спецификации

Чтобы сделать правильную оценку, нужно четко определить объем работ, который недвусмысленно должен быть отражен в *спецификации*.

Глава 22. Рецепт программы

Методологии разработки определяют, как нужно состыковывать задачи и включать их в план проекта.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 678.

Вопросы для размышления

1. Как спасти отстающий проект и привести его в соответствие с графиком?
2. Как правильно реагировать на то, что конечный срок вам устанавливают до того, как проведено технико-экономическое обоснование или планирование?
3. Как обеспечить реальную пользу плана разработки?
4. Почему программисты работают с разной скоростью? Как отразить это в плане?

Вопросы личного характера

1. Какая доля проектов, в которых вы участвовали, была выполнена в срок?
   a. Для удачных проектов: чем был обусловлен успех планирования?
   b. Для неудачных: в чем заключались главные проблемы?
2. Насколько точными бывают ваши оценки? Насколько далекими оказываются результаты от ваших оценок?

VI

Вид сверху

Воздух более разрежен, но видно все лучше. Несколько сот страниц назад мы начали свое путешествие на самом нижнем уровне, копаясь в подбрюшье создания исходного кода. В этой последней части мы завершаем свое путешествие, взбираясь на самую вершину разработки программ и обозревая низлежащие территории. Надеюсь, вы не боитесь высоты? Теперь мы посмотрим, как сложить последние части головоломки.

Глава 22. Рецепт программы

Поваренная книга для написания кода: как действительно пишутся программы в командах. Обсуждаются методологии и процессы разработки программ. Показано, как нам удается выпускать программы предсказуемым и своевременным образом (или хотя бы пытаться это сделать).

Глава 23. За гранью возможного

Взгляд на существующие стратегии написания кода: программирование приложений, игр, распределенное программирование и т. д. В каждой из отмеченных областей есть свои проблемы и особо важные качества. Зная о них, вы сможете писать код, наиболее приемлемый для каждого случая.

Глава 24. Что дальше?

Конец виден... Посмотрим, куда двигаться дальше в изучении мастерства кодирования. Все еще только начинается.

22

Рецепт программы

Методологии и процессы разработки кода

В этой главе:

- Стили программирования
- Процессы разработки программного обеспечения
- Их влияние на наш код

Говорят, что время все меняет;
на практике приходится делать это самому.

Энди Уорхол

Ингредиенты:

1 пучок программистов (желательно, свежих)

1–2 чайных ложек языка

1 целевая платформа

1 менеджер проекта

1 щепотка удачи

1 обезвоженное обучение

Модные в профессии словечки по вкусу

Способ приготовления

Замариновать программистов путем обучения. Добавить языки, целевую платформу, приправить менеджером проекта. Быстро перемешать. Модные словечки по вкусу. Равномерно сбрызнуть удачей и поставить в горячую духовку до

окончательного срока. Вынуть, вывалить и дать остыть, перед тем как подавать клиенту.

Мне известно, по крайней мере, четыре рецепта бисквита. Их различия определяются вашими предпочтениями – хотите ли вы бисквит без масла или бисквит без яиц – и способом приготовления. Программы пишутся таким же образом. Не существует рецепта или магической формулы; одну и ту же систему можно построить разными способами, каждый из которых может быть не лучше другого. Можно выбрать разные ингредиенты для процесса разработки и разные методы. Так или иначе, в результате могут испечься немного различающиеся пироги – в отношении функций, структуры, стабильности, расширяемости, сопровождаемости и т. п. Рецепты описывают *жизненный цикл программного продукта*: этапы разработки от первоначального (замысел программы) до окончательного (прекращения эксплуатации).

Мы, программисты, должны предсказуемым (и в какой-то мере воспроизводимым) образом создавать программное обеспечение, следуя определенной процедуре. Мы должны привлечь себе на помощь такую процедуру разработки, которая позволит нам создавать лучшие из возможных программ. В этой главе мы изучим некоторые рецепты создания программ; мы будем их сравнивать, противопоставлять, критиковать и смотреть, какое влияние они оказывают на создаваемый нами код.

Мы писали программы для ZX spectrum не так, как для современного наладонного PDA, и не так, как систему биржевого контроля для мэйнфрейма с высокопроизводительным веб-интерфейсом. В одиночку мы программируем иначе, чем в паре, и иначе, чем в команде из 200 человек. Выбор рецепта определяется различиями в целевой платформе и команде разработчиков (и их уровне мастерства). Искусство программирования не ограничивается работой с редактором, компиляцией, сборкой и запуском.

Хорошие программисты знают, как программировать – владеют методами и приемами работы.

Каковы же эти рецепты программирования?

Стили программирования

Стиль программирования описывает, как планируется решение задачи, как оно делится на части и моделируется на выбранном языке. Мы должны создавать *модели* решений, потому что полезная система не может целиком уместиться в голове разработчика. Стиль программирования определяет, каким образом мы разбиваем проект на управляемые части; это парадигма проектирования, служащая выражению задач вашего кода.

Разным языкам программирования соответствуют разные стили программирования. Одни стили подогнаны под конкретный язык, другие

подходят для нескольких. Стили программирования делятся на две главные категории: *процедурные* и *описательные*.

- Процедурные языки позволяют точно задать последовательность шагов, которые приводят к получению результата. Это то, к чему привыкло большинство программистов.
- Описательные языки описывают связи между переменными на языке правил вывода (или функций), а для получения результата исполнительная система языка применяет к этим правилам какой-то фиксированный алгоритм. (Это описание может стать более понятным после того, как мы рассмотрим функциональное и логическое программирование.)

Выбор языка программирования отчасти определяет стиль. (Лучше все же выбирать язык, поддерживающий тот стиль, который вы хотите использовать.) Однако выбор языка программирования – не самое главное. Вполне можно писать структурированный код на объектно-ориентированном языке, точно так же, как можно написать отвратительный код на любом языке. В нескольких последующих разделах описываются популярные стили программирования.

Структурное программирование

В этом стандартном методе процедурного проектирования применяется *алгоритмическая декомпозиция* – процесс разбиения системы на части, каждая из которых представляет собой небольшой шаг в более крупном процессе. Проектные решения направлены на поток управления и создают иерархическую функциональную структуру. Как писал Дейкстра: «Иерархические системы обладают тем свойством, что сущность, считающаяся неделимой на одном уровне, рассматривается как составной объект на следующем, более низком уровне детализации; в результате естественные гранулы пространства или времени, применимые на каждом уровне, уменьшаются на порядок, когда мы переключаем свое внимание на очередной более низкий уровень. Мы говорим о стенах в терминах кирпичей, о кирпичах – в терминах кристаллов, о кристаллах – в терминах молекул, и т. д.» Так Дейкстра популяризировал структурное программирование в своей классической статье «Go To Statement Considered Harmful» (Оператор Go To вреден). (Dijkstra 68)

Структурное программирование – это модель, сосредоточенная на управлении и следующая нисходящей модели проектирования. Вы начинаете с того, что задумываете целиком всю программу (например, `сделать_покупки`), потом разлагаете ее в серию подблоков (например, `составить_список_покупок`, `выйти_из_дома`, `дойти_до_магазина`, `выбрать_покупки`, `расплатиться_в_кассе`, `вернуться_домой`). В свою очередь каждый подблок раскладывается на части, пока не будет достигнут такой уровень, на котором легко написать реализацию в коде. Блоки собираются в целое, и на этом проект заканчивается.

Структурный подход имеет следующие последствия:

- Каждый шаг декомпозиции должен быть в пределах разумного понимания программистом. (Дейкстра сказал: «Я предлагаю ограничиться проектированием и реализацией программ, понимаемых разумом».)
- Поток управления нужно тщательно контролировать: избегайте ужасного оператора `goto` (неструктурированного перехода в произвольное место программы) и пишите функции с одним входом и одним выходом (так называемый *код SESE*).
- Структурность кода обеспечивается циклическими конструкциями и условными операторами внутри функциональных блоков. Отношение к досрочным выходам в середине цикла или из вложенного блока такое же негодующее, как к `goto`.

Распространенные языки структурного программирования – C, Pascal, BASIC и более древние, типа Fortran и COBOL. С помощью большинства процедурных языков легко пишется структурный код, хотя это не их специализация; структурные программисты часто принимают новомодные языки, не принимая новых идиом.[1]

Объектно-ориентированное программирование

Буч так описывает ОО-программирование: «Метод реализации, при котором программы организованы в виде взаимодействующих между собой объектов, каждый из которых представляет собой экземпляр некоторого класса, а классы образуют иерархическую структуру, основанную на отношениях наследования.» (Booch 94) Это тоже процедурный стиль, но он позволяет более естественным способом моделировать реальность; мы направляем свое внимание на моделируемые взаимодействующие сущности, а не на конкретный поток выполнения.

Эта модель в значительной степени строится вокруг данных (в отличие от структурного программирования, где в центре находятся процессы). Мы интересуемся здесь жизнью данных и их перемещением, а не последовательностью действий, которые нужно выполнить, чтобы решить задачу. У объектов (данных) есть поведение (они что-то делают) и состояние (которое изменяется в результате действий). На уровне языка это реализовано в виде *методов классов объектов*. ОО-программы рассматриваются как наборы взаимодействующих программных компонент, а не монолитные списки команд ЦП. ОО-проектирование позволило нам эффективно моделировать крупные системы.

В объектно-ориентированном программировании применяются следующие понятия компьютерной науки:

---

[1] Это необязательно плохо, если только программист не считает, что вышел за границы структурного кодирования, не меняя своего способа разработки кода.

Абстракция

Абстракция – как искусство избирательно уделять внимание – позволяет проектировать код так, что верхние уровни управления могут игнорировать конкретные детали реализации. Не все ли равно, что происходит при обращении к `get_next_item` – бинарный поиск по дереву, выбор элемента массива или телефонный звонок во Франкфурт? Важно, что возвращается очередной элемент (чем бы он ни был), и вызывающему коду больше ни о чем беспокоиться не нужно.

Представление Дейкстрой иерархии (вернитесь к нему и прочтите снова) открыло определенный вид абстракции.

Инкапсуляция

Инкапсуляцией называют помещение связанных друг с другом исполняемых блоков в один пакет, доступ к которому осуществляется *только* через четко определенный API: капсулу для кода. Пользователи этой капсулы могут лишь обращаться к заданному API, но не имеют прямого доступа к внутреннему состоянию. Это обеспечивает четкое разделение обязанностей, дает возможность порассуждать над метафизическими вопросами типа «*Что есть объект?*» и дает некоторые гарантии того, что никакой злоумышленник не сможет поковыряться в вашем коде, когда вы отвернетесь.

Наследование

Это механизм для создания объектов, являющихся особой версией родительских. Пусть родительский тип называется `Фигура`, а наследуют ему `Квадрат`, `Круг` и `Треугольник`. Наследники обладают более детальным и специализированным поведением (например, они знают, сколько сторон у фигуры). Как и любые идеи в программировании, наследование может служить основой для создания непонятных и странных программ, а может применяться в логически правильном, элегантном коде. Хорошие ОО-программисты умеют создавать эффективные наследственные иерархии.

Полиморфизм

Это свойство позволяет одному и тому же коду действовать с данными разных типов (которые в ОО-языках обычно называются классами), используя контекст, в котором выполняется код. Эта технология особо выделяет программирование согласно явно заданным интерфейсам, а не неявным реализациям – полиморфизм обеспечивает четкое разделение обязанностей при написании кода. Полиморфизм бывает двух видов: *динамический* и *статический*.

Динамический полиморфизм в соответствии со своим названием определяет фактическую операцию на этапе исполнения – в зависимости от типа операнда или целевого объекта. При этом часто используется иерархия наследования: клиент, работающий с типом `Фигура`, может в конкретном случае работать с объектами типа `Квадрат` или `Треугольник`, что определяется во время прогона.

Статический полиморфизм определяет, какой код нужно выполнять, на этапе компиляции. В число функций языка, обеспечивающих статический полиморфизм, входят: *перегрузка функций* (функции с одинаковыми именами принимают разные списки параметров, и компилятор определяет по аргументам, какую именно функцию нужно вызвать), *перегрузка операторов* (можно определить определенные операции для разных типов, в том числе +, !=, < и &; эти функции будут вызываться в зависимости от типа операндов) и *общие средства программирования* типа *шаблонов C++* (они позволяют перегружать шаблон в зависимости от типа параметров).

Все эти средства можно применять и в языках, не являющихся объектно-ориентированными, с помощью специальных приемов. Однако ОО-языки явно выражают их и используют для создания связанных систем.

Объектно-ориентированное программирование впервые появилось в языке Simula примерно в 1970 году, а затем было популяризировано с помощью C++ и Java. К числу немногих чисто ОО-языков программирования принадлежит Smalltalk. В нынешние времена ОО-программирование в моде, и существует много других ОО-языков; часть из них – процедурные языки с модными ОО-довесками.

Функциональное программирование

Это описательный стиль, основанный на *лямбда-исчислении*, модели программирования, привлекающей математику. Работа происходит с величинами, функциями и функциональными формами. Функциональные программы обычно компактны и элегантны, хотя редко компилируемы. Они, следовательно, зависят от среды исполнения. Эффективность программ зависит от этих сред – они бывают достаточно медлительны и требовательны к памяти.[1]

Структурированный и объектно-ориентированный стили гораздо популярнее, чем описательные языки, но это не уменьшает полезности последних. У них есть свои достоинства и области применения. Функциональные программы требуют совершенно иного подхода к проектированию кода, чем процедурные методы.

Распространенными языками функционального программирования являются Lisp (хотя в нем есть нефункциональные элементы), Scheme, Ml и Haskell.

---

[1] Это проблема не только описательных языков (например, в Java есть исполнительная среда, JVM). Однако исполнительные среды описательных языков были относительно слабо оптимизированы, поскольку во многих случаях поддерживаются академическими учреждениями, а не богатыми корпорациями.

Логическое программирование

Это еще один описательный стиль, который предполагает наличие среды исполнения, набора аксиом (логики) и целевого утверждения. Набор встроенных правил вывода (над которыми у программиста нет власти) применяется с целью определить, достаточно ли аксиом для установления истинности целевого утверждения. Выполнение программы, по существу, служит доказательству целевого утверждения.

Интерес к искусственному интеллекту послужил громадным ускорителем разработки логических языков программирования. Они широко применяются для автоматического доказательства теорем и в *экспертных системах* (которые моделируют большие предметные области и генерируют ответы на основании накопленных знаний).

Самый известный из логических языков программирования – Prolog.

Рецепты: как и что

Мы изучим два разных аспекта. В «рецептах» программирования участвуют как *процедура разработки*, так и *стиль программирования*. Это разные, но взаимосвязанные вещи:

- Процесс разработки представляет собой крупный план: он описывает шаги, которые нужно сделать для создания программного продукта. Он охватывает *всю* организацию, где ведется разработка, а не только программистов. Как правило, создание программ – работа, в которой участвует много людей; процесс показывает, как организовать массу людей, чтобы они создали какое-то согласованное целое. По крайней мере, он должен попытаться это сделать.
- Стиль программирования представляет собой мелкий план: это лежащий ниже способ расчленения, создания и склеивания компонент. Скорее всего, на него окажет влияние выбор процесса разработки, хотя это необязательно.[1] Более вероятно, что влияние окажут целевой язык и предшествующий опыт разработчика.

Оба эти аспекта разработки могут называть *методологиями*, из-за чего их легко спутать.[2] Мы уже рассмотрели стили программирования, а да-

---

[1] Например, ОО-стили часто выбираются в «итеративных и инкрементных» процессах, что обычно связано с традициями. (Если термины непонятны вам, не пугайтесь: все будет разъяснено в разделе «Итеративная и инкрементная разработка» на стр. 545.)

[2] Если вас интересует разница, то я называю стилями программирования то, что часто называют методологиями (со строчной «м»). Процессы разработки часто называют Методологиями (с прописной «М») – типа высокой и низкой церкви. Это слишком тонко. В данной главе я буду говорить о стилях и процессах.

лее обсудим процессы разработки. Важно овладеть разными имеющимися методами разработки, чтобы расширить свой кругозор и правильно выбрать процесс, если вам представится возможность выбора.

Наши программные проекты определяются теми стилями и процессами, которые мы применяем. Они неизбежно отражаются на форме и качестве кода.

Следующие разделы не являются учебником по этим темам, а представляют собой достаточно беглый обзор, позволяющий их сравнить. Если вас заинтересуют подробности, найти соответствующий учебник будет нетрудно.

Процессы разработки

Существует много процессов разработки, поскольку есть люди, которые любят их изобретать. Многие из них лишь незначительно развивают одну-две базовые модели. Эти базовые варианты мы здесь рассмотрим. Как вы убедитесь, некоторые из них тесно связаны между собой.

Выбор вами процесса разработки определяет планирование проектов, переход работы из одной фазы в другую и взаимодействие между членами команды, осуществляющей проект. Различие между процессами проявляется по разным направлениям:

Толстый/тонкий

Толстый процесс разработки тяжел и бюрократичен. Он создает массу бумажных документов, регламентирует поведение разработчиков и предполагает определенную структуру команды. Его характеризует организационная модель ISO 9000, в которой каждая производственная процедура раболепно расписана во всех деталях, несмотря на изъяны или неуместность процесса.

На другом полюсе располагаются *тонкие процессы разработки*, избегающие ненужной бюрократичности и благоприятствующие более скромным и ориентированным на людей принципам. Такую практику поддерживают процессы ускоренного программирования, описанные на стр. 547.

Последовательность событий

В некоторых процессах разработки разумно учитывается, что реальность непредсказуема, и предпринимаются попытки учесть это в планах путем выполнения нескольких итераций в цикле процесса. Это дает возможность разработчикам учесть в новой итерации результаты, полученные на предыдущей. Они могут приспособиться к естественным изменениям, происходящим по мере разработки продукта (новые требования клиентов, непредвиденные проблемы и т. п.).

Другие процессы более регламентированы и линейны и полагаются на формальный переход разработки от одной стадии к другой. Они предусматривают значительные затраты на предварительное планирование и попытки в деталях предвидеть будущее. Такие предсказания ограничивают возможность будущих коррекций направления разработки.

Направление проектирования

При *нисходящем проектировании* система создается на основе начального общего представления. Каждый пакет верхнего уровня уточняется и подвергается разбиению на составляющие. Этот процесс продолжается до тех пор, пока не будут получены спецификации продукта, позволяющие начать работу. В нисходящем проектировании велика роль планирования и хорошего представления о результирующей системе и предполагается, что в процессе разработки требования будут меняться мало.

В противоположной процедуре – *восходящем проектировании* – подробно описываются отдельные части системы, а затем отыскивается лучший способ соединить их вместе. При этом легче включить в новый проект уже существующие компоненты. В современной практике эти два подхода часто объединяются – для начального планирования создается некоторое представление о системе в целом, а затем происходит выявление и кодирование компонент и объектов низкого уровня.

Ни один из процессов разработки не лучше остальных. Существуют крайние точки зрения относительно правильного выбора позиции по каждой из этих осей. Правильная методология для любого проекта определяется рядом факторов, включая культуру разработки, существующую в организации, тип разрабатываемого продукта и опыт команды разработчиков.

Теперь пристегните ремни безопасности, и мы начнем наше головокружительное путешествие по процессам разработки программного обеспечения.

Ad Hoc

Это исходная точка, но на самом деле это анти-процесс. Здесь нет никакого процесса либо он не документирован. Каждый работает по собственному плану, никто не знает, чем занимаются остальные, и можно только надеяться, что в итоге получится что-то полезное. Может быть, и ваша команда работает так, как показано на рис. 22.1?

Когда организация не знает, как она делает свои программы, это непростительно, даже если она малочисленна и считает, что ей не нужен процесс. При таком положении нет гарантий, что программы будут выпускаться вовремя, поскольку нет подотчетности. Кто ответит за то, чтобы были реализованы все функции?

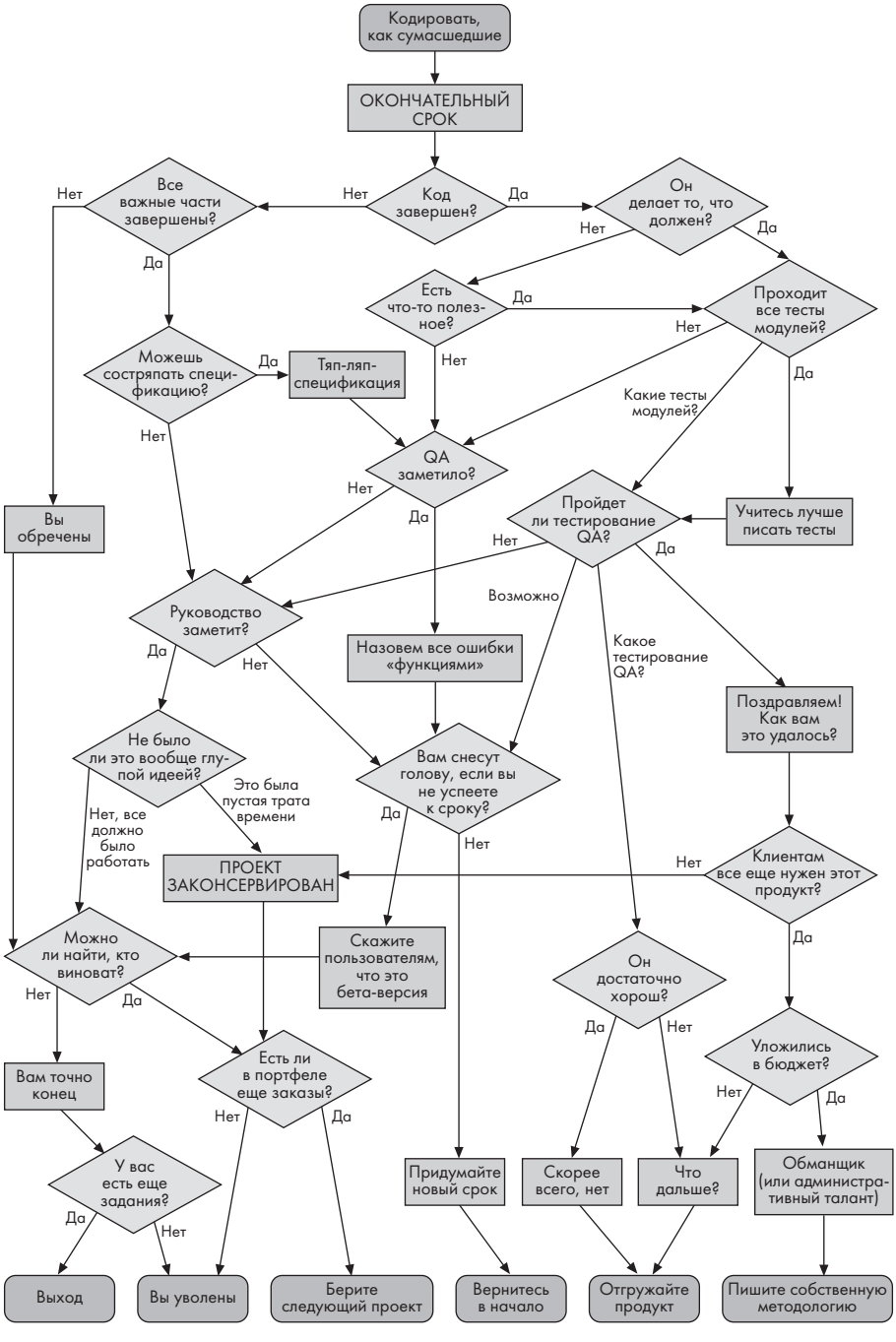


Рис. 22.1. *Путь к выпуску продукта*

Таким хаотическим способом создаются многие программы с открытым исходным кодом.[1] Если есть бесконечное число обезьян и бесконечное число компьютеров, то е́сть шанс, что получится программа, но ждать придется бесконечно долго. Даже эскиз, сделанный на салфетке, – это шаг в направлении более формального и предсказуемого процесса разработки.

Без процесса разработки команда находится в состоянии анархии. Программы появляются благодаря удаче, а не в результате целенаправленных действий.

Это случай анархии в разработке. Отдельные программисты могут старательно трудиться, и их героические усилия могут привести к появлению чего-то стоящего. Однако серьезно рассчитывать на такой результат не стоит. Вероятнее всего, команда окажется очень неэффективной и едва ли выпустит что-либо стоящее.

Каскадная модель

Каскадная модель представляет классический цикл разработки программных продуктов. Ее много критикуют за простоту (и даже за старомодность). Однако на ней в какой-то мере базируются практически все процессы разработки. У нее много недостатков, но, несмотря на них, это полезная отправная точка в изучении процессов. Она сделана по подобию более традиционного жизненного цикла в инженерном деле и была описана Ройсом в 1970 году (Royce 70). Это самый предсказуемый среди всех процессов разработки.

Идея проста: процесс разработки разбивается на несколько сменяющих друг друга стадий. Это напоминает водопад, поскольку поток устойчиво и неотвратимо движется от одного каскада к другому. Как вода в реке всегда течет в одну сторону, так и разработка последовательно проходит все стадии в направлении выпуска готового продукта.

Традиционная каскадная модель показана на рис. 22.2.[2] Вы видите пять обычный стадий, описанных во врезке «Стадии разработки». Когда одна стадия успешно закончена, с помощью некой *промежуточной процедуры* (обычно это обзорное совещание) осуществляется переход к следующей стадии. На выходе большинства стадий появляется

---

[1] Там это может не быть столь существенно, поскольку нет заказчика, который платит деньги, и формального набора требований – множество программ с открытым текстом разрабатывается, потому что этого хочется программисту. Однако хотя бы частичное введение методологии в проекты open source наверняка привело бы к улучшению программ.

[2] Это обычное упрощение первоначальной статьи Ройса. У Ройса *допускалось* движение в обратном направлении, но не очень поощрялось. Рьяные администраторы вообразили, что разработка программного обеспечения должна быть строго линейным процессом, и вскоре убрали эти пути вверх по течению; водопад был опорочен.

Стадии разработки

Каскадная модель описывает пять стадий процесса разработки программного продукта. Во многих других процессах выделяются те же фазы, но располагаются в них в другом порядке или меняют свое относительное значение.

Анализ требований заказчика

Сначала нужно определить требования к программному проекту. В них входят цели проекта, услуги, которые должен предоставлять продукт, и накладываемые на него ограничения. Этому шагу часто предшествует технико-экономическое обоснование, либо оно проводится одновременно. Обоснование должно дать ответы на вопросы: *Сможет ли* продукт работать? *Нужно ли* разрабатывать этот продукт? *Какие есть* альтернативы?

Проектирование и спецификация

Выявленные на первом этапе требования превращаются в требования к программному и аппаратному обеспечению. Затем требования к программному обеспечению преобразуются к такому виду, чтобы можно было легко реализовать их в компьютерной программе – вероятно, с помощью разбиения на отдельно разрабатываемые компоненты.

Реализация

На этом этапе создаются программы. Каждая программа или компонента является модулем и подвергается тестированию. Тестирование модулей должно обеспечить их соответствие спецификациям, определенным в предшествующей фазе.

Интеграция и тестирование

Все модули объединяются в единую систему, которая подвергается тестированию. Проверяются правильность интеграции кода, корректность работы системы в целом и ее соответствие системным требованиям. Если система успешно прошла тестирование, разработка считается завершенной.

Сопровождение

Наконец, продукт поставляется заказчику. Не думайте, что если продукт отгружен заказчику, работа над ним закончена; это наивно. Самую длинную фазу жизненного цикла программного продукта составляет его сопровождение (см. «Сопровождение существующего кода» на стр. 374). Приходится исправлять ошибки, реализовывать пропущенные и изменившиеся требования, а также выполнять другую работу по сопровождению продукта.

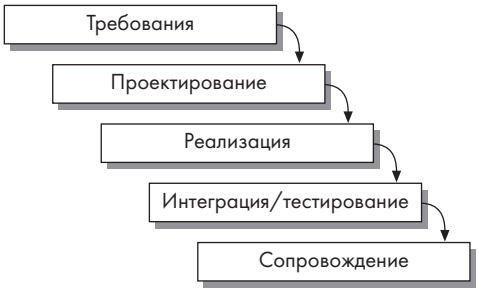


Рис. 22.2. *Традиционная каскадная модель*

какой-нибудь документ: спецификация требований, спецификация архитектуры или что-то аналогичное. Если во время проверки обнаруживаются ошибки или проблемы, о них сообщается разработчикам, и стадия откатывается в начало.

Следуя этой модели, непросто вернуться назад, чтобы внести изменения – как лососю нужно потратить массу времени и энергии, чтобы подняться вверх по течению. Но у лосося для этого есть генетическая программа, а у программиста ее нет. Это означает, что данный процесс не будет полезен, если требуется сделать изменения на поздних стадиях разработки. Требования должны быть зафиксированы до начала проектирования системы, а когда процесс уже пошел, модификации затруднены. Обычно проблемы проектирования обнаруживаются лишь на стадии тестирования системы.

В защиту этой модели нужно сказать, что она проста в управлении, по крайней мере теоретически, и служит основой большинства других моделей разработки. Каскадная модель плохо подходит для очень крупных проектов; она хороша для проекта длиной в пару недель. В иных моделях это учитывается и предлагается на протяжении проекта использовать несколько маленьких каскадных моделей.

SSADM и PRINCE

Хотя аббревиатура *SSADM* может вызвать подозрение, что участвовать в ней могут только совершеннолетние, на самом деле за ней скрывается *Structured Systems Analysis and Design Methodology – методология структурного системного анализа и проектирования*. Это структурированный и строгий метод на основе каскадного подхода – видимо, наиболее регламентированный из всех вариантов каскадной модели.

Эта модель охватывает анализ и проектирование, но не реализацию и тестирование, и является хорошо определенным открытым стандартом, широко применяемым в правительственных организациях Великобритании. SSADM состоит из пяти основных ступеней (каждая из которых подразделяется на множество мелких процедур), названия

которых достаточно содержательны, чтобы в данной книге не останавливаться на них подробно:

- Технико-экономическое обоснование
- Анализ требований
- Спецификация требований
- Разработка логического проекта
- Физическое проектирование

Модели *PRINCE* (*Projects In a Controlled Environment*) и последовавшая за ней *PRINCE2* были разработаны в 1989 и 1996 годах с целью заменить SSADM. Подобно SSADM, эти модели являются тяжеловесными и документо-центрическими. Они перечисляют регламентированные шаги (на этот раз восемь фаз), следуя которым можно изготовить продукт, который должен удовлетворить заданным требованиям и стандартам качества.

V-модель

Эта модель процесса основана на классической каскадной схеме и была разработана для управления процессами разработки программного обеспечения в правительственных и военных организациях Германии. Она имеет много общего с каскадной моделью (включая способность вызывать критику), но моделирует процесс не в виде каскада, а в виде буквы V, как на рис. 22.3.

Слева мы видим фазы разработки, ведущие к созданию программного продукта: планирование, проектирование и реализацию. Поток в правой части управляет тестированием и одобрением.[1] На каждом уровне тестирования происходит сравнение со спецификациями, созданными на соответствующей стадии слева.

Отличие V-модели от каскадной не только в ориентировке диаграммы. Этапы тестирования (в правой ветви) могут начинаться параллельно

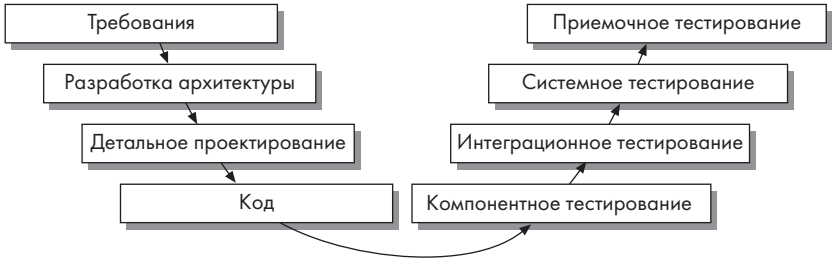


Рис. 22.3. *V-модель*

[1] Обратите внимание, что разработка течет в нижнюю сторону, как в водопаде, а тестирование оказывается восхождением в гору – неплохая модель разработки программ!

с разработкой (левая ветвь) и приобретают равное с ней значение. Это хорошо, поскольку:

- Обычно тестирование сокращают на заключительном этапе опаздывающего проекта. Это опасно. Этот факт отражается выделением тестирования как краеугольного камня процесса разработки, что помогает повысить качество продукта.
- Тестирование не должно ограничиваться конечным продуктом – необходимо рецензировать и проверять продукт на *всех* стадиях разработки, от анализа требований до законченной программы. В V-модели это отражено.
- На практике тестирование и исправление ошибок часто занимают больше половины общей длительности проекта. В каскадной модели это не находит точного отражения.
- V-модель может помочь сократить общее время разработки, потому что позволяет осуществлять этапы тестирования как только завершится очередная стадия разработки. Такая параллельная работа дает возможность приблизить завершение проекта, потому что не нужно ждать окончания фазы реализации каскадной модели, чтобы начать тестирование.

Создание прототипов

Несмотря на долгие исследования и большой опыт в процессах разработки ПО, каскадная модель остается эталонной благодаря своей четкой логике – очевидно, что разумной реализации не осуществить, пока не проведены анализ требований и проектирование. Однако в каскадной модели трудно сделать оценку программной системы до завершения разработки. Так же трудно показать продукт клиенту до того, как закончена интеграция системы и не началось ее тестирование.

Создание *прототипов* нацелено на то, чтобы обойти это ограничение. Прототипы помогают изучить и оценить реализацию в ходе разработки и уточнить требования к системе (пользователи никогда не знают, что им нужно).

Суть процесса прототипирования заключается в создании нескольких временных прототипов системы. Каждый прототип оценивается, показывается клиенту, и отзывы клиента кладутся в основу следующего прототипа. Это продолжается, пока не будет получено достаточно знаний для разработки и развертывания реального продукта.

Здесь заметна аналогия с другими видами промышленного производства. При разработке нового автомобиля создается много прототипов, пока не будет найдена правильная конструкция. То же самое мы хотим сделать с нашими программами. Однако есть важное различие, о котором нужно помнить. При создании автомобиля основных расходов требует производство, а не разработка. С программами все наоборот. Создание новых экземпляров кода ничего не стоит; затратную

часть составляет разработка. Поэтому цикл прототипирования нужно ограничивать; нельзя повторять его бессчетное число раз.

Прототипы быстро разрабатываются на языках верхнего уровня. Иногда их просто рисуют: применение автоматизированных инструментов[1] может невероятно ускорить создание прототипов. Прототипы нужны для доказательства правильности концепции, поэтому эффективность, стабильность или полнота функций не являются главными заботами при их разработке. В результате прототипы более эффективны для систем, в которых большое значение имеет интерфейс пользователя.

Прототипы позволяют управлять рисками. С их помощью мы убеждаемся, что клиентам действительно нужно то, чего они, по их словам, хотят. Можно также воспользоваться прототипами для исследования применений новой технологии или проверки применимости проектных решений в реальной обстановке.

С прототипами связан соблазн доработать неэффективный, наскоро сделанный и не вполне продуманный код до окончательной версии. Особенно характерно это для проектов, выбивающихся из графика, когда настоящая разработка не укладывается в сроки. Не имеющие специальной подготовки клиенты не видят разницы между прототипом и законченным продуктом и будут удивлены, что получить свой за-

Опасности прототипов

Выпуск прототипов в качестве продукта может вызвать серьезные проблемы сопровождения.

Одно время я сотрудничал с компанией, в которой существовала политика использования только одной библиотеки GUI для своих клиентов, написанных на Java. Но на практике в одних системах эта библиотека использовалась, а в других – нет. Когда обнаруживалась ошибка, программистам сопровождения было очень трудно разобраться в работе клиентского кода. Других библиотек GUI они не знали, и их исправления часто служили источником новых проблем. Чем чаще это происходило, тем сильнее продукты компании теряли репутацию.

Не пришлось глубоко копать, чтобы выяснить причину проблем: клиентские части, использовавшие не ту библиотеку GUI, которую нужно, были прототипами, лишь «по случайности» ставшие продуктами. Потратив какое-то время на выпуск правильного кода, можно было бы избежать многомесячной работы в будущем и сберечь репутацию компании.

[1] Например, *средства быстрой разработки приложений* (*RAD*) с простыми построителями GUI.

конченный программный продукт смогут нескоро. Для борьбы с этим требуется очень старательное руководство. Лучше всего решать эту проблему, намеренно делая свои прототипы незаконченными и не позволяя им приближаться к выпускаемому продукту. Прототип, в котором реализовано слишком много функций, это уже не прототип!

Итеративная и инкрементная разработка

В последнее время все атаки на каскадный метод представляют собой вариации на одну тему. Основное развитие заключается в выполнении разработки *итеративным и инкрементным образом*. Это подразумевает совершение многочисленных проходов (итераций) короткого цикла разработки с добавлением в каждом цикле новых функций, пока система не станет законченной. Каждый отдельный проход мини-цикла обычно следует каскадной модели и может длиться несколько недель или месяцев (в зависимости от масштабов проекта). Поэтому каждая фаза каскада проходится несколько раз. В конце каждой итерации есть готовый к выпуску продукт.

Инкрементная разработка не является ни нисходящей, ни нисходящей. В каждой выпускаемой версии присутствует полная версия кода со всеми необходимыми компонентами высокого и низкого уровней. При каждой итерации система развивается, и последующее проектирование может основываться на существующих проекте и реализации. Здесь наблюдается параллель с прототипами, но в данном случае нас не так интересуют скороспелые демонстрационные решения. При таком подходе все стадии проще и легче в управлении – и за прогрессом системы легче следить; вы знаете, какая ее часть уже построена и интегрирована.

Такого рода процесс хорошо действует в таких проектах, в которых вначале не очень понятны требования. Скажем прямо: это значительная часть всех реальных проектов. Здесь обеспечивается большая гибкость и можно избежать длительного повторного проектирования и реализации всей системы, с чем мы сталкиваемся при каскадном подходе. Итеративная и инкрементная разработка успешны, потому что соответствуют фундаментальной природе разработки программного обеспечения и помогают нам лучше контролировать свойственный ей хаос. Поскольку итеративные циклы оказываются гораздо короче, создается больше возможностей для обратной связи и коррекции: не нужно ждать конца проекта, чтобы узнать о его провале.

Спиральная модель

Спиральная модель, предложенная Барри Боэмом в 1988 году (Boehm 88), дает хороший пример итеративного и инкрементного подхода.[1]

[1] Процесс Боэма не был первой итеративной моделью, но он первым популяризировал и подчеркнул значение итерации.

Процесс разработки представляется в виде спирали, как на рис. 22.4. Спираль начинается в центре и разворачивается наружу в направлении более поздних стадий проекта. Работа начинается с очень грубого представления о системе, которое по мере перехода на последующие витки спирали становится более детализированным. При каждом обороте спирали на 360 градусов мы проходим один каскадный цикл, а каждая итерация обычно длится от шести месяцев до двух лет.

Функции определяются и реализуются в порядке убывания их приоритета: самые важные средства создаются как можно скорее. Это способ управления риском: так надежнее, поскольку, приближаясь к дню поставки, вы можете быть уверены в том, что большая часть системы завершена. На самом деле, это очень прагматичный подход: программисты не смогут тратить 80 процентов своего времени на пустячные (но интересные для них) 20 процентов системы.

Боэм разбивает спираль на четыре квадранта или четыре разные фазы:

Постановка задач

Определяются конкретные цели для данной фазы.

Оценка и уменьшение рисков

Выявляются и анализируются ключевые риски, ищется информация, позволяющая снизить эти риски.

Разработка и проверка

Выбирается подходящая модель для следующей фазы разработки.

***Рис. 22.4.** Спиральная модель*

Планирование

Происходит рецензирование проекта и составляются планы для очередного цикла спирали.

Методологии ускоренной разработки

Они появились как реакция на бюрократические и тяжеловесные методологии, сковывавшие процесс разработки программного обеспечения. Те, кто занимается ускоренным программированием, исходят из того, что разработку программного обеспечения трудно сделать предсказуемым процессом; они утверждают, что программирование сильно отличается от известных инженерных процедур типа проектирования моста.[1] Старомодные монументальные методологии только мешают тем, кто старается писать хорошие программы, и потому их нужно отбросить.

Ускоренные методологии – это собирательный термин для целого ряда процессов разработки, включая разрекламированное экстремальное программирование, а также Crystal Clear и Scrum. Ускоренные процессы сосредоточены на быстроте и сокращении рисков, а не на долгосрочном планировании и достижении (мнимой) предсказуемости.

Ускоренные процессы основываются на следующих догмах:

- Минимизировать риск путем выполнения множества небольших итеративных циклов разработки. В конце каждого цикла программное обеспечение и все, созданное в процессе, является полным, последовательным и имеет качество, пригодное для выпуска. Хотя выпуски программного обеспечения делаются редко, его можно передать клиенту для рецензирования и комментирования. Это дает клиенту уверенность в успешной работе команды.

  При ускоренном процессе разработки итерации обычно гораздо короче, чем циклы в итеративных и инкрементных процессах (обычно они длятся недели, а не месяцы).
- Минимизировать риск в результате придания большой важности комплекту автоматизированных регрессивных тестов, выполняемых непрерывно, вместо длительного цикла тестирования в конце разработки.
- Сократить объем документации, которой наводнены тяжелые процессы. В ускоренных процессах сам код рассматривается как проект и как документация по реализации. Хороший код самодостаточен и не нуждается в обременительных процессах бюрократического документирования.
- Подчеркнуть важность человеческого фактора и облегчить возможность общения – предпочтительно личного, а не посредством доку-

[1] Это религиозный спор. Ряд программистов уверен, что процесс разработки ПО можно сделать воспроизводимым и предсказуемым, но необходимых для этого зрелости и жесткости данная отрасль пока не достигла.

ментов. Заказчик (или его представитель) как можно более тесно сотрудничает с командой разработчиков, участвуя в принятии решений по реализации и назначению приоритетов.

- Считать мерой прогресса и эффективности *работающее программное обеспечение*, а не текст спецификации или мнение менеджера о месте команды в фиктивном цикле разработки. Разработчики решают проблемы и реагируют на перемены, модифицируя код по ходу разработки.

Ускоренный подход не всегда оправдан. Он эффективнее всего в маленьких проектах, где команды состоят не более чем из 10 высококвалифицированных программистов, находящихся в географической близости друг к другу. Ускоренные процессы хорошо показывают себя в областях, где значительно варьируются требования. В компаниях с культурой тяжелых процессов они приживаются с трудом.

Другие процессы разработки

Есть много других процессов разработки, являющихся вариациями на те же темы, но с характерными особенностями. Есть модифицированные каскадные процессы с перекрытием фаз или с вложенными проектами, управляемыми как миниатюрные каскадные. Метод создания *прототипа с эволюционным развитием* начинает с исходной концепции, проектирует и реализует прототип, итеративно его уточняет, пока он не становится приемлемым, и делает из него выпускаемую версию (в процессе возможно создание нескольких временных прототипов).

В *поэтапной поставке* осуществляется последовательный процесс вплоть до проекта архитектуры, после чего реализуются отдельные компоненты, которые по завершении показываются заказчику; затем при необходимости возвращаются к предыдущим этапам разработки. *Эволюционная поставка* представляет собой гибрид эволюционного создания прототипов и поэтапной поставки.

В ускоренной разработке приложений (Rapid Application Development, RAD) делается акцент на участие пользователя и небольшую численность команд разработчиков, активно применяются прототипы и средства автоматизации. В качестве некоторого реверанса в сторону других процессов временны́е рамки разработки устанавливаются заранее и считаются неизменными. После этого в проект включается столько функций, чтобы уложиться в заданный срок; некоторые функции могут оказаться принесенными в жертву.

Унифицированный процесс компании Rational (*RUP*)[1] – примечательная коммерческая методология от IBM, идущая от *Objectory Process* Айвара Джекобсона (Ivar Jacobson, 1987). Это тяжелый, но гибкий объектно-ориентированный процесс, активно применяющий UML-диа-

[1] Джим Арлоу, Айла Нейштадт «UML 2 и Унифицированный процесс», 2-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2007.

граммы, с проектированием на основе *прецедентов*[1] (прецедент описывает деятельность одного пользователя (*актера*) или его взаимодействие с системой). В нем применяются итеративная разработка, непрерывное тестирование и тщательный контроль изменений. Будучи коммерческим, он поддерживается рядом коммерческих инструментов.

Спасибо, хватит!

Если вы добрались до этого места и вам не наскучило читать, это хорошо. В конце концов, самое главное – какие выводы нужно сделать из всего этого? Программист-мастер должен на практике хорошо разбираться в процессах разработки и стилях программирования, но эти знания любой может почерпнуть из соответствующих книг. Как с пользой применить их в работе? Как они могут повысить наше мастерство?

Во всех этих процессах просматривается некоторая общность. В каждом есть этапы, описанные во врезке «Стадии разработки» на стр. 540. Действительное различие между процессами относится только к длительности и взаимному расположению этих стадий. Каждый вид деятельности необходим для создания программного обеспечения хорошего качества. Лучшие процессы гарантируют, что тестирование не оставляется напоследок, а осуществляется и контролируется непрерывно в течение всего процесса разработки.

Трудно сравнивать или оценивать разные процессы и стили программирования. Какой из них лучший? Какой гарантирует своевременную поставку продукта в рамках бюджета? Нельзя дать ответ, поскольку это неверные вопросы. Уместность того или иного процесса зависит от характера проекта и культуры вашей компании. Если у вас есть 20 программистов, которые понятия не имеют об объектно-ориентированном программировании и всегда пользуются только C, глупо будет пытаться строить OO-продукт на Java.

Выбор рецепта программы может определяться многими причинами, которые должны быть вескими. Мотивировка вашего выбора многое говорит о степени зрелости вашей организации.

Можно встретить две крайности процедуры разработки: анархию случайного метода и строгий режим регламентированного процесса. В последнем случае любые эксперименты, которые могли бы привести к появлению более изящной архитектуры, не поощряются. Реальные требования пользователя могут никогда не дойти до разработчика, потерявшись в бюрократическом процессе. Программист лишь пишет код в соответствии с переданной ему спецификацией, составленной на предыдущей стадии процесса.

---

[1] Мартин Фаулер «UML. Основы, 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2004.

Если следовать принципу *Златовласки*, то наиболее гибкий подход должен располагаться где-то посередине. Вы обязаны знать, по какому процессу работаете и где его описание. Эффективная разработка требует дисциплины: чтобы вовремя выпустить что-то на свет, нужна согласованная стратегия (наличие реалистичного графика работы – это отдельная история; см. «Игры с планами» на стр. 517). Опытные программисты знают, в чем ценность их процесса разработки, а также его недостатки. Они знают, как работать с ним и когда нужно выйти из него. Хорошие программисты не только программируют. У них есть свои рецепты, и они знают, как адаптировать их к обстановке. Вот почему наша наука все еще является ремеслом.

Важно не быть скованным и чрезмерно законопослушным в отношении принятого процесса, но у вас должна быть установленная схема для производства программного продукта. Она должна соответствовать вашей команде разработчиков – не каждой организации нужен излишне формальный процесс, требующий преодоления многочисленных препятствий и заполнения длинных форм.

Процесс, который вы принимаете, не должен быть слишком формальным и трудным в осуществлении. Обычно признаками хорошего процесса служат как раз противоположные характеристики. Однако определенный процесс должен быть.

Время от времени возникают новые методологии. Часто их встречают фанфарами и фейерверками; они претендуют на то, чтобы стать серебряной пулей, панацеей, которая облегчит разработку для наших детей и внуков. Увы, на практике этого не происходит. Если на то пошло, то какой бы жизненный цикл программы вы ни избрали, все зависит от качества программистов, входящих в команду. Если у них нет интуиции, чутья, опыта и мотивации, то, независимо от выбранного процесса, вы не сможете быть уверены, что выпустите хороший код. Зато вы сможете лучше следить за тем, насколько отстали от графика.

Выбор процесса

Есть много факторов, влияющих на правильный выбор процесса разработки. Однако выбор редко исходит из разумных оснований; процесс выбирают, потому что *мы всегда так работали*, *он достаточно хорошо работает* или *это первое, что пришло нам в голову*.

Как узнать, какой метод разработки лучше всего вам подходит? В конечном итоге, если процесс оказывается действенным – у вас в команде налажено сотрудничество и вовремя выпускается хороший продукт, – значит, вы правильно выбрали метод разработки.

Правильный выбор зависит от типа и масштаба проекта. Для небольших модификаций имеющегося базового кода не нужно большого ите-

ративного цикла разработки; для промышленных проектов длиной в три года и начинающихся с чистого листа – скорее всего, нужно. Когда процесс выбран правильно, он учитывает опыт членов команды, разработчики проявляют желание его применять, а менеджер проекта действительно понимает его суть.

С другой стороны, есть много ошибочных оснований для выбора процесса. Нет смысла переходить на новый процесс только потому, что хочется перемен; новый процесс нужно вводить, если текущая модель разработки столкнулась с проблемами. Нет смысла в попытках политических заявлений (я знаю тех, кто пытался культивировать в своей организации открытую разработку, чтобы подтолкнуть ее к раскрытию базового кода и выводу его в open source). Самая скверная мотивировка при выборе конкретного процесса – это следование моде. Популярные новомодные словечки не всегда свидетельствуют о преимуществах предлагаемого процесса.

Нужно иметь в виду, что неправильный выбор процесса может гибельно сказаться на качестве кода; вы потратите больше времени на соблюдение всех процедурных формальностей, чем на разработку программы. Хороший процесс не мешает работе. Напротив, он позволяет команде работать лучше и быстрее.

Выбор процесса имеет жизненно важное значение. Неудачные проекты в большинстве случаев оказываются такими не по техническим причинам. Почти всегда одной из главных причин оказывается плохая организация процесса.

Резюме

Создание программного продукта сродни преступлению: оно удается, когда хорошо организовано. Случается, что команда без всякой дисциплины выдает нечто примечательное и создает программный шедевр. Но это исключение. Чтобы процесс разработки был эффективным, он должен быть определен, а также понят и поддержан членами команды, обладающими достаточной квалификацией. Иначе полученный программный продукт окажется безобразным.

Для того чтобы создавать программы в соответствии с графиками, бюджетами и изменяющимися требованиями, необходимо применять опробованные процессы разработки и показавшие себя стили проектирования. Создавать программные продукты трудно, и мы рассмотрели еще один способ того, как облегчить этот процесс.

Хорошие программисты...

- Понимают стиль программирования и процесс разработки, в рамках которых им предстоит работать

Плохие программисты...

- Игнорируют вопросы процесса разработки и пытаются делать вещи по-своему

Хорошие программисты...

- Применяют свой процесс разработки для формирования отношений с другими участниками программного производства; если процесс становится тормозом, они обходят его
- Учитывают плюсы и минусы различных методов разработки и умеют выбрать тот, который лучше всего подходит в конкретной ситуации

Плохие программисты...

- Не понимают, как процесс влияет на их отношения с другими разработчиками
- Стараются не думать о таких проблемах – *это заботы менеджеров*

См. также

Глава 8. Время испытаний

Тестирование – ключевая фаза процесса разработки. Часто им пренебрегают в стремлении сдать продукт своевременно.

Глава 17. Вместе мы – сила

Совместная работа – краеугольный камень разработки крупных программных продуктов.

Глава 19. Спецификации

Спецификации часто служат мостиками между разными фазами процесса разработки.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 681.

Вопросы для размышления

1. Какое влияние оказывают друг на друга стиль программирования и процесс разработки?
2. Какой стиль программирования лучше?
3. Какой процесс разработки лучше?
4. Укажите для каждого из перечисленных в этой главе процессов разработки его место в классификации, рассмотренной на стр. 536.
5. Если процессы разработки и стили программирования являются рецептами, как должна выглядеть поваренная книга разработки программного обеспечения?
6. Можно ли, правильно выбрав процесс, сделать разработку программного обеспечения более предсказуемой и повторяемой задачей?

Вопросы личного характера

1. Каким процессом разработки и языком программирования пользуетесь вы в данное время?
   a. Было ли по этому вопросу достигнуто формальное соглашение между участниками разработки или действуют традиции?
   b. Как происходил выбор? Это выбор для данного проекта или рецепт, которым вы пользуетесь всегда?
   c. Существует ли какая-то документация по данному вопросу?
   d. Соблюдает ли команда требования процесса? Если возникают проблемы и вас прижимают к стене, продолжаете ли вы придерживаться процесса или все теории забываются, когда нужно хоть что-то представить в срок?
2. Считаете ли вы правильным текущий выбор процесса и стиля? Это лучший для данного момента способ разработки?
3. Признается ли в вашей организации, что существуют другие модели разработки, достойные рассмотрения?

23

За гранью возможного

Различные области программирования

В этой главе:

- Сравнение областей программирования
- Как эффективно работать в вашей области
- Особые навыки, требуемые в каждой области

Все, что раздражает нас в других,
может помочь нам понять себя.

Карл Юнг

Я люблю широкие обобщения и незамысловатые метафоры. Можете меня за это осуждать. Кроме того, я провел некоторое исследование. Я обнаружил, что в городе, где я живу, больше 40 церквей. Все они немного отличаются одна от другой. В них ходят люди разного типа, и они занимаются разными вещами. У них разные проблемы и способы работы. Они находятся в разных местах. Тем не менее все они делают примерно одно и то же.

Какое все это имеет отношение к программированию, спросите вы? Если вы позволите мне такую незамысловатую аналогию, то разработка программных продуктов происходит во многом таким же образом. Согласен, мы ходим колоннами к некоторому зданию каждое воскресное утро (большинство

из нас). Но посторонним действительно кажется, что мы совершаем какие-то странные ритуалы и загадочные обряды по отношению к вещам, которые неподвластны обычным человеческим существам.

В действительности, я привел это сравнение, чтобы показать, что не существует единственного способа программирования и единственной методологии, которая решает все проблемы. Нет единственного языка программирования. Существуют различные классы проблем, которые нужно решать в многочисленных областях. Работа в каждой из таких областей отличается не только технологией (т. е. инструментами и библиотеками кода), но и техникой. Каждая из них требует специфического мастерства, определенного склада ума и несколько различных способов работы. Различия могут показаться незначительными, но ничто не заменит конкретный опыт программирования определенного вида систем; в противном случае предложения работы для программистов были бы гораздо более общими. Очень важно хорошо знать свою область и понимать ее особые проблемы. В каждом конкретном разделе программирования мастер должен знать свое ремесло, уметь работать с материалом и отлично владеть инструментом.

Существуют разные виды программирования в различных предметных областях. В каждой области есть свои особые проблемы и требуются специфические умение и опыт.

В данной главе мы обсудим эти вопросы. Мы проведем экскурсию по широкой области, каковой является программирование, расскажем о некоторых предметных областях, для которых чаще всего пишем программы, отметим их различия и рассмотрим конкретные задачи и проблемы каждой из них.

Некоторые из этих сфер перекрываются. Это естественно. Границы всегда оказываются не такими четкими, как хотелось бы. Последующие описания неизбежно будут носить общий характер, поскольку все это обширные области с массой вариаций внутри. Тем не менее вы должны получить представление о том, что в них творится.

Программирование приложений

Это то, что представляет себе большинство гуманитариев, когда вы произносите слово *программирование*.[1] По-видимому, это самая крупная из рассматриваемых в данной главе категорий.

Имеется в виду программирование *приложений* – законченных программ – как правило, для однопользовательских компьютеров типа рабочей станции. В этой области в центре внимания находятся пользова-

[1] Но только не вы, конечно. Если на вечеринке признаться, чем вы занимаетесь, можно сразу погубить разговор. Если, конечно, там не собралась толпа сисадминов, но тогда вы и сами, скорее всего, постараетесь убежать!

тели и их работа с настольными системами. В силу коммерческих причин целевыми платформами обычно являются самые распространенные – в настоящее время Windows и Mac OS. Хотя сейчас много говорят о программировании для Linux, это все еще не та область, в которой сосредоточена работа над приложениями (по крайней мере, в данное время). По мере того как портативные устройства становятся все более мощными, а среды разработки приложений для них все богаче, разработка мобильных приложений переместилась из области встроенных приложений (см. раздел «Встроенное программное обеспечение» на стр. 563) в данный класс программирования приложений более общего назначения. Особые препятствия на пути разработки встроенных приложений в значительной мере устранены.

Для такого рода работы есть много языков и сред. Стандартно используются C и C++. Мы также часто встречаем Visual Basic и Delphi, Java и .NET, а также ряд библиотек и структур типа MFC и Qt. Выбор делается в соответствии с тем, что удобнее для разработчиков – хорошо им знакомо и обеспечивает все необходимые функции.

Современное программирование приложений достигло больших успехов по сравнению с началом эры персональных вычислений. У нас появились богатые функциональными возможностями среды разработки и полезные структуры кода, автоматизирующие утомительные стандартные действия. Мы располагаем поддержкой потоков исполнения, библиотеками стандартных компонент интерфейса пользователя и средствами, обеспечивающими сетевую прозрачность. Операционные системы предоставляют значительную поддержку, облегчающую программирование приложений, но это означает также, что до начала программирования нужно многое изучить. Нужно очень много знать, чтобы *действительно* понимать происходящее вокруг.

Вся эта дополнительная поддержка поднимает тот уровень, которому должна соответствовать хорошая программа. То, что считалось приемлемым поведением программы несколько лет назад, сегодня считается недопустимым. Клиент рассчитывает получить высококачественную надежную программу со стандартным интерфейсом и внешним видом, быстрой реакцией, дружественностью к пользователю (доступностью для самого неподготовленного пользователя) и массой функций (даже если на практике пользователю нужна лишь их малая часть). Продаваемые сегодня огромные профессиональные приложения являются результатом работы больших команд разработчиков и подразделений, специально занимающихся вопросами юзабилити.

Мы наблюдаем сдвиг в сторону систем, базирующихся на веб – приложений, выполняемых в броузерах посредством сетей. Мы рассмотрим их отдельно; отчасти это перекрывается со сферой распределенного программирования (см. раздел «Программирование распределенных систем» на стр. 566).

Есть два главных рынка программирования приложений: коробочные (shrink-wrap) продукты и заказные приложения (custom applications).

Коробочные продукты

Коробочные продукты разрабатываются для массового рынка. Они используются большим количеством людей – по крайней мере, в мечтах отделов маркетинга. В этом разгадка: рынок спекулятивен, поэтому продукт должен оказаться привлекательным для возможно большей части покупателей, чтобы принести доход. Так как ни один покупатель не заказывает разработку коробочного продукта и не оплачивает ее, определить выгодный рынок нужно еще до начала работы, иначе вы зря потратите время и силы. Программный продукт должен отличаться от конкурентов по функциональности, производительности или оригинальности подхода к проблеме.

Коробочный продукт можно купить в магазине аккуратно завернутым в целлофан или загрузить через Интернет. Это даже может быть веб-сервис, предоставляемый по подписке. Главное – как он продается и как это влияет на разработку.

Жизнь программистов коробочных продуктов нелегка. Вы не знаете, в какой среде будет выполняться код. Он должен хорошо себя вести со всеми версиями операционной системы, на машинах различной конфигурации, при наличии в системе других установленных библиотек и приложений, с которыми должен успешно ладить. Для тестирования это кошмар! Программистам веб-приложений вдвое легче (как вы увидите позже): у них есть контроль над сервером. Но и им приходится сражаться с проблемами совместимости броузеров: веб-страницы должны корректно отображаться на многочисленных целевых платформах.

Заказные приложения

Заказные приложения разрабатываются по специальному заданию для конкретного клиента. Поэтому здесь не столь важны привлекательный интерфейс, бесконечное количество функций или даже совершенство и отсутствие ошибок. Нет коммерческой необходимости этим заниматься. Нужно заставить программу работать. Поставить клиенту. Получить деньги. Это более надежная бизнес-модель.

Раз клиент заказывает такую программу, он будет работать именно с этой программой и ни с какой другой. В отсутствие реальной конкуренции этой программе требуется минимальный уровень качества. Дай программистам волю, и они станут ковырять и вылизывать свой код, пока он не достигнет какого-то фантастического уровня совершенства. Но в данной ситуации нет коммерческого смысла это делать. Нет ничего страшного в том, что программа будет отлично работать, но раз в неделю зависать; дешевле периодически запускать ее заново, чем пускаться в долгие поиски ошибки (если, конечно, при аварии не теряются какие-либо данные).

Краткая информация

Работать над приложениями – удовольствие.
Современные ПК обладают такими характеристиками,
что не нужно особенно беспокоиться о размере кода
или производительности и можно сосредоточиться
на написании аккуратного и элегантного кода.
Мысль о том, что твоим приложением пользуются
десятки тысяч людей во всем мире, приятно возбуждает.

Стив (прикладной программист в крупной компании)

Типичные продукты

Типичные коробочные продукты – это настольные приложения, такие как веб-броузеры, электронные таблицы и т. п. Заказные программы могут быть какими угодно, например система управления складированием для крупной компании розничной торговли, сделанная по индивидуальному заказу.

Целевая платформа

Обычно это такая же машина, как та, на которой вы занимаетесь разработкой (как правило, это x86 Windows PC).

Среда разработки

Обычно код разрабатывается на такой же рабочей станции, на которой будет выполняться программа. Современные интегрированные среды разработки (IDE) обеспечивают удобное рабочее окружение, объединяя редактор, компилятор, отладчик и справочную систему под единым интерфейсом, управляемым мышью. Существуют многочисленные дополнительные компоненты сторонних поставщиков, которые упрощают разработку стандартных задач. Применяется широкая гамма языков: от низкоуровневых C/C++ до BASIC и Java, а также языков сценариев.

Типичные задачи и проблемы

Пользователи рассчитывают получить высококачественные программы, придерживающиеся стандартных принципов интерфейсов. Требованием времени является такое количество функций, которого не упомнить ни одному пользователю. Это серьезная коммерческая необходимость и то, чем зачастую отличается один продукт от другого. Обычно новые версии продуктов содержат больше функций (и ошибок), чем решают проблем. Таково требование рынка.

Программирование игр

Захватывающий и чарующий мир программирования игр – это специфическая форма разработки приложений, обычно в коробочном виде. В значительной мере борьба ведется с помощью подкупающего маркетинга и очень удачной идеи игры. Это тонкая граница между выдающимися, успешными играми и теми, которые не вышли в призеры.

В этих играх часто создаются грандиозные 3D-среды от первого лица, обеспечивающие полный эффект присутствия. Чтобы создать захватывающие ощущения, графические возможности аппаратуры исполь-

Краткая информация

Профессиональная разработка игр – это очень интересно, но здесь существует высокая конкуренция, и разработчикам приходится следить за новейшими технологиями, укладываться в немыслимые сроки и удовлетворять не подлежащим обсуждению требованиям, меняющимся в последнюю минуту. Кровью, потом и слезами пишутся программы, встречающие суровый прием у публики и жесткую критику в специальной прессе. Однако эта работа может приносить глубокое удовлетворение – в результате вы создаете нечто такое, что видят и понимают другие люди, что доставляет им радость.

Тадеус (профессиональный программист компьютерных игр)

Типичные продукты

От первого лица, полный эффект присутствия, 3D-игры, стратегии, логические сетевые игры (пазлы).

Целевая платформа

Настольные ПК, игровые консоли, мобильные устройства (PDA и мобильные телефоны), аркадные машины.

Среда разработки

Для специализированных игровых платформ (в том числе старших моделей графических карт в ПК) созданы среды разработки, помогающие использовать их мощь. Тем не менее для того чтобы извлечь из платформы все, на что она способна, нужны очень талантливые разработчики.

Типичные задачи и проблемы

Получить отличное качество игры; сбалансировать возможности, реакцию пользователей, эстетические достоинства, атмосферу и сложность. Хорошая игра имеет фабулу, увлекающую игрока.

Для утилизации возможностей платформы требуется оптимизация.

зуются на полную мощь, а из ЦП выжимается все, что можно, для управления картами, противниками, пазлами путем серьезного моделирования физики движущихся объектов. Все это нужно координировать в реальном времени, что максимально использует аппаратные возможности. Большое значение в программировании игр имеет оптимизация кода в зависимости от платформы исполнения. По мере появления более быстрой аппаратуры проблема не снимается; чтобы выделиться среди других игр, нужна дополнительная оптимизация, чтобы создать более реалистичные ощущения на новых платформах. В этой области очень важно оставаться на переднем крае и применять новейшие технологии для достижения особых эффектов.

В современных командах, разрабатывающих игры, состав участников больше подошел бы для создания голливудских фильмов, а не обычной бухгалтерской программы. Тут и художники, и проектировщики уровней, и художники-раскадровщики, и концепт-дизайнеры.

Целевой платформой может быть достаточно навороченный ПК или специальная игровая консоль. Аппаратная часть таких машин может ускоренно осуществлять многие графические операции, а специальные утилиты управляют их мощью. Изготовители консолей предоставляют наборы для разработки (специальные версии аппаратуры и особые программные средства), облегчающие создание продуктов, загрузку кода, тестирование, отладку и позволяющие обойти аппаратную защиту, которая препятствовала бы разработке.

Игры со многими участниками предоставляют более богатые условия. Ко всему прочему добавляются сетевое взаимодействие и ухищрения с целью обеспечить приемлемое время реакции на низкоскоростных соединениях Интернета.

Качество конечного продукта определяется атмосферой игры. Доводится до ума все, что может положительно повлиять на восприятие игры: разделение на уровни, физические модели, графика, цвет нижнего белья. Не упускается ничего. Можно написать прекрасный код, программа никогда не будет зависать, будет делать все, что от нее требовалось, и работать очень эффективно. Но если в игре нет особой искры, делающей ее захватывающей, успеха ей не видать. Хитрая штука.

Системное программирование

Приложения работают поверх богатых системных библиотек: различных уровней кода, обеспечивающих сетевое взаимодействие, графические интерфейсы, многозадачность, доступ к файлам, мультимедийные возможности, управление периферийными устройствами, связь между процессами и т. д. Если прикладные программисты получают существенную поддержку от системы, то кто-то должен снабдить систему этими средствами поддержки. В этом и состоит системное программирование.

Краткая информация

Я написал стек протоколов шины USB для закрытой ОС. Я должен был изучить операционную систему, аппаратуру USB и USB-протокол, так что разбираться пришлось со многими вещами. Я позаботился о производительности, чтобы система хорошо работала. Выполняя функции посредника, я абстрагировал аппаратные интерфейсы и создал аккуратный API, которым можно было воспользоваться в приложениях. Дополнительные сложности были вызваны необходимостью сделать все независимым от платформы.

Дэйв (разработчик системных компонент)

Типичные продукты

Операционные системы, драйверы устройств, оконные менеджеры, графические подсистемы.

Целевая платформа

Поскольку в каждой среде нужна какая-то поддержка этапа исполнения, программное обеспечение системного уровня есть почти в каждом электронном устройстве. Системное программное обеспечение необходимо как самому маленькому встроенному устройству, так и самому большому мэйнфрейму.

Среда разработки

Новые драйверы устройств и компоненты операционной системы могут нарушить нормальную работу компьютера и сделать систему нестабильной, поэтому часто разработку ведут на одной машине, а выполняют код на другой. Самым распространенным языком в этой области является C, хотя библиотеки иногда пишут на других языках (популярностью пользуется C++ как предназначенный для системных разработок).

Типичные задачи и проблемы

Здесь главная задача – стабильность, поскольку речь идет о создании фундаментальных блоков целой вычислительной среды. Если приложение может аварийно завершиться и при этом сохранить результаты работы и восстановиться, то драйверу устройства такая роскошь обычно недоступна; он обязан корректно работать все время, пока запущен. Это время может быть очень продолжительным, поэтому даже малейшие утечки памяти могут вызвать серьезные проблемы.

Код должен быть эффективным (в достаточной мере) в отношении как памяти, так и скорости, а также должным образом приспособлен к особенностям функциональной среды.

Обычно системное программирование осуществляется и для рабочих станций тоже, но не предназначено конечным пользователям. Системное программное обеспечение предназначено разработчикам приложений; его публичное лицо составляют API, используемые более высокими уровнями программ. Системное ПО связано с логикой низкого уровня, взаимодействующей с компьютером на самых базовых принципах, и со вспомогательными структурами промежуточного уровня, которые, хотя и не работают напрямую с аппаратурой, но предоставляют важные сервисы для всей системы.

Типичная работа в этой сфере заключается в написании драйверов (управляющих такими устройствами, как принтеры, носители информации, устройства вывода и т. п.), создании библиотек общего доступа и средств управления дефицитными ресурсами, реализации операционных систем, действительно управляющих компьютером, и обеспечении таких компонент, как файловые системы и стеки сетевых протоколов. В этот разряд могут попасть даже компиляторы и пакеты программ установки, поскольку они обеспечивают работу программистов приложений и тесно связаны со средой исполнения программы.

Встроенное программное обеспечение

Компьютерные технологии появляются во всех областях нашей жизни, даже если мы этого не замечаем. Мы постоянно пользуемся разными устройствами и приспособлениями, от микроволновых печей до часов, от радиоприемников до термостатов. Эти электронные приборы бытовой техники требуют для своего функционирования программного обеспечения. Как правило, пользователь устройства не видит его управляющей программы. Встроенное программное обеспечение есть не только в бытовой технике: программами управляются все устройства с микроконтроллерами, например лабораторные приборы или машинки, выдающие квитанции на парковке. Мы должны написать программы, встраиваемые в аппаратные устройства: встроенное программное обеспечение.

Создатели встроенного программного обеспечения работают в жестких условиях:

- Обычно ресурсы весьма дефицитны: низкая мощность ЦП и/или малый объем памяти. Ограничения памяти относятся как к ROM (где содержится образ программы), так и RAM (памяти, в которой выполняется код и хранятся его данные). На платформах с ограниченным объемом памяти приходится с большим трудом впихивать ПО в доступное пространство. Иногда это требует весьма изобретательных (и героических) усилий, например динамической декомпрессии кода программы или данных.
- Средства интерфейса пользователя весьма ограничены: как организовать все операции пользователя с помощью двух кнопок и свето-

диода? На практике интерфейса пользователя может вовсе не существовать; никакого прямого взаимодействия с пользователем нет, а программа должна просто работать.

Эти ограничения оказывают глубокое воздействие на характер разрабатываемого кода. К сожалению, во встроенном программном обеспечении (чаще, чем в других ситуациях) приходится жертвовать чистотой кода ради получения работоспособного продукта. Если код быстрый и умещается в ПЗУ, это важнее, чем его идеологическая корректность, но большой размер и медленная работа.

Встроенные программы должны решать одну-единственную задачу и надежно работать. Присутствие программного обеспечения не должно быть заметно; устройство просто должно постоянно работать. Отказ

Краткая информация

Люблю работать с железом – здесь нужно действительно думать головой. Нужен плотный код и четкое представление о том, как работает устройство. Отладка кода может оказаться непростым делом, но такие задачи и делают эту работу интересной.

Грэм (разработчик встроенного ПО)

Типичные продукты

Управляющие программы для стиральных машин, высококачественной аудио/видеоаппаратуры, мобильных телефонов.

Целевая платформа

Небольшие заказные устройства с крайне ограниченными ресурсами и скудными пользовательскими интерфейсами.

Среда разработки

Поскольку вы работаете с заказными устройствами, инструментарий тоже часто оказывается заказным. Зачастую он далеко не такой совершенный, как у программистов приложений. (По мере развития рынка в этой области наблюдается улучшение.) Код разрабатывается в среде с кросс-компиляцией, где целевая платформа отличается от платформы разработчика. (Ясно, что нельзя компилировать C-код на стиральной машине... пока.)

Для каждого конкретного устройства пишется специальная программа. Для встроенного программирования повсеместно используется C, исключая действительно низкий уровень, на котором прибегают к ассемблеру. C++ прокладывает путь в эту область, встречается также ADA.

Типичные задачи и проблемы

Здесь можно столкнуться с самыми разнообразными проблемами, что во многом зависит от того, с какой платформой вы работаете – готовой покупной или разрабатываемой самостоятельно. Есть проблемы программирования в реальном времени (например, своевременная обработка аппаратных событий и прерываний), непосредственного интерфейса с аппаратурой и управления соединениями с периферией, а также неинтересные низкоуровневые проблемы, такие как порядок байтов в слове и структура физической памяти.

Чтобы обеспечить надежность систем, требуется тщательное тестирование.[1]

[1] Тщательного тестирования требуют любые разрабатываемые программы – не только встроенные. Во всех областях объем проводимого тестирования страдает от усердия маркетинговых и административных работников, не понимающих природы программного обеспечения. Однако настольные приложения проще обновить, чем микропрограммы устройств.

программы, как правило, недопустим; в результате устройство может оказаться физически повреждено. Сравните это с настольным компьютером, являющимся универсальной машиной. Он должен уметь работать с текстом, воспроизводить кинофильмы, показывать веб-сайты, читать электронную почту, управлять банковскими счетами и т. д. Пользователи привыкли к случайным авариям и некоторой нестабильности. Маленькие неудобства перевешиваются мощью и гибкостью. Со встроенными устройствами совсем другая история.

Хороший пример дает производство современных автомобилей. Транспортные средства содержат массу встроенных систем, управляющих самыми разными вещами: двигателем, тормозной системой ABS, средствами безопасности (такими как подушки и ремни безопасности), климатом, спидометром и т. п. Однако пользователям (в данном случае водителю и/или пассажирам) нет надобности знать о том, что под капотом урчат какие-то микропроцессоры. Им нужно, чтобы машина ехала. Вот когда откажет система управления двигателем, пользователю придется узнать о существовании программного обеспечения! Или возьмите мобильные телефоны. Очевидно, что это компьютерные устройства, но редкий покупатель воспринимает их в качестве компьютеров. Эти маленькие устройства обладают изрядной вычислительной мощностью, и все же существуют строгие границы функциональных возможностей, в рамках которых должно действовать программное обеспечение.

Встроенная система обычно представляет собой комбинацию маленького компьютера, некоторой специальной аппаратуры и операционной системы реального времени или простой управляющей программы. Она непосредственно управляет аппаратными средствами устройства. Обычно встроенные системы изготавливаются на заказ – для конкретной аппаратуры и конкретных задач. В простых встроенных системах есть только одна выполняющаяся программа – никаких сложных многопоточных программных средств, нет даже операционной системы.

Обычно код хранится в чипе памяти, доступном только для записи. Возможность его обновления существует редко, поэтому он должен корректно работать с первого раза. Если вы ошиблись, то шансов поставить версию 1.1 не будет. Одна простая ошибка, и ваше чудо технологии обернется провалом.

В последнее время память и вычислительная мощность ЦПУ подешевели, и на рынке появляется все больше массовых устройств. Встроенная среда становится мощнее, и ограничения на ресурсы ослабляются. Тем не менее всегда будет существовать потребность в очень маленьких устройствах с ограниченной мощностью, которые все же решают свои задачи.

Программирование приложений для наладонных устройств типа PDA можно считать либо уровнем встроенных программ, либо уровнем приложений – в зависимости от ваших вкусов.

Программирование распределенных систем

Распределенные системы состоят из нескольких компьютеров. Как мы увидим позднее, Всемирная паутина фактически представляет собой огромную распределенную систему, в которой информация хранится на многочисленных компьютерах, разбросанных по разным континентам, а приложения удаленно загружаются с помощью веб-броузера. Хотя дело не ограничивается броузерами. Архитектуры из нескольких машин используются во многих ситуациях. Проектирование и создание распределенных систем выдвигает целый ряд новых проблем.

Сделать программную систему распределенной может понадобиться по ряду причин. Возможно, одни компьютеры лучше, чем другие, подойдут для решения определенных задач. Возможно, система будет испытывать большую нагрузку, и для повышения эффективности ее целесообразно распределить между другими машинами в сети. Возможно, для некоторых машин существуют ограничения по физическому размещению, которые обусловливают распределение системы. Возможно, вам нужно организовать взаимодействие нового продукта с историческими системами или старым оборудованием.

Задача в том, чтобы создать систему из нескольких программ, выполняющихся на разных машинах и работающих как единое целое. Соединенные между собой с помощью сети, они могут физически разме-

щаться в одном помещении или находиться в разных концах света и использовать для связи Интернет.

Отдельные части должны быть как-то связаны вместе; программы должны обмениваться данными, и желательно вызывать функции на удаленных машинах, как если бы они локально компоновались с кодом. Это называют *удаленным вызовом процедур* (*RPC*), и такие возможности обеспечиваются рядом имеющихся *технологий программных средств промежуточного слоя* (*middleware*). Они действуют в качестве брокеров при передаче данных между машинами, предоставляя возможности обнаружения сервисов на других машинах и взаимодействия с ними, а также публикации своих сервисов, к которым могут обращаться другие программы. ПО промежуточного слоя управляет политикой взаимодействия, включающей вопросы безопасности (кому кого разрешено вызывать?), проблемы латентности в сети (что делать, если удаленный вызов функции выполняется слишком долго или компьютер отключился?), выбор синхронного или асинхронного режима вызова удаленных функций и т. д.

В некоторых системах промежуточного слоя применяются объектно-ориентированные технологии, в других используется преимущественно процедурный подход. ПО промежуточного слоя является просто коммуникационным и допускает некоторую независимость от платформы. Когда на данной платформе выполняется ПО промежуточного слоя, клиентскому коду должно быть безразлично, какова платформа, к которой происходит обращение (хоть ZX spectrum), – вызов функции должен выглядеть одинаково. Разумеется, проектируя распределенную систему, вы выберете для каждой задачи наиболее подходящую аппаратную часть. Сомнительно, что вам встретится хоть один ZX spectrum!

Распространены такие системы промежуточного слоя, как CORBA, Java RMI, Microsoft DCOM и удаленный доступ .NET Remoting. С их помощью система разбивается на элементы интерфейса пользователя, бизнес-логику (реально работающий код) и необходимые средства хранения данных (например, база данных и механизм запросов). Клиент с интерфейсом пользователя может быть программой GUI или веб-интерфейсом. Это классическая *многоярусная архитектура* (описанная в разделе «Архитектура клиент/сервер» на стр. 355). Наблюдается также появление *веб-API* – методов обмена данными с сервисами, которые используют стандартные веб-протоколы.

Грид-компьютинг (*grid computing*) и *кластерные системы* представляют собой особые механизмы распределения, созданные для вычислений (о программировании вычислений см. ниже) и позволяющие создавать высокоэффективные распределенные вычислительные алгоритмы. Кластеры являются тесно связанными системами; обычно машины находятся в одном помещении, используют одинаковые аппаратное обеспечение и ОС и соединяются специфическим кластерным

промежуточным ПО. Гриды связаны менее тесно; они могут быт разнесены географически и образовывать гетерогенную среду. Связь между ними организована через стандартные сетевые протоколы (например, HTTP/XML).

Краткая информация

Проект Smallpox, завершенный в 2003 году, был грид-проектом, который должен был найти лекарство от оспы путем проверки огромного количества молекул потенциального лекарства. Это было сотрудничество ученых, университетов и бизнеса, которое выявило 44 хороших кандидата для лечения болезни.

Типичные продукты

В системе электронной торговли работа распределяется между клиентскими приложениями (веб-интерфейс, торговый киоск и/или система заказа по телефону), бизнес-логикой (управление запасами, реализация заказов и безопасных расчетов) и общим хранилищем данных.

Целевая платформа

Различные компьютерные системы соединяются с помощью ПО промежуточного слоя, почти всегда действующего поверх стандартных сетевых протоколов.

Среда разработки

Имеется большое разнообразие. Все зависит от выбранных языков, типов объединяемых компьютеров и типа выбранного ПО промежуточного слоя. Интерфейсы с удаленным вызовом часто определяются с помощью какого-либо *языка описания интерфейсов* (*IDL*) и компилируются в представление на языке реализации, которое обеспечивает весь механизм вызовов и ловушки для всех реализаций функций, к которым нужно подключаться.

Типичные задачи и проблемы

Правильный выбор распределения сервисов между компьютерами и ускорение связи между ними. Это оказывает сильное влияние на масштабируемость распределенной системы. Система, справляющаяся с несколькими транзакциями в сутки, может не справиться с 100 транзакциями в минуту. Это вызывает необходимость тщательного проектирования. Кроме того, нужно решать проблемы доступности компьютеров и корректного поведения, когда один из компьютеров отключается от сети.

Программирование веб-приложений

В 1990 году Тим Бернерс-Ли создал первые броузер и сервер HTML, и родилась World Wide Web. Сегодня это вездесущая технология, и серверы могут предоставлять не только статические страницы с информацией, но и динамически создавать страницы с помощью программ, выполняемых на веб-сервере. Это весьма специфическая форма распределенных вычислений, в которой интерфейс пользователя – веб-броузер – находится у удаленного клиента.

Примерами такого рода приложений служат:

- Электронная торговля
- Доски объявлений, службы передачи сообщений и почтовые системы, базирующиеся на веб
- Системы заказа билетов
- Механизмы поиска в Интернете

Большинство людей не задумывается, работая с веб-приложениями; они так же естественны, как текстовый процессор. Однако эти программы заметно отличаются от обычных настольных приложений (т. н. *толстых клиентов*). Есть вещи, которые лучше удаются одной из этих систем. Без специального кодирования на JavaScript возможности интерфейса пользователя в броузере оказываются гораздо беднее.

Модель функционирования веб-приложений отличается от обычных приложений – состояние сеанса хранится на удаленной машине, которая должна управлять многочисленными одновременными соединениями с клиентами, сохраняя их состояние между запросами HTTP и корректно поступая с клиентами, прекратившими связь. Для решения этой задачи часть информации хранится на сервере (например, заказанные клиентом товары помещаются в базу данных), а часть находится у клиента на локальной машине (в виде *cookies* броузера – записей данных сеанса, хранящих идентификатор клиента/сеанса). Существуют такие среды, как ASP.NET и Java Servlets, которые ускоряют разработку веб-приложений. Есть многочисленные готовые системы, например системы управления контентом и системы электронного магазина.

Для представления и передачи информации используются многочисленные открытые стандартные протоколы и системы кодирования. Стандартным механизмом передачи данных является HTTP, а XML часто применяется для кодирования пакетов данных (например, SOAP служит интернет-протоколом связи, основанным на схеме XML).

Проблемы, с которыми сталкиваются программисты веб-приложений, в основном касаются совместимости с различными типами броузеров, которые могут встретиться, особенностей обработки ими HTML и JavaScript. Нередко приходится проектировать замысловатый HTML, который должен справляться с разнообразными ошибками, имеющими-

ся в популярных броузерах. Веб-программистам часто приходится иметь дело с историческими системами (базами данных клиентов, имеющимися системами управления заказами и т. п.), чтобы получать из них информацию, и это бывает достаточно хлопотно. Серьезной проблемой является масштабируемость. Система может отлично работать, когда ее тестируют пять пользователей. Но когда систему начинают реально эксплуатировать, ей приходится выдерживать одновременную работу с 500 пользователями. *Тестирование под нагрузкой* имеет здесь важное значение (см. раздел «Испытание под нагрузкой» на стр. 196).

Краткая информация

Веб-приложение заставляет воспринимать броузер как свою ОС. Все хорошие веб-разработчики сначала вдоль и поперек изучают технологии броузера клиента. Затем вы учитесь писать хороший код для сервера (быстрый, параллельный, с транзакциями, распределенный и корректный). Лучшее свойство Веб – непрерывное развитие и рост ожиданий пользователей. Плохо в Веб то, что ожидания пользователей все время растут, и ваш код не может стоять на месте.

Алан (программист веб-приложений)

Типичные продукты

Интерактивные системы обслуживания, требующие постоянного обновления информации и ввода пользовательских данных: заказ билетов и электронная торговля.

Целевая платформа

Исполнительной платформой является веб-сервер (обычно Apache или IIS). Выбор за вами, потому что вы сами разворачиваете веб-приложение. Клиенты суть веб-броузеры, и здесь много вариантов. У каждого есть свои особенности, и вы не можете управлять тем, какой из них будет использоваться. Приходится делать веб-страницы, совместимые с большинством броузеров.

Среда разработки

Среда состоит из конкретного веб-сервера и языка программирования приложений, на котором пишется система, выполняемая на этом сервере. Стандартные языки: Perl и PHP.

Типичные задачи и проблемы

Обеспечение совместимости с разными броузерами; масштабируемость.

Программирование масштаба предприятия

Предприятие (*enterprise*) – это скучное, постоянно звучащее слово, скорее из новояза менеджеров, чем из лексикона программистов. Буквально оно означает бизнес-организацию. Так что *программирование масштаба предприятия* (*enterprise programming*) создает системы для целых компаний, объединяя их отдельные системы в одно тесно связанное целое. Почти всегда программирование масштаба предприятия подразумевает разработку крупных распределенных систем.

Обычно они разворачиваются в интранете компании (ее внутренней сети) и связывают вместе различные подразделения, совершенствуя рабочий процесс. Системы могут иметь интерфейс к клиентам. Если в организации действует интегрированная компьютерная система, обычно бывает не слишком сложно добавить в нее автоматизированную работу с клиентами, например интерфейс магазина электронной торговли. Иногда системе масштаба предприятия требуется интерфейс к системам других компаний, например для слежения за поставками товаров.

Краткая информация

Я работаю в отделе ИТ крупного городского банка. Мы пишем программы для решения конкретных потребностей нашего бизнеса. Это важные задачи; от результатов нашей работы реально зависят прибыли компании, поэтому отношение к ней серьезное. Когда через систему ежечасно проходят многие тысячи долларов, ошибиться нельзя.

Ричард (программист приложений масштаба предприятия)

Типичные продукты

Бизнес-системы целых компаний, управляющие их коммерческими операциями.

Целевая платформа

Специально сконструированная распределенная система.

Среда разработки

Такая же, как для распределенных систем. Часто приходится работать с очень большими хранилищами данных, различными технологиями баз данных, доставшимися от прежних систем («legacy systems» на жаргоне менеджеров). Тут все помешаны на XML.

Типичные задачи и проблемы

Такие же, как для распределенных систем.

Программирование масштаба предприятия имеет много сходства с написанием заказного программного обеспечения. Продукт не требует *очень высокого* уровня качества, поскольку разрабатывается по контракту с конкретным заказчиком, а не в расчете на рыночный успех. Успех определяется не качеством (в смысле высокой устойчивости и превосходства над конкурентами по числу предоставляемых функций), а степенью удовлетворения потребностей заказчика.

Системы масштаба предприятия пишутся под конкретные машины, стоящие в вычислительном центре компании, и фиксированные настольные системы. Таким образом, вы располагаете достаточным контролем над средой исполнения и можете не беспокоиться о том, как заставить код работать под всеми версиями операционной системы и в каждой мыслимой аппаратной конфигурации. Это приятно избавляет от головной боли, которая мучает программистов приложений.

Численное программирование

Численное программирование предполагает решение научных, весьма специальных технических задач с активным использованием математических методов. Это очень специализированная область, где пишутся приложения, особо ориентированные на решение конкретных числовых задач. Часто программы рассчитаны на суперкомпьютеры – очень быстрые машины, способные проводить сложные расчеты. Это очень дорогие платформы, применяемые для специальных приложений, в которых требуются огромные объемы математических вычислений.

Например, суперкомпьютеры нужны для предсказания погоды. Они также находят применение в анимационной графике, гидродинамических расчетах и других областях, требующих сложных математических методов и вычислений.

Суперкомпьютер – это не мэйнфрейм. Последним мы называем высокопроизводительную систему, предназначенную для одновременного выполнения большого количества программ, часто используемую в качестве централизованного вычислительного ресурса в бизнес-условиях. В суперкомпьютере вся мощь нацелена на выполнение небольшого числа программ с максимальной скоростью. Есть несколько архитектур суперкомпьютеров, применяющих новейшие технические достижения и требующих различных алгоритмических подходов для полного использования своей мощности. В настоящее время и универсальные машины становятся достаточно мощными для проведения серьезных расчетов; при недостатке средств их можно объединить в кластер, что позволит составить вполне приличный суперкомпьютер.

Числовые расчеты требуют эффективных алгоритмов, способных использовать все возможности вычислительной платформы. Здесь применяют тщательно спроектированные и высоко оптимизированные библиотеки для работы с числами и явным образом распараллеливают вы-

Краткая информация

Я работаю над программными системами в инженерной фирме. Мы моделируем крупные механические установки, чтобы выяснить, где существуют или могут возникнуть в будущем физические проблемы. Моя задача – представить математическое описание реального мира и определить, как устроена (или должна быть устроена) система. Затем нужно найти правильные математические конструкции, которые представят эти системы приемлемым и точным образом.

Энди (эксперт по программированию вычислений)

Типичные продукты

Области, в которых требуется сложный математический анализ, например ядерные исследования или поиск нефтяных месторождений.

Целевая платформа

Суперкомпьютеры или компьютерные гриды и кластеры.

Среда разработки

Хотя планируется улучшить поддержку вычислений в C++ и частично такая работа ведется на C, большое количество вычислительных программ пишется на Fortran, который отлично поддерживает числовые расчеты (для чего он и был создан: трансляция формул).

Типичные задачи и проблемы

Создание эффективных алгоритмов, полностью использующих мощь суперкомпьютера.

числения при написании алгоритмов и процессов. Параллельность может относиться как к задачам, так и к данным: либо одновременно выполняется множество идентичных задач на многих ЦП, либо алгоритм конвейеризуется, при этом на разных ЦП выполняются разные его части.

Для достижения нужной производительности здесь требуется тщательная оптимизация в соответствии с характеристиками целевой платформы.

И что дальше?

Свобода от желания получить ответ очень важна для понимания проблемы.

Джидду Кришнамурти

Какое значение имеют для нас эти ниши программирования? Как должны мы действовать, зная о них? Чтобы стать хорошим программистом, настоящим мастером, нужно знать:

- В какой области вы работаете – какого рода программы делаете.
- Какое влияние оказывает эта область на архитектуру. (Это многоуровневая система предприятия или крепко скрученный встроенный код? См. главу 14.)
- Какие конструкции кода подходят и не подходят в данной области. (Например, требуется ли жертвовать ясностью и изяществом в пользу эффективности, предельно сжимать исполняемый образ или оставлять много ловушек для последующих расширений?)
- Какими инструментами пользоваться – что доступно, а что нет.
- Какой язык программирования лучше выбрать и какие идиомы кодирования применять.

Знайте свою область программирования. Изучите ее специфику. Научитесь писать отличные программы, которые удовлетворяют ее требованиям.

Резюме

За поворотом могут оказаться новая дорога или тайная калитка.

Р.Дж. Толкиен

Мы лишь обмакнули палец и попробовали на вкус разные виды программирования. Разумеется, есть и другие, которых мы не коснулись: одни более четко очерченные, другие менее. Например, *программное обеспечение, критичное с точки зрения безопасности*, управляет системами, требующими высокой надежности, такими как медицинское или авиационное оборудование. Здесь отказы недопустимы, и корректность кода должна быть проверяема. Это оказывает существенное влияние на способы проектирования и написания такого ПО.

Что мы выяснили? Общее у этих областей одно: наличие особенностей. Каждая область требует специфических проектных решений. Код уровня приложений в целом не годится для встроенной среды. Архитектура приложения для рабочей станции может не масштабироваться в применении к распределенной системе.

Из этого следует тенденция разработчиков ПО к специализации в конкретной области и мышлению соответствующими ей схемами. Понимание реальных проблем, существующих в каждой такой среде, поможет вам стать более гибким и зрелым программистом. В конечном итоге нужно знать, к какой из программистских церквей вы принадлежите, и правильно исполнять ее обряды и ритуалы.

Хорошие программисты...

- Понимают сущность проблем, с которыми сталкиваются
- Приспосабливают свой код и архитектуру к предметной области

Плохие программисты...

- Обладают примитивно узким кругозором в области программирования: не понимают, какие объективные обстоятельства требуют разработки разных типов ПО
- Пишут код, плохо соответствующий предметной области (выбирают неподходящие архитектуры или неуместные идиомы кода)

См. также

Глава 7. Инструментарий программиста

Разным предметным областям соответствуют разное качество и спектр инструментов разработки.

Глава 14. Программная архитектура

Различные предметные области требуют разных программных решений.

Контрольные вопросы

Подробное обсуждение этих вопросов можно найти в разделе «Ответы и обсуждение» на стр. 685.

Вопросы для размышления

1. Какие из рассмотренных нами областей программирования обладают наибольшим сходством между собой или общими характеристиками? Какие особенно различаются?
2. В какой из перечисленных областей труднее всего работать?
3. Следует ли стремиться стать экспертом в одной конкретной области или лучше иметь хорошую подготовку во всех без узкой специализации?
4. С какой из областей программирования следует знакомить начинающих программистов?

Вопросы личного характера

1. В какой области программирования работаете вы в настоящее время? Какое влияние оказывает она на код, который вы пишете? Какие конкретные решения в области архитектуры и реализации она потребовала от вас принять?
2. Доводилось ли вам работать в нескольких областях программирования? Было ли вам легко принимать другой взгляд на вещи и применять технологии, присущие новой сфере?
3. Есть ли среди ваших коллег люди, не понимающие причин, влияющих на то, как вы пишете код? Сталкивались ли вы с тем, что встроенное программное обеспечение пишут программисты, умеющие писать только приложения? Какой выход может быть из этой ситуации?

24

Что дальше?

Все хорошо, что хорошо кончается

В моем начале – мой конец, в моем конце – мое начало.
Томас Элиот

Поздравляем! Вы добрались до конца книги. Либо вы из тех, кто любит все испортить, начав читать с последней страницы. (Тогда сразу сообщаю: *Убийцей был дворецкий*.) Если же вы прочли все главы, то должны были:

- Научиться многим практическим приемам написания кода, благодаря которым уже усовершенствовали свою технику.
- Понять, как пишется код в реальном мире и какие уловки помогают писать приличный код в безумных условиях промышленного производства программ.
- Выработать свои персональные способы повышения мастерства. (Вы же попробовали ответить на вопросы? Если нет, сделайте это сейчас.)
- Выяснить, как писать эффективный код в составе команды, предпринимая практические меры для улучшения качества сотрудничества в вашей нынешней команде.
- Узнать об обезьянках из комиксов больше, чем вам может понадобиться.

Еще важнее, что к этому времени вы должны были понять, что выдающегося программиста отличает правильная позиция: он старается пи-

сать самый хороший код в любой ситуации, он находит общий язык с коллегами и он умеет принимать практичные решения в условиях промышленного производства программ. Мастер знает, как управлять технической задолженностью и старается заблаговременно решать проблемы, не позволяя поймать себя в ловушку.

Чтобы стать хорошим программистом, нужно выбрать правильную позицию – тот угол, под которым рассматривается создание программного обеспечения.

Но что же дальше?

Важно не прекращать задавать вопросы.
Любознательность – великая вещь. Нельзя без благоговения созерцать тайны вечности, жизни, чудесного устройства окружающего мира. Достаточно каждый день пытаться проникнуть в малую часть этой тайны.
Берегите священный дар любознательности.

Альберт Эйнштейн

Вы никогда не достигнете совершенства в искусстве программирования; лучшее, что вам доступно, – это состояние постоянного развития. Всегда найдется то, чего вы еще не знаете. Так что же делать дальше? Тот факт, что вы об этом спрашиваете, уже многое значит: одна из главнейших характеристик программиста-мастера – желание совершенствоваться.

Если бы я хотел стать хорошим футболистом, можно было бы найти какие-то книги о футболе, купить учебный фильм, а потом, взяв пива и попкорна, сесть и начать учиться, как играть в футбол. Замечательно. Через пару месяцев поинтересуйтесь у меня, чего я достиг. Если я скажу, что прочел массу книг и знаю все приемы выдающихся игроков, это не произведет на вас впечатления. Действительно, идея прочесть все об игре и изучить ее неплоха, но мастерство, приобретенное на диване, не имеет никакого практического применения.

Научиться играть в футбол можно только одним способом – выйти на поле и играть, не боясь упасть в грязь. *Мастерство приходит в результате практики*. Нужно играть с людьми, которые умеют это делать и смогут научить меня. Надо не жалеть энергии и не бояться выглядеть смешным в глазах других. Медленно, постепенно, с муками я стану лучше.

Должен вас огорчить, но путь к совершенству в кодировании точно такой же. Мало лишь прочесть эту книгу. Нужно браться за работу и *делать ее*. Так, как это требуется. Что же можно предложить на практике? Вот несколько простых идей:

- Поставьте эту книгу на полку. Не откладывая дела в долгий ящик, примените на практике все, что вы узнали. Если у вас возникнут проблемы, всегда можете открыть нужную главу.

  Проработав несколько месяцев согласно этому совету, достаньте книгу и прочтите ее еще раз. Особое внимание обратите на разделы «Вопросы личного характера» – определите, какие еще шаги нужно сделать, чтобы улучшить ваш код. Каждый раз после такой процедуры вы будете находить новые пути для повышения своего мастерства.
- Постарайтесь работать рядом с хорошими программистами и возьмите от них все, что можно. Выясните, чем хорош их код, в чем конструктивность их позиции и что из этого вы можете перенять. Добивайтесь от них советов, критики, отзывов, мнений. Просите их стать вашими учителями. (Если нужно, подмажьте их попкорном и выпивкой!)
- Программируйте и расширяйте свой кругозор. Пишите больше кода. Экспериментируйте с новыми технологиями. Беритесь за новые задачи, изучайте новые языки и технологии.
- Не бойтесь ошибиться; хорошими программистами не становятся в одночасье. По ходу своего совершенствования вы наверняка сделаете много неловких ошибок. Они не должны задержать ваш рост и становление как программиста. Если вы не будете осваивать новые приемы, то никогда ничему не научитесь и перестанете развиваться. Джордж Бернард Шоу писал, что «лучше всю жизнь делать ошибки, чем всю жизнь ничего не делать».

  Относитесь к рекомендациям и замечаниям по поводу вашего кода конструктивно. Взгляните еще раз на свою работу и поищите, что в ней можно сделать лучше.
- Развейте в себе интерес к посторонним предметам, которыми можно было бы воспользоваться как сферой приложения технических знаний. Если ваше образование ограничится программированием, вы станете очень ограниченным человеком, не способным применить мастерство программирования к проблемам окружающего мира.
- Поищите, какие классические труды написаны по вашей области. (Очевидно, одним из них является эта книга!) Достаньте их и хорошенько изучите. В каждой сфере и в каждом языке есть свои признанные гуру; вы должны знать их имена и написанные ими книги.

  Прочтите следующие классические книги:

  «The Mythical Man-Month»[1] (Brooks 95)

  Gerald M. Weinberg «The Psychology of Computer Programming», Dorset House Publishing, 1998 (Weinberg 71)

---

[1] Фредерик Брукс «Мифический человеко-месяц или как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

«Peopleware: Productive Projects and Teams»[1] (DeMarco 99)

«The Pragmatic Programmer»[2] (Hunt Davis 99)

«Code Complete»[3] (McConnell 04)

«The Practice of Programming»[4] (Kernighan Pike 99)

«Design Patterns: Elements of Reusable Object-Oriented Software»[5] (Gamma et al. 94)

«Refactoring: Improving the Design of Existing Code»[6] (Fowler 99)

Спросите у коллег, какие книги им показались ценными. Поищите соответствующие журналы, веб-сайты и конференции.

- Займитесь преподаванием. Обучите менее подготовленного программиста. Передавая свои знания, вы сами многому научитесь.
- Расширьте свое мастерство, присоединившись к профессиональной организации, такой как British Computer Society (BCS), Association for Computing Machinery (ACM) или ACCU (*www.accu.org*). Затем включайтесь в ее работу – пишите. Чем активнее ваше участие, тем больше ваши инвестиции в себя самого. Например, ACCU активно способствует сотрудничеству. Она осуществляет проекты под руководством наставников и поддерживает публикации членов в своих периодических изданиях. Эти организации проводят конкурсы программистов, обеспечивают форумы для социальных сетей и имеют много местных отделений, где можно встретить единомышленников, интересующихся мастерством программирования.
- Получайте удовольствие! Сочиняйте код для решения хитрых задач. Создавайте ПО, которым можно гордиться. Конфуций сказал: «Если вам нравится то, чем вы занимаетесь, вы больше ни одного дня в своей жизни не будете работать».

Займитесь совершенствованием своего мастерства. Не теряйте любви к программированию и желания быть в нем мастером.

---

[1] Том Демарко и Тимоти Листер «Человеческий фактор: успешные проекты и команды». – Пер. с англ. – СПб.: Символ-Плюс, 2005.

[2] Э. Хант и Д. Томас «Программист-прагматик. Путь от подмастерья к мастеру». – Пер. с англ. – Лори, 2009.

[3] С. Макконнелл «Совершенный код. Практическое руководство по разработке программного обеспечения». – Пер. с англ. – СПб.: Питер, 2005.

[4] Б. Керниган и Р. Пайк «Практика программирования». – Пер. с англ. – М: Вильямс, 2004.

[5] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб.: Питер, 2005.

[6] Мартин Фаулер «Рефакторинг. Улучшение существующего кода». – Пер. с англ. – СПб.: Символ-Плюс, 2002.

ОТВЕТЫ И ОБСУЖДЕНИЕ

Принципы развития совершенного ума:
Изучайте искусство как науку. Изучайте науку как искусство.
Развивайте свои чувства – в особенности учитесь видеть.
Проникнитесь пониманием того, что все вещи связаны между собой.

Леонардо Да Винчи

Эта часть содержит мои размышления по поводу вопросов, предлагаемых в конце каждой главы. Это не набор простых ответов – очень редко ответ заключается в *да* или *нет*. Сравните свои ответы с приведенными здесь.

Смысл вопросов в том, чтобы побудить вас мыслить, заставить глубже проникнуть в каждую тему и подстегнуть к совершенствованию своего умения программировать.

Если вы собираетесь продолжить чтение, только чтобы узнать «правильный» ответ, не дав себе труда подумать над вопросом, я настоятельно советую вам не делать этого. Даже немного времени, потраченного на размышление и оценку личной ситуации, окупится с лихвой. Как сказал Конфуций: «Я слышу и забываю. Я вижу и запоминаю. Я делаю и понимаю.»

Глава 1. Держим оборону

Вопросы для размышления

1. Можно ли переусердствовать с защитным кодом?

Да – затруднить чтение кода могут как многочисленные комментарии, так и многочисленные проверки, если они неуместны. При тщательном кодировании можно избежать лишних проверок, например правильно подбирая типы.

2. Нужно ли добавлять операторы контроля в каждую точку, где выявлена и исправлена ошибка?

В принципе это неплохая практика. Но нужно подумать, где их стоит располагать. Очень часто ошибки возникают из-за нарушения контракта API. Если вы передали мусор в функцию, нужно поместить в эту функцию какую-то проверку *предусловия*, а не делать проверку в месте вызова. Если функция возвратила мусор, нужно либо исправить ее так, чтобы больше такого не происходило (и проверить исправление), либо написать какие-то проверки *постусловий*.

Полезнее для всякой найденной и исправленной ошибки добавить новый тест модуля.

3. Следует ли исключать из окончательных сборок операторы контроля посредством условной компиляции? Если нет, то какие контрольные операторы стоит оставить в окончательной сборке?

Некоторые очень эмоционально относятся к этой проблеме. Трудно дать однозначный ответ; есть много доводов «за» и «против». Часто встречаются совсем малозначительные проверки, которым *не место* в окончательном коде. Но некоторые операторы контроля могут оказаться полезны и при реальной эксплуатации пакета.

Однако если вы оставляете какие-либо проверки в окончательной версии, их нужно скорректировать так, чтобы программа не завершала аварийно работу, встретив ошибку, а лишь записывала сообщение в журнал.

Нужно помнить, что действительные проверки ошибок этапа исполнения *нельзя* удалять, и они не должны программироваться как операторы контроля.

4. Что обеспечивает лучшую защиту – обработка исключений или операторы контроля в стиле C?

Исключения могут обеспечивать лучшую защиту. Они ведут себя особым образом: когда исключение распространяется вверх по стеку вызовов, его можно перехватить и игнорировать, подавив таким образом его действие. Благодаря этому исключения оказываются более гибким инструментом. Игнорировать оператор `assert`, прерывающий выполнение программы, нельзя; операторы контроля – механизмы более низкого уровня.

5. Следует ли включать защитную проверку пред- и постусловий *в* каждую функцию или только при *вызове* важных функций?

Несомненно – в функцию. Тогда вам придется написать проверку всего один раз. Желание вынести проверку из функции может быть обоснованным, только если нужна повышенная гибкость, чтобы по-разному реагировать на невыполнение ограничения. Но выигрыш не столь убедителен в сравнении с резким увеличением сложности и возможности отказа.

6. Насколько ограничения соответствуют идеалу защитного средства? Каковы их недостатки?

Этот механизм далек от совершенства. Лишние ограничения могут быть просто паразитами или даже представлять собой помехи. Например, можно проверять ограничение на значение параметра функции `i >= 0`. Но гораздо лучше сделать `i` беззнаковым типом, чтобы ему нельзя было присваивать отрицательные значения.

Ограничения, от которых нельзя избавиться, должны рассматриваться с определенной степенью недоверия: необходимо тщательно следить за побочными эффектами (операторы контроля могут иметь скрытые косвенные последствия) и вопросами синхронизации в отладочной и окончательной сборках, которые могут различаться в поведении. Операторы контроля должны быть логическими ограничениями, а не настоящими проверками времени исполнения, которые нельзя удалять. В код, защищающий от ошибок, могут вкрасться ошибки!

Но осторожное применение ограничений все же гораздо лучше, чем их отсутствие и беззащитность перед случаем.

7. Можно ли *обойтись* без защитного программирования?

a. Можно ли изобрести язык, который устранит необходимость в защитном программировании? Как это сделать?

b. Следует ли из этого, что C и C++ порочны, поскольку они оставляют слишком много простора для возможных проблем?

В языки наверняка можно было бы ввести некоторые функции, которые позволили бы избегать ошибок. Например, в C не проверяется индекс при доступе к элементам массива. В результате можно вызвать аварийное завершение программы, обратившись по недопустимому адресу памяти. Напротив, в среде исполнения Java значения индексов массивов всегда проверяются, поэтому там такой катастрофы произойти не может. (Недопустимые индексы все равно вызывают ошибку, но класс отказов лучше определен.)

Хотя список «усовершенствований», которые можно было бы ввести в либеральную спецификацию C (советую вам подумать и предложить их как можно больше), может быть длинным, вы никогда не придумаете такой язык, который не требует защитного программирования. В функциях всегда нужно будет проверять параметры, а в классах всегда будут нужны инварианты для проверки непротиворечивости их внутренних данных.

Несмотря на то, что C и C++ предоставляют богатые возможности для совершения ошибок, они также обладают огромной мощью и выразительностью. Считать ли это пороком языка – дело вкуса. Это благодатная тема для начала религиозных споров.

8. Какого рода код не должен вызывать заботы о защитном программировании?

Мне приходилось работать с людьми, которые отказывались вводить защитный код в старую программу, потому что она была настолько плоха, что их защита не принесла бы ощутимых результатов. С большим трудом я удерживал себя от того, чтобы их поколотить.

Можно встать на ту точку зрения, что отдельная маленькая программа из одного файла или отладочная структура обойдутся без тщательного кода защиты или строгих ограничений. Но даже в этих случаях беззаботность – это неряшливость. Мы должны стремиться к тому, чтобы постоянно занимать оборонительную позицию.

Вопросы личного характера

1. **Насколько тщательно вы обдумываете каждый оператор, который вводите? Проверяете ли вы безжалостно код возврата для каждой функции, даже если *уверены*, что она не может возвратить ошибку?**

Готов поспорить, что вы проверяете не все. Очень легко пропустить код возврата некоторых функций, особенно когда их важность считается не одинаковой. Много ли найдется программирующих на C, кто проверяет код возврата `printf`? И многие ли *знают*, что он вообще существует?

2. **Указываете ли вы пред- и постусловия в описаниях функций?**
   a. **Присутствуют ли они в неявном виде в описании работы функции?**
   b. **Если пред- и постусловия отсутствуют, отмечаете ли вы это явно в документации?**

Каким бы очевидным ни казался вам контракт (из имени функции или ее описания), явная констатация ограничений снимает всякие недоразумения; всегда помните, что областей для предположений должно быть как можно меньше. Явный текст «*Предусловия: отсутствуют*» явно документирует контракт.

Конечно, не нужно повторять в каждой функции предусловие, являющееся глобальным. Это утомительно и скучно. Если всюду в API предполагается, что значения указателей не могут быть нулевыми, лучше написать это один раз в глобальной секции.

3. **Многие компании на словах одобряют защитное программирование. Рекомендуется ли придерживаться этой стратегии в ваших условиях? Изучите исходные коды – соблюдаются ли рекомендации на практике? Насколько распространена проверка допущений с помощью операторов контроля? Насколько тщательно осуществляется проверка ошибок в каждой функции?**

Очень редко в компаниях сочетаются культура отличного кода с должным уровнем защиты. Рецензирование кода – хороший способ поднять качество кода, создаваемого командой; чем больше глаз видят код, тем больше ошибок обнаруживается.

4. **Насколько вы по своему характеру недоверчивы? Пересекая дорогу, смотрите ли вы в обоих направлениях? Можете ли вы заставить себя делать неприятную работу? Проверяете ли вы свой код на все потенциальные ошибки, какими бы маловероятными они ни казались?**
   a. **Насколько легко делать это систематически? Не забываете ли вы о необходимости проверки ошибок?**
   b. **Можно ли каким-то образом облегчить себе написание защитного кода, более скрупулезно проверяющего ошибки?**

Это противоестественно для любого человека; ждать худшего от своего тщательно отшлифованного нового кода – противоречит инстинктам программиста. Всяких неприятностей вы ждете только от тех, кто будет работать с вашим кодом. Им же далеко до вашей добросовестности!

Очень полезный прием – написать тесты для каждой функции или класса. Ряд экспертов настоятельно советуют сделать это *до того*, как писать функцию, в чем немало смысла. Это заставит вас продумать все возможные ситуации ошибок, а не полагаться беспечно на то, что ваш код с ними справится.

Глава 2. Тонкий расчет

Вопросы для размышления

1. **Нужно ли менять формат старого кода, чтобы привести его в соответствие с новым кодом? Следует ли воспользоваться для этого инструментами переформатирования?**

Обычно лучше всего оставить «исторический» код в том виде, какой он есть, даже если он уродлив и с ним трудно работать. Я стал бы переформатировать его только при абсолютной уверенности, что никто из первоначальных авторов никогда не вернется к этому коду.

В результате переформатирования теряется возможность легкого сравнения версий – в обилии изменений, касающихся форматирования, можно пропустить то существенное отличие, которое вам нужно. Кроме того, при форматировании можно занести в программу ошибки.

Что касается инструментов для форматирования кода, то это любопытные штуковины, но я бы не стал рекомендовать их применение. В некоторых фирмах требуют пропускать файлы исходного кода через форматирующие фильтры перед записью кода в хранилище. Этим достигается однородность, стерильность и единообразие формата кода. Главный недостаток такого подхода в том, что идеальных инструментов не существует; некоторые полезные нюансы авторского форматирования теряются. Не пользуйтесь средствами форматирования, если только ваша команда не состоит из одних лишь гиббонов.

2. **Распространена система, при которой исходный текст располагается в фиксированном количестве колонок. Каковы ее преимущества и недостатки? Есть ли в ней смысл?**

Данный вопрос, как и другие вопросы представления, не имеет однозначного ответа; в значительной мере это дело личного вкуса.

Я предпочитаю разбивать свой код так, чтобы он умещался на экране с 80 колонками. Я всегда так делал, поэтому тут важную роль играет привычка. Я не имею ничего против тех, кто любит длинные строки, но мне кажется, что с длинными строками труднее работать. Я настраиваю свой редактор так, чтобы он переносил длинные строки, а не показывал горизонтальную полосу прокрутки (горизонтальная прокрутка неудобна). В таких условиях длинные строки обычно портят весь эффект отступов.

Насколько я понимаю, главное достоинство колонок фиксированной ширины состоит не в облегчении печати, как многие считают, а в том, что вы получаете возможность открыть на одном мониторе несколько окон редактора, одно подле другого.

На практике в C++ образуются очень длинные строки. Это более многословный язык, чем C; он приводит к вызову членов-функций объектов, на которые ссылаются другие объекты с помощью контейнеров-шаблонов... Существуют способы управиться с обилием появляющихся при этом длинных строк, один из которых – сохранение промежуточных ссылок во временных переменных.

3. В какой мере следует детализировать стандарт кодирования?

a. Насколько строго следует придерживаться выбранного стиля? Какие наказания могут быть за уклонение от него?

b. Допустим, что стандарт оказывается непомерно мелочным и ограничительным. Каковы могут быть последствия?

За отклонение от любого стандарта кодирования следует отрубать все шесть ног.

Правильный ответ весьма зависит от того, насколько всеобъемлющ стандарт кодирования и какова принятая у вас культура кодирования. Обычно есть более серьезные проблемы, чем не там поставленная скобка, но по поводу скобок проще предъявлять претензии. Я видел много стандартов, которые были настолько ограничительными и парализующими, что несчастные программисты просто напрочь игнорировали их. Чтобы приносить пользу и быть принятым, стандарт кодирования должен оставлять некоторое место для маневра, при этом желательно привести *лучший практический* подход в качестве примера.

4. Если создается новый стиль представления, какое количество объектов или ситуаций следует регламентировать? Какие дополнительные правила представления следует установить? Перечислите их.

Если выписывать каждое правило представления отдельно, придется рассмотреть непомерно большое число случаев. Стиль кодирования – это тонкая игра многих факторов: помимо отступов в ней участвуют внутренние пробелы, выбор имен, размещение операторов, располо-

жение скобок, содержимое файлов, применение и порядок заголовочных файлов и многое-многое другое.

Ниже приводится список элементов, который, несмотря на свою длину, далеко не полон. Это хорошая отправная точка для создания стиля. На практике не все элементы одинаково важно стандартизировать. При чтении списка определите, что в каждом пункте предпочтительнее вам самому. Выясните также, какое соглашение принято в вашем текущем проекте.

Поля кода

- Левый край кода определяется количеством пробелов для отступа. Часто встречаются отступы в два или четыре пробела, хотя некоторые программисты дипломатично выбирают три. Чем меньше отступ, тем реже вы будете залезать на правое поле, но при этом создается впечатление тесноты и труднее различать уровни вложенности. Большие отступы выглядят более отчетливо, но при этом быстро кончается место в строке.
- Что использовать для отступов – пробелы или табуляцию – давний спор, в котором не один программист потерял здоровье. Пробелы лучше переносимы; в любом редакторе они выглядят одинаково. При показе кода пропорциональным шрифтом[1] табуляция обеспечивает лучшее выравнивание.
- Ширина страницы определяет форматирование правого края кода. Можно ограничить строки фиксированным количеством колонок или никак не ограничивать их, что потребует горизонтальной прокрутки окна. Фиксированная ширина обычно устанавливается в количестве 79 или 80 символов. Это сложилось исторически; 80 символов – стандартная ширина терминала, но последнюю колонку не всегда можно использовать для отображения.
- Существуют варианты выравнивания некоторых конструкций. На каком уровне помещать `public:`, `private:` и `protected:` в объявлении класса? Где ставить метки `case` в операторах `switch`? Как форматировать метки оператора `goto`, которым вы никогда не пользуетесь?[2]

Пробелы и разделение

- Фрагменты кода можно выравнивать в соответствии с некоторой внутренней схемой табуляции, например выравнивать операторы по одной колонке во всех последовательных строках. Этим достигается зрительное выделение функции блока операторов. Однако при этом приходится больше печатать с клавиатуры и труднее модифицировать код, поэтому не все программисты придерживаются

---

[1] Чаще встречается в напечатанном коде, чем в редакторе исходного текста.

[2] Потому что в нынешние просвещенные времена, разумеется, ни один программист высокого класса не станет пользоваться этими `goto`. См. раздел «Структурное программирование» на стр. 531.

такого стиля. Горизонтальная табуляция может выглядеть примерно так:

```
int   cat   = 1;
int   dog   = 2;
char *mouse = "small and furry";
```

Пробельные символы могут появляться практически всюду, и есть много способов выделения пустым пространством отдельных предложений кода. Полезно выделять пробелами операторы, например: `hamster = "cute"`. Это примерно то же, что выделение пробелами слов на письме. Альтернативный вариант – `hamster="ugly"` – выглядит слишком плотно.

- Аналогично есть много способов оформления пробелами вызовов функций. Можно воспользоваться одним из следующих форматов:

```
feedLion(mouse)
feedLion( hamster )
feedLion (motherInLaw)
```

Последний вариант многим не нравится: в математическом выражении не должно быть пробела после имени функции.

Следует ли придерживаться того же правила с ключевыми словами? Как выглядит `while(lionIsAsleep)`? Скученно. Ключевые слова не функции; они больше похожи на слова, поэтому чаще всего их отделяют пробелами.

- Если код оказывается слишком длинным для одной строки, его нужно разбить на части, но где его разбивать – тоже вопрос. Естественно разбить там, где это логично, но что для одного логика, то для другого – безумие. Обычно строки переносят в районе оператора, но делать ли это до или после него, т. е. оставлять оператор в конце строки или начинать с него новую, – дело вкуса.

Переменные

- Классический пример раздоров в C/C++ – размещение звездочки в объявлении указателя (так называемые «звездные войны»). Можете выбрать один из трех вариантов:

```
int *mole;
int* badger;
int * toad;
```

В первых двух «указательность» ассоциируется с переменной и типом соответственно. Ассоциация с типом не проходит в таких предложениях, как `int* weasel, ferret;`. Третий вариант – разумная промежуточная позиция, но она не столь распространена.

- В некоторых стандартах C/C++ требуется, чтобы все константы писались прописными буквами для лучшей их видимости. Другие утверждают, что прописными буквами следует писать только имена макросов препроцессора.

Строки кода

- Форматирование определяет, *что* именно находится в каждой строке; часто требуется, чтобы каждое предложение (statement) кода занимало отдельную строку, что отчетливо выделяет его.
- Это приводит к проблеме *побочных эффектов* в предложениях; допустим ли код типа `index[count++] = 2` или присваивание в `if`?
- В некоторых стилях представления код располагается на одной строке с открывающей скобкой:

```
for (...) { ostrich++;
            buryHead(ostrich);
          }
```

Конструкции

- Всегда ли ставить фигурные скобки, даже если в них находится всего один оператор? Можно опускать скобки, если код располагается на той же строке, например:

```
if (weAreAllDoomed) startPanicking();
```

- Часто фразы (clause) `else` выравнивают по той же колонке, что соответствующий `if`, но иногда их ставят с большим уровнем отступа.
- Насколько важно выделять *особые случаи*? В некоторых стандартах требуется, чтобы все переходы вниз в вариантах оператора `switch` помечались комментариями. Аналогично *во избежание путаницы* в циклах нужно помечать отсутствие операций, иначе следующий маленький цикл без тела, который находит конец строки C `str`, может смутить неопытного читателя:

```
char *end;
for (end = str; *end; ++end);
```

- Где помещать встраиваемые методы C++ – внутри объявления класса, вне его (сразу после) или в отдельном файле?

Файлы

- Одно из главных решений – как разбить проект на файлы и какую информацию в них поместить. Нужно ли каждый класс или функцию помещать в отдельный файл? Или файлы должны быть более мелкими или крупными, например отдельный файл для библиотеки или раздела кода? Как быть, если есть много маленьких взаимосвязанных классов? Хорошо ли иметь множество маленьких взаимосвязанных классов?[1]
- Правила разделения файла на секции различны. Одни программисты вставляют в качестве разделителя несколько пустых строк, другие – блоки комментариев, некоторые любят ASCII-искусство.

---

[1] Для Java ответ ясен: имя класса обязано соответствовать имени файла.

- В C/C++ стиль представления может определять порядок файлов, подключаемых директивой `#include`. Тут есть разные течения мысли. Одни включают сначала системные файлы, затем файлы проекта и в конце специфические для данного файла. Другие считают, что надежнее обратный порядок; это предохраняет от зависимости заголовочного файла от заголовков, обычно включаемых перед ним. В некоторых стандартах требуется, чтобы в *самих заголовочных файлах* отсутствовали `#include`, а в файлах реализации содержалось длинное перечисление.

Разное

В конкретных случаях всегда возникает масса других проблем. Как форматировать команды SQL в коде, работающем с базой данных? Нужно ли соблюдать единообразие среди нескольких языков, применяемых в проекте?

5. Что важнее — хорошее *представление* кода или хорошее *проектирование* кода? Почему?

Вопрос очень искусственный. То и другое совершенно необходимы для хорошего кода, и никогда не следует жертвовать одним ради другого. Нехорошо, если вас ставят перед таким выбором. Однако по вашему выбору можно многое сказать о вас как о программисте.

Несомненно, плохое форматирование легче исправить, чем плохой проект, особенно если воспользоваться средствами автоматизации, которые наведут в формате кода единообразие.

Существует любопытная связь между представлением и проектом: плохое представление часто свидетельствует о том, что код писал плохой программист, что, в свою очередь, может означать, что внутренняя конструкция тоже плоха. Либо это следствие того, что код сопровождался разными программистами, что привело к утрате первоначального проекта.

Вопросы личного характера

1. У вас есть свой постоянный стиль?

a. Работая с чужим кодом, какой стиль вы применяете — авторский или свой?

b. В какой мере ваш стиль кодирования определяется автоматическим форматированием, применяемым в вашем редакторе? Служит ли оно достаточной причиной для принятия определенного стиля?

Если вы не можете управлять тем, как редактор устанавливает курсор, вам нужен другой редактор (либо вы́ ничего не умеете, либо ваш редактор).

Если вы не в состоянии писать код в одном и том же стиле, ваш диплом программиста следует аннулировать. Если вы не в состоянии повто-

рить стиль представления другого программиста, вас нужно назначить сопровождать BASIC на всю оставшуюся жизнь.

Контролируйте свою позицию: программисту свойственно больше заботиться о своем коде, личных приемах и личных пристрастиях в оформлении, чем о благополучии проекта в целом. Очень часто возникает проблема *отношений личности с командой*. Если программист противится стилю, насаждаемому организацией, или не может сопровождать код так, чтобы сохранялся его стиль представления, это плохой признак, свидетельствующий о неспособности программиста видеть общую картину.

2. **Табуляция – дьявольское изобретение или благо? Объясните.**
   a. **Что вам известно о своем редакторе? Вставляет ли он символы табуляции автоматически? Какова величина табуляции в вашем редакторе?**
   b. **Некоторые *весьма* распространенные редакторы допускают отступы, состоящие одновременно из пробелов и табуляций. Затрудняет ли это сопровождение кода?**
   c. **Скольким пробелам должен соответствовать символ табуляции?**

Поскольку это вопрос, касающийся веры, я просто скажу: табуляция сакс! – и быстро ретируюсь. Я только добавлю, что хуже, чем отступы с помощью табуляции, только отступы с помощью табуляции *и* пробелов – это полный ужас!

Если ваш редактор вставляет символы табуляции (а возможно, и пробелы) незаметным для вас образом, попробуйте какое-то время попользоваться другим редактором, и вы увидите, как это неудобно. Задайте другое значение табуляции и посмотрите, во что превратится ваш код. «*Все пользуются одним редактором, поэтому нет никакой разницы*» – это непрофессиональный подход. Не все пользуются одним редактором, поэтому разница *есть*.

Вы встретите рекомендации относительно правильного выбора размера табуляции, причем старательно обоснованные. Это все очень хорошо; есть солидное исследование, доказывающее, что *три* или *четыре* пробела обеспечивают лучшие возможности для чтения. (Я предпочитаю четыре пробела, потому что не люблю нечетные числа!) Тем не менее табуляция не должна соответствовать никакому фиксированному количеству пробелов. Табуляция есть табуляция, она не пробел и не кратна пробелу. Если код структурирован с помощью табуляции, не должно иметь никакого значения, сколькими пробелами представляется табуляция – код должен легко читаться в любом случае. К сожалению, мне редко приходилось встречать код с отступами в виде табуляции, который удовлетворял бы этому требованию. Очень часто табуляцию и пробелы смешивают, чтобы выровнять код. Это удается, если табуляция установлена так же, как у автора. В остальных случаях возникает невообразимый беспорядок.

3. **Есть ли у вас любимый формат кода?**
   a. **Опишите его несколькими простыми предложениями. Не упустите важное, например формат операторов `switch` и способ разбиения длинных строк.**
   b. **Сколько предложений вам потребовалось? Стало ли это для вас неожиданностью?**
   c. **Есть ли в вашей организации свой стандарт кодирования?**
   d. **Знаете ли вы, где его найти? Рекламируется ли он? Вы его читали?**
      i. **Если да: он вас удовлетворил? Сделайте честный анализ и сообщите свое мнение авторам документа.**
      ii. **Если нет: правильно ли это? (Обоснуйте свой ответ.) Существует ли общепринятый, но неписаный стиль кода? Можете ли вы оказать влияние на то, чтобы стандарт был принят?**
   e. **Нет ли у вас *нескольких* стандартов, например для разных проектов? Если да, то как в этих проектах используется общий код?**

Проверьте, что вам известны руководства по стилю (или неписаные соглашения), в соответствии с которыми вы должны работать.

Этот вопрос отчасти вызван моим личным опытом: я работал в крупной организации, где было несколько подразделений и в каждом были свои правила оформления кода. Поскольку отдельные продукты постепенно соединялись вместе, было технологически (а также коммерчески) оправданно объединить некоторые части базового кода. В результате образовался код, в котором перемешались разные стили интерфейса, разные представления и даже разное применение языка. Все это выглядело очень неорганизованно и непрофессионально и создавало большие трудности в работе.

4. **Много ли разных форматов кода вы испробовали?**
   a. **Какой из них вам больше пришелся по душе?**
   b. **В каком существовали наиболее строгие ограничения?**
   c. **Связано ли это между собой?**

После нескольких лет работы программистом может выработаться свой особый стиль форматирования, источники и причины которого трудно вспомнить. Конечно, это сочетание кода, который вы читали или с которым работали, и ваших личных пристрастий. Присмотритесь к нему и удостоверьтесь, что ваш стиль кодирования действительно надежен. Возможно, настала пора модифицировать и усовершенствовать его.

Изменить свой стиль не так-то просто. Существует ваш старый код – нужно ли переделать его в соответствии с новым стилем или оставить в прежнем состоянии?

Возьмите текстовый редактор и поместите в него следующий фрагмент кода; он вычисляет n-е простое число. Код написан в определенном

стиле. Измените его представление согласно *своим* вкусам. Не касайтесь существа реализации.

```
/* Возвращает результат проверки num на простоту.*/
bool
isPrime( int num ) {
     for ( int x = 2; x < num; ++x ) {
         if ( !( num % x ) ) return false;
     }
     return true;
}

/* Эта функция вычисляет `n`-е простое число./
int
prime( int pos ) {
     if ( pos ) {
         int x = prime( pos-1 ) + 1;
      while ( !isPrime( x ) ) {
           ++x;
      }
      return x;
  } else {
      return 1;
  }
}
```

Это реальный код, который не нужно воспринимать как глупое и неинтересное упражнение.

Обратите внимание, что я не настаиваю на том, каким должен быть правильный ответ. Мое форматирование так же законно, как ваше и как в оригинальном коде. Вот почему это вопрос *личного характера*.

Если вы читаете ответы, не попытавшись самостоятельно разобраться с вопросами, попробуйте хотя бы этот. Книга подождет, пока вы напечатаете несколько строк...

Теперь поглядите на то, что написали.

- Какие отличия у вашей версии? Сколько конкретных изменений вы внесли?
- Для каждой модификации задайте себе вопрос: связана ли она с эстетическими предпочтениями или можно найти для нее рациональное обоснование? Проверьте это обоснование – насколько оно веско? Готовы ли вы защищать его?
- Насколько уютно вы чувствовали себя в исходном формате? Вам было трудно его читать? Смогли бы вы работать в таком стиле кодирования, если бы понадобилось? Смогли бы привыкнуть к нему?

Дайте себе дополнительные очки, если вам захотелось написать код по-другому, сделав его более эффективным, и еще очки, если вы устояли перед таким соблазном. (Преждевременная оптимизация – это порочная практика; см. раздел «Технические подробности» на стр. 274.)

Глава 3. Что в имени тебе моем?

Вопросы для размышления

1. **Хорошо ли выбраны следующие имена переменных? Ответом может быть «да» (объясните почему и в каком контексте), «нет» (объясните почему) или «не знаю» (объясните почему).**
   - **a.** `int apple_count`
   - **b.** `char foo`
   - **c.** `bool apple_count`
   - **d.** `char *string`
   - **e.** `int loop_counter`

Правильный выбор имени зависит от контекста, поэтому трудно сказать, хороши ли эти имена. Поэтому в вопросе и говорится о контексте. Очевидно, есть контексты, в которых эти имена окажутся неудачными: `apple_count` не слишком подходящее имя для счетчика грейпфрутов.

`foo` – *всегда* плохое имя. Никогда не видел, чтобы кто-нибудь считал *foo*. `loop_counter` – тоже плохо; даже если цикл слишком велик для короткого имени счетчика, можно выбрать более содержательное имя, которое отражало бы фактический смысл переменной, а не ее применение в качестве счетчика цикла.

Трудно сказать, хорошее ли имя `bool apple_count`; скорее всего, нет – булева переменная не может содержать число. Возможно, оно отражает допустимость определенного количества яблок, но тогда следовало бы выбрать другое имя: `is_apple_count_valid`.

2. **В каких случаях оправдан такой выбор имен функций? Какие типы возвращаемых значений или параметров предполагаются? Какие типы возвращаемых значений делают такие имена бессмысленными?**
   - **a.** `doIt(...)`
   - b. `value(...)`
   - c. `sponge(...)`
   - d. `isApple(...)`

Смысл каждого из этих имен зависит от их местонахождения. Все определяется контекстом; контекст обеспечивает охватывающая область видимости функции. Контекстная информация может также задаваться параметрами функции или возвращаемыми переменными.

3. **Что важнее для системы именования – легкость чтения или легкость написания кода? Как можно облегчить то или другое?**
   - **a. Сколько раз вы пишете один и тот же фрагмент кода? (Подумайте.) Сколько раз вы его читаете? Ответы на эти вопросы должны помочь определить относительную важность.**

b. Как вы поступаете при противоречиях в принципах именования? Допустим, вы работаете над кодом C++ в стиле camelCase и должны модифицировать библиотеку STL (используется_подчеркивание). Как лучше поступить в такой ситуации?

Я работал с базовым кодом C++, в котором такая коллизия правил именования применялась с пользой. Во внутренней логике использовался camelCase, а в библиотеках и компонентах, расширявших стандартные библиотеки, применялись правила_именования STL. Это было достаточно удобно и разграничивало отдельные части проекта.

К сожалению, не всегда все проходит так удачно. Я часто сталкивался с кодом, где изменение стиля не могло быть объяснено никакими разумными причинами.

4. При каком размере цикла следует дать осмысленное имя переменной цикла?

Все зависит от длины вашей веревочки. Очевидно, что цикл из 100 строк, в котором счетчик назван `i`, не лучший вариант.[1] Вставляя в цикл дополнительный код, проверьте, не стоит ли одновременно изменить имя счетчика цикла.

5. Если в C `assert` является макросом, почему его имя пишут строчными буквами? Зачем дают макросам выделяющиеся имена?

`assert` не выделяется заглавными буквами, потому что `assert` не выделяется заглавными буквами. В идеале следовало бы это делать, но таковы правила, и приходится смириться с тем, что это имя макроса второго сорта. Увы.

Огонь полезен, но он может быть опасен. То же с макросами. Макросы и определения констант `#defined` опасны – принятие соглашения о выделении их имен ПРОПИСНЫМИ БУКВАМИ позволяет избежать нехороших конфликтов с обычными именами. Это столь же разумно, как надевать защитные очки, когда рядом бродит лунатик с острой палкой.

Ввиду того что макросы могут причинить столько неприятностей, следует выбирать имена, конфликты с которыми маловероятны. Но самое главное – старайтесь всеми мерами избегать препроцессоров.

Длинные расчеты легче читать, если поместить промежуточные результаты во временные переменные. Предложите хорошие правила составления имен для переменных этого типа.

Плохие имена временных переменных: `tmp`, `tmp1`, `tmp2` и т. д, а также `a`, `b`, `c` и т. д. К сожалению, они часто встречаются на практике.

Как и все прочие объекты, имена временных переменных должны быть осмысленны (например, `радиус_круга` в тригонометрических расчетах или `количество_яблок` в программе для садоводства). На практике

[1] Да и сам цикл из 100 строк – не лучший вариант.

хорошие имена способствуют документации внутренней логики и показу происходящего.

Если вы столкнетесь с величиной, назначение которой трудно объяснить определенным именем, и если это действительно произвольная промежуточная переменная, то вы поймете, почему так распространено имя `tmp`. По мере сил старайтесь избегать таких имен, как `tmp` – попытайтесь разбить вычисления каким-то более осмысленным образом.

6. Каковы достоинства и недостатки принципов именования стандартной библиотеки вашего языка?

Стандартные библиотеки часто показывают образцы применения языка, поэтому полезно следовать принятым в них соглашениям. Такой стиль именования окажется знаком другим программистам, поэтому они реже будут сталкиваться с неприятными неожиданностями и будут уютно чувствовать себя с вашим кодом.

С другой стороны, библиотеки не всегда дают лучшие образцы, поэтому будьте осторожны! Хорошим примером служит отвратительное название макроса `assert` в C.

7. Может ли надоесть имя? Правильно ли пользоваться одним и тем же именем локальной переменной в разных функциях? Правильно ли пользоваться локальными именами, которые перекрывают (и скрывают) глобальные имена? Почему?

Вполне допускается повторение имени локальной переменной в различных контекстах. Иногда это оказывается хорошей практикой: зачем каждый раз выбирать новое имя для счетчика цикла? Это только затруднило бы чтение кода.

Не закрывайте глобальные имена именами локальных переменных; это вносит путаницу и указывает на хрупкость кода.

8. Объясните сущность венгерской нотации. Каковы ее достоинства и недостатки? Согласуется ли она с современными принципами разработки кода?

Венгерская нотация – это правило, которое добавляет к именам переменных и функций таинственные префиксы, обозначающие тип. Оно употребляется преимущественно в C-коде. Есть несколько диалектов, имеющих тонкие различия, но наиболее употребительные префиксы венгерской нотации приведены в табл. 1.

Венгерская нотация была довольно неприятна в C (к тому же она стала ненужной, когда язык стал значительно более типизированным) и быстро вызывает тошноту в C++, поскольку плохо масштабируется, когда вы определяете многочисленные новые типы.

Если вы действительно хотите запутать программиста сопровождения, воспользуйтесь венгерской нотацией, а потом, через несколько

месяцев, поменяйте типы всех переменных, не меняя их имен (потому что менять их слишком долго). Это реальная слабость данной системы.

Таблица 1. Стандартные префиксы венгерской нотации

Префикс	Значение
`p`	указатель на... (`lp` означает *long pointer*, старую проблему архитектуры; если не знаете о ней – и не надо)
`r`	ссылка...
`k`	константа
`rg`	массив
`b`	булев (`bool` или C `typedef`)
`c`	`char`
`si`	`short int`
`i`	`int`
`li`	`long int`
`d`	`double`
`ld`	`long double`
`sz`	строка `char`, оканчивающаяся нулем (заметьте: *не* `p`)
`S`	`struct`
`C`	`class` (можно придумывать свои сокращения для классов)

Бойтесь венгерской нотации, как чумы.

Некоторые схемы именования имеют тайную склонность к венгерской нотации. Возьмите, например, `foo_ptr` и `m_foo`, приводившиеся в этой главе. Есть и другие изобретательные системы схожего типа: некоторые программисты называют свои глобальные переменные `theFoo`, а переменные-члены – `myFoo`. Наверное, это свидетельствует о том, что в венгерской нотации заложена хорошая идея, но, доведенная до логической крайности, она становится тиранической системой. Будьте настороже.

9. Часто классы содержат члены-функции, предназначенные для *чтения* и *записи* значений определенных свойств. Какие схемы именования таких функций существуют и какие из них предпочтительней?

Хотя иногда утверждают, что наличие методов get и set свидетельствует о слабости конструкции, классы, написанные подобным образом, встречаются очень часто. В некоторых языках есть даже встроенная поддержка этих операций.

Есть несколько разных схем именования. Если вы пишете на C++ в стиле camelCase и у вас есть свойство *foo* типа *Foo*, можно выбрать:

```
Foo &getFoo();
void setFoo(const Foo &) const;
```

или

```
Foo &foo();
void setFoo(const Foo &) const;
```

или, может быть

```
Foo &foo();
void foo(const Foo &) const;
```

Выбор может определяться стандартом кодирования или вашими личными эстетическими пристрастиями. В этом случае я бы нарушил правило «*Имя функции всегда должно содержать глагол*» и выбрал второй вариант, поскольку он выглядит в коде наиболее естественно. Смотрите сами.

Если метод чтения при первом запуске должен выполнить длинные вычисления (даже если результат буферизуется для следующих обращений), то я бы поостерегся. Это уже не обычная функция, извлекающая значение, и приведенные схемы не отражают этого. `Tree::numApples` – хорошее имя для метода чтения, если только он не замирает на минуту, пока система распознавания образов не обнаружит все яблоки. В этом случае я бы предпочел, чтобы имя отражало такое поведение. `Tree::countApples()` наводит на мысли о существенных действиях благодаря глаголу в имени.

Вопросы личного характера

1. **Искусны ли вы в выборе имен? Какими из предложенных правил вы пользуетесь на сегодняшний день? Занимаетесь ли вы составлением имен и применением этих правил сознательно или это происходит стихийно? В какой области вы могли бы улучшить свою практику?**

Вернитесь к разделу «Технические подробности» на стр. 79. Сравните приведенные там правила и последний написанный вами фрагмент кода. Есть ли соответствие? В какой мере ваши имена определяются существующими правилами кодирования (к чему вас призывали на стр. 86), а в какой вы используете собственные правила?

2. **Есть ли в вашем стандарте кодирования правила для выбора имен?**
   a. **Охватывает ли он все рассмотренные нами ситуации? *Достаточно ли* он полон? Есть ли от него польза или он представляет собой формальность?**
   b. **Насколько детализированными должны быть правила именования в стандарте кодирования?**

Иногда стандарт кодирования с пространной схемой именования *осложняет* выбор имен – правил так много, что запомнить их и следовать им

всем становится трудно. Критически отнеситесь ко всему, что более ограничивает, чем принципы, изложенные в главе 3.

Для опытного мастера выбор хороших имен становится привычным, и ему не нужно для этого обращаться «за помощью» к стандартам. Авторы стандартов часто утверждают, что их стандарты помогут в выборе имен менее опытным программистам. Однако эти стандарты оказываются, как правило, не столь полезными – прегрешения неопытных программистов не ограничиваются плохим выбором имен. Качество их работы необходимо проверять с помощью рецензирования.

3. **Какое самое неудачное имя встретилось вам в последнее время? Были ли случаи, когда имена вводили вас в заблуждение? Как бы вы изменили их, чтобы избежать подобных проблем в будущем?**

Когда оно вам встретилось – при официальном рецензировании чьей-то работы или при сопровождении старого, забытого кода?[1] Лучше всего, когда плохие имена обнаруживаются и корректируются сразу после своего появления (вы еще помните, как правильно назвать этот объект). И сил при этом тратится меньше всего. Разбираться с именами спустя несколько месяцев бывает гораздо труднее.

4. **Приходится ли вам портировать код с одной платформы на другую? Как это повлияло на ваш выбор имен файлов, других объектов и общей структуры кода?**

В старых файловых системах количество символов в имени файла было ограничено. В результате имена файлов были некрасивыми и непонятными. Если вашей целевой платформой не является такая архаичная система, можете смело игнорировать такие ограничения.

Полиморфизм на основе имен файлов – хитрый способ замены кода при сборке проекта. К нему часто прибегают при выборе реализации для конкретной платформы в переносимом коде. Можно настроить пути поиска заголовочных файлов так, чтобы один и тот же `#include` загружал разные файлы в зависимости от платформы сборки.

Глава 4. Литературоведение

Вопросы для размышления

1. **Группировка взаимосвязанного кода делает более заметными эти связи. Как можно осуществить это группирование? Какие методы документируют эти связи наиболее заметным образом?**

Очевидно, для группировки можно применять такие средства, как общие префиксы и суффиксы имен, расположение в файловой системе, размещение сущностей в одном классе или структуре, пространстве

[1] Очевидно, что если это ваш код, в нем не может быть никаких проблем!

имен C++/C#, пакете Java, исходном файле или библиотеке кода. Можете ли вы предложить другие способы?

Сильнее всего связи, поддерживаемые языком – они более наглядны и подвергаются автоматической проверке. Однако близость размещения кода создает более мощную связь, чем может показаться вначале. Порядок тоже имеет большое значение – первый элемент всегда кажется более важным, чем остальные. Используйте эти обстоятельства для документирования своего кода.

2. Следует избегать в коде *волшебных чисел*. Считать ли ноль магическим числом? Какое имя вы дали бы константе, представляющей ноль?

Число «ноль» обладает магическими свойствами во многих контекстах; в коде C оно применяется в качестве значения *нулевого* указателя и начального значения в большинстве циклов. Чем *можно* заменить 0 ?

- Общая константа с именем `ZERO` ничуть не лучше, чем просто `0`; магия сохраняется. Из имени не следует, чем является нуль на самом деле – значением нулевого указателя или значением инициализации цикла? Этот подход не достигает цели.
- Разные имена для всех нулевых констант вызывают большие неудобства из-за необходимости многочисленных вариаций на тему `for (int i = SOME_ZERO_START_VALUE; i < SOME_END_VALUE; ++i)`. Никакой новой информации эти имена все равно не несут.

Над именами нулевых констант нужно хорошенько подумать. Очевидный вариант – что-то вроде `NO_BANANAS`, что означает *нулевое количество бананов*. Однако префикс `NO_` можно спутать с сокращением для числа (например, `NUM_`).

3. В самодокументируемом коде для передачи информации активно используется контекст. Приведите пример и покажите, как некоторое имя по-разному интерпретируется в различных функциях.

Есть много способов использования контекста для целей документирования. Пусть у нас есть класс `Cat`. Его члены-функции не обязательно называть `setCatName`, `setCatColor` и т. д.; часть *cat* неявно следует из контекста класса.

У многих английских слов есть несколько значений. Надо полагать, что переменная `count` (подсчет) в функции поиска будет содержать иную информацию, нежели одноименная в базе данных вампиров (граф). Ближе к реальности: переменная `name` в классе `Cat` явно будет хранить кличку кошки, тогда как та же переменная в классе `Employee` будет, скорее всего, хранить имя человека, его фамилию и обращение. Имя переменной одно, контексты разные. Как можно чаще используйте контекст при условии, что он очевиден.

4. Можно ли реально рассчитывать на то, что человек, впервые видящий некоторый самодокументируемый код, сможет в нем полностью разобраться?

Да, это наша цель, и она достижима. Однако читателю все равно потребуются обзорная и проектная документация, описывающая систему в целом, ее работу и структуру. Если эти данные оказываются в комментариях к коду, то место для них выбрано неудачно (или это очень маленькая система).

Когда код хорошо документирован, новичку должно быть совершенно понятно, чем занимается конкретный *раздел кода*. Полная документация по API показывает назначение любой функции, обращение к которой может встретиться новичку.

5. Если код действительно самодокументируемый, какой объем дополнительной информации необходим?

Все зависит от размера проекта. Необходимы функциональные спецификации и проектные документы. Обзор реализации также может потребоваться, и обязательно потребуются детальные спецификации тестирования.

Для документирования конструкции отдельного раздела кода достаточно хороших литературных комментариев, и тогда другая документация не нужна.

6. В каких случаях может потребоваться, чтобы в каком-то фрагменте кода мог разобраться кто-либо иной, кроме его автора?

Это реальность промышленного производства программ. Бессовестный программист может воспользоваться тем, что никто кроме него не разбирается в коде, чтобы закрепить за собой рабочее место. Написав код, который никто не может разгадать, вы гарантируете себе занятость до конца жизни (или до конца существования компании – в зависимости от того, что случится раньше). Недостаток такого решения в том, что до конца дней придется возиться со своей грязной стряпней.

На практике код, который не доступен для понимания другим программистам, представляет собой опасность. Если вы уйдете из компании, перейдете в другой отдел, получите повышение или у вас просто не будет времени для сопровождения, нужно кому-то передать вашу работу. И даже если до этого не дойдет, в один прекрасный день, когда вы уже забудете, как работает ваш код, обнаружится критическая ошибка, которую нужно немедленно исправить.

Рецензирование кода способствует тому, чтобы код был понятен и должным образом документирован.

7. Эту простую С-функцию *пузырьковой сортировки* можно усовершенствовать. Что именно в ней нехорошо? Напишите улучшенный, самодокументируемый вариант.

```
void bsrt(int a[], int n)
{
     for (int i = 0; i < n-1; i++)
```

```
            for (int j = n-1; j > i; j--)
                if (a[j-1] > a[j])
                {
                    int tmp = a[j-1];
                    a[j-1] = a[j];
                    a[j]   = tmp;
                }
    }
```

Прежде всего, никогда не применяйте пузырьковую сортировку. Есть множество лучших способов. Возможно, для вашего языка есть гораздо более удачная функция в общей библиотеке; например, для C имеется `qsort`. Я воспользовался здесь пузырьковой сортировкой только как простым примером.

Интерфейс функции *совершенно* неясен. Имя функции малопонятно, а имена параметров бессмысленны. Хотелось бы также увидеть комментарий с документацией API, но в своем переработанном варианте я его опущу.

Внутри кода также царит неразбериха. Его задача станет гораздо понятнее, если выделить код, меняющий местами элементы массива, выделить в отдельную функцию `swap`. Тогда читателю будет проще разобраться. Еще немного труда, и получаем следующий код:

```
void swap(int *first, int *second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}

void bubblesort(int items[], int size)
{
    for (int pos1 = 0; pos1 < size-1; pos1++)
        for (int pos2 = size-1; pos2 > pos1; pos2--)
            if (items[pos2-1] > items[pos2])
                swap(&items[pos2-1], &items[pos2]);
}
```

Это приличный C-код, хотя неплохо было бы его немного модифицировать. В зависимости от ваших пристрастий могут появиться фигурные скобки вокруг тела цикла. Ради эффективности можно было бы сделать из `swap` макрос. Тем не менее это не самая толковая оптимизация; явно следует выбрать более эффективный алгоритм сортировки.

В C++ я бы сделал `swap` встроенной, а параметры передавал по ссылке (что отражает тот факт, что они будут изменены). Самое лучшее – воспользоваться средством `std::swap`, предоставляемым библиотеками языка.

8. Работа со средствами документирования кода обнаруживает ряд интересных проблем. Выскажите свое мнение по следующим вопросам:

a. Следует ли при проверке документации *проверять* также и *код*, сравнивая с комментариями в файлах исходного кода, или *проверять спецификации*, сравнивая со сгенерированной документацией?

b. Где следует документировать протоколы и прочие не относящиеся к API вопросы?

c. Документируете ли вы закрытые/внутренние функции? Если это код C/C++, где поместить такую документацию — в файле заголовка или в файле реализации?

d. Если система большая, что лучше — создать один большой документ по API или несколько меньших по отдельным областям? В чем преимущества каждого из подходов?

По этому вопросу я думаю следующее:

a. Проверьте сгенерированные спецификации. Не обращайте слишком много внимания на расположение комментариев в исходном файле. Вы рецензируете содержимое, а не код.

b. Не нужно думать, что документация должна располагаться в *заголовочном файле* или *файле реализации*. Даже если средства документирования одобряются, нет ничего дурного в наличии отдельных «обычных» документов. Напишите в них о своем протоколе.

c. Задокументируйте все внутренние функции, которые этого требуют. Не обязательно писать подробную документацию по всем закрытым частям. Эти документы, если они достаточно велики, следует перенести в файл реализации, чтобы открытый интерфейс выглядел аккуратно и просто.

d. То и другое! Примените инструмент несколько раз, чтобы создать один крупный документ и документы для каждой подсистемы.

9. Если вы работаете с кодом, в котором отсутствует грамотная документация, и вам необходимо внести изменения или добавить новые методы или функции, что правильнее — снабдить их грамотными комментариями или оставить недокументированными?

Мастер *стремится* документировать и автоматически испытывает потребность написать блок комментариев. Если для кода существует отдельный документ-спецификация, то ваша документация должна войти в него наравне со всем прочим. Либо можно начать добавлять литературные комментарии. Следите, однако, за тем, чтобы автор не почувствовал себя обиженным!

10. Можно ли написать самодокументируемый код на языке ассемблера?

Можно попытаться, но это будет нелегко. Код ассемблера не очень выразителен; вы программируете не на уровне смысла, а на уровне «*глупый процессор, делай так!*». Код будет состоять в основном из блоков

комментариев (для ассемблера это вообще правильная практика). За исключением меток подпрограмм, для самодокументирования нет большого простора.

Вопросы личного характера

1. **Какой из встречавшихся вам кодов был лучше всего документирован? В чем были его особенности?**
   a. **Было ли у этого кода много внешних спецификаций? Многие ли из них вы прочли? Можно ли быть уверенным, что вы достаточно знаете о коде, вовсе не прочтя их?**
   b. **В какой, по вашему мнению, мере качество этой документации было обязано личному стилю программирования автора, а в какой – требованиям и наставлениям той организации, в которой он работал?**

Если код хорошо документирован, отдельные документы с описанием необязательны. Хорошее документирование достигается с помощью правильных имен, разделения на логические модули, применения простых технологий, четкого форматирования, регистрации предположений и толковых комментариев. Фирменный стиль полезен, но не заменяет умного и тонкого программирования. Если человек идиот, то он может написать отвратительный код даже при соблюдении самых строгих правил.

2. **Если вы программируете на разных языках, различаются ли ваши стратегии составления документации для каждого из них?**

Языки обладают разной степенью выразительности, поэтому возможности документирования в рамках синтаксиса языка различаются. Наряду со всем остальным это влияет на объем необходимых комментариев. Самодокументируемый код лучше удается, когда вы работаете с тем языком, с которым лучше всего знакомы.

3. **Каким образом вы выделили важные фрагменты в последнем коде, который писали? Постарались ли сделать малозаметной закрытую информацию?**

Внимательно отнеситесь к этому вопросу – естественно говорить себе, что *ты написал хороший код*. Но посмотрите на него так, будто его автор – кто-то другой. Отнеситесь к нему критически.

4. **Если вы работаете над кодом в составе группы, часто ли ее участники обращаются к вам с просьбой пояснить работу вашего кода? Смогли бы вы избавиться от таких отвлечений путем лучшего документирования кода?**

Хорошее решение этой проблемы включает в себя два пункта:

a. Если вопрос действительно касается какого-то неясного места в вашем коде, то, разъяснив его спрашивающему (и уточнив, что в действительности он *хотел* знать), задокументируйте каким-то спосо-

бом эти данные. Можно потом подтвердить их, послав по электронной почте.

b. Если вопрос уже разъяснен в документации, ткните спрашивающего носом в это место со словами RTFM.[1]

Глава 5. Заметки на полях

Вопросы для размышления

1. **Как могут различаться *необходимость* в комментариях и их *содержание* в следующих типах кода:**
   a. **Язык ассемблера низкого уровня (машинный код)**
   b. **Сценарии командного интерпретатора**
   c. **Однофайловая среда тестирования**
   d. **Крупный проект C/C++**

Язык ассемблера наименее выразителен и дает меньше всего средств для написания самодокументируемого кода. Вследствие этого в коде ассемблера требуется больше всего комментариев, причем на гораздо более низком уровне, чем в других языках – в комментариях кода на ассемблере обычно разъясняется и «почему», и «как».

Особой разницы между тремя оставшимися вариантами нет. Сценарии командного интерпретатора (shell scripts) бывают достаточно трудны для чтения; в этом отношении они представляют собой прото-Perl. Полезно писать подробные комментарии. В большой программе на C/C++ предпочтительным может оказаться грамотное программирование.

2. **Есть инструменты для подсчета процента строк комментариев в исходном тексте. Насколько они полезны? Насколько точно могут они оценить качество комментариев?**

Такая метрика позволяет оценить код, но не следует придавать ей большого значения. Она не точно отражает качество кода. В самодокументируемом коде может вообще не быть комментариев. Пространная история версий или длинное заявление об авторских правах могут занимать большую часть мелких файлов и влиять на эту метрику.

3. **Если вам встретился непонятный код, как лучше внести в него некоторую ясность – добавив комментарии с вашим пониманием его работы или переименовав переменные/функции/типы, дав им более содержательные имена? Какой подход может оказаться проще? Какой подход будет безопаснее?**

Оба подхода применимы в зависимости от ситуации. Переименование можно было бы счесть лучшим подходом, но оно опасно, если вы не знаете в точности, что делает функция. Может оказаться, что новое

[1] Read The (...) Manual.

имя будет столь же плохим. Меняя имя, убедитесь, что хорошо понимаете суть переименовываемого объекта.

Проверьте с помощью тестов модулей, что ваши модификации не нарушили работу кода.

4. **Если вы написали блок комментариев к C/C++ API, куда его лучше поместить – в заголовочный файл, где объявляется функция, или в файл, где находится ее реализация? В чем преимущества и недостатки каждого из расположений?**

Этот вопрос вызвал большие споры в одной фирме, где я работал. Некоторые доказывали, что описание должно находиться в файле `.c`. Когда оно находится рядом с функцией, сложнее написать неточный комментарий или написать код, который не соответствует документации. Кроме того, больше шансов, что комментарий изменится синхронно с модификацией функции.

Если же описание помещено в заголовочный файл, то оно находится рядом с открытым интерфейсом, что вполне логично. Кто станет искать в реализации документы по открытому API?

Инструменты грамотного программирования должны уметь извлекать комментарии из любого места, но иногда проще прочесть комментарии в исходном коде, чем запускать специальный инструмент – преимущество грамотного программирования. Я предпочитаю помещать комментарии в заголовочные файлы.

Конечно, в Java и C# есть только один файл с исходным кодом; обычно используются комментарии в формате Javadoc или C# XML.

Вопросы личного характера

1. **Рассмотрите внимательно файлы исходного кода, над которыми работали в последнее время. Прочтите свои комментарии. Так ли они хороши, если быть честным? (Ручаюсь, что при чтении кода вам захочется сделать несколько изменений!)**

Читая и рецензируя собственный код, легко пропустить комментарии, полагая, что они точные или хотя бы адекватные. Полезно потратить некоторое время на их чтение, чтобы оценить качество. Попросите кого-нибудь из коллег, которым вы доверяете, высказать свое (конструктивное) мнение по поводу вашего стиля комментариев.

2. **Как вы добиваетесь того, чтобы ваши комментарии были действительно полезными, а не представляли собой личные заметки, никому кроме вас не понятные?**

Вот некоторые правила, которым нужно следовать: пишите полными предложениями, не пользуйтесь сокращениями, аккуратно форматируйте комментарии и пишите их общепринятым языком (включая слова из предметной области). Избегайте шуток, лишних фраз и того, в чем вы не до конца уверены.

Слабости вашего стиля комментариев обнаружатся во время рецензирования.

3. **Придерживаются ли ваши коллеги одинакового стандарта написания комментариев, возможно, с небольшими различиями?**
   a. **Чьи комментарии лучше других? Почему? Чьи самые плохие? Есть ли корреляция между качеством их комментариев и качеством кода в целом?**
   b. **Не думаете ли вы, что если потребовать от членов вашей команды придерживаться определенного стандарта кодирования, то качество комментариев может вырасти?**

Пользуйтесь рецензированием кода, чтобы оценить качество комментариев ваших коллег и повысить степень единообразия комментирования в вашей команде.

4. **Включаете ли вы в исходные файлы историю их модификации? Если да:**
   a. **Поддерживаете ли вы ее вручную? Зачем, если система управления версиями может автоматически делать это вместо вас? Насколько точно ведется история?**
   b. **Разумна ли такая практика *в действительности*? Как часто возникает потребность в таких данных? Каковы преимущества хранения их в исходном файле по сравнению с иным механизмом?**

Людям свойственно неаккуратно вести журналы, даже если у них самые лучшие намерения. Это требует большого объема ручной работы, которой пренебрегают, когда времени мало. Следует вести журнал с помощью специальных средств и правильно выбирать место для информации (им, по моему мнению, не должен быть файл исходного текста).

5. **Добавляете ли вы свои инициалы или помечаете каким-то иным способом свои комментарии, которые делаете в чужом коде? Указываете ли вы дату комментирования? Когда и зачем вы это делаете – полезна ли такая практика? Сослужили ли вам когда-либо пользу чьи-то инициалы или отметка о дате?**

Для некоторых комментариев такая практика полезна. В других случаях это неудобно – приходится пробираться через эти помехи, чтобы добраться до существенного материала.

Лучше всего пользоваться временными комментариями `FIXME` (исправить) или `TODO` (что сделать), которыми помечается незаконченная работа. В окончательной версии сохранять их, пожалуй, не стоит.

Глава 6. Людям свойственно ошибаться

Вопросы для размышления

1. **Эквивалентны ли такие механизмы сообщения об ошибках, как *возвращаемые значения* и *исключительные ситуации*?**

Возвращаемые значения эквивалентны глобальным *переменным состояния*, поскольку оба механизма могут передавать один и тот же код (хотя на переменную состояния проще не обращать внимания). С помощью обоих методов можно написать код, действующий аналогичным образом.[1]

Исключения – совсем другое дело. В них участвуют новые потоки управления, что сильно отличается от простых кодов причины. Они тесно связаны с исполнительной средой языка и программы. Хотя можно моделировать исключения с помощью собственного кода, распространяющего ошибки, нужно тщательно определить:

- Представление ошибок в виде произвольных объектов, а не простых целочисленных кодов.
- Иерархию классов исключений и возможность перехвата через базовый класс.
- Распространение исключений любыми функциями, даже теми, где отсутствуют операторы `try`, `catch` или `throw`.

Именно последний пункт лучше всего демонстрирует, почему эти два метода не эквивалентны. Будучи реализованы на уровне языка, исключения нисколько не навязываются для использования в коде. Самодельная имитация должна уметь обрабатывать отказ в каждой точке. Каждая функция обязана возвращать код ошибки – даже если сама не может быть причиной сбоя, – чтобы распространять информацию об ошибке дальше. Это требует существенного адаптирования кода.

2. Можете предложить какие-нибудь реализации возвращаемых типов *кортежей*? Не ограничивайтесь каким-либо одним языком программирования. В чем достоинства и недостатки возвращаемых значений типа кортежа?

В C для каждого возвращаемого типа можно создать `struct`, связав с кодом причины отказа. Это будет выглядеть примерно так:

```
/* Объявление возвращаемого типа */
struct return_float
{
    int reason_code;
    float value;
};

/* Функция, использующая его ... */
return_float myFunction() { ... }
```

Это некрасиво, скучно писать, трудно применять и тяжело читать. Можно воспользоваться шаблонами C++ или генериками Java/C#,

[1] Тем не менее они не совсем совпадают. В C++ можно возвратить значение типа *proxy*, для которого в деструкторе определено поведение. Тем самым механизм кода возврата обретает дополнительные возможности.

чтобы автоматически строить данную конструкцию, либо привлечь класс C++ `std::pair`. В промышленном коде C++ можно встретить оба подхода. Тот и другой – нудные: требуются дополнительные объявления и механизмы, возвращающие эти типы. В некоторых языках, например в Perl, поддерживаются списки произвольных типов – такой способ реализации значительно проще. В функциональных языках тоже есть такая возможность.

Мы уже увидели определенные недостатки такой технологии: она очень докучлива в коде и не доставляет никакого удовольствия при чтении. Кроме того, это не идиоматическая практика кодирования. Возможен выигрыш в производительности, если возвращается не один аргумент, а несколько, но это не очень убедительно, если только вы не работаете на уровне машинного кода. Заметное преимущество состоит в том, что отдельный код успешности операции не смешивается с возвращаемыми значениями.

3. В чем различия между реализациями механизма исключительных ситуаций в разных языках?

Мы рассмотрим четыре основные реализации: C++, Java, .NET и структурированные исключения Win32. Исключения Win32 связаны с функционирующей платформой, остальные – со своими языками. Языки *могут* быть реализованы с использованием соответствующих средств платформы, на которой происходит работа, или без них.

Во всех случаях применяется идентичный подход: исключение генерируется оператором `throw` и перехватывается потом оператором `catch`, помещаемым после кода, заключенного в блок `try`. Все они следуют модели завершения исполнения программы.

В Java, .NET и Win32 есть также конструкция `finally`. Она содержит код, выполняемый как при нормальном, так и при аварийном выходе из блока `try`. В это место можно, например, поместить код для уборки и знать, что он всегда выполнится. В C++ можно моделировать `finally`, но это малоприятное занятие.

Чистые (raw) исключения Win32 (за вычетом поддержки в языке, предоставляемой компилятором) не производят уборку при развертывании стека, потому что в ОС нет понятия деструктора. Их нужно применять с осторожностью – они предназначены для обработки ситуаций, более близких к сигналам, чем к ошибкам в логике кода.

Исключения Java (потомки `Throwable`) и исключения C# (потомки `Exception`) автоматически выполняют обратную трассировку, очень полезную при дальнейшей отладке. В .NET CLI позволяет генерировать что угодно, но C# не предоставляет такой возможности (хотя предоставляет возможность перехвата). Другие языки .NET могут генерировать что угодно.

4. **Сигналы – это классический механизм UNIX. Нужны ли они по-прежнему, когда мы располагаем современными технологиями типа исключительных ситуаций?**

Да, они по-прежнему нужны. Сигналы являются частью стандарта ISO C, так что обойтись без них в любом случае не просто. Сигналы ведут свое происхождение от реализаций UNIX (до) System-V. Это асинхронный механизм для сообщения о проблемах/событиях системного уровня. Исключения решают другую проблему, сообщая об ошибках логики кода, которые могут достичь обработчика. Нет смысла генерировать исключения для событий типа сигналов, особенно в модели с завершением программы – она не обеспечивает асинхронной обработки.

5. **Какая структура кода лучше всего подходит для обработки ошибок?**

На этот вопрос не существует ответа. В разных ситуациях эффективны разные стратегии. Важно лишь надежно обнаруживать и обрабатывать ошибки с помощью ясного, понятного и доступного для сопровождения кода.

6. **Как бы вы поступили с ошибками, возникшими в вашем коде обработки ошибок?**

С ошибками, сообщения о которых поступают из обработчиков событий, следует поступать так же, как с любыми другими ошибками. Однако все быстро становится отвратительным, когда обработчики ошибок вложены в обработчики ошибок, вложенные в обработчики ошибок. Следите, чтобы этого не происходило, и поищите более толковый способ организации своего кода.

Хороший способ – выполнять в обработчиках только такие операции, которые гарантированно будут успешными (или обещают не генерировать исключений). В этом случае жить становится гораздо легче.

Вопросы личного характера

1. **Насколько полно реализована обработка ошибок в вашем нынешнем программном проекте? Как это отражается на стабильности программы?**

Есть прямая связь между надежной обработкой ошибок и стабильностью кода. Либо от вашей программы не требуется надежность, либо она должна систематически обнаруживать и обрабатывать все аварийные состояния. Если это не является основополагающим принципом философии программы, вы не получите надежной системы.

2. **Занимаетесь ли вы обработкой ошибок во время написания кода или она является неприятным отвлечением, которое вы откладываете на более позднее время?**

Нелюбовь к обработке ошибок естественна; кому может понравиться постоянно думать о негативных аспектах функционирования про-

граммы?[1] Однако прислушайтесь к следующему важному совету: не откладывайте это дело в долгий ящик. В противном случае вы обязательно пропустите важные ошибки, которые в один прекрасный день приведут к неожиданному поведению программы. Выработайте у себя привычку думать об ошибках сразу.

3. **Возьмите последнюю написанную вами функцию (достаточного размера) и тщательно проанализируйте ее код. Найдите все необычные ситуации и условия возможного возникновения сбоев. Все ли они учтены в коде обработки ошибок?**

   Теперь предложите проанализировать свой код кому-то постороннему. Не стесняйтесь! Нашлись ли другие опасные ситуации? Почему? Как это характеризует код, над которым вы работаете?

Это хорошо продемонстрирует, насколько тщательно вы программируете в действительности. Старательно проделайте это упражнение и *обязательно* привлеките кого-то постороннего. Даже самые опытные программисты пропускают отдельные случаи возможных ошибок.[2] Если маловероятно, что они проявят себя как ошибки, можно не обратить на них внимания и жить дальше под угрозой возможных неприятностей.

При использовании исключений пропустить возникновение ошибки не так легко – исключения проложат себе дорогу вверх по стеку вызовов независимо от того, обрабатываете вы их или нет. И в этом случае можно написать плохой код, если он не обрабатывает исключения надежным образом (он может завершить работу, оставив систему в плохом состоянии или с неосвобожденными ресурсами) или слишком рьяно осуществляет перехват (берясь за ошибки, которые на текущем уровне не могут быть обработаны; именно поэтому не пишите `catch(...)`, что перехватывает все исключения).

4. **Как вам проще справляться со сбойными ситуациями – с помощью *возвращаемых значений* или *исключений*? Вы уверены, что умеете писать код, корректно обрабатывающий исключительные ситуации?**

В некоторой степени это зависит от ваших привычек. Исключения дополняют и расширяют возврат значений. Тот, кто пользуется исключениями, разбирается и в возвращаемых значениях, но обратное верно не всегда. Возвращаемые значения более очевидны, и потому их легче правильно применять.

Если же вы применяете исключения, нужно знать, чего опасаться. Надежная обработка оказывает влияние на *весь* ваш код, а не только на те части, где генерируются и перехватываются ошибки. Надежная обработка исключений представляет собой большую и сложную тему,

---

[1] Если у вас есть такая склонность, из вас может получиться очень хороший тестер программ. Но не бросайтесь менять место работы – по-настоящему скрупулезные программисты встречаются редко.

[2] К примеру, часто ли в C проверяют ошибки, возвращаемые `printf`?

которая требует глубокого изучения. Не следует недооценивать ее влияние на стиль вашего программирования.

Глава 7. Инструментарий программиста

Вопросы для размышления

1. **Что важнее — всем членам команды разработчиков пользоваться одинаковой IDE или каждому выбрать для себя наиболее подходящую среду? Каковы последствия использования разных инструментов в одной команде?**

Каждый профессиональный программист должен ответственно подойти к выбору инструментов, обеспечивающих для него высшую производительность. Все программисты разные, и естественно, что они могут предпочитать разные инструменты. Если выбор осуществляется исходя из практических соображений, это положительно отражается на общей продуктивности команды. Но требуя от самостоятельно мыслящих программистов, чтобы они пользовались конкретными инструментами, трудно добиться от них хорошей работы.

Если члены команды пользуются разными средами разработки, то их совместная работа должна быть правильно организована. Их код должен быть *идентичен*, а при редактировании чужого файла его форматирование не должно меняться.

2. **Какой минимальный набор инструментов должен быть в распоряжении каждого программиста?**

Нельзя обойтись хотя бы без такого набора:

- Элементарный редактор
- Минимальная поддержка языка (компилятор, интерпретатор или то и другое в зависимости от языка)
- Компьютер, на котором они выполняются

Но при таком минимальном наборе нельзя эффективно программировать. Для серьезной работы нужны дополнительные инструменты.

- Необходима система контроля версий, иначе работать становится опасно.
- При наличии достаточного комплекта библиотек не нужно будет изобретать колесо и снизится риск появления ошибок.
- Необходим также инструмент сборки для создания программной системы.

Такой минимальный набор более реалистичен. Чем больше базовых инструментов вы введете, тем проще будет вести разработку и тем лучше будет готовый код.

3. **Какие инструменты мощнее — основанные на командной строке или использующие GUI?**

Стоит надрать вам уши, если вы стали отвечать на этот вопрос. Командная строка и GUI – разные вещи. Весь сказ.

Возникает интересный философский вопрос: *Что в этом контексте означает «более мощный»?* Имеющий больше экзотических функций? Более простой в применении? Быстрее работающий? Или легче вписывающийся в производственный цикл? Дайте сначала определение и исходя из него обоснуйте ваш ответ. Тогда я, может быть, пощажу ваши уши.

4. **Есть ли строительные инструменты, не являющиеся программами?**

Мы уже отнесли в категорию инструментов языки и библиотеки, поэтому ответ будет положительным. Можно привести другие хорошие примеры:

- Регулярные выражения
- Графические компоненты («виджеты» GUI)
- Сетевые сервисы
- Стандартные протоколы и форматы (например, XML)
- Диаграммы UML
- Методологии проектирования (например, карточки CRC)

5. **Какое качество важнее всего для инструмента?**
   a. **Универсальность интерфейса**
   b. **Гибкость**
   c. **Настраиваемость**
   d. **Мощь**
   e. **Легкость освоения и применения**

Все они важны. Относительная ценность может меняться в зависимости от типа применяемых инструментов и ситуации, в которой они используются.

Мощь важна; инструменты должны быть *достаточно* мощными для решения поставленной задачи, иначе ваша жизнь превратится в кошмар. Если бы это было не так, программисты редактировали бы свой код с помощью Notepad или vi.

Вопросы личного характера

1. **Какие инструменты входят в ваш стандартный набор? Какими из них вы пользуетесь ежедневно? Какими пользуетесь несколько раз в неделю? К каким обращаетесь в редких случаях?**
   a. **Насколько уверенно вы пользуетесь ими?**
   b. **Насколько полно вы используете возможности каждого инструмента?**
   c. **Как вы учились работать с ними? Пытались ли когда-нибудь совершенствовать свои навыки работы с ними?**

d. Известны ли вам более удачные инструменты?

Важнейшим является последний в списке вопрос. Честно признайтесь, существуют ли более эффективные инструменты, чем те, которыми вы пользуетесь? Стоит оглядеться по сторонам. Если найдете лучшие инструменты, достаньте их и поэкспериментируйте с ними.

2. Насколько современны ваши инструменты? Имеет ли для вас значение, что их версии не самые новые?

Устаревшие инструменты могут служить источником неприятностей, равно как и новейшие версии. Самые большие неприятности возникают, когда версия одного из инструментов не соответствует другим, участвующим в инструментальной цепочке. Из-за отличия версий могут обнаружиться скрытые функциональные несоответствия, из-за которых цепочка будет работать неверно. Симптомом этого чаще оказывается не сбой в работе цепочки, а код, ведущий себя необъяснимым образом.

Устаревшие инструменты могут страдать от ошибок, устраненных в более новых версиях. Вы можете не почувствовать необходимости обновления, пока вас не ударит та ошибка, которая исправлена в новых версиях. Задним умом все крепки. Отстав в обновлении, вы можете оказаться в заложниках у инструментов, которые более не поддерживаются, выпущенных более не существующими компаниями. Это может обусловить серьезные проблемы в важном проекте.

Разумеется, всегда можно загрузить и установить новую версию инструмента. Однако по ряду причин обновление может оказаться нецелесообразным. Например, оно окажется для вас слишком дорогим. Либо оно потребует обновления вашей ОС или других важных элементов технологической линии, что не всегда разумно.

3. Чему вы отдаете предпочтение – интегрированным наборам (типа визуальной среды разработки) или цепочкам отдельных инструментов? В чем преимущества *противоположного* подхода? Насколько велик ваш опыт работы *в том и другом* стиле?

Необдуманный ответ может стоить вам ушей (см. ответ на вопрос 3 в разделе «Вопросы для размышления» на стр. 612). Попробуйте составить список преимуществ другого способа работы, чтобы избежать узкого и предвзятого взгляда.

4. Соглашаетесь ли вы с настройками своего редактора по умолчанию или стремитесь переделать в нем все на свой вкус? Какой подход «правильнее»?

Изучив возможности настройки своего редактора, можно научиться работать с ним максимально эффективно. В таком случае тонкая настройка может оказаться более правильной. Позиция прагматика, вероятно, должна быть где-то посередине (хороший пример *принципа Златовласки*: любые крайности редко приносят успех). Нет смысла

настраивать функции, которыми вы не пользуетесь. Некоторые вещи не имеют особого значения – меня ничуть не волнует, какая цветовая схема используется в редакторе. Зато есть существенные вещи – я не хочу оставлять стиль форматирования кода, устанавливаемый по умолчанию, если он слишком вычурный.

Гораздо лучше кодировать в тщательно подобранном вами стиле, чем полагаться на установки редактора по умолчанию. Этого может *потребовать* стиль, принятый в фирме. Я лучше настрою свой редактор, чтобы он автоматически форматировал код так, как мне нужно, чем буду мучиться с перемещением курсора после каждого нажатия ENTER.

Такого рода соображения касаются не только редакторов, но и всех программных инструментов, допускающих настройку.

5. **Как вы определяете долю своего бюджета, предназначенную для приобретения инструментов? Как вы решаете, стоит ли инструмент своих денег?**

Все зависит от того, в какой организации вы работаете и какого рода деятельностью занимаетесь. Если бюджет вашего проекта сравним с ВВП небольшого государства, тогда стоимость инструментов не имеет значения – купите себе лучшие инструменты (они могут оказаться не самыми дорогими) и радуйтесь жизни. А вот одинокий хакер, работающий дома, не всегда может позволить себе самые высококлассные инструменты. Часто для домашнего употребления оказывается достаточно свободно распространяемых инструментов.

На самом деле, бесплатно распространяемые инструменты часто отличаются очень высоким качеством, из-за чего трудно определить, в каких случаях стоит раскошелиться на инструменты. Если вы платите за инструменты, то обычно можете рассчитывать на хорошую поддержку, а также исправление обнаруженных ошибок и доработку в будущем. Однако это не всегда удается – компании закрываются, а выпуск продуктов прекращается. Это *может быть* аргументом в пользу выбора наиболее популярных и широко используемых инструментов. Численность спасает.

Если нет других разумных критериев, то нужно исходить из того, что чем дороже инструмент, тем больше должна быть коробка, в которой он продается. Если он стоит кучу денег, а коробочка маленькая, не покупайте его!

Глава 8. Время испытаний

Вопросы для размышления

1. **Напишите комплект тестовых примеров для кода определения наибольшего общего делителя, приведенного в начале главы. Постарайтесь проверить как можно больше случаев. Сколько отдельных контрольных примеров у вас получилось?**

a. **Сколько успешно прошло?**
b. **Сколько не прошло?**
c. **С помощью своих тестов найдите ошибки и исправьте код.**

Есть много тестов, которые нужно выполнить, хотя комбинаций, ввод которых недопустим, очень мало. Сначала о недопустимых данных: проверка на ноль. Это значение может быть как допустимым, так и недопустимым (не видев спецификации, нельзя сказать с уверенностью), но код должен поступить с ним разумным образом.

Затем напишите тесты для комбинаций обычных входных данных (например, 1, 10 и 100 в любом порядке). После этого проверьте числа, у которых нет общих множителей, такие как 733 и 449. Проверьте работу для каких-нибудь очень больших значений и отрицательных чисел.

Как писать такие варианты тестирования? Напишите простую функцию блочного тестирования и поместите ее в автоматизированную тестовую структуру. Не нужно в каждом тесте программно вычислять, каким должен быть правильный результат[1]; просто сравните его с известной числовой константой. Старайтесь, чтобы код тестирования был как можно проще.

```
assert(greatest_common_divisor(10, 100) == 10);
assert(greatest_common_divisor(100, 10) == 10);
assert(greatest_common_divisor(733, 449) == 0);
... еще тесты ...
```

Для такой простой функции оказывается удивительно много тестов. Можно, конечно, доказывать, что для столь небольшого кода проще провести изучение, рецензирование и убедиться в корректности, чем дотошно писать комплект тестов. Этот аргумент может показаться справедливым. А что если впоследствии кто-нибудь модифицирует код? При отсутствии тестов придется тщательно перепроверять код, о чем можно легко позабыть.

Нашли ли вы ошибку в `greatest_common_divisor`? Сейчас будет подсказка. Если не хотите испортить загадку, отвернитесь... *Попробуйте подать на вход отрицательный аргумент*. Вот более надежная (и эффективная) версия, написанная на C++:

```
int greatest_common_divisor(int a, int b)
{
    a = std::abs(a);
    b = std::abs(b);
    for (int div = std::min(a,b); div > 0; --div)
    {
        if ((a % div == 0) && (b % div == 0))
           return div;
```

[1] Это открыло бы дорогу появлению новых ошибок кодирования – представьте себе мучения из-за ошибок в тестовом коде!

```
        }
        return 0;
    }
```

2. Какие различия между тестированием приложения электронной таблицы и системы автоматического пилотирования самолета?

В идеале ни там, ни тут не должно быть ошибок. В такой утопической ситуации та и другая будут протестированы исчерпывающим образом и только после этого сданы. На практике происходит немного иначе. Если можно смириться с тем, что электронные таблицы будут время от времени давать сбой,[1] то автопилот не должен содержать никаких ошибок. Когда на карту ставятся человеческие жизни, программы разрабатываются совсем иным образом – гораздо более формально и гораздо тщательнее. Они строго тестируются. Действуют стандарты безопасности.

3. Нужно ли тестировать тестовый код, который вы пишете?

Если долго думать на эту тему, то голова заболит. Если начать тестировать тестовый код, то как можно быть уверенным, что тестовый код для тестового кода вашего тестового кода корректен? Фокус в том, чтобы писать *как можно более простые* тесты. Тогда самой вероятной ошибкой тестирования станет отсутствие каких-то важных проверок, а не проблемы фактических строк тестового кода.

Пишите код тестов как можно проще, чтобы избежать в нем ошибок.

4. В чем разница между тестированием, проводимым программистом, и тестированием в отделе QA?

Тестеры чаще проводят тестирование в стиле черного ящика и обычно работают только с продуктом в целом. Они редко работают на уровне кода, потому что продукт обычно является исполняемой программой; библиотеки кода поставляются относительно редко.

Программисты чаще проводят тестирование в стиле «белого ящика», проверяя, что их творения работают в соответствии с замыслом.

Тайная задача любого программиста, пишущего тесты, – доказать, что его код работает, а не найти случаи, когда он не действует! Легко написать кучу тестов, которые покажут, какой замечательный у меня код, сознательно избегая тех мест, которые кажутся мне подозрительными. Это хороший аргумент в пользу того, что тестовые структуры должен создавать не автор кода, а кто-то другой.

5. Нужно ли писать набор контрольных примеров для каждой отдельной функции?

---

[1] Жаль, но мы вынуждены с этим смириться.

Не нужно уходить в крайности. Некоторые функции настолько просты, что достаточно проверить их визуально. Однако не будьте небрежны и помните, что читать код нужно *цинично*. Для простых функций получения и установки свойств не нужно уймы отдельных тестов.

Каков должен быть размер кода, чтобы заслужить создание тестовой структуры? Обычно, если он становится таким сложным, что это необходимо. Если с первого взгляда нельзя сказать, что код корректен, тогда напишите какие-то тесты.

6. Существует методика, требующая сначала писать тесты, а потом код. Какого рода тесты вы стали бы писать?

Пока не написан никакой код, это могут быть только тесты черного ящика. Либо они, либо у поборников этой методологии должен быть дар предвидения.

7. Следует ли писать тесты для C/C++, чтобы проверить правильность обработки параметров, имеющих значение NULL (нулевых указателей)? В чем ценность такого теста?

Если предполагается, что на вход может поступить ноль, то, конечно, это нужно проверять.

Но проверка нулевых указателей требуется не всегда. Если вы не задаете особого поведения для нулевого указателя, то ваша функция вправе завершиться, когда вы передадите ей недопустимый указатель. В таком случае нулевой указатель не лучше, чем указатель на освобожденный или недопустимый адрес памяти. Едва ли можно проверить, что код выдержит все недопустимые указатели.

Однако *может* оказаться полезным написать такой код, который устойчив к нулевым указателям, поскольку они возникают во многих случаях. Многие процедуры выделения памяти возвращают нулевые указатели в случае неудачи, и неопределенные указатели часто тоже устанавливаются в ноль. Если собака кусается, на нее следует надеть намордник.

8. Первоначально тестирование может происходить не на той платформе, для которой пишется код, – она может быть недоступна для вас. Что лучше – отложить тестирование, пока вы не получите в свое распоряжение целевую платформу, или начать его сразу?

Если код предназначен для работы в другой среде (например, на мощном сервере или во встроенном устройстве), как обеспечить адекватность тестирования?

Все зависит от характера тестируемого вами кода. Что это – простая функция, выполняющая вспомогательную работу, или логика доступа к аппаратным средствам? Нужно представлять себе, чем отличаются платформа разработки и целевая среда. Ограничения на объем памяти или скорость процессора могут повлиять на выполнение кода. Возможно, для большей части вашего кода это несущественно, и локального комплекта тестов будет вполне достаточно.

Если ваш код использует особые возможности целевой платформы (многопроцессорность или специфическую аппаратуру), то полное тестирование в их отсутствие не провести. Можно воспользоваться эмуляторами, чтобы убедиться в работоспособности кода, но окончательного ответа они не дадут.

Опасно откладывать тестирование до того момента, когда станет доступна целевая платформа. К тому времени накопится большой объем кода, полностью проверять который у вас не будет ни времени, ни желания. Наибольшая надежность достигается, если тестирование проводится как можно раньше.

9. Как определить, что проведено достаточно тестов? Когда можно *остановиться*?

Тестирование не может доказать отсутствие дефектов, поэтому нельзя с уверенностью сказать, что оно проведено в достаточном объеме. Задача может оказаться бесконечной, и хотелось бы предложить такой план тестирования, который оказался бы реалистичным.

Для простых блоков кода при тестировании методом черного ящика достаточно успешно выполнить все тесты из раздела «Выбор контрольных примеров для блочного тестирования» на стр. 200.

Степень адекватности и полноты тестирования можно измерить, взглянув на процесс под определенным углом. Есть несколько основных стратегий.

Тестирование по охвату

План тестирования определяется как мера *охвата* программы. Например, можно запланировать выполнить каждую строку кода хотя бы один раз, пройти каждую ветвь условных операторов или продемонстрировать хотя бы раз выполнение всех системных требований.

Тестирование по количеству ошибок

Основано на фильтрации определенного процента программных ошибок. Задается гипотетическое количество ошибок – обычно исходя из прежнего опыта. Затем вы решаете найти и обезвредить, скажем, 95% от этого числа.

Тестирование по типам ошибок

При этом подходе внимание обращается на те места, где чаще всего возникают ошибки. Например, тестируются все граничные значения и выявляются все ошибки смещения на единицу.

Исходя из этого можно установить критерии прекращения тестирования:

- Успешно проходит определенный процент регрессивных тестов (при этом *неудачные* тесты не являются определяющими).
- Охват кода, функциональности или требований достигает заданного уровня.

- Процент ошибок падает ниже заданного уровня.

Помимо этого существуют некоторые физические пределы, обычно фиксированные, которыми, в конечном счете, определяется завершение тестирования:

- Достижение предельных сроков (тестирования или сдачи готового продукта). Разработка обладает неприятным свойством затягиваться и съедать время, предназначавшееся для тестирования, что требует тщательного административного контроля.
- Израсходование средств, выделенных на тестирование (достойный сожаления критерий остановки).
- Окончание срока альфа- или бета-тестирования.

В большинстве организаций решение о прекращении тестирования и поставке продукта принимается в связи с достижением предельных сроков. Это компромисс между известными остающимися ошибками, степенью их тяжести и частотой проявления с одной стороны и необходимостью выхода на рынок – с другой. Тесты позволяют принять обоснованное решение относительно степени готовности продукта.

Вопросы личного характера

1. **Для какой части своего кода вы пишете тесты? Вас это удовлетворяет? Входят ли ваши тесты в процедуру автоматизированной сборки? Как вы проводите тестирование остального кода? Адекватно ли оно? Собираетесь ли вы изменить свою процедуру?**

Не считайте себя обязанным писать тесты для каждого кусочка кода. Но не забывайте и головой думать. Чтобы реализовать простую функцию, обычно не требуется особых размышлений – потому и пишут ее не задумываясь, откуда и результат: глупые ошибки. Поскольку для простой функции и тест нужен простой, может быть, стоит его написать. В моей фирме действует простое правило: для *каждого* фрагмента кода есть свой тест, иначе он не включается в базовый код.

Важно осуществлять адекватное и достаточное тестирование, чувствуя за него ответственность, а не уклоняться от неприятной задачи. Спросите себя, от скольких ошибок, с которыми вы столкнулись в последнее время, можно было бы уберечься при наличии хорошего набора тестов? Так обеспечьте себе его!

Если ваши тесты не входят в систему сборки, как вы можете гарантировать их выполнение и прохождение через них всего кода?

2. **Какие у вас отношения с сотрудниками отдела QA? Какое мнение у них сложилось о вас?**

Очень важно установить хорошие рабочие отношения между отделом QA и разработчиками программного обеспечения. Между ними часто возникает соперничество; тестирующее подразделение рассматривается как кучка людей, стремящихся помешать разработке и затруднить

выпуск окончательной версии, а не тех, кто помогает создать стабильный продукт. Обычно тестеры и разработчики сидят в разных концах и не слушают никого, кроме своих предводителей.

Покончите с этим.

Угостите их кофе. Пригласите на ленч. Отправьтесь вместе в бар. Примите все меры для прекращения разделения на *своих* и *чужих*.

Наладьте профессиональные рабочие отношения. Старайтесь предоставлять им добротный, протестированный код, а не все, что вы впопыхах написали. Если вы будете кидать им мусор, с которым они вынуждены будут разбираться, у них сложится впечатление, что вы относитесь к ним как к слугам, а не как к коллегам по работе.

3. Как вы обычно поступаете, обнаружив ошибку в своем коде?

Реакция бывает разной:

- Отвращение и разочарование
- Желание обвинить кого-то другого
- Счастье или просто *восторг*
- Пытаетесь делать вид, что этого не было, не обращаете внимания, надеетесь, что все уладится (как будто *такое* возможно).

Кое-что из перечисленного настолько ошибочно, что, я надеюсь, вы до этого не опускаетесь. Вам кажется, что только ненормальный может предположить, что можно *испытывать счастье* при обнаружении ошибки? Но это вполне рациональная реакция для программиста, озабоченного проблемами качества; гораздо лучше, если ошибки обнаружатся во время разработки, чем пользователи найдут их во время работы.

Степень вашего восторга будет зависеть от того, на какой стадии разработки будет найдена ошибка. Обнаружение критической ошибки за день до выпуска продукта никого не сделает счастливым.

4. Пишете ли вы отчет по каждой обнаруженной в коде проблеме?

На самом деле нет необходимости делать это для каждой отдельной ошибки. Если вы не записываете отчет об ошибке в базу данных, то нужно методически делать заметки, чтобы не забыть о ней. По этой причине может оказаться удобнее использовать с самого начала систему отслеживания ошибок. Необходимость подачи сообщений об ошибках может возникнуть в том случае, если продукт отгружается с таким запозданием, что *нужно* известить людей об остающихся проблемах.

Как только вы делаете доступной очередную версию кода, *следует* публиковать данные обо всех имеющихся ошибках. Это должно демонстрировать, что вы знаете обо всех проблемах и работаете над их решением.

Обнаружив ошибку в коде, нужно написать тест, который ее демонстрирует, и включить его в свой комплект автоматических проверок,

выполняемых при регрессивном тестировании. Это своего рода документация ошибки и гарантия того, что она не встретится вновь.

5. Какой объем тестирования обычно осуществляют инженеры проекта?

Важно знать, чего от вас ждут, и обеспечить соответствующий уровень тестирования. Но *не ограничивайтесь* этим – делайте то, что *необходимо*.

Напишите модульные тесты для *всех* созданных вами фрагментов кода. Если нужно модифицировать чужой код, напишите сначала для него тест, если его еще нет. Благодаря этому вы узнаете, как он работает в настоящее время, что нужно исправить и как проверить, что ваши модификации ничего не испортили.

Глава 9. Поиск ошибок

Вопросы для размышления

1. Следует ли требовать, чтобы ошибки исправлял тот программист, который написал код? Или программист, обнаруживший ошибку, лучше сможет ее исправить?

Свежий взгляд на проблему всегда полезен. Во время отладки этот тезис позволяет решить известную проблему, связанную со склонностью программиста видеть то, что он *хотел воплотить* в своем коде, а не то, что реально делает этот код, из-за чего многие ошибки остаются незамеченными.

С другой стороны, автору кода, скорее всего, проще внести исправление. Он знает свой код вдоль и поперек (в идеале), знает, какие последствия может иметь конкретное изменение. Он раньше других может указать, где находится источник ошибки.

На практике выбор того, кто исправит ошибку, может определяться наличием свободного времени у программистов, а также наличием других обязанностей у членов команды. Если ошибка в программе известна с незапамятных времен, автор кода может быть уже недоступен. Он может покинуть компанию, перейти в другой проект или (что самое ужасное) перейти на руководящую работу.

2. Как определить, что лучше – воспользоваться отладчиком или подумать?

Даже отладчик нужно применять разумным образом. (Помните золотое правило отладки?)

Я пользуюсь следующим практическим правилом: не запускай отладчик, не определив, какую информацию хочешь от него получить. Опасность заключается в том, что можно бродить с отладчиком по коду, не зная толком, что ты ищешь. Так можно потратить уйму времени и ничего не получить в результате.

3. Чтобы искать и исправлять ошибки в незнакомом коде, следует сначала изучить его. Однако темпы работы организаций, производящих программное обеспечение, часто не позволяют уделить достаточно времени изучению и освоению ремонтируемой программы. Какая стратегия будет оптимальной в таких условиях?

В своих мечтах вы можете поколотить тех, кто составил такой график, и выделить столько времени, сколько надо, чтобы исправить ошибку, как она того требует. Однако проснемся...

Лучше всего постараться постепенно изучить код. Двигайтесь по нему с предельной осторожностью и не доверяйте своим впечатлениям: всегда проверяйте, действительно ли код делает то, что должен по вашим представлениям. Если вам покажется, что вы нашли источник ошибки, поинтересуйтесь, не знаком ли кто-то из вашей команды с подозрительным кодом. Обсудите с ними свои планы. Часто во время объяснения ситуации *вам* становится ясной очевидная вещь, которую вы пропустили.

4. Предложите технологию, препятствующую возникновению ошибок утечки памяти.

Есть несколько хороших методов:

a. Воспользуйтесь языками, в которых их возникновение наименее вероятно, например Java или C#. (И в этих языках могут возникнуть утечки. Знаете ли вы, в каких ситуациях?)

b. Пользуйтесь «безопасными» структурами данных, которые сами управляют выделением памяти, освобождая вас от этих хлопот.

c. Во избежание проблем применяйте полезные идиомы языка, такие как `auto_ptr` в C++.

d. Будьте последовательны и методичны при выделении памяти. Убедитесь, что на каждое выделение есть освобождение и оно всегда происходит.

e. Пропустите свой код через утилиты проверки работы с памятью и убедитесь в отсутствии ошибок.

5. В каких случаях оправдана кавалерийская атака на поиск и устранение ошибки в противовес более методичному подходу?

Всегда нужно обдумывать свои действия. Даже скороспелые исправления должны быть хорошо обдуманы. Не пытайтесь густо расставить в своем коде контрольные точки и начать копаться во внутренностях; думайте о том, как устроен код и какие действия от него предполагались.

Внутренний голос и непосредственные реакции могут помочь быстро найти ошибку в *очень маленькой программе* (скажем, из нескольких десятков строк). Но если в программе тысячи строк, необходимо понять, как она работает. Ничто не может заменить понимание сути происходящего. Нет ничего плохого в том, чтобы исследовать программу в отладчике, но контрольные точки нужно выбирать с умом.

Вопросы личного характера

1. **Каким количеством отладочных технологий/инструментов вы обычно пользуетесь?**

Ответ очевиден: *никаким*. Вы всегда с первого же раза пишете идеальный код!

2. **Какие стандартные проблемы и ловушки существуют в языках, которыми вы пользуетесь? Какие меры вы предпринимаете против появления соответствующих ошибок в вашем коде?**

О таких проблемах необходимо знать. Этим эксперты и отличаются от рядовых программистов. Если ты не знаешь, где живут драконы, то не сможешь уберечься от них.

3. **Какие ошибки чаще встречаются в вашем коде – сделанные по невнимательности или связанные с более тонкими проблемами?**

Если вы постоянно натыкаетесь на одни и те же мелкие промахи с языком, значит, нужно писать код более внимательно. Не пишите код в спешке. Проверьте написанное и еще раз прочтите – в конечном счете вы сбережете время. Классическая ошибка – исправить ошибку, не протестировав исправление, а потом столкнуться с нежелательными побочными эффектами своего «исправления».

Нет ничего позорного в том, что в коде встретились ошибки. С кем не бывает. Нужно только следить за тем, чтобы не допускать глупых ошибок, которые можно предотвратить.

4. **Умеете ли вы пользоваться отладчиком на своей рабочей платформе? Служит ли он вашим повседневным инструментом? Опишите свои действия в случае:**
   a. **Обратной трассировки**
   b. **Изучения значений переменных**
   c. **Изучения значений полей в структуре**
   d. **Выполнения произвольной функции**
   e. **Переключения контекста потока**

Постоянно пользоваться отладчиком – *излишество*. Никогда не применять его – тоже *неправильно*. Не бойтесь воспользоваться отладчиком, но и не опирайтесь на него постоянно. Разумно пользуясь отладчиком, вы очень быстро отыщете местонахождение ошибки.

Глава 10. Код, который построил Джек

Вопросы для размышления

1. **Зачем при наличии удобной интегрированной среды обращаться к утилите командной строки make, если среда позволяет собрать проект, нажав одну кнопку?**

Помимо понимания того, что в действительности происходит при нажатии кнопки сборки, умение пользоваться *make* открывает дорогу к более мощному и гибкому построению программ. Графические инструменты сборки редко приближаются по своим возможностям и гибкости к make-файлам. Упрощение часто оказывается полезным, и инструменты с GUI помогают разработчикам быстро создавать программы, но за все приходится платить.

Графические инструменты сборки плохо масштабируются и малопригодны для достаточно крупных проектов. Синтаксис make не прост, но эта утилита предоставляет значительно более широкие возможности. Например, make-файлы допускают вложенные каталоги и иерархическую сборку. Примитивные инструменты GUI предоставляют только один уровень глубины – вложенность проектов внутри рабочего пространства.

Можно слышать жалобы на сложность make и трудности ее применения. Это действительно так, но то же справедливо в отношении любых мощных инструментов – можно получить увечья при неправильном их применении.

Это не значит, что вы должны выкинуть все графические инструменты сборки и начать писать кучу make-файлов, которые их заменят. Совсем наоборот: для каждой работы хорош свой инструмент. Сравните простоту и интеграцию с мощью и расширяемостью, и в каждом случае выберите подходящий инструмент.

2. Почему необходимо рассматривать извлечение исходного кода как отдельный этап перед сборкой?

Это логически разные этапы. В правильно устроенной системе сборки у вас должна быть возможность получить любую версию программного продукта вне зависимости от ее давности, а затем выполнить ту команду, которая ее соберет. После этого вы должны иметь возможность очистить дерево и собрать его заново с помощью той же команды, ничего не записывая в систему.

Вы ничего не теряете, разбив это на два этапа. Легко написать скрипт, который объединит оба этапа в одну процедуру извлечения/сборки – он будет полезен для ночной сборки. В этих скриптах ночной сборки важно каждый раз начинать со свежего исходного дерева (чтобы избежать проблем, имевшихся в предыдущем дереве). Это хорошая проверка вашего исходного дерева: удалив его и выполнив полную сборку заново, вы удостоверитесь в наличии всех файлов и их свежести (можно забыть что-то сохранить в системе).

При включении в этап сборки извлечения исходного кода могут также возникнуть следующие проблемы:

- Вы не хотите, чтобы система сборки автоматически получала файлы из хранилища при выполнении сборки. Обычно не требуется, чтобы при каждой новой сборке все исходные данные менялись.

Важно управлять кодом, с которым вы работаете, а не подчиняться прихотям системы сборки.

- Существует проблема начальной загрузки: если извлечение кода входит в процедуру сборки, то откуда взять исходное дерево, чтобы начать сборку? Придется получать его вручную! Либо потребуются дополнительные заклинания, чтобы частично загрузить части дерева, управляющие сборкой, а потом уже выполнить настоящую загрузку и сборку. Этим путем идти не нужно.

3. Куда следует записывать промежуточные (например, объектные) файлы, возникающие при сборке?

В некоторых системах сборки объектные файлы размещаются рядом с исходными, которые их породили. Более развитые системы могут создавать параллельное дерево каталогов и собирать объекты в *нем*, оставляя исходные каталоги в неприкосновенности. Когда исходные и сгенерированные файлы разделены, наблюдается бóльший порядок. Есть в этом и отрицательная сторона: сложнее осуществлять поиск в иерархии. Повторную компиляцию можно инициировать, удалив объектный файл `.o`, но при раздельных деревьях для этого придется дальше удалиться от источника.

Другой хороший подход – размещение промежуточных файлов внутри исходного дерева, но в собственном подкаталоге; они не будут путаться с исходными файлами, но все же будут под рукой. В итоге получится иерархия каталогов, примерно как на рис. 1.

Это удобный способ сборки *нескольких целей* из одного исходного дерева – у каждой цели оказывается собственный подкаталог сборки.

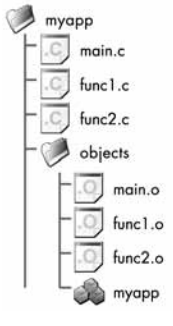


Рис. 1. *Помещение генерируемых объектных файлов в подкаталог*

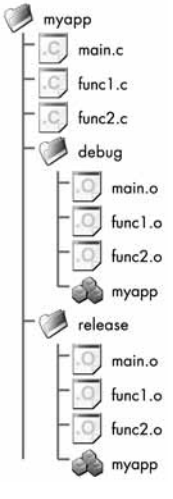


Рис. 2. *Еще лучше: помещение генерируемых объектных файлов в именованный подкаталог*

В ином случае сборка могла бы начаться как отладочная, а завершиться как окончательная, и этап сборки стал бы катастрофой. Такой подход приводит к дереву, показанному на рис. 2.

4. Следует ли при добавлении в систему сборки автоматизированного набора тестов запускать его автоматически после сборки или лучше вводить для запуска тестов отдельную команду?

Ввести отдельную команду нетрудно (например, сделать в make-файле цель *tests* и после `make all` ввести `make tests`). Однако есть шанс, что этот дополнительный шаг не будет выполнен – он не обязателен. Таким образом, тестирование будет пропущено. С учетом человеческих особенностей это вполне вероятно. Непротестированный код может вызвать самые разнообразные проблемы, и весь труд по написанию тестов окажется напрасным. Лучше включить тесты в основную процедуру сборки.

Автоматически запускать тестирование под нагрузкой на этом этапе не стоит. Оно может отнять слишком много времени, и потому его лучше оставить для ночной сборки. Постройте структуру для автоматического выполнения этих тестов, но в обычной сборке не запускайте ее.

5. Какую сборку выполнять ночью – *отладочную* или *финальную*?

Обе. Очень важно как можно раньше проверить конфигурацию окончательной сборки. Отладочные сборки не должны попадать в отдел QA и тем более выходить за пределы компании.

Важно проверять работоспособность процедур окончательной и отладочной сборок не только при организации системы сборки, но на постоянной основе. Крайне легко сорвать ту или другую процедуру в результате малозначительной модификации. Если откладывать проверку сборки до того момента, когда наступит окончательный срок, и окажется, что она не работает, это будет крайне неприятным сюрпризом.

Выполняемые модули, генерируемые при отладочной и окончательной сборках, могут сильно различаться. Поведение некоторых компиляторов в отладочном и окончательном режимах разительно расходится. Существует один популярный компилятор, который в отладочных сборках расширяет буферы данных, так что нехватка памяти не вызывает проблем и проходит незамеченной – едва ли это можно считать хорошей помощью в отладке. Если вы проверяли только отладочную сборку, то при переходе в окончательный режим перед самой поставкой продукта неизбежно столкнетесь с проблемами.

6. Напишите для make правило, которое с помощью компиляторабудет автоматически генерировать информацию о зависимостях. Покажите, как пользоваться этой информацией в make-файле.

Это можно сделать несколькими способами, что частично зависит от того, как вы получаете данные о зависимостях от своего компилятора. Допустим, что некий гипотетический компилятор принимает параметр `-dep`, выражающий просьбу создать наряду с объектным файлом файл

зависимостей. Допустим, что формат этого файла уже соответствует формату зависимостей make.[1] GNU Make позволяет задать правило компиляции, побочным эффектом которого будет генерация зависимостей:

```
%.o: %.c
    compiler -object %.o -dep %.d %.c
```

Вы можете включить все сгенерированные файлы зависимостей прямо в make-файл с помощью следующей строки в его конце:

```
include *.d
```

Вот и все! Разумеется, это лишь простейший способ. Есть много путей развить его дальше. Например:

- Отправить файлы зависимостей в отдельный каталог. Тогда они не будут загромождать рабочий каталог и скрывать важные файлы.
- Написать правило `include` так, чтобы оно включало только нужные файлы `.d`. Кругом могут находиться другие файлы `.d`, которые не должны включаться, и применение маски `*` может нарушить работу make. Такая проблема может легко возникнуть: если вы уберете из make-файла какой-то исходный файл и не почистите перед тем дерево, то старые файлы `.o` и `.d` будут болтаться в рабочих каталогах, пока вы не удалите их вручную.
- Если позволит компилятор, можно написать отдельное правило для создания `.d`-файлов, сделав их равноправными участниками сборки. Недостаток этого в замедлении процесса сборки: для каждого исходного файла компилятор будет вызываться *дважды*.

7. Рекурсивная сборка — популярный метод создания модульной системы сборки, охватывающей несколько каталогов. Однако ей свойственен ряд недостатков. Опишите их и предложите альтернативные методы.

Здравый смысл подсказывает, что все большие проекты, собираемые с помощью make-файлов, должны пользоваться техникой рекурсии. Однако, несмотря на мощь рекурсивной make, ей присущи фундаментальные недостатки, которые не следует игнорировать. Необходимо понимать, как работает (или не работает) рекурсивная make, потому что она очень часто встречается; необходимо знать связанные с ней проблемы, чтобы выбрать лучшее решение.

Почему рекурсивная make может стать обузой? Она таит ряд ловушек:

Скорость

Ну, *о-о-очень медленно* работает. Если вы запустите повторную сборку дерева, которое уже готово, рекурсивная сборка все равно честно пройдет по всем каталогам. Для проекта хорошего размера

[1] Это вполне оправданное допущение: есть много систем, действующих таким образом.

это займет массу времени, что неоправданно, когда никаких действий не требуется.

Каждый каталог собирается посредством *отдельного* вызова.[1] Это исключает ряд возможностей оптимизации: общие включаемые файлы будут обрабатываться раз за разом снова. Хотя файловые системы умеют кэшировать данные, все равно здесь возникают лишние потери. В разумной системе сборки каждый файл нужно прочесть только один раз.

Зависимости

Рекурсивная make не может корректно отслеживать зависимости; make-файлы в подкаталогах не имеют возможности получить все сведения о зависимостях. make-файл вашего модуля может обнаружить, что его локальный исходный файл `func1.c` зависит от заголовочного файла `shared.h` в другом каталоге. Он благополучно станет компилировать `func1.c` после каждой модификации `shared.h`. Но `shared.h` может автоматически генерироваться другим модулем по некоторому шаблону, скажем `shared.tmpl`. Ваш модуль не будет знать об этой дополнительной зависимости. Даже если бы он знал о ней, он не смог бы собрать `shared.h` – это просто не его работа. Таким образом, после модификации `shared.tmpl` исходный файл `func1.c` не будет собран заново, как следовало бы.

Эту прореху можно залатать, только если собирать `shared.h` *раньше*, чем модуль `func1.c`. Чтобы обеспечить правильность сборки, программисту необходимо тщательно выбирать *порядок* рекурсии.[2] Чем более опосредованными оказываются зависимости, тем больше беспорядка возникает в итоге.

Сталкиваясь с такими проблемами, программисты придумывают всякие некрасивые способы их обойти, например выполнение нескольких проходов сборки дерева или ручное удаление некоторых файлов, которое инициирует повторную сборку. Такие приемы приводят к лишнему удлинению сборки и неоправданному усложнению процедуры.

Перенесение вины на разработчика

Make была придумана, чтобы справиться со сложностью повторной сборки кода. Рекурсивная make ставит все с ног на голову и снова заставляет вас заниматься процедурой сборки. Как мы видели, программисту приходится управлять порядком рекурсии и заставлять каждый make-файл обходить ограничения.

---

[1] Представьте себе только накладные расходы, связанные с запуском всех этих дочерних процессов!

[2] Это очко в пользу графических средств – в них нет рекурсии, и обычно они правильно справляются с зависимостями.

Скрытость

Проблемы рекурсивной make не бросаются в глаза. По этой причине многие благосклонно относятся к рекурсии. Когда же что-то перестает ладиться, это проявляется странным образом. Причину проблемы редко удается установить, и ее списывают на влияние «нечистой силы».

Тем самым создаются дополнительные условия для появления опасно неустойчивых систем.

Эти проблемы несправедливо приписывают самой make, утверждая, что в ней есть дефекты. Но в данном случае make оказывается лишь невинным свидетелем. Порок не в make, а в том, *как* мы ею *пользуемся*. Все проблемы создаются рекурсией, которая мешает make правильно делать свое дело.

Как же справиться с этими неприятностями? Очевидно, не хотелось бы отказываться от вложенности в деревьях исходного кода. Нужен такой процесс сборки, который поддерживает вложенность, но не ведет сборку рекурсивно. Это не слишком сложно; такой прием называется *вложенной make*. Он просто предполагает помещение всех данных о сборке в один главный make-файл. При этом нет необходимости в отдельных make-файлах для каждого подкаталога. Главный make-файл управляет всеми подкаталогами исходных файлов внутри себя.

Вопреки распространенному мнению, рекурсивное применение make – плохой прием. Вместо него следует пользоваться более надежной процедурой вложенной make.

Может показаться, что такой подход сложнее и менее гибок. Как управиться с большим деревом сборки с помощью лишь одного make-файла?

Эта задача облегчается с помощью ряда практических методов реализации:

- Воспользуйтесь механизмом make для включения файлов. Поместите в каждом каталоге список содержащихся в *нем* исходных файлов – так будет гораздо проще и понятнее. Сохраните список в файле, дав ему имя типа `files.mk`, и включайте этот файл из главного `Makefile`.
- Можно сохранить модульность рекурсивной make – вход в *любой* каталог и выполнение там команды `make` – путем определения промежуточных целей. Эти цели образуют специфические части проекта. Организация модульной сборки таким способом может оказаться более осмысленной, чем подход рекурсивной make на основе случайного обхода каталогов, и гарантирует правильную сборку каждой промежуточной цели.

Вложенная make нисколько не сложнее рекурсивной; на практике она даже может быть *проще*. С ее помощью сборка становится надежнее, точнее и быстрее.

Вопросы личного характера

1. **Умеете ли вы выполнять различные виды компиляции с помощью своей системы сборки? Каким образом можно собрать из одного и того же источника и с помощью одних и тех же make-файлов и отладочную, и финальную версии продукта?**

В одном из предыдущих ответов мы видели хорошее решение этой проблемы: собирать объекты в разных подкаталогах с помощью сценариев сборки исходя из типа сборки (один каталог для отладочных файлов, другой для окончательных).

В GNU Make этого можно достичь с помощью имен файлов. Например:

```
# Определим исходные файлы
SRC_FILES = main.c func1.c func2.c

# Тип сборки по умолчанию (если не указан)
BUILD_TYPE ?= release

# Создаем имена объектных файлов
# (Это формула GNU Make, которая меняет
# суффикс .c на суффикс .o)
OBJ_FILES = $(SRC_FILES:.c=.o)

# А теперь хитрость: добавим к именам объектных
# файлов префикс (чудеса GNU Make)
OBJ_FILES = $(addprefix $(BUILD_TYPE)/, $(OBJ_FILES))
```

Очевидно, что для выбранного типа сборки (`BUILD_TYPE`) можно поменять и другие параметры, например флаги компилятора. Не забудьте, что понадобится правило, которое создаст подкаталоги, иначе компилятору некуда будет писать результат. Вот как это делается в UNIX:

```
$(BUILD_TYPE):
    mkdir -p $(BUILD_TYPE)
```

Теперь можно ввести следующие две команды – сначала одну, потом другую – и быть уверенным, что система сборки отлично справится с ними:

```
BUILD_TYPE=release make all
BUILD_TYPE=debug make all
```

Можно сделать более простую систему без этой техники подкаталогов, но в ней придется делать чистку при каждом изменении `BUILD_TYPE`.

2. **Хороша ли система сборки в вашем текущем проекте? Обладает ли она качествами, о которых говорилось в этой главе? Можно ли ее усовершенствовать? Легко ли будет:**
   a. **Добавить новый файл в библиотеку?**
   b. **Добавить новый каталог в код?**
   c. **Удалить или переименовать файл с кодом?**

d. Добавить новую конфигурацию сборки (например, для демонстрационной версии)?

e. Собрать две конфигурации по одному экземпляру исходного кода, не зачищая дерево в промежутке между сборками?

Это говорит о том, насколько хорошо вы знаете процесс сборки и насколько легко его сопровождать. Полезно сравнить свой механизм сборки с другими проектами – тогда вы увидите, где ваши процессы неадекватны и требуют усовершенствования.

Возьмем, к примеру, перемещение и переименование исходных файлов. То и другое часто незаметно происходит при рефакторинге. Эти простые действия могут привести к некорректному определению системой зависимостей и к сборке кода с ошибками. Я сталкивался с такими проблемами неоднократно; не сразу замечаешь возникшие проблемы.

Часто у программистов «нет времени», чтобы поправить систему сборки; они слишком спешат выдать готовый продукт. Это опасное заблуждение. Сценарии сборки составляют часть кода и так же требуют сопровождения и развития, как любые другие исходные файлы. Надежность и безопасность системы сборки настолько важны, что нельзя считать потерянным время, потраченное на их доработку. Это время – ваши инвестиции в базовый код.

3. Приходилось ли вам когда-либо создавать систему сборки с самого начала? Чем вы руководствовались при ее конструировании?

Как и в любой задаче программирования, на ваше решение оказывают влияние многие факторы:

- Ваш прежний опыт
- Ваши знания
- Ваше *текущее* представление о задаче
- Ограничения существующей технологии
- Имеющееся время

Обычно вскоре после начала эксплуатации выясняется, насколько удачными были ваши решения. Никогда с самого начала не бывают ясны все требования, и возникают проблемы, которых никто не мог предвидеть:

- Меняются требования – если проект действительно оказывается успешным, могут потребоваться версии на разных языках или для других архитектур процессора. Система сборки должна допускать расширение.
- Может потребоваться перенесение кода на новую инструментальную цепочку сборки, хотя никто и никогда не предполагал, что такой вариант возникнет.

Легкость, с которой могут быть сделаны эти модификации, служит свидетельством качества вашего проекта. Каждое изменение способст-

вует накоплению ценного опыта, который пригодится вам при создании новой системы.

4. **Всем приходится иногда сталкиваться с недостатками системы сборки. При написании сценария сборки точно так же можно допустить ошибки, как при программировании обычного кода.**

   С какими сбоями системы сборки вы сталкивались и как вы смогли исправить или даже предотвратить их?

В число обычных ошибок сборки входят:

- Неправильный сбор информации о зависимостях.
- Некорректная обработка системных сбоев, таких как отсутствие места на диске или некорректные права доступа к файлам; бывает, что сборка продолжается, и ничто не указывает на сбой одного из этапов.
- Проблемы с контролем версий: некорректное слияние или выбор неправильных версий кода.
- Ошибки конфигурирования библиотек, часто с использованием несовместимых или устаревших версий.
- Неумение программистов пользоваться системой сборки и совершение ими глупых ошибок.

Если что-то пошло неожиданным путем, отчасти проблема вызвана системой сборки.

Глава 11. Жажда скорости

Вопросы для размышления

1. **Оптимизация — это поиск компромисса, т. е. отказ от одного положительного свойства кода в пользу другого, более желательного. Опишите, какого рода компромиссы возникают в стремлении увеличить производительность программы.**

Вот решения, которые оказывают значительное влияние на производительность программы:

- Количество функций в противовес размеру кода
- Скорость программы в противовес расходу памяти
- Хранение и кэширование в противовес вычислению по требованию
- Защищенный подход в противовес незащищенному; оптимистичный в противовес пессимистичному
- Приближенные вычисления в противовес точным вычислениям
- Встроенные функции в противовес вызову функций; монолитность в противовес модульности
- Индексирование массива в противовес поиску в списке

- Передача параметра по ссылке или адресу в противовес передаче копии
- Аппаратная реализация в противовес программной
- Жесткий, прямой доступ в противовес косвенному доступу
- Предопределенное, фиксированное значение в противовес переменному и настраиваемому
- Выполнение во время компиляции в противовес выполнению на этапе исполнения
- Локальная функция в противовес удаленному вызову
- Отложенные вычисления в противовес немедленным вычислениям
- «Умный» алгоритм в противовес понятному коду

2. Посмотрите на альтернативы оптимизации, перечисленные в разделе «Доводы против оптимизации» на стр. 271. Допускаются ли при этом компромиссы, и если да, то какие?

Некоторые из этих альтернатив можно считать оптимизациями – в зависимости от того, в какой мере система находится под вашим контролем. Если вы указываете, на какой аппаратной платформе будет выполняться ваша программа, то использование более быстрой машины будет оптимизацией. Если нет, то это, скорее, обходной путь.

Многие из альтернатив влекут скрытый рост сложности. Например, полагаясь на определенную конфигурацию платформы (т. е. выполнение некоторых сервисов или фоновых программ), вы создаете особые зависимости от среды, которые могут оказаться незамеченными и пропущенными во время установки или последующего сопровождения.

3. Объясните значение следующих терминов и связь между ними:

- **Производительность (Performance)**
- **Эффективность (Efficiency)**
- **Оптимизированность (Optimized)**

Эффективность кода определяет его *производительность*. Оптимизация означает повышение эффективности кода с целью увеличения его производительности. Обратите внимание, что ни один из этих терминов не описывает непосредственно *скорость выполнения*; нужным качеством может быть не скорость, а объем памяти или пропускная способность.

4. Какие наиболее вероятные узкие места могут быть в медленно работающей программе?

Распространено заблуждение, что разные программы конкурируют за ЦП и что плохой код захватывает все процессорное время. Иногда бывает, что процессор работает почти вхолостую, а производительность при этом ужасная. Программа может застрять по ряду причин:

- Интенсивный обмен между оперативной памятью и файлом подкачки на жестком диске.

- Ожидание доступа к диску
- Ожидание конца медленной транзакции базы данных
- Плохая организация блокировок

5. Как устранить необходимость в оптимизации? Какие методы позволяют избежать появления неэффективного кода?

Мы видели, насколько важно с самого начала *проектировать* систему с учетом ее производительности. Это можно делать только тогда, когда есть четкое представление о требованиях к производительности.

Хорошо продумав архитектуру, пишите разумный код. Нужно знать, какие конструкции в выбранном вами языке наиболее эффективны, и избегать использования неэффективных. Например, в C++ лучше передавать ссылки `const`, а не создавать дорогостоящие временные копии.[1]

Полезно иметь примерное представление об относительной стоимости разных операций. Если положить, что процессор выполняет одну операцию в секунду, то функция обычно выполняется за считанные секунды, виртуальная функция – за 10–30 секунд, поиск на диске занимает несколько месяцев, а интервал между нажатиями клавиш составляет несколько лет. Попробуйте сделать подобные оценки для таких операций, как выделение памяти, получение блокировки, создание нового потока и поиск в простой структуре данных.

6. Как влияет на оптимизацию наличие нескольких потоков?

Многопоточность может вызвать не меньше проблем, чем пытаются решить с ее помощью. Необдуманная архитектура потоков может привести к появлению новых узких мест, особенно если возникают долгие взаимные блокировки.

Многопоточные программы труднее профилировать, если только профайлер не обеспечивает хорошую поддержку потоков. Если потоки должны взаимодействовать, нужно подумать, как будет происходить выполнение в целом, когда несколько потоков управления переплетаются между собой.

7. Почему мы пишем *неэффективный* код? Главное, что мешает нам сразу воспользоваться высокопроизводительными алгоритмами?

Есть много совершенно законных причин, по которым не удается написать оптимизированный код с первой попытки:

- Неизвестно, в каких условиях будет в конечном счете работать программа. Если нет реальных данных для проведения тестирования, трудно выбрать конструкцию, которая будет работать лучше всего.

---

[1] С другой стороны, такие ссылки могут отрицательно сказаться на других аспектах производительности. Копии гарантируют отсутствие проблем с алиасами; некоторые оптимизации компилятора не выполняются, если могут возникнуть переменные алиасы. Как всегда, нужно прикинуть, что для вас лучше.

- Достаточно трудно вообще заставить программу *работать*, не говоря уже о том, чтобы работать *быстро*. Чтобы проверить техническую осуществимость, мы выбираем конструкции, которые легко реализовать, быстро создав прототип.
- «Высокоэффективные» алгоритмы могут быть более сложными и трудными для реализации. Естественно, что программисты избегают их, поскольку там легко допустить ошибку.

Программистам часто кажется, что время выполнения кода пропорционально труду, потраченному на его написание.[1] Можно написать какой-нибудь код синтаксического анализа файлов за несколько часов, но выполняться он будет невыносимо долго, потому что диски работают медленно. Сложный код, который вы писали несколько дней, может выполниться всего за несколько сотен циклов процессора. В действительности, ни эффективность фрагмента кода, ни время, которое нужно потратить на его оптимизацию, никак не связаны с тем временем, в течение которого вы его написали.

8. **Тип данных `List` реализован с помощью массива. Какова будет алгоритмическая сложность следующих методов `List` в худшем случае?**
   a. **Конструктор**
   b. `append` **— добавление нового элемента в конец списка**
   c. `insert` **— вставка нового элемента в заданном месте между двумя существующими элементами**
   d. `isEmpty` **— возвращает true, если в списке нет элементов**
   e. `contains` **— возвращает true, если в списке есть указанный элемент**
   f. `get` **— возвращает элемент с указанным индексом**

Оценки для худшего случая будут такими:

a. Сложность конструктора будет `O(1)`, поскольку ему нужно лишь создать массив; вначале список пуст. Однако следует учесть, что размер этого массива влияет на сложность конструктора – в большинстве языков массивы при создании полностью заполняются объектами, даже если вы не сразу собираетесь ими пользоваться. Если конструкторы этих объектов нетривиальны, выполнение конструктора `List` займет некоторое время.

   Размер массива может не фиксироваться – конструктор примет еще один параметр, который определит этот размер (фактически задав максимальный размер списка). Тогда сложность метода становится `O(`*n*`)`.

b. В среднем операция `append` имеет сложность `O(1)`: она должна просто записать элемент массива и обновить размер списка. *Но* если массив полон, придется выделять новую память, копировать и освобождать память – в худшем случае сложность не меньше `O(`*n*`)` (зависит от производительности менеджера памяти).

[1] Выглядит глупо, но в эту ловушку легко попадают.

c. `insert` имеет сложность в среднем `O(n)`. Может потребоваться вставить элемент в начало списка. Для этого придется сместить все элементы на одну позицию, чтобы освободить место для первого элемента. Чем больше в `List` элементов, тем больше времени займет эта операция. И здесь в худшем случае потребуется заново выделять память, что может оказаться гораздо сложнее, чем `O(n)`.

d. Если только ваша реализация не окажется совсем уж дикой, сложность `isEmpty` составит `O(1)`. Размер списка известен, так что возвращаемое значение – простая операция над этим числом.

e. `contains` имеет сложность `O(n)` в предположении, что содержимое списка не упорядочено. В худшем случае требуется найти несуществующий элемент, и тогда придется обойти все элементы списка.

f. `get` имеет сложность `O(1)` благодаря реализации в виде массива. Выбор элемента массива – операция с постоянным временем. Если реализовать `List` в виде *связанного списка*, то эта операция будет иметь сложность `O(n)`.

Вопросы личного характера

1. Насколько важна производительность кода в проекте, над которым вы работаете в данное время (только честно)? Чем обусловлено данное требование к производительности?

Требования к производительности не должны выбираться произвольно. Они должны быть обоснованы, а не взяты с потолка. Все требования к производительности важны; нет спецификаций, которые не существенны. Сколько внимания потребует некоторое конкретное требование, зависит от того, насколько трудно его удовлетворить. Независимо от трудности необходимо предложить решение, при котором оно может быть удовлетворено.

2. Когда прошлый раз вы проводили оптимизацию:

a. Применяли ли профайлер?

b. Если да, какой прирост эффективности он показал?

c. Если нет, как вы выяснили, было ли достигнуто улучшение?

d. Проверили ли вы работоспособность кода после проведенной оптимизации?

e. Если да, насколько тщательно вы провели тестирование?

f. Если нет, то почему? Можете ли вы быть уверены, что код по-прежнему правильно работает во *всех* случаях?

Только очень существенное увеличение производительности можно обнаружить без профайлера или других хороших средств хронометража. Человек может обманываться в своих ощущениях – после того как вы изрядно попотели над тем, чтобы ускорить программу, она всегда *покажется* вам более быстрой.

Тщательно измеряйте рост производительности, и если он несущественен, отмените изменения в коде. Лучше сохранить ясность кода, чем достичь незначительного прироста скорости и получить логику, которую невозможно сопровождать.

3. **Если вы еще не занимались оптимизацией кода, над которым работаете в данное время, попробуйте угадать, какие его части самые медленные и какие потребляют больше всего памяти. Теперь пропустите свой код через профайлер и проверьте, насколько верны оказались ваши предположения.**

Возможно, вас удивят результаты. Чем больше профилируемая программа, тем менее вероятно, что вы правильно оцените эти узкие места.

4. **В какой степени заданы требования к производительности вашей программы? Есть ли у вас конкретный план проверки того, что ваш код удовлетворяет этим критериям?**

Если нет четкой спецификации, никто не может предъявить вам претензии по поводу скорости работы программы.

Глава 12. Комплекс незащищенности

Вопросы для размышления

1. **Что такое «безопасная» программа?**

Безопасная программа должна противостоять попыткам неправильного употребления, взлома или применения для решения задач, которые не имелись в виду изготовителем. Это не просто надежная программа; надежный код отвечает спецификациям и не дает сбой при несколько повышенных требованиях. Однако надежная программа может разрабатываться без учета требований безопасности и допускать утечку важной информации при определенных исключительных условиях. Иногда лучше, чтобы при неправильном употреблении программа аварийно завершилась, чем давала нежелательную выдачу. Таким образом, безопасная программа может аварийно завершаться!

Определение безопасной программы зависит от требований безопасности, предъявляемых к приложению. Отчасти они зависят от того, что могут предоставить вспомогательные службы (ОС и другие приложения). Исходя из этого для приложения могут быть определены следующие задачи:

Секретность

Система не предоставляет информацию тем, кто к ней не допущен. Они получат сообщение «доступ запрещен» или вообще не узнают, что такая информация существует.

Целостность

Система не позволяет неавторизованным лицам изменять данные.

Готовность к работе

Система работает непрерывно – даже в случае атаки. Трудно защититься от *всех* мыслимых угроз (что если кто-то отключит питание?), но можно бороться с многими угрозами путем повышения степени избыточности в проекте или путем обеспечения быстрой перезагрузки после атаки.

Аутентификация

Система проверяет, что пользователи являются теми, за кого себя выдают, обычно с помощью механизма имя пользователя/пароль.

Аудит

Система записывает сведения обо всех важных операциях, чтобы перехватить или отследить действия злоумышленников.

2. Какие входные данные нужно проверять в безопасной программе? Какого типа проверка необходима?

Все входные данные нужно проверять. В их число входят параметры командной строки, переменные окружения, вводимые через графический интерфейс данные, данные веб-форм (даже если JavaScript проверяет их на стороне клиента), URL в кодировке CGI, содержимое cookies, содержимое файлов и имена файлов.

Нужно проверять размер вводимых данных (если это не просто числовые переменные), допустимость их формата и фактическое содержимое данных (вхождение чисел в допустимый диапазон, отсутствие встроенных строк запросов).

3. Как можно защищаться против атак со стороны группы доверенных пользователей?

Это не так просто. Они получили определенный уровень прав, потому что считается, что они не злоупотребят им. Большинство пользователей не станет умышленно неправильно эксплуатировать вашу программу, но найдется небольшое число таких, которые попытаются нарушить правила в своих интересах.

Есть несколько способов борьбы с этим:

- Регистрируйте все операции в журнале, чтобы знать, кто, когда и какие изменения осуществил.
- Потребуйте, чтобы все действительно важные операции подтверждались двумя пользователями.
- Заключите каждую операцию в транзакцию, которую нельзя откатить.
- Периодически делайте резервное копирование данных, чтобы их можно было восстановить в случае утраты.

4. Где может произойти переполнение буфера, допускающее создание эксплойта? Какие функции особенно подвержены переполнению буфера?

Переполнение буфера является, пожалуй, самой большой угрозой безопасности, и злоумышленнику бывает легко воспользоваться этой уязвимостью. Оно может всегда возникнуть при обращении к структуре, размещенной по нескольким адресам – либо при копировании данных в/из нее, либо при обращении к ней с целью получить конкретный элемент. Чаще всего источником опасности оказываются массивы и строки.

Обычно эта ситуация возникает в процедурах ввода пользовательских данных, хотя это не единственное место – переполнение буфера может возникнуть в любом коде, обрабатывающем данные. Буфер может находиться как на стеке (где размещаются локальные переменные функции), так и в куче (пуле динамически распределяемой памяти).

5. Можно ли полностью исключить возможность переполнения буфера?

Да – если старательно проверять входные данные каждой функции и обеспечить, чтобы стек программы, ведущий к каждой точке ввода (возможность, предоставляемая функциями ввода ОС или библиотекой этапа исполнения вашего языка), был безопасен.

Вот некоторые основные приемы защиты вашего кода:

- Применяйте язык, в котором нет буферов фиксированного размера, например язык, поддерживающий автоматическое расширение строк. Но опасность представляют не только строки: обратите внимание на массивы с проверкой границ и безопасные карты хеш-таблиц.
- Если нет поддержки со стороны языка, *нужно* проверять на соответствие диапазону все вводимые данные.
- В C всегда применяйте более безопасные функции стандартной библиотеки `strncpy`, `strncat`, `snprintf`, `fgets` и т. д. Не применяйте подпрограммы из `stdio`, такие как `printf` и `scanf`, – нельзя гарантировать их безопасность.
- Не применяйте библиотеки сторонних разработчиков, безопасность которых может вызвать сомнения.
- Пишите свой код в управляемой среде выполнения (типа Java или C#). В этом случае атаки с использованием переполнения буфера становятся практически неосуществимыми – среда исполнения автоматически перехватывает большинство переполнений.

6. Как защитить память, используемую приложением?

Три момента, когда надо задуматься о безопасности памяти:

а. *Перед использованием*. Когда вы получаете какой-то объем памяти, в ней содержатся произвольные значения. Не пишите код, который полагается на значения, содержащиеся в неинициализированной памяти. Взломщик может воспользоваться этим для атаки на ваш код. Для пущей надежности обнулите всю выделенную память, перед тем как ее использовать.

b. *Во время использования.* Заблокируйте память с секретными данными, чтобы ее нельзя было выгрузить на диск. Очевидно, что ОС должна быть защищенной; если одно приложение может прочесть память другого – пиши пропало.

c. *После использования.* Прикладные программисты часто забывают, что при освобождении памяти ее нужно очистить и только потом возвращать в ОС. Если не делать этого, зловредный процесс может найти в памяти ваши секретные данные.

7. Характерна ли для C и C++ принципиально более низкая защищенность, чем для других языков?

C и C++ ответственны за большую часть незащищенных приложений и *позволяют* писать код, содержащий классические уязвимости. Вне всякого сомнения, с ними нужно быть настороже; даже опытные разработчики должны быть внимательны при написании кода на C/C++, чтобы не допускать переполнения буфера. Эти языки не очень способствуют написанию защищенных программ.

Однако другие языки не избавляют вас от всех проблем защищенности – лишь от тех, что стали знаменитыми благодаря C и C++. В другом языке вы, скорее всего, избежите проблем с переполнением буфера, но при этом не должно возникать ложного чувства безопасности; остается множество проблем, которых нельзя избежать в самом языке. *Каким бы* языком вы ни пользовались, нужно помнить о проблемах безопасности – нельзя выбрать «безопасный» язык и забыть об их существовании.

На самом деле, переполнение буфера – это такая проблема, которую очень легко проверить и решить. Когда требуется написать защищенное приложение, выбор языка не существенен в сравнении с другими проблемами.

8. Учтен ли опыт C для проектирования C++ как более защищенного языка?

В C++ появился абстрактный тип `string`, который сам управляет своей памятью. Это существенно помогает избежать переполнения буфера, хотя для тех, кто ищет неприятностей, сохранились обычные массивы `char` в стиле C. Другое полезное средство – `vector`, массив с управлением памятью. Однако обе эти структуры можно при желании обмануть – рассказать вам как?

C++ можно было бы считать более опасным, чем C, из-за того, что он часто держит указатели функций в куче (там хранятся таблицы виртуальных функций). Если атакующему удастся изменить какой-нибудь из этих указателей, то он сможет передать управление своему зловредному коду.

Во многих отношениях C++ более защищен – или, скорее, его проще использовать защищенным образом. Однако при его разработке защи-

щенность не была основной целью, и у него есть собственные проблемы безопасности, о которых должен быть осведомлен разработчик.

9. Как можно узнать, что ваша программа скомпрометирована?

В отсутствие средств обнаружения вам остается лишь следить, не стала ли система вести себя необычным образом. Трудно назвать это научным подходом. То, что система взломана, может оставаться секретом неограниченно долго. Даже если жертва (или изготовитель программного продукта) *обнаружит* атаку, он может предпочесть не распространяться об этом, чтобы не привлечь новых нарушителей. Какая компания публично объявит, что ее продукт содержит бреши в системе защиты? Даже если она выпустит заплатку с исправлением, не все ею воспользуются и во многих действующих системах останется широко известный дефект защиты.

Вопросы личного характера

1. Каковы требования к безопасности, предъявляемые в вашем текущем проекте? Каким образом были они установлены? Кто осведомлен о них? В каких документах они отражены?

Дайте честный ответ. Нетрудно сочинить что-нибудь благовидное. Но если требования к защите не зарегистрированы формально, значит, ваш проект несерьезно относится к требованиям безопасности. Каждый разработчик должен знать о них и уметь осуществить.

2. Какая была самая страшная ошибка защиты в выпущенных вами приложениях?

Важно знать о ней, даже если это уже стало древней историей. Нужно помнить, какую ошибку вы совершили, чтобы не оставалось никаких шансов повторить ее в будущем. Если вам неизвестно ни о каких прежних уязвимостях, возможно, вы недостаточно тщательно проверяли свой продукт.

3. Сколько бюллетеней по вопросам безопасности было выпущено по поводу вашего приложения?

Были ли они связаны с глупыми ошибками и небрежностью разработчиков или происходят от более глубоких проблем архитектуры? Большинство проблем, попадающих в бюллетени, относятся к первому виду.

4. Проводили ли вы *аудит безопасности* программы? Какие проблемы он выявил?

Если только проверкой не занимался специалист по безопасности, какие-то виды уязвимостей наверняка окажутся пропущенными. Однако и тогда аудит вскроет вопиющие проблемы и окажется полезен.

5. Кто, как вам кажется, вероятнее всего может попытаться атаковать вашу систему? В какой мере на это влияют:

- **Ваша компания**

- **Тип пользователя**
- **Тип продукта**
- **Популярность продукта**
- **Конкуренция**
- **Платформа, на которой он работает**
- **Нахождение в сети и видимость системы широкой публике**

На каждого может кто-нибудь напасть: злонамеренный пользователь, беспринципные конкуренты и даже террористы. Разве можно кому-то верить?

Глава 13. Важность проектирования

Вопросы для размышления

1. **Как масштаб всей задачи влияет на проект программного обеспечения и работу по его созданию?**

Чем больше проект, тем сильнее он нуждается в проектировании архитектуры в сравнении с проектированием кода на низком уровне. Больше времени требуется потратить на проектирование на предварительном этапе, поскольку неправильный выбор будет иметь более тяжелые последствия.

2. **Что лучше – хорошо документированный плохой проект или плохо документированный хороший проект?**

Хорошая документация – неотъемлемая часть хорошего проекта. В плохом проекте хорошая документация – это путь к коду, даже если это ярко освещенная аллея, ведущая к помойной яме. Она хотя бы покажет вам, что этого кода лучше не касаться.

Если код достаточно простой, вам не потребуется объемистая документация, но если программа сложная, работать с ней в отсутствие приличного описания становится трудно.

Что лучше? Лучше хороший проект без документации: если проект действительно высокого качества, он *должен* быть понятен и самодокументирован.

3. **Как оценить качество проекта по фрагменту кода? Можно ли количественно отразить его простоту, элегантность, модульность и т. п.?**

Качество трудно измерить; в значительной мере это эстетическая оценка проекта. Почему мы считаем картину прекрасной? Это из разряда тех вещей, которые нельзя взять в руки и пересчитать. Задним числом вы можете оценить, насколько просто оказалось разобраться в коде или модифицировать его. Но впервые столкнувшись с каким-то кодом, трудно его оценить. Если есть две архитектуры, A и B, и мне кажется, что A красивее, но на практике B оказывается удобнее и легче допускает модификацию, то трудно утверждать, что архитектура A лучше.

Качество можно оценить, только *посмотрев* на код. Обычно достаточно небольшого фрагмента; если малая часть хороша, то, скорее всего, и остальной код будет достаточно высокого качества. Такая оценка оправдывается не всегда, но все же удобна. Реалистичен такой подход: если маленький фрагмент плох, готовьтесь к тому, что базовый код в целом ужасен. Если маленький фрагмент выглядит прилично, не исключено, что базовый код в целом таит в себе более тонкие проблемы.

Оценить качество проекта помогут инструменты, которые исследуют код и генерируют графики и документацию.

4. Может ли проектирование быть коллективной деятельностью? Насколько важны навыки групповой работы при разработке хорошего проекта?

Весьма важны. Задачи программирования редко решаются в одиночку. В промышленном производстве программного обеспечения в большинстве крупных проектов участвует несколько проектировщиков. Даже если работа поделена на части, в какой-то момент эти части нужно стыковать между собой – как и проектировщиков. Даже если проектировщик один, он должен успешно задокументировать и передать свой проект.

5. Есть ли зависимость между проектом и наиболее предпочтительной для него методологией?

Да, в некоторых проектах охватываемая ими область делает излишними определенные методы проектирования. Например, если вы пишете драйверы устройств, полномасштабный процесс ОО-проектирования едва ли окажется полезен.

Если вы работаете над очень формализованным проектом, например для правительственного учреждения, вы будете вынуждены применять весьма формализованный процесс, в котором документируется каждая стадия и обосновывается каждое принятое проектное решение. Разница с исследовательским проектом в научной лаборатории может быть весьма существенной.

6. Каким образом можно определить, является ли данный проект сильно связным или слабо сцепленным?

В конечном счете нужно смотреть на код и выяснять, как он связан внутри себя, но это так скучно! Для проекта на C или C++ можно оценить степень его связанности, посмотрев на директивы `#include` в начале файла. Если они многочисленны, то связанность может быть умопомрачительной. С другой стороны, можно воспользоваться программами анализа, которые наглядно отобразят свойства вашего кода.

7. Если вам приходилось решать аналогичную задачу проектирования в прошлом, поможет ли это определить, насколько сложна проблема в *данном* случае?

Опыт полезен в проектировании, поэтому учитесь и применяйте знания на практике. Но проявляйте здравый смысл в отношении своего опыта; не следует действовать автоматически. Разные ситуации выдвигают разные требования – не следует чрезмерно полагаться на внешнее сходство разных задач.

Если вы умеете пользоваться молотком, не нужно думать, что не существует никаких других задач, кроме забивания гвоздей.

8. Допустимы ли эксперименты в проектировании?

Да, все проекты экспериментальные, пока они не реализованы и не признаны успешными. Возьмите принцип «первый раз делай на выброс», описанный Фредериком Бруксом (Brooks 95). В пользу экспериментирования можно сказать многое.

Проектирование – итеративный процесс; на каждой итерации можно опробовать альтернативы и выбрать ту, которая наиболее разумна. Чем больше итераций и чем меньше область, охватываемая каждой из них, тем менее болезненными окажутся любые ошибочные проектные решения.

Вопросы личного характера

1. Обратитесь к прошлому и вспомните, как вы учились проектировать код. Смогли бы вы передать полученные вами знания абсолютному новичку в этом деле?

Сколь многому вы по правде *могли бы* научить и что должно зависеть от способностей и опыта ученика? Можете ли вы, основываясь на своем опыте, разработать ряд упражнений, которые помогут другим?

Не следует с самого начала предлагать новичку разработку крупной системы. Первым заданием должен быть небольшой законченный проект, а затем можно предложить ученику развить имеющуюся программу, зорко наблюдая за его действиями.

Большинство программистов, обучаясь проектированию, не получают такого рода поддержки. Они учатся путем проб и ошибок. Попробуйте заняться обучением и наставничеством – это поможет вам развить собственные способности.

2. Какой опыт у вас есть в использовании конкретных методологий проектирования? Удачный или неудачный? Каким в результате получился код? Какой выбор мог оказаться более удачным?

Остался ли у вас неприятный осадок от прежнего опыта и предпочтений? Если вы не умеете применять какую-то методологию, вам будет тяжело и неуютно работать с ней. Матерый С-программист вполне может невзлюбить любые объектно-ориентированные проекты, а его собственные ОО-конструкции окажутся ужасными. Но это не значит, что ОО-программирование – неправильный подход.

3. **Считаете ли вы, что нужно строго придерживаться выбранной методологии?**

Метод проектирования – это инструмент, приспособление. Как и языком проектирования, им следует пользоваться до той поры, пока он сохраняет *полезность*. Если он перестает быть полезным, это уже не инструмент! Методология окажется бездейственной, если никто в вашей команде не будет знать, как ее применять; пользуйтесь тем, что всем известно, или сначала научите их новому.

4. **Какой код из тех, что вам встречались, был спроектирован лучше всего? Хуже всего?**

Я думаю, что вам нетрудно будет вспомнить, какой код был хуже всего спроектирован. Плохой код не дает забыть о себе, как больное место. Хорошо спроектированный код выглядит просто и очевидно, не вызывая желания воскликнуть: «Какой замечательный проект!». Возможно, вы даже не обратите внимания на то, сколько сил потребовало его проектирование.

5. **Язык программирования является, по существу, инструментом для реализации вашего проекта, а не святыней, о которой можно спорить. Насколько важным является знание идиом языка?**

Очень важным, иначе получится никому не понятный код.

Некоторые архитектурные решения могут определяться языком, но на код низкого уровня язык реализации оказывает *огромное* влияние. Очевидный пример: при кодировании на Java не пользуйтесь плоской процедурной архитектурой – это абсолютно ошибочно.

6. **Считаете ли вы программирование *технической дисциплиной*, *ремеслом* или *искусством*?**

Если говорить просто, все зависит от того, как этим заниматься. В нем есть элементы одного, другого и третьего.

Мне больше нравится представлять программирование как ремесло – оно требует навыков, мастерства, дисциплины и опыта. Его продукты могут быть одновременно практичны и красивы. В программировании есть элемент искусства; это творческий процесс. С этим артистизмом связано владение инструментами и технологиями. Все это – признаки мастерства.

Глава 14. Программная архитектура

Вопросы для размышления

1. **Объясните, где заканчивается *архитектура* и начинается *проектирование программного продукта*.**

По правде говоря, то и другое можно определить так, как вам больше нравится. В стандартном употреблении есть следующие различия:

- **Архитектура** – проект структуры верхнего уровня. Она учитывает далеко идущие последствия выбранных решений, влияющие на стоимость создания и сопровождения, общую сложность системы, возможности расширения в будущем и задачи маркетинга. Архитектура разрабатывается в начале проекта. Она оказывает значительное влияние, по крайней мере, на последующее проектирование программного продукта.
- **Проектирование программного продукта** – это работа на следующем уровне, более детальная и целенаправленная. Она касается деталей кода, таких как структуры данных, сигнатуры функций и конкретная передача управления от одного модуля к другому. Проектирование программного продукта осуществляется помодульно. Его результаты имеют гораздо меньшее значение для системы в целом.

Точная граница между тем и другим отчасти определяется масштабом проекта. Создание программного продукта является итеративным и инкрементным процессом – хотя вначале разрабатывается архитектура, результаты проектирования продукта могут оказать обратное влияние на архитектуру.

2. Как плохая архитектура может сказаться на качестве системы? Есть ли такие части, на которые недостатки архитектуры не оказывают влияния?

Плохая архитектура обесценивает все попытки написать хороший программный продукт. Архитектура оказывает фундаментальное влияние на качество кода. Если есть какой-то код, на который не повлияла плохая архитектура, то это либо отдельная библиотека, либо он по-настоящему не связан с системой.

3. Трудно ли исправить выявившиеся недостатки архитектуры?

На ранних этапах формирования продукта относительно легко воздействовать на архитектуру. Но когда в соответствии с этой архитектурой началась разработка и в жертву ей принесен определенный объем капиталовложений (в смысле проектирования и кодирования), весьма и весьма трудно что-либо изменить. С таким же успехом можно попытаться переписать с самого начала весь продукт.

Вот почему так важно с самого начала правильно определить архитектуру. Рефакторингу можно подвергнуть небольшие участки кода, но не все основание структуры.

Конечно, разорвать на части программный продукт и начать все заново проще, чем в реальном физическом строительстве, но этого нельзя делать по экономическим соображениям. Обычно нам дают единственную возможность правильно выбрать архитектуру, и если ее упустить, то расплачиваться придется в течение всего жизненного цикла программной системы.

4. В какой мере архитектура оказывает влияние на:

a. Конфигурацию системы

b. Ведение журналов

c. Обработку ошибок

d. Безопасность

Архитектура оказывает глубокое влияние на все перечисленное, или, правильнее, все перечисленное оказывает глубокое влияние на архитектуру. Прежде чем серьезно браться за разработку архитектуры, необходимо определить, какие требования предъявляются в этих областях. В дальнейшем будет трудно вносить такие возможности в код, не говоря уже об изменении архитектуры.

a. Архитектура определяет, *что* можно конфигурировать (в каком объеме) и *как* это делать. Тип механизма конфигурации определяется рядом факторов: значением общей компоненты «администратора конфигурации», необходимостью в удаленном конфигурировании и определением лиц, которым разрешено конфигурирование (только разработчикам или также тем, кто устанавливает, сопровождает или эксплуатирует систему). Все это фундаментальные проблемы архитектуры.

b. Отдельные компоненты могут заносить информацию в журнал с помощью общего механизма или индивидуальных средств. Архитектура определяет, какой из способов предпочтительнее, как получать доступ к журналам и какая информация нуждается в регистрации. Она должна учитывать требования как разработчиков, так и пользователей. Должны ли окончательные версии вести журнал данных для разработчика?

c. Аспекты архитектуры, относящиеся к обработке ошибок, включают в себя наличие или отсутствие центрального сервиса регистрации ошибок в журнале и системы сообщений об ошибках (каким образом ошибка проникает из жалких серверных компонент в стерильный графический интерфейс пользователя?). В архитектуре также определяется тип механизма обработки ошибок: центральная таблица кодов ошибок, общих для всех компонент, или общая иерархия исключений. Решается и проблема обработки ошибок, генерируемых компонентами сторонних разработчиков.

d. Проблемы безопасности зависят от типа разрабатываемого программного обеспечения. У распределенной системы торговых точек, использующей для связи Интернет, будет иная система требований к защите, чем у маленькой программы, размещаемой на отдельном компьютере. Безопасность – важный вопрос, и ее нельзя присоединить к системе в последний момент; эти проблемы нужно решать на ранних стадиях проектирования архитектуры.

5. Какой опыт или подготовка необходимы для того, чтобы называться *программным архитектором*?

Можно *называть* себя архитектором, но нельзя в одночасье овладеть знаниями и опытом либо чудесным образом приобрести мудрость, необходимую для принятия правильных архитектурных решений.

Хорошие архитектурные разработки требуют наличия богатого опыта – изучения, проектирования и доработки программных систем. Приобрести его можно только в результате самостоятельного труда, а не наблюдения за работой других. Следует с опасением относиться к тем, кто начинает называть себя архитектором, выпустив единственную версию продукта.

Можно заниматься разработкой архитектуры и не называться архитектором; наличие этого звания часто определяется организационной структурой и культурой компании. Для того чтобы претендовать на этот титул, формальной квалификации не требуется, хотя в некоторых странах закон запрещает называться архитектором чего бы то ни было без профессиональной аккредитации.

6. Должна ли стратегия рыночного поведения оказывать влияние на архитектуру? Если да, то каким образом? Если нет, то почему?

Да, коммерческие соображения обязательно влияют на архитектуру. В противном случае система, которую вы построите, окажется нежизнеспособным продуктом; вы быстро станете безработным, а ваша компания – несостоятельным должником.

Мы *обязаны* думать о коммерческих результатах выбора архитектуры, например оценивать последствия сбоев и издержки, связанные с возвратом продукта или его обслуживанием на месте. Архитектура должна минимизировать такого рода проблемы (например, с помощью удаленного доступа и хорошей диагностики можно снизить стоимость обслуживания).

Коммерческие соображения влияют также на такие области архитектуры, как средства поддержки клиентов (в частности, определяя степень сложности администрирования системы), метод установки (с помощью обученного персонала или автоматически с CD), техническое обслуживание и система оплаты.

7. Какими особенностями должна обладать архитектура, рассчитанная на *расширение*? А рассчитанная на максимальную *производительность*? Какое влияние эти проектные цели оказывают на систему и в каких отношениях они находятся между собой?

Есть несколько архитектурных решений, которые вытекают из этих двух требований:

- **Расширяемость** достигается с помощью таких архитектурных средств, как подключаемые модули (плагины), программный доступ к коду (отражение), дополнительные языковые связки, средства написания сценариев и дополнительные уровни косвенности.

- **Производительность** достигается путем совершенствования архитектуры, соблюдения в ней умеренности. Нужно удалить все ненужные компоненты и обеспечить своевременность и адекватность соединений. Для повышения скорости обработки данных можно прибегнуть к буферизации.

Как можно видеть, оба эти требования имеют мало общего; каждая ловушка, устроенная для возможного расширения, съедает какую-то, пусть малую, долю производительности. Дополнительные уровни косвенности имеют свою цену – косвенность. Если ваша задача – обеспечить возможность расширения, это приемлемая цена. Хорошая архитектура предполагает правильный выбор компромиссов на верхнем уровне, обеспечивающий удовлетворение технических требований.

Вопросы личного характера

1. **Насколько широк диапазон архитектурных стилей, которые вы применяете в своей работе? В каком из них у вас больше опыта и как это влияет на программы, которые вы пишете?**

Архитектура оказывает на нас влияние разными путями. Разные архитектурные стили ведут к разным технологиям проектирования и кодирования. Мы – рабы своих привычек, и эти технологии определяют, как мы мыслим и пишем код, даже если в дальнейшем приходится работать с другой архитектурой.

Полезно знать несколько разных архитектур и уметь работать с ними. На практике каждый выбирает себе какой-то конкретный стиль. Вы должны понимать, какое влияние оказывает на код данная архитектура, и следить за тем, чтобы при смене архитектуры код, который вы пишете, ей соответствовал.

2. **Каков ваш личный опыт работы с архитектурами, оказавшимися успешными или неудачными? Какие особенности сделали их правильным выбором или помехой в работе?**

Во-первых, нужно определить, какую архитектуру считать успешной. Значит ли это, что у нее есть технические достоинства? Или система оказалась коммерчески выгодной? Или то и другое вместе? У вас может быть свой собственный ответ.

Если система не выдерживает нагрузки из-за неудачной архитектуры, это обычно связано с тем, что данная архитектура не смогла обеспечить возможности расширения. Она не смогла обеспечить важные дополнительные функции. В таком случае продукт неизбежно теряет свою долю рынка, уступая более находчивым конкурентам. Можно вспомнить массу продуктов, которые сгинули в подобной ситуации.

Другую опасность представляет наследие; большие капиталовложения в архитектуру могут оказаться огромным препятствием. Чтобы расстаться со старой системой или архитектурой и начать все с самого начала, нужны глубокое понимание сути проблем и немалое мужест-

во. При переработке продукта всегда нужно помнить уроки предшествующих версий.

Архитектура, перегруженная функциями, столь же опасна, как бедная ими. Если архитектура поддерживает слишком много возможностей, продукт становится чрезмерно сложным, громоздким и недопустимо медленным. Обычно это приводит к тому, что самые незначительные модификации требуют внесения изменений во многие компоненты.

3. **Предложите каждому разработчику в вашем текущем проекте изобразить архитектуру системы — самостоятельно (без обсуждения с коллегами) и без обращения к системной документации или коду. Сравните их схемы. Что вас больше всего поражает в результатах их труда, исключая чисто художественные достоинства?**

Плохо, если у них получатся непохожие картинки. Нет ничего страшного, если обнаружатся мелкие расхождения; каждый может пропустить какие-то мелкие компоненты, и каждый может проявить особый интерес к какой-либо части системы. Но если окажется, что на схемах присутствуют весьма различающиеся между собой компоненты или разные способы связи между ними, то это будет означать, что у команды нет единой мысленной модели кода. Можно с большой уверенностью ждать катастрофы. Соберите всех разработчиков и выработайте у них общее представление о системе.

Если все схемы *окажутся* похожими, можете вздохнуть с облегчением. Дополнительные очки вы получите, если и расположение компонент на каждом листе бумаги окажется аналогичным. Это свидетельствует о наличии центральной спецификации архитектуры и, что еще важнее, о понимании ее каждым.

4. **Есть ли в вашем текущем проекте общедоступное описание архитектуры? Давно ли оно обновлялось? Какими видами представления вы пользуетесь? Если вам потребуется рассказать о системе новому сотруднику или будущему клиенту, какие данные вам реально потребовались бы в документе?**

Оцените, насколько ваша документация отличается от реальности. Какие возможности у вас есть для исправления ситуации? В напряженной обстановке коммерческой разработки вам редко представится возможность получить в графике время, достаточное для документирования архитектуры в целом, но можно запланировать сделать это по частям во время проектирования и выработки спецификаций новых модулей. Таким образом, можно по частям создать хороший обзор архитектуры.

5. **Сравните архитектуры своей системы и ваших конкурентов. Благодаря каким особенностям своей архитектуры вы рассчитываете добиться успеха своего проекта?**

Необходимо понимать, какие особенности вашей архитектуры должны обеспечить выполнение всех технических требований и гарантировать успех. (Если вы руководствовались при ее разработке какими-то

иными соображениями, это чревато серьезными проблемами.) Мы уже видели, что архитектура оказывает огромное влияние на формирование программной системы и ее качество, а следовательно, в значительной мере определяет успех или провал вашего продукта. Редко можно встретить успешный программный продукт с плохой архитектурой. Если вы и найдете такой, едва ли его успех будет долгим.

Архитектура должна быть в состоянии поддерживать, по крайней мере, те же базовые функции, что и конкурирующие системы, и хорошо поддерживать особые функции, благодаря которым покупатели могут отдать предпочтение вашему продукту, а не какому-то другому. Простые функции, не зависящие от поддержки со стороны архитектуры, редко бывают так же привлекательны, как базовая функциональность, глубоко укорененная в системе.

Глава 15. Программное обеспечение – эволюция или революция?

Вопросы для размышления

1. **Какая метафора лучше всего отражает развитие программного продукта?**

Никакая. Бессмертные слова Фореста Гампа: «Программа – это то, что ведет себя как программа». (Groom 94) Создание кода вызывает много ассоциаций, но ни одна метафора не передает все тонкости, как нельзя передать словами красоту восхода солнца.

Аналогии могут вводить в заблуждение; программы очень сильно отличаются от любых физических объектов, и потому их создание ни на что не похоже. Они имеют меньше физических ограничений и допускают большее воздействие.

Какая-то капля истины есть в каждой метафоре. Пользуйтесь ими в разумной мере и не ограничивайтесь одной точкой зрения.

2. **Если принять для разработки программного продукта метафору человеческой жизни, о чем говорилось во вступлении, каким реальным событиям соответствуют такие этапы, как:**
   - **зачатие**
   - **рождение**
   - **рост**
   - **созревание**
   - **выход в мир**
   - **средний возраст**
   - **усталость**
   - **выход на пенсию**
   - **смерть**

Несмотря на все несовершенство метафор, данная много говорит нам о жизненном цикле программной системы. Явно неразумно перемещать стадии разработки на место предшествующих им – нельзя выпустить продукт, пока он не созрел. Можно, конечно, но последствия будут тяжкими.

Зачатие

Компания видит нишу для нового продукта. Определяются требования рынка к нему. Принимается решение о разработке.

Рождение

Запускается программный проект. Привлекаются проектировщики и программисты. Определяется архитектура. Начинают писать код.

Рост

Код развивается, и программа растет. Она обретает все большую функциональную полноту. Нависают сроки сдачи.

Созревание

Наконец, код готов. Он проходит все тесты к удовлетворению отдела QA. Считается, что задача выполнена и, можно надеяться, с не слишком большим опозданием.

Выход в мир

Выходит версия программы 1.0. Она успешно удовлетворяет потребностям рынка.

Средний возраст

Клиенты активно пользуются программой. После выхода нескольких версий она приобретает дополнительные функции и несколько разбухает.

Усталость

В конечном счете, программа уступает более удачному конкурирующему продукту, у которого больше функций и выше производительность. Новые клиенты не появляются, а старые требуют обновления. Развивать программу становится тяжело и даже невыгодно.

Выход на пенсию

Наконец компания решает отказаться от разработки и прекратить поддержку. Объявляется, что поддержка прекращается через x месяцев – формальное извещение о *кончине*. Разработка прекращается, хотя некоторые действия по сопровождению производятся.

Смерть

Наступает неизбежное: полностью прекращаются разработка и сопровождение. Поддержка больше не осуществляется. Жизнь продолжается, и скоро никто не вспомнит ни как пользоваться программой, ни даже как она называется.

3. Есть ли предел продолжительности жизни программного продукта – как долго можно разрабатывать и совершенствовать программу, прежде чем начать работу заново?

Это больше зависит от рынка, на котором распространяется программа, чем собственно от ее качества. Код может жить вечно, если его хорошо сопровождать и разумно расширять. Однако технологии быстро устаревают, а мода меняется. Операционные системы быстро эволюционируют, аппаратные платформы устаревают, и то, что когда-то было новейшей технологией, лидирующей на рынке, через несколько лет оказывается никому даром не нужным. Поддержание конкурентоспособности программы требует большого труда. Для этого может потребоваться непрерывно добавлять новые функции или переносить программу на новые платформы.

Программное обеспечение с открытым исходным кодом тоже не защищено от подобных проблем конкуренции и рынка, а иногда еще более уязвимо для них. Денежные интересы могут быть незначительными или вообще отсутствовать, но есть реальный рынок с развивающимися технологиями, облегченным доступом и большими возможностями перехода с одного продукта на другой.

4. Соответствует ли объем кода степени зрелости программы?

Нет. У меня было много случаев, когда в результате *удаления* из системы части кода система существенно выигрывала. Дублирование кода может привести к значительному росту его объема без заметного расширения функций. Воспользовавшись внешними библиотеками, вы получите массу новых функций без заметного увеличения объема кода проекта.

Часто мерой прогресса разработки считают количество *строк кода*. Такие метрики бесполезны, если не относиться к ним правильным образом. Они всего лишь характеризуют *объем написанного кода*, а не его качество или изящество архитектуры. Во всяком случае, это не мера функциональности.

5. Важно ли обеспечить *обратную совместимость* при сопровождении кода?

Все зависит от конкретного проекта и способа его развертывания. Как правило, очень важно сохранить обратную совместимость при модификации кода, особенно в отношении форматов файлов, структур данных и протоколов связи. Редкие приложения могут оправданно отойти от этого требования – только системы с немногочисленной базой клиентов и отсутствием необходимости в хранении, извлечении или передаче исторических данных.

Следует также задуматься о *совместимости снизу вверх*. Имеется в виду проектирование кода, допускающего расширение и способного пережить последующие события. Ошибка Y2K хорошо свидетельствует

о том, что бывает при игнорировании этого правила, когда последствия оказываются дорогостоящими и могут быть катастрофичными.

6. **В каком случае деградация кода происходит быстрее – при его модификации или сохранении в неизменном состоянии?**

Быстрее всего код деградирует, когда вы пытаетесь его изменить. Дело в том, что если оставить свой код и дать ему возможность медленно загнивать, то ваши конкуренты не преминут воспользоваться этим, сделав ваш код никому не нужным. По вашему *продукту* будет звонить колокол, но *код* останется таким же прекрасным, каким был в молодости.

Небрежное сопровождение и неряшливое расширение могут действительно искалечить код. Слишком легко при решении одних проблем внести новые ошибки. Требование скорейшего изменения приводит к модификациям, которые ухудшают понятность кода и его структуру. Сопровождение кода часто делает его недоступным для сопровождения.

Чтобы избежать этого, нужны хорошие программисты и грамотное руководство проектом.

Вопросы личного характера

1. **Чем вам больше приходится заниматься – написанием нового кода или модификацией имеющегося?**
   a. **Если это новый код, то пишется ли он для вновь создаваемых систем или расширений существующих?**
   b. **Зависит ли от этого, *как* вы пишете код? Каким образом?**

В этих разных ситуациях вступают в игру разные силы. При расширении существующего кода или приспособлении нового программного обеспечения к старой структуре приходится предварительно проводить *обширное* изучение того, как работает существующая программа. Иначе вы напишете плохой код, который недостаточно хорошо стыкуется с имеющимся, что станет причиной больших проблем в будущем.

Если код пишется заново, нужно учитывать будущие потребности модификации. Код должен быть понятным, расширяемым и податливым, чтобы избежать будущих проблем.

2. **Есть ли у вас опыт работы с уже готовым базовым кодом? Если да:**
   a. **Повлияло ли это на ваш нынешний уровень мастерства? Чему вы в результате научились?**
   b. **Каким был этот код по большей части – плохим или хорошим? Каким образом вы это оценивали?**

Опыт нескольких лет работы позволяет отличить хорошее программное обеспечение от плохого. Становится ясно, какие симптомы должны вызвать тревогу, и можно быстро определить, что код нуждается в осторожном отношении.

Несмотря на некоторый мазохизм, работа с чужим скверным кодом может принести пользу – вы узнаете, чего делать *нельзя* и как недальновидность одного программиста может впоследствии осложнить жизнь другого. Это поможет вам понять важность ответственного отношения к коду, который вы пишете.

3. **Случалось ли вам вносить изменения, ухудшавшие качество кода? Почему?**

Обычные причины (или отговорки) следующие:

- В то время я не мог придумать ничего лучше.
- У меня не было времени, потому что нужно было быстро сдать код.
- Другие способы потребовали бы слишком много работы.
- Я мог модифицировать только доступный мне код – проблема была в коде другой команды или в двоичной библиотеке, исходного кода которой у меня не было.

Ни одно из этих оснований нельзя считать удовлетворительным.

Вы получите лишние очки, если предложите контраргументы для всех этих объяснений и предложите выход из каждой ситуации. Например, если от вас требуют быстрого выпуска кода, можно сделать простую и быструю модификацию, а после выпуска продукта вернуться к работе над ним и осуществить более грамотное решение.

4. **Через сколько ревизий прошел код вашего текущего проекта?**
   a. **В какой мере менялась функциональность от одной версии к другой? Как менялся код?**
   b. **Осуществлялось ли развитие *наудачу*, *по плану* или каким-то промежуточным образом? В чем это проявляется сейчас?**

Вот несколько важных соображений.

a. Одно не обязательно связано с другим. Иногда очень простое изменение функциональности требует глубокой переработки кода. Я видел много проектов, которые сталкивались с такой ситуацией, когда архитектура системы не поддерживала новые требования и должна была быть радикально изменена.

   Я сталкивался и с обратным: версии были функционально идентичны, но за одинаковой внешностью скрывался почти полностью измененный код. Нет смысла переписывать весь проект, если система приближается к концу своей жизни, но если у нее хорошие коммерческие перспективы, а нынешний код не сможет удовлетворить будущим требованиям, такая переделка может быть оправдана.

   Выпуск новой версии без новых функций может оказаться самоубийственным с коммерческой точки зрения – клиенты не станут обновлять имеющуюся версию, если не увидят для этого оснований. Поэтому в качестве приманки нужно предложить несколько

небольших новых функций или немного смошенничать (например, «*Эта версия содержит важные исправления ошибок*»).

b. Необходимо знать историю своего базового кода, чтобы понять, как он приобрел свой нынешний вид, и грамотно осуществлять модификацию и приводить в порядок.

5. Какие меры принимаются в вашей команде, чтобы модификацией кода не занималось одновременно несколько программистов?

Воспользуйтесь для управления модификацией кода *системой контроля версий*. Блокировка файлов, загружаемых из системы, не позволит модифицировать их нескольким программистам одновременно. Но этого недостаточно. Сразу за одной модификацией в системе может быть сохранена другая, которая ей противоречит. Нужно тщательно *управлять* разработкой, чтобы каждый программист знал, чем занимаются его коллеги и кто за какие изменения отвечает. Рецензирование кода помогает обнаружить и исправить такого рода проблемы.

Хороший комплект регрессивных тестов позволит проверить, что модификация не нарушила работу кода.

Глава 16. Кодеры

Вопросы для размышления

1. Сколько нужно программистов, чтобы заменить электрическую лампочку?

Вопрос некорректен. Это аппаратная проблема, а не программная. Предложите решить ее инженерам. Впрочем, если инженеры пожелают решить проблему программным способом...

2. Что лучше — энтузиазм, но слабое мастерство (при некоторой компетентности) или выдающийся талант, но отсутствие мотивации?

a. У кого код окажется лучше?

b. Кто окажется лучшим программистом? (это разные вещи)

Что больше влияет на результирующий код — техническая компетентность или социальная позиция?

Есть разные типы программных систем, и при создании системы каждого типа требуется свой набор умений. Таким образом программисты находят для себя нишу во встроенных системах, веб-службах, финансовых системах и т. п. Задача программирования может также зависеть от уже имеющегося кода. Вы можете писать:

- Простые «учебные» программы
- Новые системы, создаваемые с чистого листа
- Расширения существующих программ
- Сопровождение старой базы кода

В каждом случае нужен особый уровень мастерства и дисциплины и особый подход к разработке. Об этом говорится в следующем вопросе. Не всякий программист, способный написать «игрушку» для себя, сможет создать новую профессиональную систему.

В любом случае качество полученного кода определяется как вашей технической компетентностью, так и вашим *отношением* к задаче – одно должно дополнять другое. Если вам недостает какого-то технического навыка, то ваша позиция должна включать признание этого и желание компенсировать.

Ваша позиция может оказать больше влияния на код, который вы пишете, чем уровень ваших знаний в настоящее время. Если ваши знания недостаточны, но вы стремитесь хорошо справиться с работой, то вы скорее достигнете положительного результата. Также более вероятно, что вы станете учиться и совершенствовать свое мастерство.

3. **Мы пишем разные программы в зависимости от «унаследованного» кода. Чем различается написание следующих типов кода:**
   - **«несерьезная» программа**
   - **абсолютно новая система**
   - **расширение существующей системы**
   - **сопровождение старого кода**

Казалось бы, между этими сценариями нет большой разницы, но в действительности они требуют весьма различных подходов.

«Несерьезная» программа

Это может быть небольшая интересная поделка для личного пользования или маленькая утилита в помощь разработке крупной системы. От такой программы не требуются абсолютная надежность, глубоко продуманная архитектура или исчерпывающий набор функций. Она должна лишь обеспечить решение текущей проблемы, после чего ее можно выкинуть.

Быстрота и легкость разработки, по всей видимости, имеют бóльшее значение, чем красота архитектуры или идеологическая чистота производственного процесса.

Новая система

Создание заново системы профессионального уровня требует серьезной проектной и плановой работы. Необходимо учесть, как система будет применяться и расширяться, и обеспечить надлежащую документацию.

Расширения

Проектов, в которых новая система строится с нуля, немного. Чаще приходится расширять существующий код, добавляя в него новые функции. Новый код должен корректно вписаться в существующую систему. Чтобы достичь этого, нужно досконально разобрать-

ся в исходном коде и уметь проводить модификации, которые хорошо уживаются с имеющимся кодом.

Сопровождение

Чаще всего программисты занимаются сопровождением существующего кода, исправляя дефекты и поддерживая его работоспособность в меняющихся окружающих условиях. Для этого нужен тщательный, методичный подход. Может понадобиться широкое исследование, которое потребует напрячь ваши дедуктивные способности, так как редко можно встретить систему, достаточно хорошо документированную для сопровождения, особенно если она стареет и приближается к концу своего существования.

4. **Если программирование – это искусство, то каким должно быть соотношение обдумывания и планирования, с одной стороны, и интуиции и инстинктивности – с другой? Чем руководствуетесь вы – инстинктом или планом?**

Как мы уже видели, эффективные программисты применяют оба подхода. Интуиция и творческое чутье позволяют создавать элегантный код. Параллельно с ними действует вдумчивое планирование, гарантирующее надежность, практичность и своевременный выпуск кода.

Нельзя вывести оптимальное соотношение или формулу для того и другого. Эффективные программисты владеют обоими подходами и умеют проявлять сдержанность в их применении.

Глава 17. Вместе мы – сила

Вопросы для размышления

1. **Почему программы пишут командами? В чем преимущества относительно самостоятельной разработки?**

Разрабатывать программы в одиночку, видимо, проще; не нужно работать вместе с другими странными личностями, не нужно координировать работу или страдать от неэффективного руководства. Нетрудно, однако, увидеть и многие преимущества групповой работы над программным обеспечением.

Команде доступно решение более крупных задач в результате разделения их на части и передачи отдельным разработчикам. Код можно создавать быстрее. В группе таланты разработчиков объединяются, образуя нечто большее, чем простая сумма частей. Если отсутствует четко определенный проект или предварительный эскиз, то более широкий спектр навыков и знаний в группе имеет явное преимущество; коллективная работа фильтрует идеи и предлагает лучшие решения. Рецензирование со стороны коллег обеспечивает лучшее качество работы.

Существует еще личная мотивация: технари любят работать над крутыми проектами. В составе команды вы можете работать над система-

ми, выходящими за пределы ваших личных возможностей. Это может быть продукт, который требует больше, чем по силам поднять одному человеку, или он требует какого-то специального мастерства, или дает возможность работать рядом с более опытными программистами.

На практике даже отдельно работающий программист входит в некую команду. Если вы не работаете вместе с другими программистами, то все равно входите в команду компании в целом, работающей над созданием законченного продукта. Без других участников ваша программа никогда не могла бы быть выпущена.

2. Опишите симптомы хорошей и плохой групповой работы. Каковы необходимые условия хорошей работы и как проявляется плохая групповая работа?

Для эффективной работы команды должны быть обеспечены следующие условия:

- Правильный подбор людей, имеющих все необходимые технические навыки.
- У членов команды может быть разный уровень опыта; они должны уметь учиться у других. Если набрать команду из одних стажеров, она вряд ли добьется успеха. (Однако их гораздо легче обучить и нацелить на нужное дело, чем шайку псевдогуру, имеющих о себе высокое мнение.)
- Типы личности участников команды должны дополнять друг друга. Чтобы достичь успеха, команда должна иметь в своих рядах вдохновителей и энтузиастов, а не тех, кто будет снижать ее боевой дух.
- Четкие и реальные цели (еще лучше, если это интересный проект, который членам команды действительно хотелось бы довести до успешного завершения).
- Мотивация (финансовая или эмоциональная).
- Нужные спецификации должны быть получены как можно раньше, чтобы все знали, над чем они работают, и отдельные фрагменты стыковались между собой.
- Хорошее администрирование.
- Как можно меньшая численность команды, но не слишком маленькая. Чем больше людей, тем труднее наладить групповую работу: больше каналов передачи данных, больше усилий по координированию, больше точек возможного провала. Следует избегать неоправданной сложности.
- Четкая и всеми понимаемая технологическая процедура разработки, которой следуют все члены команды.
- Поддержка со стороны компании, а не препятствия и бюрократизм.

С другой стороны, есть надежные индикаторы того, что команда не может работать эффективно. В этом списке есть как внутренние, так и внешние факторы:

- Нереалистичный график работы и срок окончания, устанавливаемый до того, как команда может оценить объем работы.
- Неясные задачи проекта и отсутствие технических требований к проекту.
- Нарушение коммуникативных процессов.
- Плохие или неподготовленные руководители команд.
- Плохо определенные роли и ответственность участников – кто и за что отвечает?
- Неправильное личное отношение к работе и личные цели.
- Некомпетентность членов команды.
- Оценка отдельных разработчиков не по заслугам, наличие любимцев.
- Оценка разработчиков по критериям, не соответствующим задачам команды.
- Большая текучесть кадров.
- Отсутствие изменений в процедуре руководства.
- Отсутствие обучения и наставничества.

3. Сравните групповую разработку программ и метафору строительства (см. раздел «Действительно ли мы собираем программы?» на стр. 240). Позволяет ли она лучше понять групповую работу?

Есть несколько метафор для описания нашей работы (например, *спортивная команда,* или *хоровое общество* Демарко, или *фабрика*, о которой мы шутим здесь) (DeMarco 99). Проблема любой метафоры в том, что она справедлива лишь отчасти. У программирования есть свои задачи и проблемы. Химическая технология отличается от гражданского строительства, а то отличается от съемки фильма, а съемка отличается от написания программ.

Метафора стройки полезна, хотя и не идеальна. В конце концов, мы сооружаем программу в соответствии с планом, из отдельных блоков (одни из которых мы изготавливаем сами, а другие покупаем или достаем). Вот полезные параллели:

- Вам нужна бригада: нельзя построить в одиночку ни небоскреб, ни высокотехнологичную программную суперструктуру уровня предприятия.
- У бригады есть задача: она должна завершить строительство в соответствии с назначенными сроками и выделенным бюджетом.
- Кто-то является заказчиком работы, имея цель: это конечная цель работы.
- Все члены бригады заняты разными делами: разделение ролей помогает выполнить работу. Есть архитекторы, строители, плотники, водопроводчики, электрики, прорабы, конторские работники, охрана и пр. Каждый вносит свой важный вклад.

- Есть ответственные члены команды: прораб управляет людьми.

Но, конечно, здания сильно отличаются от программ. Здания нельзя сооружать итеративным или инкрементным образом. Любые изменения в спецификации здания приводят к дорогостоящей разборке и последующей постройке заново. В нашей области, имеющей дело с чистой мыслью, можно уничтожить конструкцию и собрать ее заново с очень небольшими издержками (но с учетом времени и стоимости рабочей силы). В программировании легче строить абстрактные интерфейсы между блоками. Технические области разные, но это не значит, что нельзя найти полезные параллели с другими профессиями.

4. Какие факторы – внешние или внутренние – больше всего угрожают эффективности команды разработчиков?

Те и другие угрожают сорвать вашу работу. Внутренние факторы таковы:

- Неэффективные члены команды
- Конфликты
- Беспорядок
- Критические ошибки, выявленные на поздних стадиях
- Неточные планы

Они соединяются с такими внешними факторами, как:

- Нечеткие или меняющиеся технические требования
- Нереалистичные сроки
- Плохое администрирование
- Корпоративный бюрократизм

Все это крайне осложняет жизнь разработчиков программного обеспечения. Давление внутренних и внешних факторов в равной степени может расстроить работу вашей команды, но широко распространено мнение, что большинство проектов проваливается не по техническим причинам.

С уверенностью можно сказать одно: факторов, разрушающим образом влияющих на эффективность команды, гораздо больше, чем факторов, способствующих успеху. Поэтому нужно старательно оберегать работу своей команды, стремясь изолировать ее как от внутренних, так и от внешних угроз.

5. Как размер команды влияет на ее динамику?

Чем больше в команде людей, тем больше она требует:

- Затрат на координацию
- Затрат на коммуникации (с новыми людьми появляются и новые пути коммуникаций, число которых увеличивается экспоненциально)
- Затрат на кооперацию
- Зависимости от других (прямой и косвенной)

Каждое из этих обстоятельств связано с дополнительными затратами труда. Однако понятно, что команда программистов может создать более крупный продукт, чем работающий в одиночестве кодировщик. Отсюда следует, что должно существовать оптимальное соответствие между численностью команды и размером задачи; оно меняется в зависимости от типа разрабатываемой системы.

По мере роста команды растет вероятность, что отдельные программисты начнут работать спустя рукава, поскольку на общем фоне это будет незаметно. В «Мифическом человеко-месяце» Брукса показывается, что подключение к проекту новых работников необязательно приводит к его ускоренному завершению (Brooks 95).

В крупном проекте больше шансов, что успех или провал будут определяться административными способностями, и больше возможностей для администраторов вызвать катастрофический провал.

В целом, чем меньше команда разработчиков, тем лучше; но она должна быть достаточно велика, чтобы суметь выполнить задачу.

6. Как защитить команду от проблем, создаваемых неопытными членами?

Неопытные программисты будут существовать всегда. Это верно для любой профессиональной области. Во многих профессиях новобранцы в той или иной форме проходят период ученичества и должны завершить академическое образование. Это гарантирует, что их мастерство будет на каком-то приемлемом уровне. Несмотря на обилие академических курсов программирования (разного качества), в нашей профессии не принято никакого формального вида ученичества. Наставничество над молодыми программистами – великолепный способ быстро довести новичков до разумного стандарта.

Есть несколько приемов, делающих работу неопытных кодировщиков менее рискованной:

- Будьте разумны и не ждите от них чудес. Давайте стажерам посильные задания.
- Следите за их ростом и старайтесь, чтобы они не боялись задавать вопросы и говорить о проблемах.
- Не требуйте от них слишком большого начального опыта: пусть пользуются популярными языками и инструментами, которые потребуют меньше времени для набора скорости.
- Не применяйте новейшие технологии и приемы.
- Стандартизируйте набор инструментов в каждой команде, чтобы новичкам достаточно было освоить его один раз.
- Учите их.
- Рецензируйте их код.
- Выберите им наставников.
- Программируйте с ними в парах.

Вопросы личного характера

1. **Что представляет собой команда, в которой вы работаете в данное время? Какому из описанных стереотипов она более соответствует?**
   a. **Создавалась ли она такой намеренно?**
   b. **Требуются ли в ней перемены?**
   c. **Это здоровая команда?**

   Какие факторы, по вашему мнению, мешают хорошей групповой работе?

   Если вы еще не сделали этого, заполните «План действий» на стр. 445. Разберитесь, как можно улучшить вашу команду, и начните перемены.

Подумайте, как вы станете осуществлять необходимые изменения. Поставьте перед собой задачи и через несколько месяцев снова оцените здоровье команды.

В число часто возникающих в команде проблем входят:

- Несбалансированность состава
- Неэффективность членов команды
- Плохое администрирование
- Нереалистичные конечные сроки
- Меняющиеся технические требования
- Отсутствие взаимопонимания

2. **Вы умеете работать в команде? Что вы можете сделать, чтобы успешнее сотрудничать с коллегами и создавать лучшие программы?**

Взгляните еще раз на характеристики личности в разделе «Личное мастерство и качества, необходимые для работы в команде» на стр. 426. Определите, в какой мере они свойственны вам и как вы можете усовершенствоваться.

3. **Какие именно обязанности возлагаются на инженера-программиста в вашей команде?**

Какая ответственность и какие права есть у инженера-программиста? Существует ли иерархия программистских должностей? Если да, то чем различаются эти должности? Входят ли в их сферу обязанностей:

- Определение границ и целей проекта
- Анализ
- Оценка временных затрат
- Архитектура
- Проектирование
- Рецензирование
- Управление проектом
- Наставничество

- Контроль производительности
- Составление документации
- Интеграция систем
- Тестирование (на каком уровне?)
- Работа с заказчиком
- Планирование модификаций или новых версий

Ответы могут различаться в зависимости от компании или проекта. Есть ли в вашей команде четкие линии подотчетности? Есть ли технические руководители или наставники, за которыми закреплены программисты?

Существует ли описание ваших функциональных обязанностей? Есть ли у вас список персональных задач? Если да, то занимаетесь ли вы их решением или этот список некорректен?

Глава 18. Защита исходного кода

Вопросы для размышления

1. Как надежно передать свой исходный код другим людям?

Самый простой вариант для коммерческого исходного кода – вообще не передавать его, что позволяет избавиться от многих проблем. Если же вы вынуждены передать код, то не забудьте выбрать для него лицензию и заключить с заказчиком соглашение о неразглашении коммерческой тайны. Следите за своей аудиторией и ее размером и, если это необходимо, примите меры, чтобы код не получил дальнейшего распространения.

Для проектов с открытым исходным кодом это не имеет большого значения; они по сути своей поставляются в виде исходников.

Перед выпуском продукта проверьте наличие в каждом файле с исходным текстом четких уведомлений об авторских правах и лицензионном соглашении.

Есть несколько механизмов выпуска исходного кода с разными возможностями обеспечить его передачу только в надежные руки:

- Допустить посторонних в вашу систему контроля версий. Можно ограничить их действия с помощью учетной записи, имеющей доступ только для чтения, а если код в открытом доступе, можно создать общую *анонимную* учетную запись.

  Очевидно, что для доступа к вашему серверу системы контроля версий пользователям необходимо предоставить некоторый уровень прав и возможность подключения по сети, поэтому требуется тщательное администрирование – и чтобы ваши клиенты ничего не повредили, и чтобы взломщики не могли увидеть ваш код.

- Сделать *тарбол* (*tarball*) исходного дерева (создать архив из сжатых файлов – по имени команды UNIX *tar*). Этот архив можно передать по почте, по FTP или на CD. Нужно лишь выбрать достаточно надежный метод.

Снабдите свой код описанием версии и данными о ревизии исходного дерева (обычно это номер версии системы контроля или номер сборки). Пометьте в системе какой-нибудь меткой включенный в выпуск код, чтобы его можно было легко извлечь заново.

2. Какая из двух моделей редактирования файлов в хранилище (блокировка загруженных файлов или их параллельная модификация) лучше?

Ни одна из моделей не лучше и не хуже другой. Каждой свойственны свои проблемы редактирования файлов, и при возникновении коллизий модификаций пользователи должны действовать по-разному.

- Модель с блокировкой требует загрузить из системы файл и зарезервировать его, прежде чем выполнять любые модификации. При этом гарантируется, что никто больше не сможет модифицировать этот файл и помешать вам работать, поскольку у вас сохраняется монопольный доступ к нему, пока вы не сохраните его в системе или не освободите, оставив в исходном состоянии. Недостаток в том, что зарезервированный файл остается заблокированным, пока владелец его не освободит. Нельзя знать, когда это произойдет.

  Если владелец сидит за соседним с вами столом, то можно выяснить это у него, хотя и неприятно. Но если он находится на другом континенте, в другом часовом поясе или случайно оставит файл заблокированным, уйдя в отпуск, то вы попали. В лучшем случае вы можете освободить файл, добравшись до компьютера владельца. Это, несомненно, приведет потом к ссорам и неприятностям.
- В параллельной модели такой проблемы нет и можно беспрепятственно продолжать кодировать в любой момент. Скрытая опасность заключается в возможности конфликтов между модификациями. Если Фред изменит в `foo.c` строки с 10 по 20, а Джордж – строки с 15 по 25, начинается гонка! У первого программиста, который сохранит файл в системе, не возникнет никаких проблем, поэтому если победит Фред, в хранилище будет помещен его файл с измененными строками с 10 по 20. Но когда Джордж попытается сохранить свою работу, система сообщит ему, что его дерево исходного кода устарело, и ему придется слить изменения, внесенные Фредом, со своим вариантом `foo.c`. Пять конфликтующих строк придется сливать вручную; Джордж должен будет дополнительно разбираться с изменениями Фреда и пытаться объединить их со своими. Лишь после этого он сможет сохранить результаты своей работы.

  Это не идеал, но на практике такое случается редко, и большинство конфликтов легко разрешается. Чаще всего случается так, что Фред модифицирует строки с 10 по 20, а Джордж – с 40 по 50; моди-

фикации не противоречат одна другой, и система может выполнить слияние автоматически. Если вы все же столкнетесь с параллельными конфликтующими модификациями, это может свидетельствовать о необходимости проведения рефакторинга кода.[1]

Ни один из режимов не идеален, но каждый работает прекрасно. Выбор режима определяется конкретной системой контроля версий, технологией разработки и культурой организации, в которой вы работаете.

3. Как различаются требования к системе управления версиями для территориально распределенной и компактно размещенной команд разработчиков?

Если система умеет работать с территориально распределенными сайтами, то уж с командой разработчиков, размещенных в одном месте, она точно справится, поэтому нас больше интересуют дополнительные требования, предъявляемые к обслуживанию нескольких площадок. В их число входят:

- Наличие масштабируемой архитектуры клиент/сервер.
- Возможность эффективной работы через низкоскоростные сетевые соединения (которые часто встречаются на вспомогательных площадках) *либо* наличие действительно высококачественной связи между сайтами. Низкоскоростные соединения требуют применения сжатия данных и разумных коммуникационных протоколов (например, пересылаться должны не файлы целиком, а различия между ними).
- Наличие централизованного администрирования учетных записей пользователей, чтобы разные площадки могли без помех взаимодействовать между собой.

Есть две основные архитектуры: глобальная сеть и репликация удаленных хранилищ. В первом случае все клиенты подключаются к центральному серверу, размещенному на главной площадке. Для этого нужны быстрые и надежные каналы связи между площадками. Во втором случае нагрузка на связь уменьшается в результате копирования хранилища на удаленные серверы в часы низкой загруженности каналов. Однако при этом сильно осложняется процедура разработки; необходимо понимать, что хранилища не синхронизируются между собой непрерывно, и разрабатывать разумные стратегии разделения труда, чтобы избежать конфликтов между осуществляемыми разработками.

При оценивании систем управления исходным кодом нельзя забывать об этих требованиях, даже если все ваши разработчики находятся в одном месте. В дальнейшем вам может потребоваться подключить дополнительную площадку или обеспечить поддержку для программистов, работающих на дому. Учитывайте это при выборе системы.

---

[1] Мартин Фаулер «Рефакторинг: улучшение существующего кода». – Пер. с англ. – СПб: Символ-Плюс, 2002.

4. Чем нужно руководствоваться при выборе системы управления исходным кодом?

Вот хорошие критерии выбора системы управления исходным кодом:

Надежность

Убедитесь, что это проверенная технология и с ней вы не потеряете в какой-то момент свои исходные файлы. Сервер должен быть надежен, а не зависать каждые пару дней.

Емкость

Инструмент должен хорошо масштабироваться и быть пригодным как для больших команд и больших проектов, так и для маленьких. Если нагрузка увеличится, не потребуется ли слишком много дискового пространства, хватит ли пропускной способности сети и останется ли приемлемым время выполнения? Нужно ли будет синхронизировать хранилища на разных площадках или система сможет работать через низкоскоростные каналы?

Гибкость

Все ли необходимые вам операции и отчеты предоставляет система? Все ли нужные вам типы файлов она поддерживает? Может ли она работать с двоичными файлами? Поддерживается ли Unicode? Существуют ли версии каталогов, позволяющие переименование и перемещение файлов? Есть ли поддержка атомарных наборов изменений или версии каждого файла ведутся независимо?

Ветвление

Система *должна* поддерживать ветвление для работы с несколькими версиями, несколькими разновидностями продукта, для параллельной разработки функций или помощи в логической разработке. Допускается ли вложенное ветвление? Легко ли объединять версии или это слишком сложно?

Платформы

Проверьте, что бы система работала на всех платформах, во всех аппаратных конфигурациях и операционных системах, с которыми вы имеете дело.

Стоимость и лицензирование

Система должна быть доступна вам по средствам (не забывайте, что есть *абсолютно* бесплатные системы). Выясните, нужно ли оплачивать лицензии для каждого клиента. Иногда это влечет скрытый рост расходов; при расширении вашей команды придется платить поставщику системы.

Аудит

Система должна вести учет пользователей, сделавших изменения в файлах – не нужно заставлять всех заводить себе учетную запись. Система должна поддерживать вашу политику доступа, позволяя при

необходимости ограничивать права модификации. Нужно ли вам автоматическое извещение о том, что в файлы внесены изменения?

Простота

Инструмент должен быть прост в применении, настройке и развертывании. Это особенно важно, если у вас нет освобожденного администратора системы.

5. **Как разделить при групповой разработке новейший код, над которым идет активная работа, и стабильную версию?**

Нужна некоторая стратегия, которая позволит разделить их в системе управления исходным кодом. Возможны такие варианты:

- Не разделять. У каждого есть новейший код, и он должен уметь работать с ним. Не сохранять в системе ничего, что явно работает неправильно.
- Создать ветви. Разработку по каждому направлению ведите в отдельной ветви и объединяйте ветви в надлежащих стабильных точках. При такой системе проблемы объединения выясняются только при слиянии ветвей. Это возлагает дополнительные обязанности на ответственного за слияние (им может быть разработчик одной из ветвей или особый системный интегратор).
- Воспользоваться меткой *stable* (*стабильная*), применив ее ко всему исходному дереву в качестве *исходного уровня*. Разработчики получают файлы из этого помеченного уровня, а затем посылают разрабатываемые ими компоненты в последнюю версию. После этого они могут работать и сохранять изменения, не трогая общего стабильного дерева. Когда можно считать новую разработку стабильной (пригодной для общего употребления), метка перемещается. Это изменение подхватывают другие разработчики, когда они в очередной раз синхронизируются с базовым уровнем.

Ваш выбор будет зависеть от возможностей системы управления версиями и принятой культуры разработки.

Вопросы личного характера

1. **Приносит ли вашей команде пользу система контроля за исходным кодом?**

В конечном итоге, помогает ли ваша система разрабатывать программное обеспечение и способствует ли сотрудничеству в большей мере, чем альтернативные варианты? Ответьте на следующие вопросы относительно своей системы:

- Тот ли это инструмент и достаточен ли его набор функций?
- Есть ли у вас специальный администратор системы или она управляется от случая к случаю?
- Все ли умеют ею пользоваться?

- Есть ли у вас система обучения? Существует ли интеграция вашей системы с управлением дефектами или средством отслеживания ошибок?

Ответьте на следующие вопросы, касающиеся управления ресурсами:

- Есть ли согласие относительно того, какой текст должен сопровождать сохраняемые в системе данные, а также относительно использования других метаданных системы управления версиями?
- Есть ли у вас последовательная система меток для важных версий исходного дерева?
- Есть ли у вас четкая (и документированная) система ветвления, обеспечивающая корректное объединение ветвей?
- Есть ли возможность автоматического создания сопроводительного документа к выпускаемой версии по данным из хранилища исходного кода?
- Можете ли вы повторять старые сборки? Если инструментальная цепочка изменилась, приняли ли вы меры к сохранению совместимости кода?
- Можете ли вы собрать продукт исключительно из данных, находящихся в хранилище, или вам потребуются дополнительные файлы?
- Какое значение придается всем этим проблемам в вашей команде?

2. Есть ли у вас резервная копия вашей текущей работы? Считается ли в вашей команде важным выполнять резервное копирование? Когда создаются резервные копии?

Если вы взяли на себя труд написать некоторый код, видимо, в нем есть потребность, а значит, должна быть его резервная копия. Есть несколько уровней, на которых может применяться резервное копирование:

- Резервное копирование персональных рабочих станций. Оно гарантирует, что ничего не пропадет с вашего жесткого диска или из вашей песочницы с исходным деревом.
- Резервное копирование сервера с хранилищем системы контроля версий. Оно гарантирует, что вы не потеряете файлы центрального исходного дерева и историю их версий.

Последнее особенно важно: не делать копий хранилища исходного кода – преступная глупость. Если на вашей рабочей станции хранятся только те области, которые вы разрабатываете в песочнице, не столь критично, существуют ли их резервные копии; в каждый данный момент есть не так много кода, не записанного в хранилище (помните о том, чтобы сохранять *понемногу и часто*), поэтому потеря локального диска не критична.

Подумайте также о резервном копировании документов и прочих создаваемых вами данных, не являющихся исходным кодом. Либо запишите их куда-нибудь в хранилище, либо поместите в соответствующее

место на файловом сервере, для которого создается резервная копия. Если не пользоваться системой контроля версий, придется организовать хранение версий документов вручную – хранить исторические версии спецификаций так же важно, как и версий исходного кода.

В многопользовательской среде порядок создания резервных копий определяет системный администратор. Обычно копирование происходит ночью, когда компьютеры менее активны и в копируемых системах меньше изменяющейся информации. (Как быть, если разработка проекта ведется на нескольких континентах с существенной разницей в часовых поясах?)

3. На каких машинах хранится ваш исходный код?

Очевидно, он хранится на серверах и рабочих станциях разработчиков в сети компании. Они находятся в офисе компании и надежно защищены межсетевым экраном. Но подумайте, не хранится ли код также на лэптопах и домашних машинах сотрудников. Насколько конфиденциальна работа? Как защитить эти машины цифровым и физическим образом?

Глава 19. Спецификации

Вопросы для размышления

1. Что лучше – плохая спецификация или ее полное отсутствие?

Недостоверная или сильно устаревшая спецификация определенно хуже. Она заведет читателей в тупик, из-за чего они потеряют много времени. Ложные данные легко могут привести к появлению ошибочного кода, для исправления которого потребуется много времени, сил и средств.

Если спецификация двусмысленна или в ней отсутствует важная информация, можно только надеяться, что читатели достаточно опытны, чтобы понять суть проблемы и правильно интерпретировать данные. Если повезет, то все они сделают одинаковые предположения в отношении отсутствующих данных. Спецификация должна быть самостоятельной и не требовать проявления интуиции от читательской аудитории.

Если спецификация слишком многословна и скрывает информацию, тогда, возможно, лучше (в конечном счете) переписать ее.

Количество фактических неточностей в спецификациях вашей компании может быть просто ужасающим! По собственному опыту замечу, что в очень немногих компаниях бывает комплект хороших спецификаций.

2. Насколько детальной должна быть хорошая спецификация?

Ответ: *достаточно детальной*, где значение «достаточно» зависит от проекта, команды, содержимого, качества сопутствующих документов и фазы луны. Лишние детали определенно могут мешать; ясно, что

при определенном уровне детализации спецификация превращается просто в код. Однако неясности в ключевых областях могут привести к катастрофическим последствиям.

3. Нужно ли требовать, чтобы все документы в компании/проекте были выдержаны в одинаковом стиле?

Это примерно так же важно, как соблюдать единый стиль кодирования. То есть существует масса более важных вещей, о которых нужно беспокоиться, даже если это первое, что бросается в глаза в спецификации. Необходимость во внешнем единообразии обусловливается (частично) тем, выходят ли документы за пределы компании. Когда документы исполнены единообразно, в одном стиле и по одинаковому шаблону, это создает впечатление профессионализма.

В конечном счете, важнее содержимое ваших документов, а не их вид.

4. Как нужно хранить документы? Например, стоит ли создавать для них указатель (по типу или проекту)?

Нужно иметь возможность быстро найти и получить написанный документ. Неважно, какая система хранения применяется, лишь бы она была известна и применялась всеми.

Обычно имеет смысл хранить все документы в одном центральном файловом хранилище и группировать их по рабочему заданию (это может быть проект, заказчик, компонента или функция). Для облегчения поиска полезно вести центральный список всех хранящихся документов. Однако придется потратить время для его сопровождения, а если этого не делать, он быстро станет бесполезным.

В больших компаниях есть специальные служащие, которые занимаются хранением и получением документов. Несмотря на их компетентность, их наличие добавляет лишние шаги в рабочую процедуру и дополнительные звенья в производственную цепочку.

Следует хранить документы, соблюдая какую-то форму контроля версий, и следить за соответствием версий документов и версий кода. Это входит в стратегию управления конфигурацией (см. раздел «Управление конфигурацией» на стр. 456).

5. Как организовать рецензирование спецификации?

Рецензирование документов похоже на рецензирование кода. Оно обычно происходит на совещании, при проведении которого нужно соблюсти некоторые важные условия: правильно отобрать рецензентов и заранее распространить рецензируемые материалы, чтобы обеспечить достаточно времени для подготовки.

Альтернативой является виртуальное рецензирование путем опроса по электронной почте или рассылки печатных экземпляров и изучения пометок рецензентов.

При рецензировании нужно обратить внимание на ряд аспектов, о важности которых следует договориться заранее:

- Качество содержания. (Полнота, корректность и т. д.? Это имеет первостепенное значение.)
- Качество стиля представления. (Соответствует ли документ правилам проекта?)
- Качество стиля текста. (Пишет ли автор не хуже Шекспира или как пятилетний ребенок? В программных спецификациях плохо и то, и другое!)

Что касается совещания, сначала лучше высказать общие комментарии о материале и подходе в целом. (Но будьте осторожны: на этом этапе можно легко уйти в сторону, начав обсуждать конкретные технические проблемы.) Затем можно обсудить специфику материала. Так как все рецензенты ознакомились с материалом заранее и уже накопили комментарии, обычно можно последовательно обсуждать раздел за разделом. Если разделы большие, можно при необходимости проходить их по абзацам.

6. Делает ли самодокументирование кода ненужными все спецификации? А какие-либо отдельные типы?

Не вполне. Самодокументирующийся код может избавить от необходимости в спецификациях архитектуры или других сопроводительных документах. Грамотная документация API в комментариях к коду может даже *иногда* заменить функциональные спецификации, если документы достаточно детальны. Однако будьте осторожны: вместо того чтобы писать много документации в грамотных комментариях, может оказаться проще ввести ту же информацию в текстовом процессоре. Документация грамотного кода никогда не заменит спецификацию требований или тестирования.

Полный комплект автоматизированных контрольных примеров *может* заменить спецификацию тестирования программной компоненты, если тесты понятны и просты в сопровождении. Однако их редко бывает достаточно, чтобы заменить тесты проверки конечного продукта.

7. Как можно организовать работу над документом нескольких авторов?

С трудом – немногие системы документирования предоставляют такие же средства для совместной работы, как системы управления исходным кодом. Если вас устроит формат документации на основе HTML, попробуйте воспользоваться средствами для совместной работы над текстами вики-сайтов.

В противном случае придется поделить документ на части и раздать их отдельным авторам. Естественно, что в результате эти разделы будут отличаться по стилю и качеству и будут исходить из разных наборов предположений; учтите это при объединении отдельных частей. Возможно, лучше сделать из этих частей отдельные документы и затем

создать обобщающий документ. Для координации работы необходимо назначить руководителя, который будет собирать отдельные части и добиваться написания их в срок.

Другой подход – поручить все написание документации одному человеку, но под строгим контролем коллег. Содержимое и структура документа заранее утверждаются на совещании, а потом автор творит документ в одиночку и представляет на обсуждение группы.

Во всех случаях нужно проявлять осторожность, потому что коллективное сочинительство может привести к появлению вымученных документов и отнять много времени.

Вопросы личного характера

1. **Кто решает, что должно содержаться в ваших документах?**

Это определяется принятым в компании процессом разработки, шаблоном документа или традициями. Если существует традиция, это не значит, что она хороша. Следите за тем, чтобы типы создаваемых вами документов, а также их содержимое действительно были полезны для вашего процесса разработки программного обеспечения.

2. **Рассмотрим ваш текущий проект. Есть ли у вас для него:**
   a. **Спецификация требований**
   b. **Архитектурная спецификация**
   c. **Проектная спецификация**
   d. **Функциональная спецификация**
   e. **Какая-нибудь еще спецификация**

   Являются ли они актуальными? Являются ли они полными? Знаете ли вы, где находятся самые свежие версии? Есть ли у вас доступ к старым версиям?

Если какие-то из них у вас отсутствуют или их качество неудовлетворительно, то почему? Как можно исправить положение?

Кому поручено следить за актуальностью документов? Поддержка версий документов – важный аспект создания спецификаций; проверьте наличие ясного плана ее осуществления.

3. **Есть ли у вас контроль версий ваших документов? Если да, то как он организован?**

Известно несколько технологий управления версиями документов:

- Хранить их в системе контроля исходного кода.
- Воспользоваться системой управления документами (или даже управления рабочим процессом).
- Воспользоваться файловой системой: ввести номер версии в имя файла, содержащего документ (можно хранить архив старых версий в отдельном каталоге).

- Сохранять старые версии в виде почтовых вложений, отправляемых некоторому «условному» пользователю (фантастика, но я *столкнулся* с такой системой в одной компании).

Какую бы систему вы ни выбрали, она должна решать такие проблемы:

- Обеспечивать простоту применения и доступность документов
- Два человека на должны одновременно редактировать один и тот же документ
- Последняя выпущенная версия должна быть отделена от экземпляра, над которым в данное время идет работа
- Не допускать случайного уничтожения документа или переписывания документа не с тем номером версии
- Поддерживать журнал учета внесенных изменений
- Обеспечивать легкость получения документа с заданным номером версии

Глава 20. Рецензия на отстрел

Вопросы для размышления

1. Зависит ли количество рецензентов от объема рецензируемого кода?

Практически нет. Если код имеет особую важность, можно привлечь больше рецензентов или специально подобрать рецензентов, обладающих большим опытом.

Впрочем, если код слишком большой, нужно не приглашать дополнительных рецензентов, а переписать его заново!

2. Какие инструменты полезны при рецензировании кода?

Здравый смысл, пара глаз и острый ум!

Есть также ряд программных инструментов, которые могут оказать пользу. Есть много средств, позволяющих проанализировать код и оценить его качество и относительный риск, который он представляет для всего базового кода. Эти средства умеют прослеживать порядок исполнения, определять чаще всего выполняемые участки кода и вычислять коэффициент сложности кода каждой функции. Последняя метрика очень полезна, когда нужно выявить фрагменты кода, в первую очередь подлежащие рецензированию. Графические средства могут помочь разобраться со структурой кода и его зависимостями (это особенно полезно для обзора иерархий классов в объектно-ориентированных языках).

3. Когда нужно рецензировать код — до или после обработки утилитами проверки исходного кода?

После. Рецензенты могут сами воспользоваться этими средствами при подготовке рецензий, а авторы должны всеми имеющимися способами

проверить свой код, прежде чем передавать его на рецензирование. Глупо не сделать этого. Зачем рецензентам тратить время на код, который и без них можно было легко улучшить? Лучше потратить его на более интересные задачи.

Если во время рецензирования обнаруживается проблема, следует задуматься над тем, нельзя ли впредь обнаруживать ее автоматически с помощью какого-то инструмента.

4. Какая подготовка нужна перед совещанием по рецензированию?

Автор должен удовлетворительным образом завершить написание кода (иначе рецензенты напрасно потратят свое драгоценное время). Председатель должен организовать совещание так, чтобы оно прошло успешно. Что более существенно, до начала совещания каждый рецензент должен:

- Прочесть и понять спецификацию.
- Ознакомиться с кодом.
- Составить список проблем и вопросов (это требует дисциплинированности; если не заставить себя это сделать, легко ограничиться поверхностным просмотром кода, чего недостаточно для глубокого рецензирования).

При методическом изучении во время совещания всегда обнаружится что-нибудь, упущенное вами ранее. Несмотря на это, предварительная подготовка необходима, чтобы не тратить напрасно чужое время на совещании.

5. Как отличить замечания рецензентов, которые нужно реализовать сразу, от тех, которые нужно взять на заметку для следующего проекта?

Решение нужно принять исходя из следующего:

- Насколько важна выявленная проблема?
- Является ли она делом личного вкуса или нарушает принятые договоренности?
- Сколько времени понадобится для решения проблемы?
- Какое влияние окажет модификация на прочий код?
- Насколько ошибочен (или обманчив) код в отсутствие исправления?
- Насколько хрупкой или опасной является модификация?
- В какой стадии находится разработка проекта? Если близок срок выпуска, нужно вносить только *существенные* изменения.

Простого правила нет. Если совещание не выработало общего мнения, то окончательное решение принимает председатель. Иногда проблемы квалифицируются посередине между *нужно исправить* и *хорошо бы иметь*, и тогда автор реализует столько срочных исправлений, сколько успеет в оставшееся время. Остальные можно отложить до следующего цикла разработки компоненты.

6. Как провести виртуальное совещание по рецензированию?

Виртуальное рецензирование обычно проводят с помощью электронной почты. Его организует председатель, обычно являющийся центральным узлом связи. Автору явно не следует доверять организацию связи, потому что иначе ему будет легко отобрать понравившиеся комментарии и игнорировать неприятные.

Используя этот метод, вы сталкиваетесь с интересным вопросом: должны ли рецензенты видеть комментарии друг друга? При виртуальном рецензировании организовать обсуждение гораздо сложнее, особенно если вся почта направляется только председателю. Однако циркулярная рассылка большого числа почтовых сообщений быстро начинает раздражать и уводит в сторону. В качестве альтернативы можно встречаться в виртуальной комнате для переговоров, воспользоваться программой мгновенного обмена сообщениями, создать телеконференцию или почтовый список рассылки.

Альтернативный способ виртуального рецензирования – это рассылка распечаток рассматриваемого кода. Рецензенты пишут комментарии на своих экземплярах и возвращают их автору. Аналогичную систему можно осуществить с помощью вики: опубликуйте свой код в вики, и пусть рецензенты комментируют его. Не столь важен формат, в котором происходит рецензирование, сколь сам факт его проведения.

7. Насколько полезны неформальные рецензии кода?

Неформальное рецензирование гораздо лучше, чем полное его отсутствие, но поскольку оно не столь досконально, то выявляет меньше недостатков (при равном качестве рецензентов).

Хотя следующие термины официально не определены, Макконнелл описывает два типа неформального рецензирования (McConnell 96):

Критический анализ (walkthrough)

Это совершенно неофициальные собрания программистов, которые вместе просматривают код. Можно делать это в редакторе и тут же вносить изменения.

Чтение кода (code reading)

Автор рассылает код рецензентам, которые комментируют его и возвращают обратно.

Вопросы личного характера

1. Подвергается ли рецензированию код в вашем проекте? Достаточно ли проводится рецензирований?

Даже если такие совещания проходят более или менее регулярно, вполне вероятно, что рецензирование проводится в недостаточном объеме. Эта практика недооценивается; если видно, что код работает, то кажется, что и тратить время на его обсуждение нецелесообразно.

Такая позиция легкомысленна. Время, которое требуется для поиска причин ошибочной работы программы, часто значительно превышает затраты на рецензирование. Рецензирование кода – разумный и практичный способ управления процессом разработки и обеспечения высокого качества программного обеспечения.

Каким образом вы могли бы оказать положительное влияние на ситуацию с рецензированием в вашем нынешнем проекте?

2. **Есть ли у вас программисты, чей код считается выше того, чтобы быть рецензируемым?**

К уважаемому гуру-программисту часто относятся с таким трепетом, что никому и в голову не приходит предложить рецензирование его работы. Или все боятся. Такое почитание ошибочно и опасно.

На мой взгляд, гуру иногда пишут код, более достойный рецензирования, чем код любых других авторов, – полный непонятных и недоступных для сопровождения загадок. То, что они никогда не выставляют свой код для рецензирования, отражает их неправильное отношение к задаче и команде. Ничей код не может быть освобожден от рецензирования; весь код должен быть тщательно изучен.

3. **Какой процент вашего кода когда-либо проходил рецензирование?**

Можно не сомневаться, что этот объем невелик, иначе вы выпадаете из общих рядов. Насколько формальными были обсуждения? Насколько полезными они были и какое влияние оказали на качество кода в конечном итоге?

Какая часть вашего кода, не подвергавшегося рецензированию, была написана в паре? Какой объем *следовало бы* рецензировать? Какая часть непрорецензированного кода была критически важным коммерческим кодом? Сколько ошибок проникло в готовый продукт и сколько из них вызвало впоследствии проблемы?

Даже если культура вашего проекта не предполагает проведение рецензирования, возьмите себе за правило предлагать делать официальную рецензию вашей работы. Не смущайтесь тем, что никто больше так не поступает – ваш код будет выше среднего уровня!

Глава 21. Какой длины веревочка?

Вопросы для размышления

1. **Как спасти отстающий проект и привести его в соответствие с графиком?**

Один из способов не стать свидетелем провала проекта – это как можно скорее бежать из него, как крысе с тонущего корабля. Однако это не очень профессионально!

Если проект отстал от графика, мало что поможет исправить положение – если только в графике не отведено необычно много времени на непредвиденные обстоятельства. Можно попробовать следующие стратегии:

- Изменить сроки проекта; попробуйте договориться с клиентом о более поздней доставке.
- Изменить объем функциональности первой версии, отложив реализацию недостающих функций до следующего выпуска. Лучше сделать меньше, но лучше и вовремя, чем реализовывать ненужные функции и отставать от графика.

Не будьте безрассудны и не бросайте на проект новых разработчиков, чтобы ускорить работу. Брукс понятно описал, насколько плоха эта идея, особенно при угрозе провала проекта (Brooks 95). Нынешние разработчики будут отвлекаться на то, чтобы ввести в курс дела новых, а руководство большой командой потребует дополнительного труда. Любая прибыль наверняка окажется меньше, чем стоимость дополнительного персонала.

2. Как правильно реагировать на то, что конечный срок вам устанавливают до того, как проведено технико-экономическое обоснование или планирование?

Будьте благоразумны! Фиксированный срок поставки может быть законным деловым требованием: вы получите деньги, если поставите программу вовремя, в противном случае вы не получите ничего. Не всегда удается делать продукт идеологически правильно, отодвинув для этого конечный срок или ограничив объем функций.

Иногда при проектировании полезно заранее знать о предполагаемых сроках завершения проекта. Это поможет определить, насколько правильным и хорошо продуманным может быть ваш проект, оценить объем кода и возможности расширения его в будущем. В конечном итоге, будет ясно, что предпочесть: решение на скорую руку или красиво сконструированный код, который следовало бы писать в идеале. Будет легче решить, что нужно купить, а что изготовить самостоятельно, а также установить уровень качества, которому должен соответствовать конечный продукт.

Разъясните, что такой способ разработки программного обеспечения неидеален. *Возможно*, к вам прислушаются, и впредь ваши менеджеры не будут обещать поставок в такие рискованные сроки; это легкомысленное поведение, в котором на карту ставятся успех проекта и будущее организации.

3. Как обеспечить реальную пользу плана разработки?

Для высококачественных планов разработки характерны:

Точность

Они включают в себя задачи, необходимые для постройки программного продукта, и основаны на разумных временных оценках.

Детальность

Вместо нескольких крупных задач с грубыми оценками затрат есть много мелких задач, последовательность решения которых тщательно определена. Затраты на маленькие задачи можно оценить более точно, поэтому надежность плана в целом становится выше.

Если вы считаете, что задача состоит из нескольких частей, явно укажите это в плане. Например, решение задачи зависит от стороннего поставщика и определяется моментом поставки им продукта, после которого последует период интеграции и исправления ошибок.

Согласованность

Все согласны с планом: руководство удовлетворено уровнем рисков, а программисты согласны с оценками сроков, ничего не упущено и отражены все зависимости.

Наглядность

На основе планов отдельные разработчики и менеджеры принимают важные решения. Об изменениях сроков сообщается посредством планов. Ведется учет версий плана и регистрируется ход его выполнения.

Контролируемость

Если не следить за выполнением графика, то сделанные оценки сроков становятся пустыми цифрами. Успехи должны сравниваться с планом. Эта оценка служит для корректировки курса проекта разработки.

4. Почему программисты работают с разной скоростью? Как отразить это в плане?

Программисты отличаются друг от друга во многих отношениях:

- У них различные технические возможности, и они по-разному рассуждают о задачах. Это влияет на качество их работы.
- Различие в уровне опытности приводит к разным конструктивным решениям.
- Люди по-разному относятся к работе, на что влияют: ответственность за прежние проекты, степень лояльности компании или увлеченности проектом, отношение к профессии программиста, внешние обязанности (семейные, общественные и т. д.).
- Одни обладают высокой мотивацией и готовы часами работать сверхурочно, чтобы завершить проект. Другие предпочитают не задерживаться после работы и отправляются развлекаться.

Программисты различаются между собой не только длительностью выполнения заданий. Различаются качество их кода, разумность конструкций и количество ошибок в написанных программах. Даже один и тот же программист каждый раз по-разному решает одну и ту же задачу – по мере накопления опыта, он делает это все лучше.

Чтобы учесть все это в плане проекта, проверьте, каким программистам назначены все задачи. Если задача не входит в основную область компетенции программиста, увеличьте в графике срок ее выполнения или выделите дополнительное время на непредвиденные обстоятельства. Дайте ему сначала дополнительное задание для разогрева и проверьте, учтено ли время, которое может потребоваться на обучение.

Вопросы личного характера

1. **Какая доля проектов, в которых вы участвовали, была выполнена в срок?**
   a. **Для удачных проектов: чем был обусловлен успех планирования?**
   b. **Для неудачных: в чем заключались главные проблемы?**

Описать провал проще, чем успех; выявить единственную причину, по которой что-то не задалось, проще, чем указать сложный баланс удачно совпавших обстоятельств. Когда в проекте все идет хорошо, кажется, что так и должно быть.

Решению проблем и уменьшению рисков в плане способствует итеративная и инкрементная разработка. Правильно понятые рабочие задания, детальный план и хороший начальный проект тоже важны. Качественное тестирование, выполняемое безотлагательно и часто, тоже делает разработку значительно более безопасной. Талантливые разработчики тоже приносят много пользы!

2. **Насколько точными бывают ваши оценки? Насколько далекими от ваших оценок оказываются результаты?**

Это такой вид мастерства, который можно совершенствовать вечно. Опыт – большой учитель. Надеюсь, ваши последние оценки точнее, чем прежние. Разве не так?

Если вам пока не предлагали сделать оценку времени, начните практиковаться сами! Напишите мини-план текущей задачи, над которой работаете. Сделайте оценки для небольших частей этого плана и посмотрите, насколько вы окажетесь точны. Дополнительная выгода заключается в том, что вам придется хорошенько задуматься над тем, чем вы занимаетесь, и предложить хороший проект. Кроме того, вам придется выделить достаточно времени для тестирования, отладки и составления документации – все это полезные вещи.

Глава 22. Рецепт программы

Вопросы для размышления

1. **Какое влияние оказывают друг на друга стиль программирования и процесс разработки?**

Они могут не иметь никакого отношения друг к другу, но лучше, если, начиная проект, вы задумаетесь о них обоих.

Итеративные процессы проще реализуются с помощью методологий программирования, поддерживающих разделение на компоненты – объектно-ориентированную парадигму. Линейные процессы пригодны для всех стилей программирования, но они необязательно оказываются лучшим выбором.

Наибольшее влияние на выбор оказывают прежний опыт разработчика и его личные предпочтения в отношении стиля программирования.

2. Какой стиль программирования лучше?

Хитрый вопрос! Если вы решили на него ответить, то положите книжку и отшлепайте себя 30 раз мокрой тряпкой.

3. Какой процесс разработки лучше?

И на этот вопрос вы клюнули? Вам поможет только электрошоковая терапия от 9-вольтового аккумулятора.

4. Укажите для каждого из перечисленных в этой главе процессов разработки его место на осях классификации процессов (см. стр. 536).

Вначале краткое резюме: классификация *толстый/тонкий* относится к объему бюрократизма и бумаготворчества, свойственному процессу, *последовательность* описывает, насколько линеен и предсказуем процесс, а *направление проектирования* означает, откуда это проектирование ведется – от мелких деталей реализации или от обзора общего плана.

Ad hoc (специальный процесс)

Кто сумеет навести в этом порядок? Специальный процесс может оказаться в любом месте любой оси и даже постоянно перемещаться. Разработчики в процессе Ad hoc обычно не склонны к бюрократизму, но дисциплина у них отсутствует – одни вещи выпадают из их внимания, другие повторяются снова и снова. Никакой последовательности нет вообще, поэтому данный антипроцесс вообще выходит за рамки всех оценок, и если у него и есть какой-то проект, то он, возможно, не имеет ничего общего с тем, что будет создано!

Каскадная модель

Это довольно толстый, весьма линейный процесс. Обычно он приводит к нисходящему проектированию, хотя это не обязательно.

SSADM

Тут оценка по шкале толщины максимальная: бумаготворчество и подробно документируемые этапы в достатке. По оси последовательности полным ходом приближается к линейности.

V-модель

Еще один толстый линейный процесс (хотя некоторые его части в целях эффективности явно распараллелены). Как и другие вари-

анты каскадной модели, имеет склонность к нисходящему проектированию.

Создание прототипов

Явно циклический процесс (хотя путем ограничения количества прототипов можно придать процессу разработки некоторый уровень линейности). Здесь присутствует тенденция к смещению в сторону лагеря тонких процессов, иногда чрезмерная: одних лишь прототипов недостаточно для регистрации требований пользователей или проектных решений, поэтому при разработке прототипов можно легко уклониться от регистрации требований в спецификациях.

Итеративный и инкрементный процесс

Также нелинейный по замыслу, этот процесс можно сделать сколь угодно бюрократичным, но в некоторых вариантах (особенно демонстрируемых «ускоренным программированием») может быть весьма тонким. Итеративные и инкрементные процессы тяготеют к середине оси направления проектирования – на каждой итерации мы осуществляем проектирование от верхнего уровня до нижнего. Эти проектные решения пересматриваются в следующем цикле, и снова повторяется проектирование на верхнем и нижнем уровне.

Спиральная модель

Толстая версия итеративного и инкрементного процесса.

Методологии ускоренного проектирования

Ускоренные процессы – тонкие и нелинейные. Они не фиксируют направление проектирования; вы его непрерывно меняете. Сравните проектирование с поездкой в Париж на машине: в обычном случае вы поворачиваете машину в сторону Парижа и едете; в ускоренном процессе вы начинаете езду, а потом непрерывно крутите руль. *Возможно*, вы даже составите план средней части своего движения, прежде чем найдете лучший маршрут для выезда из своего города.

Не забывайте, что в каждой организации реализация конкретной модели процесса обязательно подгоняется под принятые в ней правила работы. (Это совершенно нормально.) Эти поправки могут быть весьма существенными. Например, можно базировать разработку на V-модели, но стремиться сделать переход между фазами как можно более облегченным, чтобы избежать ненужной бюрократии.

5. Если процессы разработки и стили программирования являются рецептами, как должна выглядеть поваренная книга разработки программного обеспечения?

Наверное, она была бы слишком похожа на учебник по программированию. Картинок, от которых текут слюнки, там нашлось бы немного! Можно предположить, что как различаются рецепты «голого повара»[1]

[1] Если вам показалось это грубым, загляните на *www.jamieoliver.com*.

и Рэйчел Рей, так и в гипотетической рецептурной книге найдут отражение различные подходы разработки программного обеспечения.

Поваренных книг для программистов встречается немного, потому что не так часто люди охотятся за новыми рецептами. Такого рода вещи возникают, только когда рекламные механизмы начинают раскручивать очередное «великое изобретение».

6. **Можно ли, правильно выбрав процесс, сделать разработку программного обеспечения более предсказуемой и повторяемой задачей?**

Индустрия программного обеспечения пока не готова выступать с такими заявлениями. Как бы старательно мы ни гомогенизировали процесс разработки, качество кода в конечном счете определяется качеством (например, опытом, способностями, интуицией и чутьем), а также особым настроением (например, способностью к концентрации, пребыванием в зоне или постоянными отвлечениями, см. стр. 523) программистов, которые выполняют работу. Опытный мастер создаст более красивую, устойчивую и модную конструкцию, чем ученик-новичок.

При таком расхождении трудно создавать программы неизменного качества, даже при наличии строго регламентированного процесса. Если вы попробуете еще раз создать ту же самую программу с теми же самыми программистами и тем же самым производственным процессом, вы не получите того же самого результата. Сегодня команда примет одно решение, а через день – другое, что приведет к появлению совершенно другой программы, у которой будут свои недостатки и сильные стороны. (Из области гипотез: та же команда должна научиться на своих ошибках и со второй попытки создать другой, возможно, лучший продукт.)

Методы ускоренного программирования учитывают это и прославляют непредсказуемость создания программ. Они пытаются справиться с неопределенностью путем выбора прагматичных подходов, минимизирующих неотъемлемый риск непредсказуемой задачи.

Вопросы личного характера

1. **Каким процессом разработки и языком программирования пользуетесь вы в данное время?**
   a. **Было ли по этому вопросу достигнуто формальное соглашение между участниками разработки или действуют традиции?**
   b. **Как происходил выбор? Это выбор для данного проекта или рецепт, которым вы пользуетесь всегда?**
   c. **Существует ли какая-то документация по данному вопросу?**
   d. **Соблюдает ли команда требования процесса? Если возникают проблемы и вас прижимают к стене, продолжаете ли вы придерживаться процесса или все теории забываются, когда нужно хоть что-то представить в срок?**

Это вопрос для определения того, насколько организована ваша команда разработчиков и каким способом ведется разработка – целена-

правленным или случайным. Вы действительно *умеете* изготавливать программный продукт или все еще надеетесь, что сможете выполнить работу благодаря героическим усилиям нескольких членов команды?

Можете ли вы назвать конкретный стандарт, на основании которого строите свою работу? Он документирован? Он понятен? Всем понятен – *всем* разработчикам, *всем* менеджерам процесса и *всем*, кто играет какую-то роль в производственном процессе?

2. **Считаете ли вы правильным текущий выбор процесса и стиля? Это лучший для данного момента способ разработки?**

Если вы не знаете, то как вы производите программное обеспечение? Если ваш способ не самый лучший, какой был бы лучше и почему?

Опасайтесь методов ad hoc. Я видел много организаций, в которых нет установленного способа; один человек создает чистые ОО-проекты, а другой избегает ОО и пишет в структурном стиле. В итоге получается уродливый и непоследовательный код.

3. **Признается ли в вашей организации, что существуют другие модели разработки, достойные рассмотрения?**

Выясните, кто принимает решения по такого рода вопросам – разработчики, руководитель команды программистов или менеджеры? Достаточно ли компетентны эти люди в вопросах разработки программного обеспечения? Выясните, почему они предпочитают работать нынешним способом: какие задачи они уже решили? Часто применение необычной процедуры разработки имеет исторические причины – в организациях стихийно, а не сознательно, складывается определенный тип практики работы.

Трудно ли будет убедить вашу организацию принять другую модель процесса?

Глава 23. За гранью возможного

Вопросы для размышления

1. **Какие из рассмотренных нами ниш программирования обладают наибольшим сходством между собой или общими характеристиками? Какие особенно различаются?**

Общего больше, чем может показаться вначале. Вот некоторые перекрывающиеся области:

- Программирование игр и веб-приложений можно считать специфическими видами программирования приложений.
- Веб-программирование – это вид программирования распределенных приложений.
- Программирование масштаба предприятия может иметь вид веб-приложений.

- Системное программирование может осуществляться для встроенных платформ.
- Численное программирование можно оптимизировать с помощью параллельных и распределенных вычислений.

2. В какой из перечисленных областей труднее всего работать?

Каждый вид программирования выдвигает свой набор проблем, а каждая отдельная программа имеет индивидуальные сложности. Иначе для занятия программированием не требовалось бы особого мастерства и этим мог заниматься каждый дурак. (Тот факт, что много идиотов *все же* занимается программированием, здесь не обсуждается!)

Более «сложными» сферами программирования можно считать те, которые требуют более формального процесса для обеспечения нужного качества. Например, особенно чревата опасностями разработка программ, надежность которых критически важна (о чем говорится в «Резюме» на стр. 574). Не допускающие двойного толкования спецификации, крайне формализованные разработка и тестирование и сертификация в соответствии со стандартами – вот что характерно для этой области наряду с устойчивостью к отказам.

В частности, численное программирование вызовет затруднения у того, кто не склонен к математике и разработке сложных алгоритмов. Здесь требуется дополнительная математическая подготовка.

3. Следует ли стремиться стать экспертом в одной конкретной области или лучше иметь хорошую подготовку во всех областях без узкой специализации?

Разбираться во всех областях полезно. Однако для настоящего успеха в какой-то области требуется специальное мастерство и опыт, который можно получить только в результате практической деятельности. Чтобы получить такой ценный опыт, следует сосредоточиться на одной конкретной области. Винсент ван Гог отмечал: «Если овладеть одной областью и хорошо в ней разбираться, то одновременно начинаешь постигать и разбираться во многих других». Изучите конкретные сложные вопросы, которые выделяют вашу сферу среди других.

4. С какой из областей программирования следует знакомить начинающих программистов?

Авторы курсов программирования редко задумываются на эту тему. Это досадный пробел; многие курсы ориентированы на программирование не в реальном мире, а в какой-то теоретической, соединяющей в себе противоположные свойства области. Конечно, так гораздо проще учить программированию, и обучаемые реже сталкиваются с проблемами. Но важно разбираться в том, как правильно принимать конструктивные решения, находясь в гуще промышленного производства программного обеспечения, и этому нужно как-то обучать.

В сравнении с прочими областями программирование приложений относительно не обременено специфическими ритуалами и порядками, поэтому можно считать, что с данной областью проще всего знакомить начинающих программистов. Поскольку обучаемые редко имеют представление обо всей сфере разработки программ, это, скорее всего, именно то, что они и предполагали изучать.

Вопросы личного характера

1. **В какой области программирования работаете вы в настоящее время? Какое влияние оказывает она на код, который вы пишете? Какие конкретные решения в области архитектуры и реализации она потребовала от вас принять?**

Важно понимать, какого типа код вы пишете, чтобы принимать правильные программные решения. Если вы не можете объяснить, какое влияние оказывают на ваш код проблемы предметной области, то, видимо, вы не дали себе достаточного труда подумать над тем, чем вы занимаетесь. Программное обеспечение предназначено для существования в определенной среде и потому должно формироваться этой средой.

2. **Доводилось ли вам работать в нескольких областях программирования? Было ли вам легко принимать другой взгляд на вещи и применять технологии, присущие новой сфере?**

Не поддавайтесь искушению не обращать внимания на эти различия и бездумно перескакивать из одной области в другую. Это может привести к тому, что вы станете писать плохой код. Возможно, вы слишком поздно заметите, что пишете не тот код, – когда вас замучают поиски ошибок и попытки оптимизировать систему с тем, чтобы она удовлетворяла первоначальным техническим требованиям (например, в отношении размера кода или масштабируемости). Если только тогда вы обнаружите, что ваша работа не соответствует среде, для которой она предназначена, то вы окажетесь в скверном положении.

3. **Есть ли среди ваших коллег люди, не понимающие причин, влияющих на то, как вы пишете код? Сталкивались ли вы с тем, что встроенное программное обеспечение пишут программисты, умеющие писать только приложения? Какой выход может быть из этой ситуации?**

Программисты, не приспосабливающие свою работу к требованиям предметной области, ставят ваш проект под угрозу. Если они не понимают, что данной области внутренне присущи определенные ограничения (масштабируемость, производительность, размер кода, способность к взаимодействию и т. п.), их код не будет соответствовать спецификациям, и они окажутся слабым звеном в вашей цепочке разработки.

Обнаружить это можно с помощью рецензирования кода и архитектуры, а также программирования парами.

Библиография

(Alexander 79)

Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press, 1979. 0195024028.

(Aristotle)

Aristotle (384–322 BC). *Rhetoric*. Book 1, Chapter 11, Section 20. 350 BC.

(Beck 99)

Beck, Kent. *Extreme Programming Explained*. Addison-Wesley, 1999. 0201616416.[1]

(Belbin 81)

Belbin, Meredith. *Management Teams: Why They Succeed or Fail*. Butterworth Heinemann, 1981. 0750659106.[2]

(Bentley 82)

Bentley, Jon Louis. *Writing Efficient Programs*. Prentice Hall Professional, 1982. 013970244X.

(Bersoff et al. 80)

Bersoff, Edward, Vilas Henderson, and Stanley Siegel. *Software Configuraion Management: An Investment in Product Integrity*. Longman Higher Education, 1980. 0138217696.

(Boehm 76)

Boehm, Barry. «Software Engineering». *IEE Transactions on Computers*. Vol. C-25, No. 12, pp. 1,226–1,241. 1976. http://www.computer.org/tc.

(Boehm 81)

Boehm, Barry. *Software Engineering Economics*. Prentice Hall, 1981. 0138221227.

---

[1] Кент Бек «Экстремальное программирование». – Пер. с англ. – СПб.: Питер, 2002.

[2] Мередит Белбин «Команды менеджеров. Секреты успеха и причины неудач». – Пер. с англ. – Hippo, 2003.

(Boehm 87)

Boehm, Barry. «Improving Software Productivity». *IEEE computer*, Vol. 20, No. 9. 1987.

(Boehm 88)

Boehm, Barry. «A Spiral Model of Software Development and Enhancement». *IEEE computer*, Vol. 21. May 5, 1988.

(Booch 97)

Booch, Grady. *Object Oriented Analysis and Design With Applications.* Benjamin/Cummings, 1994. Second Edition. 0805353402.

(Briggs 80)

Briggs Myers, Isabel. *Gifts Differing: Understanding Personality Type.* Consulting Psychologist's Press, 1980. 0891060111.

(Brooks 95)

Brooks, Frederick P., Jr. *The Mythical Man Month.* Addison-Wesley, 1995. Anniversary Edition. 0201835959.[1]

(DeMarco 99)

DeMarco, Tom, and Timothy Lister. *Peopleware: Productive Projects and Teams.* Dorset House, 1999. Second Edition. 0932633439.[2]

(Dijkstra 68)

Dijkstra, Edsger W. «Go To Statement Considered Harmful». *Communications of the ACM*, Vol. 11, No. 3, pp. 147–148. 1968.

(Doxygen)

van Heesch, Dimitri. *Doxygen.* http://www.doxygen.org.

(Economist 01)

«Agility counts». *The Economist*. September 20, 2001.

(Fagan 76)

Fagan, Michael. «Design and code inspections to reduce errors in program development». *IBM Systems Journal*, Vol. 15, No. 3. 1976.

(Feldman 78)

Feldman, Stuart. «Make – A Program for Maintaining Computer Programs». *Bell Laboratories Computering Science Technical Report 57.* 1978.

(Fowler 99)

Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999. 0201485672.[3]

---

[1] Фредерик Брукс «Мифический человеко-месяц или как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

[2] Демарко Т., Листер Т. «Человеческий фактор: успешные проекты и команды». – Пер. с англ. – СПб.: Символ-Плюс, 2005.

[3] Мартин Фаулер «Рефакторинг: улучшение существующего кода». – Пер. с англ. – СПб: Символ-Плюс, 2002.

(Gamma et al. 94)

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 0201633612.[1]

(Gosling et al. 94)

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2000. Second Edition. 0201310082. http://java.sun.com.

(Gould 75)

Gould, John. «Some Psychological Evidence on How People Debug Computer Programs». *International Journal of Man-Machine Studies*. 1975.

(Groom 94)

Groom, Winston. *Forrest Gump*. Black Swan, 1994. 0552996092.[2]

(Hoare 81)

Hoare, Charles. «The Emperor's Old Clothes». *Communications of the ACM*, Vol. 24, No 2. ACM, 1981.

(Humphrey 97)

Humphrey, Watts S. *Introduction to the Personal Software Process*. Addison-Wesley, 1997. 0201548097.

(Humphrey 98)

Humphrey, Watts S. «The Software Quality Profile». *Software Quality Professional*. December 1998. http://www.sei.cmu.edu/publications/articles/quality-profile/.

(Hunt Davis 99)

Hunt, Andrew, and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 1999. 020161622X.[3]

(IEEE 84)

IEEE Standard Glossary of Software Engineering Terminology. ANSI/IEEE, 1984. ANSI/IEEE Standard 729.

(ISO 84)

ISO7498:1984(E) Information Processing Systems – Open Systems Interconnection – Basic Reference Model. International Standard for Information Systems, 1984. ISO Standard ISO 7498:1984(E).

(ISO 98)

ISO/IEC 14882:1998, Programming Languages – C++. International Standard for Information Systems, 1998. ISO Standard ISO/IEC 14882:1998.

---

[1] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб.: Питер, 2001, 2007.

[2] Уинстон Грум «Форрест Гамп». – Пер. с англ. – Амфора, 2004.

[3] Э. Хант и Д. Томас «Программист-прагматик. Путь от подмастерья к мастеру». – Пер. с англ. – Лори, 2009.

(ISO 99)

ISO/IEC 9899:1999, Programming Languages – C. International Standard for Information Systems, 1999. ISO Standard ISO/IEC 9899:1999.

(ISO 05)

ISO/IEC 23270:2003, Information technology – C# Language Specification. International Standard for Information Systems, 2005. ISO Standard ISO/IEC 23270:2003.

(Jackson 75)

Jackson, M.A. *Principles of Program Design*. Academic Press, 1975. 0123790506.

(Javadoc)

Javadoc. Sun Microsystems, Inc. http://java.sun.com/products/jdk/ javadoc.

(Kernighan Pike 99)

Kernighan, Brian W., and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999. 020161586X.[1]

(Kernighan Plaugher 76)

Kernighan, Brian W., and P.J. Plaugher. *Software Tools*. Addison- Wesley, 1976. 020103669X.

(Kernighan Plaugher 78)

Kernighan, Brian W., and P.J. Plaugher. *The Elements of Programming Style*. McGraw-Hill, 1978. 0070341990.

(Kernighan Ritchie 88)

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. Second Edition. 0131103628.[2]

(Knuth 92)

Knuth, Donald. *Literate Programming*. CSLI Publications, 1992. 0937073806.

(Kurlansky 99)

Kurlansky, Mark. *The Basque History of the World*. Jonathan Cope, 1999. 0224060554.

(McConnell 96)

McConnell, Steve. *Rapid Development*. Microsoft Press, 1996. 1556159005.

(McConnell 04)

McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004. Second Edition. 0735619670.[3]

---

[1] Б. Керниган и Р. Пайк «Практика программирования». – Пер. с англ. – М: Вильямс, 2004.

[2] Б. Керниган, Д. Ритчи «Язык программирования Си». – Пер. с англ. – Невский Диалект, 2000.

[3] С. Макконнелл «Совершенный код. Практическое руководство по разработке программного обеспечения». – Пер. с англ. – СПб.: Питер, 2005.

(Meyers 97)

Meyers, Scott. *Effective C++*. Addison-Wesley, 1997. Item 34: Minimize complication dependencies between files. 0201924889.[1]

(Miller 56)

Miller, George A. «The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information». First published in *Psychological Review*, 63, pp. 81–97. 1956.

(Myers 86)

Myers, Ware. «Can software for the Strategic Defense Initiative ever be error-free?» *IEEE computer*. Vol. 19, No. 10, pp. 61–67. 1986.

(Page Jones 96)

Page-Jones, Meilir. *What Every Programmer Should Know About Object-oriented Design*. Dorset House Publishing Co., 1996. 0932633315.

(Royce 70)

Royce, W.W. «Managing the Development of Large Software Systems». Proceedings of IEEE WESCON, August 1970.

(Simpsons 91)

Simpsons, The. «Do the Bart Man». Geffen, 1991. GEF87CD.

(Stroustrup 97)

Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, 1997. Third Edition. 0-201-88954-4.

(UML)

Unified Modeling Language. Object Management Group. http://www.uml.org.

(Vitruvius)

Vitruvius Pollio, Marcus (c. 70–25 BC). *De Architectura*. Book 1, Chapter 3, Section 2.

(Weinberg 71)

Weinberg, Gerald. *The Psychology Of Computer Programming*. Van Nostrand Reinhold, 1971. 0932633420.

(Wulf 72)

Wulf, William A. «A Case Against the GOTO». Proceedings of the twenty-fifth National ACM Conference, 1972.

[1] Скотт Мэйерс «Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ». – Пер. с англ. – ДМК Пресс, 2006.

Алфавитный указатель

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Ю

Я