
Второе юбилейное издание к 20-летию выхода книги
Программист-прагматик
Ваш путь к мастерству
Дэвид Томас
Эндрю Хант
Предисловие Сарона Йитбарека

Программист-прагматик

Второе юбилейное издание
к 20-летию выхода книги

The Pragmatic Programmer

20th Anniversary Edition

David Thomas
Andrew Hunt

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Программист-прагматик

Второе юбилейное издание
к 20-летию выхода книги

Дэвид Томас
Эндрю Хант

Диалектика

Москва ◆ Санкт-Петербург
2020

ББК 32.973.26-018.2.75
Х19
УДК 681.3.07

ООО "Диалектика"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *И.В. Берштейна*
Под редакцией канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Хант, Эндрю, **Томас**, Дэвид.

Х19 Программист-прагматик: 2-е юбилейное издание. : Пер. с англ. — СПб. : ООО "Диалектика", 2020 — 368 с. : ил. — Парал. тит. англ.

ISBN 978-5-907203-32-7 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Эндрю Хант, Дэвид Томас

Программист-прагматик
2-е юбилейное издание

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907203-32-7 (рус.)

ISBN 978-0-13-595705-9 (англ.)

Оглавление

Предисловие 16

Глава 1. Философия прагматизма 29

Глава 2. Прагматичный подход 57

Глава 3. Основные инструментальные средства 107

Глава 4. Прагматичная паранойя 139

Глава 5. Гибкость или ломкость 167

Глава 6. Параллельность 213

Глава 7. По ходу кодирования 239

Глава 8. До начала проекта 297

Глава 9. Прагматичные проекты 319

Приложение А. Послесловие 343

Приложение Б. Библиография 347

Приложение В. Возможные ответы на упражнения 349

Предметный указатель 364

Содержание

Об авторах 15

Предисловие 16

Глава 1. Философия прагматизма 29

Тема 1. Это ваша жизнь 30
Тема 2. Кот съел мой исходный код 31
Доверие в команде 32
Взятие на себя ответственности 32
Тема 3. Программная энтропия 34
Прежде всего — не навредить 36
Тема 4. Суп из камней и вареные лягушки 37
Со стороны селян 38
Тема 5. Достаточно хорошее программное обеспечение 40
Идите на компромиссы с пользователями 41
Знайте меру 42
Тема 6. Ваш багаж знаний 43
Ваш багаж знаний 43
Создание своего багажа знаний 44
Цели 45
Возможности для обучения 47
Критическое мышление 47
Тема 7. Общайтесь! 49
Знайте, с кем вы общаетесь 50
Знайте, что вам требуется сказать 51
Выбирайте удобный момент 51
Выбирайте стиль общения 52
Подавайте свои идеи в привлекательной форме 52
Привлекайте тех, с кем общаетесь 53
Учитесь слушать 53
Отвечайте людям 53
Документация 54
Краткие итоги 54

Глава 2. Прагматичный подход 57

Тема 8. Сущность качественного проектирования 58
Принцип ETC — это ценность, а не правило 58
Тема 9. DRY — пороки дублирования 60
Принцип DRY не только для кодирования 61
Дублирование в исходном коде 62

Дублирование в документации 64
Представительное дублирование 67
Дублирование среди разработчиков 68
Тема 10. Ортогональность 69
Что такое ортогональность 69
Преимущества ортогональности 71
Проектирование 72
Инструментальные средства и библиотеки 74
Кодирование 75
Тестирование 76
Документация 76
Как уживаться с ортогональностью 77
Тема 11. Обратимость 79
Обратимость 80
Гибкая архитектура 81
Тема 12. Трассирующие пули 83
Код, сверкающий в темноте 84
Трассирующие пули не всегда попадают в цель 87
Трассирующий код в сравнении с прототипированием 87
Тема 13. Прототипы и памятные записки 89
Что подлежит прототипированию 90
Как пользоваться прототипами 91
Прототипирование архитектуры 91
Как *не* следует пользоваться прототипами 92
Тема 14. Предметно-ориентированные языки 93
Некоторые предметно-ориентированные языки 94
Характеристики предметно-ориентированных языков 96
Компромисс между внутренними и внешними предметно-ориентированными языками 97
Внутренний предметно-ориентированный язык почти даром 98
Тема 15. Оценивание 100
Какой точности оценки достаточно? 100
Откуда берутся оценки 101
Оценивание сроков выполнения проектов 103
Что ответить на просьбу что-нибудь оценить 105

Глава 3. Основные инструментальные средства 107

Тема 16. Сила простого текста 109
Что такое простой текст 109
В чем сила простого текста 110
Наименьший общий знаменатель 112

Тема 17. Игры в скорлупки 113
Ваша собственная оболочка 114
Тема 18. Эффективное редактирование 116
Что означает свободное владение редактором 116
Стремление к свободному владению редактором 117
Тема 19. Контроль версий 119
Все начинается с исходного кода 120
Ветвление 121
Контроль версий как центральный узел проекта 122
Тема 20. Отладка 124
Психология отладки программ 125
Мысленная установка на отладку 125
С чего начинать отладку 126
Стратегии отладки 127
Программист в чужой стране 128
Бинарный поиск 129
Метод резинового утенка 131
Элемент удивления 133
Контрольный список вопросов по отладке 134
Тема 21. Манипулирование текстом 134
Тема 22. Технические дневники 137

Глава 4. Прагматичная паранойя 139

Тема 23. Проектирование по контракту 140
Принцип проектирования по контракту 141
Реализация проектирования по контракту 145
Проектирование по контракту и аварийное завершение 146
Семантические инварианты 146
Динамические контракты и агенты 148
Тема 24. Мертвые программы не лгут 149
Принцип "поймал–отпустил" — только для ловли рыбы 150
Аварийное завершение вместо отправки на свалку 151
Тема 25. Утвердительное программирование 152
Утверждения и побочные эффекты 153
Оставляйте включенным режим утверждений 154
Тема 26. Как сбалансировать ресурсы 156
Вложенное выделение ресурсов 159
Объекты и исключения 159
Баланс исключений 160
Когда нельзя сбалансировать ресурсы 161
Проверка баланса 162

Тема 27. Не опережайте свет фар вашего автомобиля 163
Черные лебеди 165

Глава 5. Гибкость или ломкость 167

Тема 28. Развязывание 168
Крушения поездов 170
Изъяны глобализации 173
Наследование усугубляет связывание 175
Все дело в изменениях 175
Тема 29. Манипулирование реальным миром 176
События 176
Конечные автоматы 177
Проектный шаблон "Обозреватель" 181
Модель "издатель–подписчик" 182
Реактивное программирование, потоки данных и события 183
События вездесущи 185
Тема 30. Преобразовательное программирование 186
Обнаружение преобразований 189
В чем же здесь польза 193
А как насчет обработки ошибок 194
Преобразования преобразуют программирование 198
Тема 31. Налог на наследование 199
Немного предыстории 199
Трудности применения наследования для совместного использования кода 200
Лучшие альтернативы 202
Наследование редко является ответом 207
Тема 32. Конфигурирование 208
Статическая конфигурация 208
Конфигурация как служба 209
Не пишите морально устаревший код 210

Глава 6. Параллельность 213

Тема 33. Разрывание временного связывания 214
В поисках параллельности 215
Возможности для достижения параллельности 216
Возможности для достижения параллелизма 217
Выявить возможности проще всего 219
Тема 34. Общее состояние — неверное состояние 219
Неатомарные обновления 220
Множественные транзакции ресурсов 224
Обновления без транзакций 225

Другие виды исключительного доступа 226
Доктор, мне больно... 226
Тема 35. Акторы и процессы 227
Акторы могут быть только параллельными 227
Простой актор 228
Отсутствие явной параллельности 232
Erlang подготавливает почву 232
Тема 36. Классные доски 233
Классная доска в действии 235
Системы обмена сообщениями могут быть подобны классным доскам 236
Но не все так просто... 237

Глава 7. По ходу кодирования 239

Тема 37. Прислушивайтесь к своим инстинктам 241
Боязнь пустой страницы 241
Борьба с собой 242
Как прислушиваться к своим инстинктам 243
Время играть! 243
Не только *свой* код 244
Не только код 245
Тема 38. Программирование по совпадению 245
Как программировать по совпадению 246
Как программировать обдуманно 250
Тема 39. Быстродействие алгоритмов 252
Что подразумевается под оценкой алгоритмов 252
Асимптотическое обозначение 253
Разумное оценивание алгоритмов 255
Быстродействие алгоритма на практике 256
Тема 40. Рефакторинг 259
Когда следует выполнять рефакторинг 261
Порядок выполнения рефакторинга 263
Тема 41. Тестировать, чтобы кодировать 265
Обдумывание тестов 265
Кодирование на основе тестов 266
Применяя разработку, на основе тестирования, нужно знать, куда идти 268
Возврат к коду 270
Модульное тестирование 271
Тестирование соответствия контракту 271
Специальное тестирование 273
Создание тестового окна 273
Культура тестирования 274

Тема 42. Тестирование на основе свойств 276
Контракты, инварианты и свойства 276
Генерация тестовых данных 277
Выявление неудачных допущений 278
Тесты на основе свойств способны удивлять 281
Тесты на основе свойств помогают проектировать 282
Тема 43. Будьте осторожны 283
Другие 90% 283
Основные принципы защиты 284
Здравый смысл и криптография 288
Тема 44. Именование 291
Уважение к культуре 293
Согласованность 294
Переименовывать еще труднее 295

Глава 8. До начала проекта 297

Тема 45. Западня требований 298
Миф о требованиях 298
Программирование как терапия 299
Требования — это процесс 300
Поставьте себя на место клиента 301
Требования и правила 302
Требования и реальность 303
Документирование требований 303
Излишне подробная спецификация 305
“Еще одну мятную вафельную пластинку...” 305
Ведение словаря терминов проекта 305
Тема 46. Решение неразрешимых головоломок 307
Степени свободы 308
Идите своим путем! 309
Судьба благоволит подготовленному уму 310
Тема 47. Совместная работа 311
Парное программирование 312
Групповое программирование 313
Что следует делать? 313
Тема 48. Сущность гибкости 315
Гибкий процесс вообще невозможен 316
Что же тогда делать? 317
И это движет проект 318

Глава 9. Прагматичные проекты 319
Тема 49. Прагматичные команды 320
Никаких разбитых окон 321
Сваренные лягушки 321
Планирование пополнения багажа знаний 322
Внешнее общение команды 323
Не повторяйтесь 323
Трассирующие пули в команде 324
Автоматизация 325
Знайте, когда остановиться 325
Тема 50. Кокосами не обойтись 326
Все дело в контексте 327
Один и тот же подход годится не всем 328
Главная цель 329
Тема 51. Начальный набор инструментальных средств программиста-прагматика 331
Ведение проекта путем контроля версий 332
Строгое и непрерывное тестирование 332
Затягивание сетки 336
Полная автоматизация 337
Тема 52. Доставляйте удовольствие своим пользователям 338
Тема 53. Гордость и предубеждение 340

Приложение А. Послесловие 343
Нравственный ориентир 344
Представляйте будущее таким, каким вы хотите его видеть 345

Приложение Б. Библиография 347

Приложение В. Возможные ответы на упражнения 349

Предметный указатель 364

Отзывы о втором издании книги

"По мнению некоторых, в своей книге Энди и Дэйв поймали молнию в бутылку; вряд ли кому-нибудь удастся в ближайшее время написать книгу, способную взбудоражить целую отрасль так же, как и эта. Впрочем, молния иногда поражает одно и то же место дважды, и эта книга служит явным тому доказательством. Ее обновленное содержимое обеспечит ей неизменное присутствие в списке самых лучших книг по разработке программного обеспечения в течение последующих 20 лет, т.е. там, где ей и надлежит быть".

ВМ (Вики) Брассер (VM (Vicky) Brasseur),
директор программы Open Source Strategy
(Стратегия открытого исходного кода)
в компании Juniper Networks

"Если вы стремитесь сделать свое программное обеспечение легко модернизируемым и сопровождаемым, держите под рукой эту книгу. В ней вы найдете немало практических советов как технического, так и профессионального характера, которые еще многие годы сослужат вам верную службу в ваших проектах".

Андреа Гуле (Andrea Goulet),
генеральный директор компании Corgibytes;
учредитель компании LegacyCode.Rocks.

"Это одна из тех книг, которые вывели меня из прежней колеи и направили на путь успешной профессиональной деятельности в области разработки программного обеспечения. Читая ее, я открыл для себя возможности стать искусным мастером своего дела, а не просто шестеренкой в большом механизме. Это одна из самых знаменательных книг в моей жизни".

Оби Фернандес (Obie Fernandez),
автор книги ***The Rails Way****.*

"Для новых читателей эта книга может послужить увлекательным введением в мир современных практик разработки программного обеспечения, в становлении которого не последнюю роль сыграло первое издание книги. Те, кто уже читал первое издание книги, вновь откроют в ней поучительные выводы и мудрые практические советы, которыми она отличается в первую очередь. Материал настоящего издания тщательно отобран и обновлен".

Дэвид А. Блэк (David A. Black),
автор книги ***The Well-Grounded Rubyist****.*

"Я бережно храню на своей книжной полке старый экземпляр первого издания этой книги. Мне не раз приходилось перечитывать ее, чтобы изменить свое отношение к занятию программированием. В новом издании изменилось все и ничего: сейчас я читаю книгу на своем планшете iPad, разбирая примеры исходного кода на современных языках программирования, но при этом основополагающие принципы, понятия и подходы остаются по-прежнему актуальными и повсеместно применимыми. Спустя двадцать лет эта книга остается востребованной, как никогда прежде. И мне радостно осознавать, что нынешние и будущие разработчики смогут многое почерпнуть из кладезя знаний Энди и Дэвида, как это некогда удалось и мне".

Сэнди Мамоли (Sandy Mamoli),
наставник по гибкой разработке программного обеспечения,
автор книги ***How Self-Selection Lets People Excel.***

"Первое издание этой книги, вышедшее двадцать лет назад, коренным образом изменило направление моей профессиональной деятельности. Новое издание может сделать то же самое и в вашей жизни".

Майк Кон (Mike Cohn),
автор книги ***Succeeding with Agile, Agile Estimating and Planning, and User Stories Applied.***

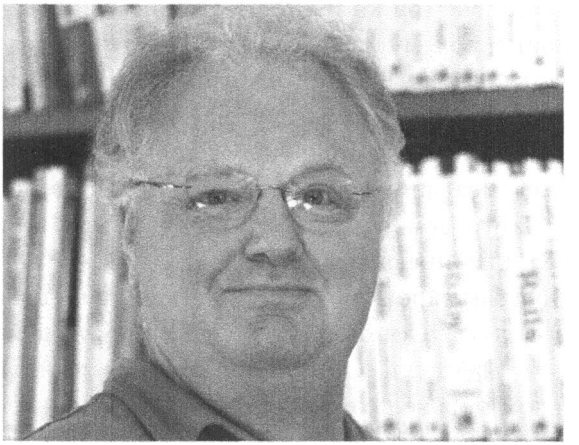
Посвящения

Посвящается Джельет и Элли, Захари и Элизабет, Генри и Стюарту.

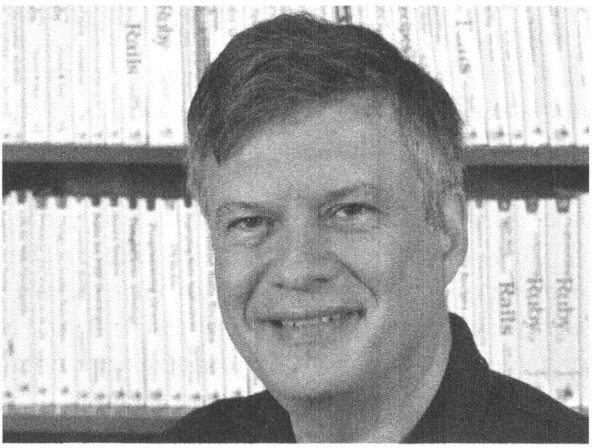
Об авторах

Дэйв Томас и **Энди Хант** являются признанными на международном уровне ведущими авторитетами в сообществе разработчиков программного обеспечения. Они консультируют и выступают с докладами по всему миру. Совместно они основали издательство Pragmatic Bookshelf, выпускающее завоевывающие награды передовые книги для разработчиков программного обеспечения и являются авторами Манифеста гибкой разработки (Agile Manifesto).

В настоящее время Дэйв преподает в колледже, увлекается резьбой по дереву и экспериментирует с новыми технологиями и парадигмами программирования. Энди пишет научно-фантастические повести, активно музицирует и любит повозиться с техникой. Но больше всего оба они стремятся постоянно учиться.



pragdave.me



toolshed.com

Предисловие

Помню, как Дэйв и Энди впервые сообщили в Tweeter о новом издании этой книги. И это было важное известие. Я наблюдал, с каким воодушевлением сообщество программистов встретило это известие, и оживленно откликнулся на него, предвидя, что двадцать лет спустя настоящее издание окажется столь же востребованным, как и прежде.

И то, что книга с подобной историей получила такие отклики, говорит о многом. Мне выпала честь прочитать рукопись этой книги, чтобы написать настоящее предисловие к ней, и тогда я понял, почему она вызвала такой переполох. И хотя это техническая литература, назвав ее таковой, можно сослужить ей плохую службу. Ведь техническая литература нередко отпугивает своим содержимым, изобилующим сложными описаниями, непонятными терминами, замысловатыми примерами, невольно вызывающими у читателя ощущение собственной тупости. Чем опытнее автор книги, тем легче читателю забыть, что при изучении новых концепций он является начинающим.

Несмотря на многолетний опыт программирования, Дэйву и Энди пришлось преодолеть немалые трудности, чтобы писать книгу с воодушевлением людей, только что освоивших уроки данной профессии. Они не обращаются с читателем свысока, не считают его знатоком и даже не предполагают, что он читал первое издание книги. Они воспринимают читателей такими, какими они есть на самом деле, — программистами, которые стремятся совершенствоваться. И чтобы помочь читателям в этом, они посвящают данной цели немало страниц своей книги, делая один практический шаг за другим.

Откровенно говоря, они уже делали это и прежде. Ведь первое издание книги изобиловало реальными примерами, новыми идеями и практическими советами, которые и ныне актуальны для приобретения навыков программирования и выработки умения мыслить как программист. И все же в настоящее обновленное издание были внесены два важных усовершенствования.

Первое усовершенствование вполне очевидно и состоит в исключении устаревших ссылок и примеров, которые были заменены новым, современным содержимым. В настоящем издании вы не найдете примеры инвариантов циклов или сборочных машин. Не обращаясь к прежним примерам, Дэйв и Энди постарались составить материал таким образом, чтобы читатели смогли извлечь из него наибольшую пользу, прорабатывая отдельные уроки. В настоящем издании такие старые принципы, как “не повторяться” (don't repeat yourself, DRY), очи-

щены он накопившейся пыли и отреставрированы, и сверкают новыми яркими красками.

Но по-настоящему привлекательным настоящее издание делает именно второе усовершенствование. После написания первого издания книги у ее авторов была возможность поразмыслить над тем, что они пытались в ней сказать, что ее читатели должны были вынести из нее и воспринять ее так, как хотелось бы авторам. Они получили отзывы на первое издание, выяснили, что было не так, что требовало уточнения, а что было неверно понято. За двадцать лет книга дошла до рук и сердец программистов во всем мире, а Дэйв и Энди, изучив отзывы о ней, сформулировали новые принципы и понятия.

Они усвоили важность поддержки и осознали, что поддержка, бесспорно, присуща разработчикам программного обеспечения в большей степени, чем представителям большинства других профессий. Поэтому они начали настоящее издание с простого, но мудрого посыла: "это ваша жизнь". Он напоминает нам о нашем потенциале в развитии созданной нами кодовой базы, нашей работе и карьере. Он задает тон всей остальной части книги, которая представляет собой нечто большее, чем очередная техническая литература, изобилующая примерами исходного кода.

Эта книга выгодно отличается от многочисленной технической литературы на данную тему ясным пониманием ее авторов, что в действительности означает быть программистом. Программирование — это попытка облегчить будущее, а следовательно, труд коллег. Это умение быстро оправиться после совершенных оплошностей, воспитание хороших привычек и овладение арсеналом инструментальных средств. Но само программирование является лишь частью мира программиста, который исследуется в этой книге.

Мне пришлось потратить немало времени, размышляя о жизненном пути в программировании. Я не вырос на программировании, не изучал его в колледже и не увлекался в юности техникой. Я приобщился к области программирования к 25 годам, когда мне пришлось самому разбираться, что означает быть программистом. Сообщество программистов заметно отличается от других сообществ, в которые мне довелось входить. В этом сообществе заметна особая преданность обучению и практичности, которая не только оживляет, но и отпугивает.

Для меня вхождение в такое сообщество сродни открытию совершенно нового мира или, по крайней мере, нового города, где мне нужно познакомиться с соседями, выбрать свой продуктовый магазин, найти лучшую из кофеен. Потребуется немало времени, чтобы сориентироваться на местности, найти наиболее эффективные маршруты, избегать улиц с интенсивным движением, знать, когда наиболее вероятны транспортные пробки. Погодные условия также могут быть иными, чем в старом месте жительства, а следовательно, придется сменить свой гардероб.

Первые несколько недель, а то и месяцев пребывания в новом городе могут оказаться ужасными. И в этот период было бы замечательно иметь дружелюбного знающего соседа, давно живущего в городе. Кто может провести для вас экскурсию по кофейням? Только тот, кто хорошо знает местную культуру, чувствует пульс жизни города и поможет вам не только почувствовать себя как дома, но и стать полезным членом сообщества. И такими соседями для вас, читатели, окажутся Дэйв и Энди.

У относительно начинающего может легко вызвать потрясение не столько сам акт программирования, сколько процесс его становления как программиста. Ведь для этого потребуется определенный сдвиг в образе мышления, изменение привычек, видов поведения и ожиданий. Процесс вашего становления лучшим программистом не произойдет лишь потому, что вы знаете, как программировать. Ему должно удовлетворять твердое намерение и тщательно продуманная практика. И эта книга служит руководством по эффективному становлению лучшим программистом.

Но имейте в виду, что в книге ничего не говорится о том, как следует программировать. Никаких философских рассуждений и обоснований по этому поводу вы в ней не найдете. Авторы просто и ясно поясняют, как стать программистом-прагматиком и выбрать свой подход к программированию, оставляя за читателями право самим решать, желают ли они стать именно такими программистами. Если вы считаете, что быть программистом-прагматиком — не для вас, то авторы ничего против этого не будут иметь. Но если вы все же решите им стать, они станут вашими дружелюбными соседями, показывающими путь к желанной цели.

Сарон Йитбарек (Saron Yitbarek),
учредитель и генеральный директор компании
CodeNewbie Host of Command Line Heroes.

Вступление ко второму изданию

В далекие 1990-е годы мы работали с компаниями, испытывающими затруднения в своих проектах. И оказалось, что мы каждый раз спрашивали у них одно и то же: "Может, вам лучше сначала тестировать свои программные продукты, а затем поставлять их? Почему код создается только на машинах разработчиков и никто не спрашивает о нем мнения пользователей?"

Чтобы сэкономить время новых клиентов, мы стали делать заметки, которые в конечном счете превратились в эту книгу. И к нашему удивлению, книга, по-видимому, нашла живой отклик у читателей, не потеряв свою популярность в течение прошедших двадцати лет.

Но двадцать лет — немалый срок для разработки программного обеспечения. Если попытаться переместить разработчика во времени из 1999 года и ввести его в современную команду, он будет бороться с чуждым ему новым миром. Но ведь и мир 1990-х годов в равной степени чужд современному разработчику. Ссылки в настоящем издании на литературу по CORBA, CASE, индексированным циклам были бы в лучшем случае забавны и, вероятнее всего, сбивали бы с толку.

В то же время прошедшие двадцать лет не оказали никакого влияния на то, что называется общими принципами. Ведь могла измениться технология, но не люди. Некогда пригодные нормы практики и методы остаются таковыми и поныне. Эти вопросы, рассматривавшиеся ранее в книге, теперь вполне созрели.

И когда настало время выпустить настоящее 20-летнее юбилейное издание, нам пришлось принять нелегкое решение. С одной стороны, мы могли бы просмотреть и обновить описание тех технологий, на которые мы ссылались в предыдущем издании, и на этом поставить точку. А с другой стороны, мы могли бы пересмотреть свои рекомендации по поводу тех норм практики, которые мы рекомендовали раньше, исходя из опыта, накопившегося за прошедшие двадцать лет. Но в конечном счете мы сделали и то и другое.

В итоге книга стала чем-то подобным *кораблю Тесея*[1]. Около одной трети тем в ней оказались совершенно новыми, а остальные были частично или полностью переписаны. Мы намеревались сделать изложение материала более понятным, уместным и, как можно надеяться, непреходящим.

[1] Известный философский парадокс: если с годами заменить каждую вышедшую из строя часть корабля, то останется ли он в итоге тем же самым кораблем?

Нам пришлось принять ряд трудных решений. В частности, мы опустили приложение "Первоисточники" как потому, что было бы невозможно сохранить его актуальность, так и потому, что нужные первоисточники читателю проще найти самостоятельно. Кроме того, мы реорганизовали и переписали темы, касающиеся параллелизма, принимая во внимание современное изобилие аппаратного обеспечения для параллельных вычислений и недостаток способов их организации. Наконец, мы добавили материал, отражающий перемены в отношениях и средах: от движения за гибкую разработку, началу которого мы способствовали, до растущего принятия идиом функционального программирования и потребности принимать во внимание конфиденциальность и безопасность.

Любопытно, однако, что споров по поводу содержимого настоящего издания у нас было меньше, чем при написании первого издания этой книги. Мы оба считали, что выявить важный материал на этот раз было проще.

Так или иначе, настоящее издание увидело свет, так что пользуйтесь им во благо, возможно, приняв новые нормы практики, а может быть, решив, что некоторая часть предлагаемого нами материала вам не подходит. Но в любом случае мы надеемся, что вы глубже вникнете в свое ремесло и откликнетесь на нашу книгу. А самое главное, не забудьте позабавиться.

Структура книги

Эта книга написана как собрание кратких тем, каждая из которых самодостаточна и посвящена конкретному предмету. В тексте вы найдете многочисленные перекрестные ссылки, помогающие ввести каждую тему в соответствующий контекст. Отдельные темы можно читать в произвольном порядке, поскольку эта книга составлена таким образом, чтобы не читать ее от корки до корки.

Периодически в книге вам будут встречаться врезки под заголовком "Совет *nn*" (например, "Совет 1. Заботьтесь о своем ремесле"). Мы рассматриваем подобные советы как обращающие на себя внимание места в тексте книги. Они имеют самостоятельное значение и служат для того, чтобы пользоваться ими ежедневно в практической деятельности.

В настоящем издании были включены упражнения и задачи там, где это было уместно. Как правило, упражнения имеют относительно простые решения, тогда как задачи допускают многие решения. Чтобы дать представление о ходе своих рассуждений, мы выделили решения упражнений в отдельное приложение к книге, хотя лишь немногие из них имеют единственное *правильное* решение. А предлагаемые задачи могут послужить основанием для групповых обсуждений или самостоятельных работ на специализированных курсах по программированию. Кроме того, в конце книги приведен перечень литературы (книг и статей), на которую делаются явные ссылки в тексте самой книги.

Что означает имя

"— Когда я беру слово, оно означает то, что я хочу, не больше и не меньше, — сказал Шалтай-Болтай презрительно"[2].

Льюис Кэрролл, ***Алиса в Зазеркалье***

Текст книги испещрен различными профессиональными терминами: как обычными словами, несколько искаженными, чтобы выразить какой-то особый технический смысл, так и жуткими словами, намеренно выбранными для обозначения отдельных понятий из вычислительной техники теми специалистами в данной области, которые пренебрежительно относятся к языку. При первом употреблении термина делается попытка определить понятие, которое он обозначает, или хотя бы намекнуть на его смысловое значение. Но, на наш взгляд, одни термины так и остались невостребованными, тогда как другие (например, *объект* и *реляционная база данных*) — настолько распространены, что не требуют дополнительного определения. Если же вам встретится незнакомый прежде термин, не пропускайте его, а найдите время, чтобы выяснить его обозначение в Интернете или справочнике по вычислительной технике. И если ваши поиски завершатся удачно, то сообщите нам по электронной почте, чтобы мы добавили найденное вами определение термина в последующее издание книги.

Несмотря на все сказанное выше, мы все же решили отомстить специалистам по вычислительной технике. В частности, мы решили пренебречь некоторыми вполне пригодными терминами для обозначения отдельных понятий. Почему? А потому, что существующая ныне терминология, как правило, ограничивается конкретной предметной областью или стадией разработки. Но ведь один из основных принципов данной книги состоит в том, что большая часть рекомендуемых нами методик оказывается универсальной. Так, модульность применима к исходному коду, проектным решениям, документации и организации команд разработчиков. Когда же требовалось употребить обычный термин в более широком контексте, то возникало недоразумение, поскольку нам не удавалось преодолеть границы той смысловой нагрузки, которую нес первоначальный термин. И когда происходило нечто подобное, мы решали пренебречь языковыми нормами, изобретая свои термины.

[2] Перевод Н.М. Демуровой.

Исходный код и другие ресурсы

Большая часть исходного кода, приведенного в этой книге, взята из компилируемых исходных файлов и свободно доступна для загрузки с нашего веб-сайта[3]. Там приведены также ссылки на другие полезные ресурсы наряду с обновлениями книги и новостями о других разработках программистов-прагматиков.

Присылайте нам свои отзывы

Мы будем рады вашим откликам на книгу. Пишите нам по адресу электронной почты `ppbook@pragprog.com`.

Благодарности за второе издание книги

Мы выгодно воспользовались буквально тысячами интересных бесед о программировании, проведенных за последние двадцать лет, встречая разных людей на конференциях, курсах, а иногда и просто в самолете. Каждая из таких бесед расширила наше представление о процессе разработки программного обеспечения и способствовала обновлению настоящего издания. Благодарим всех участников этих бесед и просим известить нас, если мы в чем-то не правы.

Благодарим также участников процесса создания бета-версии этой книги. Ваши вопросы и комментарии помогли нам лучше разъяснить отдельные вопросы, затронутые в книге.

Прежде чем перейти к бета-версии этой книги, мы поделились ее рукописью с несколькими людьми, чтобы они оставили свои комментарии к ней. Благодарим ВМ (Вики) Брассера (VM (Vicky) Brasseur), Джеффа Лангра (Jeff Langr) и Кима Шриера (Kim Shrier) за подробные комментарии, а Хозе Валима (Jose Valim) и Ника Катберта (Nick Cuthbert) — за техническое рецензирование.

Благодарим Рона Джеффриса (Ron Jeffries) за предоставленную нам возможность воспользоваться примером из судоку. Выражаем большую признательность сотрудникам издательства Pearson, согласившихся на то, чтобы мы составили эту книгу по-своему. Особая благодарность выражается незаменимой Джанет Ферлоу (Janet Furlow), которая мастерски справляется со всем, за что бы она ни бралась. В частности, она следила за тем, чтобы мы поэтапно выполняли свою работу над книгой в срок.

Наконец, выражаем во всеуслышание слова искренней признательности всем программистам-прагматикам, сделавшим программирование лучше для всех за последние двадцать лет. Теперь остается еще двадцать лет.

[3] См. по адресу `https://pragprog.com/titles/tpp20/source_code`.

Из вступления к первому изданию

Эта книга поможет вам стать более совершенным программистом. Вы можете быть самостоятельным разработчиком, членом команды в крупном проекте или консультантом, одновременно работающим со многими компаниями. Так или иначе, эта книга поможет вам в индивидуальном плане лучше выполнять свою работу. Она не носит теоретический характер, а сосредоточена на практических вопросах и применении накопленного опыта для принятия более обоснованных решений. Термин *прагматик* происходит от латинского слова *pragmaticus*, обозначающего "опытный в деле" и производного, в свою очередь, от греческого слова πραγματικός, обозначающего "пригодный для использования". Это книга о делании.

Программирование — это ремесло. В простейшем случае оно сводится к тому, чтобы заставить компьютер сделать то, что требуется программисту или пользователю его программы. Программист выступает отчасти в роли слушателя, советчика, переводчика и диктатора. Он пытается уяснить туманные исходные требования и найти способ выразить их таким образом, чтобы вычислительная машина смогла воздать им должное. Он старается задокументировать и организовать результаты своих трудов таким образом, чтобы другие могли их понять и опереться на них. Более того, программист старается сделать все это, несмотря на неумолимые сроки выполнения проекта. Каждый день программист творит небольшие чудеса, а это трудное дело.

Многие люди готовы оказать помощь программисту. Так, поставщики инструментальных средств на все лады расхваливают чудеса, которые способны творить их программные продукты. Знатоки методологии обещают, что их методики гарантированно принесут ожидаемые результаты. И каждый заявляет, что его язык программирования самый лучший, а операционная система дает ответы на все мыслимые вопросы.

Все это, конечно, не соответствует действительности, ведь простых ответов не бывает. Не существует *самого лучшего* решения, будь то инструментальное средство, язык программирования или операционная система. А могут быть лишь такие системы, которые более пригодны в конкретных обстоятельствах.

Именно здесь и может пригодиться прагматизм, который означает, что не следует привязываться к какой-нибудь конкретной технологии, но нужно иметь достаточно обширные знания и опыт, позволяющие принимать правильные решения в конкретных обстоятельствах. Знания проистекают из понимания основных принципов вычислительной техники, а опыт — из обширного ряда практических проектов. Теория в сочетании с практикой составляют прочный фундамент для программиста.

Свой подход программист приспосабливает к текущим обстоятельствам и окружению. Он оценивает относительную важность всех факторов, оказывающих влияние на проект и пользуется своим опытом для выработки подходящих решений. И делает он это постоянно по мере продвижения своей работы. Программисты-прагматики доводят свою работу до конца и делают ее хорошо.

Кому адресована эта книга

Она адресована тем, кто стремится стать более эффективным и продуктивным программистом. Вы, вероятно, испытываете разочарование от того, что не раскрываете, как вам кажется, свой истинный потенциал. А может быть, вы обращаете внимание на своих коллег, которые пользуются инструментальными средствами с целью повысить производительность своего труда. Возможно, в своей нынешней работе вы пользуетесь старыми технологиями и хотите узнать, как приложить новые идеи к тому, что вы делаете.

Мы не претендуем ни на то, чтобы ответить на все (или хотя бы большинство) вопросы, возникающие у программистов, ни на применимость наших идей во всех возможных случаях. Мы можем лишь сказать, что если вы выберете наш подход, то быстро приобретете необходимый опыт, производительность вашего труда возрастет и вы станете лучше понимать весь процесс разработки программного обеспечения. И в конечном счете вы сможете писать более качественные программы.

Что означает быть программистом-прагматиком

Каждый разработчик неповторим, отличаясь своими сильными и слабыми сторонами, симпатиями и антипатиями. Со временем каждый разработчик создает свою собственную среду, отражающую его индивидуальность в той же неумолимой степени, как и его увлечения, одежда или прическа. Но если вы являетесь программистом-прагматиком, то вам присущи многие из перечисленных ниже общих характеристик.

- **Своевременно принимаете новшества на вооружение и быстро приспосабливаетесь к ним.** Инстинктивно чувствуете преимущества отдельных технологий и методик, и вам нравится опробовать их. Если вам дают что-то новое, вы быстро осваиваете его, присоединяя приобретенное к остальным своим знаниям. Ваша уверенность рождается с опытом.
- **Любознательны.** Стремитесь задавать вопросы. Например: как вы это сделали? Возникли ли у вас трудности с данной библиотекой? Что такое квантовые вычисления, о которых я слышал? Каким образом реализуются символические ссылки? Задавая подобные вопросы, вы как барахольщик собираете мелкие факты, каждый из которых может оказать влияние на какое-нибудь решение многие годы спустя.

- **Мыслите критически.** Редко принимаете что-то на веру, не получив сначала подтверждающие факты. Так, если коллеги говорят: "Потому что надо делать так!" или если поставщик обещает решить все ваши проблемы, вы нюхом чуете предстоящие трудности.
- **Реалистичны.** Стараетесь понять потаенный характер каждого возникающего у вас затруднения. Подобный реализм дает вам ясное понимание, в чем именно состоят трудности и как долго их придется преодолевать. Глубоко понимая, что процесс *должен* быть трудным или его завершение *отнимет* время, вы приобретаете необходимую выдержку, чтобы справиться с ним.
- **Мастер на все руки.** Прилежно стараетесь освоить обширный ряд технологий и сред, работая над тем, чтобы быть в курсе новых разработок. И хотя ваша текущая работа может потребовать специальной квалификации, вы всегда готовы перейти в новые сферы деятельности и принять новые вызовы.

Мы оставили рассмотрение большинства основных характеристик напоследок. Они присущи всем программистам-прагматикам и настолько просты, что их можно сформулировать в виде отдельных советов, как показано ниже.

Совет 1 Заботьтесь о своем ремесле

Мы считаем, что разрабатывать новое программное обеспечение нет смысла, если не позаботиться о том, чтобы делать это как следует.

Совет 2 Думайте! О своей работе

Чтобы стать программистом-прагматиком, мы призываем вас думать о том, что вы делаете в тот момент, когда вы это делаете. Это не единовременная ревизия текущих норм практики, а непрерывная критическая оценка каждого принимаемого решения, производимая каждый день и в каждом проекте. Никогда не двигайтесь на автопилоте. Постоянно думайте, критически оценивая свою работу в реальном времени. Старый девиз корпорации IBM "ДУМАЙ!" (THINK!) является мантрой для программиста-прагматика.

Если такая работа покажется вам трудной, значит, вы выказываете *реалистичную* характеристику. На это потребуется немного вашего драгоценного времени, которого, вероятно, и так не хватает. Но в награду за усердие вы более активно вовлекаетесь в свое любимое дело, чувствуя, что вполне владеете неуклонно расширяющимся рядом предметов, а также получаете удовольствие от

постоянного совершенствования. А в долгосрочной перспективе потраченное вами время окупится сторицей, когда вы и ваша команда станете работать более эффективно, чтобы писать легко сопровождаемый код и проводить меньше времени на совещаниях.

Индивидуальные прагматики и крупные команды

Некоторые считают, что индивидуальности нет места в крупных командах или сложных проектах. "Программное обеспечение — техническая дисциплина, — говорят они, — которая нарушается, если отдельные члены команды принимают решения самостоятельно". Мы категорически не согласны с таким утверждением.

В построении программного обеспечения *должна* присутствовать техническая составляющая, но это совсем не мешает проявлять индивидуальное мастерство. Рассмотрим в качестве примера крупные кафедральные соборы, построенные в Европе в период Средневековья. Воздвижение каждого из них потребовало немалых затрат, измеряемых тысячами человеко-часов в течение многих десятилетий. Усвоенные уроки передавались от одного поколения строителей к другому поколению, которое совершенствовало строительную технику своими индивидуальными достижениями. Но ведь плотники, каменотесы, резчики и стеклодувы были ремесленниками, воплощавшими технические требования в нечто целое, выходившее за пределы исключительно механической стороны строительства. *Даже те, кто лишь тешут камни, должны всегда представлять себе целые соборы.*

Во всей структуре проекта всегда найдется место индивидуальности и мастерству. И это особенно справедливо для текущего состояния разработки программного обеспечения. Это состояние может показаться через сто лет таким же архаичным, как и методы, применявшиеся средневековыми строителями кафедральных соборов, современным инженерам-строителям, хотя мастерство в данной отрасли по-прежнему в почете.

Это непрерывный процесс

Турист, однажды посетивший Итонский колледж в Англии, спросил садовника, как ему удается поддерживать лужайку в идеальном состоянии. "Очень просто, — ответил тот. — Нужно лишь смахивать росу каждое утро, стричь траву каждый день и сволакивать ее раз в неделю".

"И это все?" — спросил турист.

"Абсолютно все! — ответил садовник. — Делайте это в течение 500 лет, и у вас тоже получится прекрасная лужайка".

Прекрасные лужайки требуют небольшого ухода, как, впрочем, и отличные программисты. Консультанты по управленческим вопросам любят вставлять словечко *кайзен* в свои беседы. Термин *кайзен* по-японски обозначает понятие непрерывного процесса внесения многих мелких усовершенствований, что послужило одной из главных причин для коренных изменений в производительности и качестве изготовления японских товаров и широко размножившееся по всему миру. Кайзен применяется и на индивидуальном уровне, где отдельные личности ежедневно работают над совершенствованием собственных навыков, пополняя свой арсенал новыми инструментальными средствами. Но, в отличие от итонских лужаек, они начинают замечать результаты буквально через считанные дни. А с годами они изумляются, насколько расцвел их личный опыт, и развились их индивидуальные навыки.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное сообщение, либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

Глава 1

Философия прагматизма

Эта книга о вас. Будьте уверены, что программирование — это *ваша* карьера, а самое главное — *это ваша жизнь*, и она принадлежит вам. И вы приступили к чтению этой книги потому, что знаете, что можете не только сами стать более искусным разработчиком, но и помочь другим разработчикам стать лучше. Следовательно, вы способны стать *программистом-прагматиком*.

Что же отличает программистов-прагматиков? На наш взгляд, это отношение, стиль и философский подход к задачам и их решениям. Их мышление выходит за пределы непосредственной задачи, ставя ее в более крупный контекст, чтобы получить общую картину. Ведь как иначе быть прагматичным без этого более крупного контекста и как в таком случае находить разумные компромиссы и обоснованные решения?

Еще один ключ к успеху состоит в том, что программисты-прагматики берут на себя ответственность за все, что делают, как обсуждается в разделе "Кот съел мой исходный код". Будучи ответственными, программисты-прагматики не сидят без дела, безучастно наблюдая за тем, как их проекты буквально распадаются на части из-за небрежения ими. Поэтому в разделе "Программная энтропия" далее в этой главе поясняется, как поддерживать свои проекты в безупречном состоянии.

Большинство людей находят перемены трудными — иногда по веским основаниям, а иногда просто по укоренившейся инерции. Поэтому в разделе "Суп из камней и вареные лягушки" рассматривается стратегия побуждающих к действию перемен и (ради равновесия) представлена поучительная история об одном земноводном, пренебрегшем опасностями, скрывающимися в постепенных переменах.

Одно из преимуществ, дающих понимание контекста, в котором вы работаете, заключается в том, что вам просто легче выяснить, насколько качественно разработанное вами программное обеспечение. Иногда близкое к идеальному качество оказывается лишь одним из возможных вариантов, но зачастую приходится идти на компромиссы. Более подробно этот вопрос рассматривается в разделе "Подходящее программное обеспечение".

Безусловно, чтобы все это осилить, потребуется обширная база знаний, поэтому обучение — это непрерывно продолжающийся процесс. В связи с этим в разделе "Ваш багаж знаний" описываются некоторые стратегии поддержания динамики данного процесса.

Наконец, никто из нас не работает в вакууме. Все мы проводим немало времени, общаясь с другими людьми. Поэтому в разделе "Общайтесь!" перечислены способы, помогающие делать это лучше.

Прагматичное программирование происходит от философии прагматичного мышления. И в этой главе закладывается основание для такой философии.

Тема 1 Это ваша жизнь

"Я в этом мире не для того, чтобы жить по вашим ожиданиям, а вы — не для того, чтобы жить по моим".

Брюс Ли[4]

Это *ваша* жизнь, и она принадлежит вам. Вы ее ведете и творите.

Многие разработчики жалуются нам на свою разочарованность. Их проблемы различны. Одни ощущают застой в своей работе, а другие считают, что технический прогресс обошел их стороной. Они полагают, что их недооценивают, им недоплачивают или в их командах отравляющая атмосфера. Возможно, они хотят переехать в Азию или в Европу, а может быть работать дома. И в ответ на все эти жалобы мы всегда задаем один и тот же вопрос: "Почему вы не можете это изменить?"

Разработка программного обеспечения должна находиться близко к началу любого доступного списка требующихся специальностей. Навыки разработчиков востребованы, их знания распространяются за пределы географических границ, они работают в удаленном режиме, получая за свой труд хорошую плату. И вообще, они могут делать почти все, что хотят.

Но по какой-то причине разработчики противятся переменам. Они выжидают, надеясь, что положение дел переменится к лучшему. Они пассивно наблюдают за тем, как их навыки устаревают, жалуясь на то, что их компании ничему их не обучают. Они рассматривают рекламные объявления экзотических стран в городском транспорте, а затем выходят под холодный дождь и бредут на работу. Поэтому ниже приведен самый важный совет в этой книге.

Совет 3 У вас есть свобода выбора

[4] Bruce Lee — гонконгский и американский киноактер, режиссер, сценарист, продюсер, популяризатор и реформатор в области китайских боевых искусств. — *Примеч. пер.*

Вас заедает среда? Вам надоела работа? Попробуйте исправить положение, но не затягивайте свою попытку до бесконечности. По этому поводу Мартин Фаулер говорит следующее: "Вы можете сменить организацию или изменить свою организацию"[5].

Если вам кажется, что технический процесс обходит вас стороной, наверстайте упущенное, изучая в свое личное время то, что вам кажется интересным. Тем самым вы инвестируете в самих себя — так что заниматься самообразованием в свободное от работы время вполне благоразумно.

Желаете работать в удаленном режиме? Поинтересуйтесь в одном месте. Если вам откажут, поищите другое место, где вас возьмут на работу. Данная отрасль дает вам немало замечательных возможностей, поэтому проявите заранее инициативу, чтобы воспользоваться ими.

Другие разделы, связанные с данной темой

- **Тема 4.** Суп из камней и вареные лягушки.
- **Тема 6.** Ваш багаж знаний.

Тема 2 Кот съел мой исходный код

"Величайшей из всех слабостей есть страх показаться слабым".

Ж.Б. Боссюэ[6],
Политика из Святого Писания
(Politics from Holy Writ), 1709 г.

Одним из краеугольных камней прагматичной философии является идея взять на себя ответственность за свои действия в отношении продвижения по карьерной лестнице, обучения и образования, выполнения своего проекта и повседневной работы. Программисты-прагматики сами заботятся о своей карьере и не боятся признавать свое невежество или ошибки. Это, конечно, не самая приятная сторона программирования, но она все же проявляется даже в самых удачных проектах. Порой дело не ладится, несмотря на проведенное тестирование, грамотно составленную документацию и сплошную автоматизацию. И тогда выпуск программного продукта задерживается в связи с возникшими непредвиденными техническими трудностями.

Если происходит нечто подобное, мы стараемся обращаться с командой как можно более профессионально. А это означает быть честными и откровенны-

[5] См. по адресу `http://wiki.c2.com/?ChangeYourOrganization`.

[6] J.B. Bossuet — французский проповедник и богослов XVII века, а также писатель и епископ. — *Примеч. пер.*

ми. Мы можем гордиться своими способностями, но должны признавать и свои недостатки: невежество и ошибки.

Доверие в команде

Прежде всего, ваша команда должна доверять вам и полагаться на вас, а вы — спокойно полагаться на каждого из ее членов. Доверие в команде совершенно необходимо для проявления творческой инициативы и сотрудничества, как следует из литературы, в которой исследуется данный вопрос[7]. В здоровом окружении, основанном на доверии, вы можете смело высказывать свои мысли, представлять свои идеи на суд коллег и полагаться на других членов команды так же, как и они на вас. Ведь без доверия...

Представьте, что высокотехнологичная команда ниндзя незаметно проникает в самое логово преступников. После нескольких месяцев тщательного планирования и изнуряющих тренировок вы, наконец-то, добрались до места. И теперь, когда настал ваш черед настроить систему лазерного наведения, вы неожиданно для остальных заявляете: "Извините, ребята, у меня нет с собой лазера. Мой кот играл с красной точкой лазерного целеуказателя, и поэтому я оставил его дома". Нарушенное подобным образом доверие вряд ли удастся восстановить.

Взятие на себя ответственности

Ответственность — это нечто такое, на что вы активно соглашаетесь. В частности, вы берете на себя обязательство обеспечить правильность выполнения какой-то работы, но совсем не обязательно имеете непосредственный контроль над каждой стадией данного процесса. Помимо того, что вы стараетесь выполнить свою работу как можно лучше, вам приходится также анализировать ситуацию на наличие неподконтрольных вам рисков. Вы имеете полное право не брать на себя ответственность за непредсказуемую ситуацию или такую, когда риски слишком велики или этические последствия слишком постыдны. И тогда вам придется принимать решение, исходя из ваших собственных ценностей и суждений.

Если вы все же *берете* на себя ответственность за конечный результат, то должны учитывать, что вам придется нести ответственность за него. И если вы допустите оплошность, как это делаем все мы, или совершите ошибку в суждении, то признайте ее честно и постарайтесь предложить другие варианты выхода из затруднительной ситуации.

Не сваливайте вину на кого-то или что-то другое и не ищите оправданий. Не вините во всем трудности у поставщика, язык программирования, руководство

[7] Наглядный пример мета-анализа см. в статье *Trust and team performance: A meta-analysis of main effects, moderators, and covariates* (Доверие и производительность команды. Мета-анализ основных эффектов, посредников и ковариаций), доступной по адресу `https://psycnet.apa.org/doiLanding?doi=10.1037%2Fapl0000110`.

или коллег. Любой из этих факторов может сыграть свою отрицательную роль, но именно *вы* должны найти правильные решения, а не оправдания.

На случай, если существует риск, что поставщик вас подведет, у вас должен быть план экстренных мероприятий. Если же выйдет из строя устройство памяти, а вместе с ним исчезнет весь исходный код и у вас не окажется его резервной копии, — это будет вашей оплошностью. Оправдание перед начальством вроде "кот съел мой исходный код" просто не пройдет.

Совет 4 Предлагайте варианты разрешения затруднений, а не оправдания и извинения

Прежде чем подходить к кому-нибудь и объяснять причины, по которым что-то нельзя сделать, что-то запаздывает или не работает, остановитесь и послушайте себя. Обратитесь к резиновому утенку или котенку на своем мониторе, чтобы опробовать на нем, звучат ли ваши оправдания благоразумно или нелепо и как они будут звучать перед вашим начальством.

Воспроизведите весь разговор в уме и подумайте, что, вероятнее всего, ответит другой его участник. Спросит ли он вас, пробовали ли вы сделать вот это или не учли ли вы вот то? Как вы на это ответите? Прежде чем идти к нему, чтобы сообщить плохие новости, подумайте, можно ли опробовать что-нибудь еще? Иногда вы просто *знаете*, что он вам ответит, так что избавьте его от лишних хлопот.

Вместо оправданий предлагайте свои варианты разрешения затруднений. Не говорите, что чего-то нельзя сделать. Лучше поясните, что *можно* сделать, чтобы выйти из трудного положения. Некоторый исходный код должен быть удален? Скажите об этом и объясните важность этого рефакторинга (см. раздел "Тема 40. Рефакторинг" главы 7 "По ходу кодирования").

Требуется ли вам время на прототипирование, чтобы найти наилучший путь продолжить разработку (см. раздел "Тема 13. Прототипы и памятные записки" главы 2 "Прагматичный подход")? Требуется ли вам внедрить более совершенное тестирование (см. разделы "Тема 41. Тестировать, чтобы кодировать" главы 7 "По ходу кодирования" и "Строгое и непрерывное тестирование" главы 9 "Прагматичные проекты") или его автоматизация, чтобы не повторять тестирование снова вручную?

Вам могут понадобиться дополнительные ресурсы для окончательного решения задачи тестирования, а возможно, придется уделить больше времени взаимодействию с пользователями. Или же все дело просто в вас самих, а значит, вам требуется более углубленно изучить какую-нибудь методику или технологию. Помогут ли вам в этом какие-нибудь книги или курсы? Не бойтесь задаваться подобными вопросами или признаваться в том, что вам нужна помощь.

Старайтесь избавиться от слабых оправданий, прежде чем произносить их вслух. Если вы все же считаете, что должны оправдаться, сделайте это сначала перед своим котом. И если уж ваш Мурзик готов взять всю вину на себя, то...

Другие разделы, связанные с данной темой

- **Тема 49.** Прагматичные команды, **глава 9** "Прагматичные проекты".

Задачи

- Как вы отреагируете, если кто-нибудь (например, банковский служащий, автомеханик или чиновник) придет к вам со слабым оправданием? Какое у вас составится в итоге мнение о них и об их организации?
- Если вы, так уж вышло, говорите: "Я не знаю", непременно прибавьте: "Но я найду выход". Это отличный способ признать, что вы чего-то не знаете, но все же берете на себя ответственность как профессионал.

Тема 3 Программная энтропия

Несмотря на то что разработка программного обеспечения не подвержена большинству физических законов, все мы сильно подвержены воздействию неуклонного роста энтропии. Физический термин *энтропия* обозначает величину беспорядка в системе. К сожалению, законы термодинамики утверждают, что энтропия во вселенной стремится к максимуму. Если же беспорядок возрастает в программном обеспечении, то такое явление называется деградацией последнего. Некоторые могли бы назвать это явление более оптимистическим термином "технический долг", подразумевающим, что когда-нибудь они погасят этот долг, — хотя этого, скорее всего, не произойдет. Но как бы ни называлось данное явление, техническим долгом или деградацией, оно может распространяться бесконтрольно.

Имеется немало факторов, способствующих деградации программного обеспечения, и самый важный из них, по-видимому, имеет отношение к психологии или культуре в работе над проектом. Психология может оказаться весьма тонким свойством вашего проекта, даже если вы работаете над ним самостоятельно. Какими бы совершенными ни были составленные планы или участники проекта, он все равно может быть подвергнут деградации и разрушению в течение всего срока своего действия. Тем не менее существуют и такие проекты, которые успешно противостоят естественной тенденции к беспорядку и ухитряются завершиться вполне удачно, несмотря на огромные трудности и постоянные задержки и помехи.

В чем же тогда отличие? В старых кварталах городов одни здания красивы и чисты, тогда как другие выглядят как трухлявые развалины. Исследователи в сфере преступности и упадка городов открыли замечательный пусковой механизм, очень быстро превращающий чистое, нетронутое, нежилое здание в разрушенную и заброшенную трущобу[8]. Это разбитое окно.

Оказывается, что если не отремонтировать хотя бы одно разбитое окно в течение какого-нибудь значительного периода времени, у жильцов возникнет ощущение заброшенности здания, а следовательно, отсутствия заботы городских властей о его состоянии. И, как следствие, разбивается еще одно окно, а люди начинают захламлять здание всяким мусором. На стенах такого здания появляются граффити и начинается серьезное разрушение его конструкции. В течение относительно короткого периода времени здание разрушается настолько, что у его владельца не возникает никакого желания отремонтировать его, и тогда ощущение заброшенности становится вполне реальным.

Почему же возникает такое отличие? Как показывают исследования, проведенные психологами[9], безнадежность может оказаться заразной. Так, если рассмотреть вирус гриппа в тесном помещении, то явное пренебрежение эпидемической ситуацией способно лишь утвердить в мысли, что *ничего*, вероятно, нельзя исправить, никого это не волнует и все обречены на заболевание гриппом. Подобным же образом негативные мысли могут распространяться среди членов команды, образуя порочный круг.

Совет 5 Нельзя жить с разбитыми окнами

Совет "Нельзя жить с разбитыми окнами", по существу, означает, что нельзя мириться с неудачными проектными решениями или плохо написанными фрагментами кода, оставляя их неисправленными. Исправляйте их, как только обнаружите. Если же времени на исправление недостаточно, закомментируйте неисправный код, выведите сообщение "Не реализовано" или же подставьте вместо него фиктивные данные подобно тому, как временно заколачивают разбитые окна. Словом, предпримите какое-то *действие*, чтобы предотвратить дальнейший ущерб и тем самым показать, что вы владеете ситуацией.

Нам приходилось видеть, как нормально функционирующие системы быстро приходят в негодность, как только начинают разбиваться окна. Имеются и другие факторы, способствующие деградации программного обеспечения, и мы еще коснемся их в дальнейшем, а до тех пор следует подчеркнуть, что пренебрежение *ускоряет* деградацию быстрее, чем любой другой фактор.

---

[8] См. *The police and neighborhood safety* [WH82].

[9] См. *Contagious depression: Existence, specificity to depressed symptoms, and the role of reassurance seeking* [Joi94].

Можно, конечно, возразить, что ни у кого нет времени на обследование проекта и его полную очистку от разбитого стекла. В таком случае лучше запланировать установку мусорного контейнера или переезд в соседнее здание. Но в любом случае не позволяйте энтропии одержать верх.

Прежде всего — не навредить

У Энди некогда был баснословно богатый знакомый. Его дом был в идеальном порядке, обставленный дорогим антиквариатом, предметами искусства и т.д. Однажды гобелен, висевший слишком близко к камину, загорелся. Пожарные быстро прибыли на место, чтобы потушить пожар и спасти дом от разрушительного огня. Но прежде чем затащить свои длинные и грязные шланги в дом, они остановились, чтобы раскатать мат между входной дверью и очагом пожара, поскольку им не хотелось испортить ковер — в самый разгар пожара.

Возможно, это и весьма крайний пример. Ведь первая обязанность пожарных — погасить огонь, пренебрегая сопутствующим ущербом. Но они трезво оценили ситуацию, будучи уверенными в своей способности справиться с огнем, и действовали аккуратно, чтобы не нанести имуществу лишний ущерб. Именно таким образом следует обращаться и с программным обеспечением, не нанося ему сопутствующий ущерб лишь потому, что оно находится в каком-то критическом состоянии. Ведь и одно разбитое окно — это уж слишком.

Одного разбитого окна (т.е. плохо написанного фрагмента кода, неудачного управленческого решения, с которым команде приходится мириться в течение всего проекта) достаточно, чтобы все начало приходить в упадок. Если окажется, что вы работаете над проектом с большим количеством разбитых окон, вам очень легко может прийти в голову следующая мысль: "Вся остальная часть этого кода написана скверно, и я поступлю точно так же". И совсем не важно, что до этого момента проект находился в отличном состоянии. В первоначальном эксперименте, на основании которого была выдвинута теория "разбитого окна", заброшенный автомобиль стоял нетронутым целую неделю. Но однажды в нем было разбито одно окно, после чего автомобиль был обчищен и перевернут вверх дном в считанные *часы*.

Подобным же образом, присоединившись к проекту, где код безупречен (т.е. написан ясно, изящно и грамотно), вам следует действовать крайне осторожно, чтобы не навредить, как пожарные в упомянутом выше примере. Даже если пожар в самом разгаре (поджимают сроки, приближается дата выпуска, выставочная демонстрация и т.д.), *вы* не должны стать первым, кто испортит все дело или нанесет дополнительный ущерб. Просто скажите себе: "Никаких разбитых окон".

Другие разделы, связанные с данной темой

- **Тема 10.** Ортогональность, **глава 2** "Прагматичный подход".
- **Тема 40.** Рефакторинг, **глава 7** "По ходу кодирования".
- **Тема 44.** Именование, **глава 7** "По ходу кодирования".

Задачи

- Помогите укреплению команды, обследовав окрестности проекта. Выберите два или три "разбитых окна" и обсудите со своими коллегами возникшие трудности и возможные пути их разрешения.
- Можете ли вы предсказать, когда разобьется первое окно? Какова будет ваша реакция? Если это стало следствием чужого решения или приказа начальства, то что вы можете с этим поделать?

Тема 4 Суп из камней и вареные лягушки

Три солдата, возвращавшихся с войны домой, проголодались. Когда они увидели перед собой село, то воспряли духом, поскольку были уверены, что селяне накормят их. Но когда они добрались до села, то обнаружили двери домов запертыми на замок, а окна закрытыми. После многих лет войны селянам не хватало еды, а ту, что у них была, они припрятывали.

Невзирая на постигшую их неудачу, солдаты вскипятили котелок воды, аккуратно положив в него три камня. Пораженные селяне вышли посмотреть на происходящее.

"Это суп из камней", — пояснили солдаты. — "И это все, что вы в него положили?" — спросили селяне. "Абсолютно все, хотя говорят, что он будет лучше на вкус, если положить в него немного моркови..." Один из селян удалился, тотчас вернувшись с корзиной моркови из своих припрятанных запасов.

Пару минут спустя селяне снова спросили: "Это все?". "Пожалуй, — ответили солдаты, — пару картофелин придадут ему больше густоты". И тогда удалился другой селянин.

В течение последующего часа солдаты перечислили другие ингредиенты, которые могли бы сделать суп еще вкуснее: говядину, лук-порей, соль, зелень и специи. И всякий раз какой-нибудь другой селянин удалялся, чтобы принести что-нибудь из своих личных запасов.

В итоге солдаты сварили большой котел горячего, дымящегося супа. Они извлекли из него камни и сели вокруг котла вместе со всем селом, чтобы впервые за многие месяцы насладиться сытным обедом, которого ни одному из них уже давно не доводилось отведывать.

Из этой истории о супе из камней следуют два поучительных вывода. Солдаты хитро воспользовались любопытством селян, чтобы заполучить у них еду. Но еще важнее, что солдаты действовали подобно катализатору, объединив селян

вместе и тем самым показав им, что совместно они могут сотворить нечто такое, чего не удалось бы сделать каждому из них в отдельности. И это называется синергическим эффектом, когда в конечном счете выигрывают все.

Время от времени и вам, может быть, придется подражать этим солдатам. Ведь вы можете оказаться в такой ситуации, когда точно знаете, что и как нужно делать. Перед вашим мысленным взором вся система, и вы знаете, что с ней все в порядке. Но стоит вам попросить разрешение взяться за доводку всего в целом, вы встретите препятствия и недоуменные взгляды. Люди сформируют комитеты, бюджеты потребуют утверждения, и все только усложнится. Каждый будет защищать свои ресурсы, что иногда называется "изначальной апатией".

И тогда наступает время выставить камни. Тщательно обдумайте, что именно вы *можете* обоснованно попросить. Разработайте и покажите это в готовом виде другим, чтобы изумить их. И тогда они скажут: "Конечно, было бы лучше, если бы мы это добавили..." Притворитесь, что это неважно. Не вмешивайтесь до тех пор, пока они не начнут просить вас внедрить функциональное средство, которое вам первоначально требовалось. Ведь людям легче присоединиться к уже достигнутому успеху. Приоткройте им перспективы на будущее, и они сплотятся вокруг вас[10].

Совет 6 Будьте катализатором перемен

Со стороны селян

С другой стороны, история о супе из камней повествует о едва заметном и постепенно употребляемом лукавстве, а также о чрезмерной сосредоточенности на чем-то одном. Ведь думая о камнях, селяне забывают обо всем остальном. Все мы попадаем в подобную западню едва ли не каждый день, когда что-нибудь незаметно подкрадывается к нам.

Нам не раз приходилось наблюдать подобные симптомы, когда проекты медленно, но верно выходили из подчинения. Большинство бедствий в программном обеспечении начинается с едва заметных мелочей, и большинство проектов постепенно выходят за установленные пределы. Системы отклоняются от своих спецификаций одно функциональное средство за другим, тогда как одна "заплата" за другой вставляется во фрагмент кода до тех пор, пока в нем не останется ничего первоначального. Зачастую именно накапливающиеся постепенно мелочи портят нравы и разрушают команды.

[10] Делая это, вам, возможно, будет удобнее придерживаться линии поведения, приписываемой контр-адмиралу д-ру Грейс Хоппер (Grace Hopper), которая говорила: "Легче попросить прощения, чем получить разрешение."

Совет 7 Не забывайте об общей картине

Честно говоря, мы никогда не пробовали этого сами, но знатоки говорят, что если взять лягушку и опустить ее в кипящую воду, она тотчас выпрыгнет из нее наружу. Но если погрузить лягушку в кастрюлю с холодной водой, а затем постепенно нагреть ее, то лягушка не заметит постепенного повышения температуры и сварится в кастрюле.

Следует, однако, иметь в виду, что случай с лягушкой отличается от случая с разбитыми окнами, обсуждавшегося ранее в разделе "Тема 3. Программная энтропия". Согласно теории разбитых окон люди теряют всякую охоту бороться с энтропией, поскольку воспринимают ее как безразличие всех остальных. А лягушка просто не замечает никаких изменений.

Не уподобляйтесь пресловутой лягушке. Внимательно следите за общей картиной происходящего. Постоянно присматривайте за тем, что происходит вокруг вас, а не только за тем, что вы делаете сами.

Другие разделы, связанные с данной темой

- **Тема 1.** Это ваша жизнь.
- **Тема 38.** Программирование по совпадению, **глава 7** "По ходу кодирования".

Задачи

- Рецензируя рукопись первого издания этой книги, Джон Лакос поднял следующий вопрос: солдаты сознательно вводят все больше в заблуждение селян, хотя перемены, которые они ускоряют, идут всем только на пользу. Но если сознательно вводить все больше в заблуждение лягушку, то можно нанести ей вред. Можете ли вы определить, готовите ли вы суп из камней или лягушек, когда пытаетесь ускорить перемены? Является ли такое решение субъективным или объективным?
- Быстро ответьте, не глядя на потолок, сколько осветительных приборов висит над вами? Сколько выходов из помещения, где вы находитесь, и сколько в нем присутствует людей? Есть ли там что-нибудь чуждое или выглядящее неуместным? Это упражнение в *осведомленности об окружающей обстановке* — метод, практикуемый в разных кругах: от бойскаутов до летчиков. Выработайте сначала в себе привычку смотреть и замечать все, что происходит вокруг вас, а затем перенесите ее на свой проект.

Тема 5 Достаточно хорошее программное обеспечение

"Для лучшего добро сгубить легко".

*У. Шекспир, **Король Лир**, действие 1, сцена 4*

Есть старый анекдот об американской компании, заказавшей 100 тысяч микросхем у японского производителя. В спецификации на микросхемы было, в частности, указано, что брак должен составлять 1 штуку на 10 тысяч хороших микросхем. Заказ был доставлен через несколько недель и состоял из одной крупной коробки, содержавшей тысячи микросхем, и другой, мелкой, в которой было всего лишь десять микросхем. К этой коробке была приклеена этикетка, на которой было написано "Бракованные микросхемы".

Хорошо, если бы и у нас контроль качества был на таком же уровне. Но реальность не позволяет нам производить ничего совершенно идеального, и особенно бездефектного программного обеспечения. Ведь время, технология и темперамент словно сговорились против этого.

Но это и не повод для разочарования. Как описал Эд Юрдон в своей статье *When good-enough software is best* (Когда достаточно хорошее программное обеспечение оказывается наилучшим) в журнале *IEEE Software*, [You95], можно приучить себя писать достаточно хорошие программы, которые достаточно хороши для пользователей, будущих сопровождающих их программистов и для спокойствия вашего духа. В этом случае вы работаете более продуктивно, а пользователи ваших программ удовлетворены в большей степени. А кроме того, вы можете обнаружить, что ваши программы только выигрывают от сокращения их инкубационного периода.

Прежде чем продолжить, необходимо пояснить, о чем, собственно, здесь идет речь. Слово "подходящий" совсем не подразумевает небрежно или плохо написанный код. Все системы должны удовлетворять требованиям пользователей, чтобы считаться удачно спроектированными, а также соответствовать основным нормам производительности, конфиденциальности и безопасности. Мы просто ратуем за то, чтобы дать пользователям возможность участвовать в процессе принятия решения о том, когда произведенный программный продукт следует считать достаточно хорошим для удовлетворения их потребностей.

Идите на компромиссы с пользователями

Как правило, вы пишете программы для других людей. И даже иногда не забываете выяснить, что же им, собственно, требуется[1]. Но спрашиваете ли вы их вообще, *насколько хорошее* программное обеспечение они хотят получить? Иногда у них просто нет никакого выбора. Если вы разрабатываете кардиостимулятор, автопилот или низкоуровневую библиотеку, предполагающую широкое распространение, требования будут более строгими, а возможности вашего выбора — ограниченными.

Но если вы работаете над совершенно новым программным продуктом, то на вашу разработку накладываются самые разные ограничения. Так, специалистам по сбыту требуется сдержать свои обещания, конечные пользователи могут составлять планы, исходя из сроков поставки, а ваша компания, как всегда, испытывает дефицит денежных средств. Поэтому было бы непрофессионально пренебрегать требованиями всех этих пользователей, просто дополнив программу новыми функциональными возможностями или "отполировав" исходный код еще раз. Мы ни в коем случае не сеем панику. Ведь в равной степени непрофессионально обещать невозможные временные рамки выполнения работ и нарушать элементарные правила проектирования, чтобы уложиться в крайний срок. Область действия и качество производимой вами системы должны обсуждаться как часть требований к ней.

Совет 8 Включайте в требования к системе вопрос о ее качестве

Нередки случаи, когда приходится идти на разные компромиссы. Как ни странно, но многие пользователи готовы применять не до конца доведенное программное обеспечение *сейчас*, чем ждать еще целый год, пока оно не будет доведено до состояния готовой к выпуску полноценной версии. Хотя на самом деле то, что может понадобиться через год, все равно окажется совершенно другим. И с этим согласятся отделения информационных технологий (ИТ) многих компаний со скромными бюджетами. Хорошее программное обеспечение сегодня зачастую оказывается более предпочтительным, чем совершенное программное обеспечение завтра. Если вы предоставляете что-нибудь пользователям для экспериментирования как можно раньше, их отзывы могут привести к более совершенному окончательному решению (см. раздел "Тема 12. Трассирующие пули" главы 2 "Прагматичный подход").

[1] Это, конечно, шутка!

Знайте меру

В какой-то степени программирование подобно живописи. Вы начинаете с чистого холста и некоторых основных исходных материалов. Используя сочетание науки, искусства и ремесла, вы решаете, что с ними делать. Сначала вы набрасываете общую форму, затем раскрашиваете исходное окружение и, наконец, заполняете детали. При этом вы постоянно отходите назад и критическим взглядом оцениваете то, что уже сделано. А время от времени вы отбрасываете холст и начинаете все с самого начала. Но художники скажут вам, что все тяжкие труды пойдут прахом, если не вовремя остановиться. Так, если накладывать один слой краски на другой, деталь на деталь, то *в краске потеряется сама живопись*.

Старайтесь не испортить грамотно написанную программу чрезмерными украшательствами и усовершенствованиями. Просто имейте в виду, что она никогда не будет совершенной. (Подробнее об основных принципах написания кода и несовершенстве этого мира речь пойдет в разделе "Пока вы программируете" главы 7.)

Другие разделы, связанные с данной темой

- **Тема 45.** Западня требований, **глава 8** "До начала проекта".
- **Тема 46.** Решение неразрешимых головоломок, **глава 8** "До начала проекта".

Задачи

- Просмотрите инструментальные средства и операционные системы, которыми регулярно пользуетесь. Сможете ли вы найти в них признаки того, что их производители и/или разработчики спокойно выпускают программное обеспечение, заранее зная, что оно несовершенно? Как его пользователь вы, скорее всего:
  1) подождете до тех пор, пока не будут исправлены все программные ошибки;
  2) примите как должное некоторые программные ошибки, учитывая сложность программного обеспечения;
  3) или же выберете более простое программное обеспечение, но с меньшими дефектами?
- Рассмотрите влияние модульности на выпуск программного обеспечения. Потребуется ли больше или меньше времени, чтобы получить тесно связанный монолитный программный блок требуемого качества по сравнению с системой, состоящей из очень слабо связанных модулей или микрослужб? Каковы преимущества и недостатки каждого из этих методов проектирования?

- Можете ли вы сказать о распространенном программном обеспечении, что оно страдает от *раздутости функциональных средств*, т.е. содержит намного больше функциональных средств, чем вам вообще может потребоваться? Ведь каждое функциональное средство может стать источником программных ошибок и брешей в защите. И чем больше таких средств, тем труднее их находить, когда потребуется ими воспользоваться. Чувствуете ли вы себя в опасности попасть в такую западню?

Тема 6 Ваш багаж знаний

"Инвестиции в знания приносят наибольшие дивиденды".

Бенджамин Франклин

Ох, уж этот старина Бенджамин Франклин — никогда не упустит случай произнести нравоучительную проповедь! Ну как же? Ведь если бы мы рано ложились спать и рано вставали, то разве не стали бы отличными программистами? Ранней пташке — ранний корм... и что же тогда станет с ранним червячком? Но в данном случае Бенджамин попал не в бровь, а в глаз. Ведь ваши знания и опыт являются ныне самыми важными профессиональными ресурсами.

К сожалению, эти ресурсы *исчерпываются*[2]. Ваши знания устаревают по мере разработки новых методик, языков и сред. Изменение в расстановке сил на рынке способно сделать ваш опыт устаревшим или неуместным. Принимая во внимание постоянно растущие темпы перемен в технологическом обществе, это может произойти довольно быстро.

По мере убывания ваших знаний убывает и ваша ценность для вашей компании или клиента. И нам хотелось бы воспрепятствовать такой тенденции, вообще искоренив ее. Ваши способности учиться чему-то новому как раз и составляют самый важный стратегический ресурс. Но как приобрести способность учиться чему-то новому и как знать, чему именно следует учиться?

Ваш багаж знаний

Мы предпочитаем рассматривать все, что программисты знают о вычислениях, прикладных областях, в которых они работают, а также весь их опыт как *багаж знаний*, которыми они обладают. Управление багажом знаний очень похоже на управление финансовым портфелем по описываемым ниже руководящим принципам.

[2] *Исчерпывающийся ресурс* — это нечто такое, чья ценность убывает со временем. К характерным его примерам относится склад, заполненный бананами, а также билет на футбол.

1. Серьезные инвесторы по привычке регулярно вкладывают во что-нибудь свои денежные средства.
2. Ключом к успеху в долгосрочной перспективе является диверсификация.
3. Смышленые инвесторы составляют свои портфели, находя определенное равновесие между консервативными и высокодоходными, хотя и слишком рискованными инвестициями.
4. Инвесторы пытаются приобретать ценные бумаги по малой цене, а продавать их по высокой цене, чтобы получить наибольшую прибыль.
5. Портфели должны периодически пересматриваться и заново уравновешиваться.

Чтобы быть успешным в своей профессии, необходимо инвестировать в собственный багаж знаний, следуя таким же руководящим принципам, как и описанные выше для финансового портфеля. Правда, управление такого рода инвестициями — это такой же навык, как и любой другой, а следовательно, ему можно научиться. Самое главное — сначала заставить себя, а затем развить в себе привычку делать это. С этой целью выработайте соответствующую процедуру, чтобы следовать ей до тех пор, пока она не укоренится в вашем сознании. И тогда вы обнаружите, что впитываете новые знания автоматически.

Создание своего багажа знаний

- **Инвестируйте регулярно.** Подобно финансовому инвестированию, вы должны не менее *регулярно* инвестировать в багаж своих знаний, пусть даже и в небольших объемах. И здесь привычка важна в такой же степени, как и объемы инвестиций, поэтому зарезервируйте для этой цели время и место, чтобы не отвлекаться ни на что другое. Несколько примерных целей перечислены в следующем разделе.
- **Диверсифицируйте свои инвестиции.** Чем более разнообразны ваши знания, тем ценнее вы сами. Вы должны как минимум знать все особенности той технологии, которой пользуетесь в настоящий момент. Но не останавливайтесь на этом. Перемены в вычислительной технике происходят очень быстро, и если какая-нибудь технология весьма востребована сегодня, то завтра она окажется практически бесполезной или, по крайней мере, невостребованной. Чем больше технологий вы освоите, тем легче вам будет приспосабливаться к переменам. И не забывайте все *остальные* навыки, которые могут вам потребоваться в любой момент, в том числе и навыки, приобретенные в нетехнических областях знаний.

- **Управляйте риском.** Технологии бывают самые разные: от рискованных, но и потенциально высокорентабельных и до малорискованных, но и малорентабельных. Не стоит вкладывать все свои средства ни в высокорискованные акции, которые могут внезапно обесцениться, ни тратить их консервативно, упуская при этом неплохие возможности. Не кладите все яйца в одну корзину, полагаясь на какую-то одну технологию.
- **Покупайте дешево, а продавайте дорого.** Освоить появившуюся технологию еще до того, как она найдет широкое распространение, пожалуй, так же трудно, как и найти акции, продающиеся по цене существенно ниже рыночной. Но затраченные усилия могут вполне окупиться сторицей. Так, изучать язык Java в то время, когда он только появился и был малоизвестен, было рискованно, но усилия тех, кто принял его сразу, окупились впоследствии сторицей, когда он стал мейнстримом в данной отрасли.
- **Пересматривайте и заново балансируйте багаж своих знаний.** Информационная отрасль развивается очень динамично. Новоиспеченная технология, в которую вы начали инвестировать в прошлом месяце, сегодня может оказаться окаменевшим ископаемым. Возможно, вам стоит освежить знания технологий баз данных, которыми вы уже давно не пользовались, а может быть, лучше подумать о новой работе и попытаться изучить другой язык...

Самый первый из всех перечисленных выше руководящих принципов легче всего выполнить, поэтому он еще раз упоминается ниже.

Совет 9 Регулярно инвестируйте в свой багаж знаний

Цели

Теперь, когда известны некоторые руководящие принципы управления багажом знаний, необходимо выяснить самый лучший путь для приобретения интеллектуального капитала, который можно инвестировать в этот багаж знаний. Ниже приведен ряд рекомендаций по этому поводу.

- **Каждый год изучайте хотя бы один новый язык.** Разные языки позволяют по-разному решать одни и те же задачи. Изучив самые разные подходы к решению задачи, вы расширяете горизонты ее осмысления, избегая опасности застрять в тупике. Кроме того, изучение многих языков облегчается благодаря большому разнообразию свободно доступного программного обеспечения.

- **Читайте по одной книге из технической литературы каждый месяц.** Несмотря на обилие кратких статей и порой надежных ответов в Интернете, для углубленного понимания предмета требуются книги с подробным его изложением. Просмотрите среди предложений в книготорговой сети интересную литературу, связанную с вашим текущим проектом[3]. Как только чтение технической литературы войдет у вас в привычку, читайте ежемесячно по одной книге. Освоив технологии, которыми вы пользуетесь в настоящий момент, обратитесь к какой-нибудь другой технологии, *не* связанной с вашим текущим проектом.

- **Читайте и нетехническую литературу.** Очень важно не забывать, что компьютерами пользуются *люди*, потребности которых вы стараетесь удовлетворить. Вы работаете с людьми, вам дают работу другие люди, а некоторые люди даже пытаются незаконно проникнуть в вашу вычислительную систему. Поэтому не забывайте о человеческой стороне вашей профессии, которая требует совершенно иных навыков для межличностного общения (и хотя мы иронично называем их "легкими", на самом деле приобрести их трудно).

- **Посещайте курсы повышения квалификации.** Поищите интересные курсы в местном колледже, университете или в Интернете, а возможно, и на очередной выставке или конференции.

- **Участвуйте в местных пользовательских группах или собраниях.** Изоляция может оказаться убийственной для вашей карьеры, поэтому познакомьтесь с людьми, работающими за пределами вашей организации. Старайтесь принимать активное участие во встречах по интересам, а не только приходить и слушать.

- **Экспериментируйте с разными средами.** Если раньше вы работали только с Windows, уделите время работе с Linux. Если вы пользовались только сборочными файлами и текстовым редактором, опробуйте логически развитую интегрированную среду разработки с современными функциональными средствами, и наоборот.

- **Оставайтесь в курсе дела.** Читайте новости и публикации в Интернете о технологиях, отличающихся от тех, которыми вы пользуетесь в своем текущем проекте. Это отличный способ выяснить, каким опытом их применения обладают другие люди, ознакомиться с применяемой ими конкретной терминологией и т.д.

[3] Возможно, мы субъективны, но отличную подборку литературы можно найти по адресу `https://pragprog.com`.

Очень важно не прерывать инвестирование в багаж своих знаний. Как только вы освоите какой-нибудь новый язык или технологическое новшество, переходите к следующему, чтобы изучить его.

Совершенно не важно, пользуетесь ли вы любой из освоенных вами технологий в своем проекте или просто указываете их при составлении резюме. В процессе обучения вы расширяете горизонты своего мышления, открываете новые возможности и способы сделать что-нибудь конкретное. При обучении большое значение имеет взаимное обогащение идеями. Старайтесь применить усвоенные уроки в своем текущем проекте. И даже если конкретная технология в вашем проекте не применима, то вы все равно можете позаимствовать из нее какие-нибудь идеи. Так, если вы овладеете принципами объектно-ориентированного программирования, то сможете писать процедурные программы уже иначе. А уяснив парадигму функционального программирования, сможете иначе написать объектно-ориентированный код и т.д.

Возможности для обучения

Итак, вы взахлеб читаете книги, держитесь в курсе самых передовых разработок в вашей области, что совсем не так легко. Когда же кто-то обращается к вам с вопросом и вы не имеете ни малейшего представления, что ему ответить, то открыто признаетесь в этом. Но это *не должно* вас останавливать. Примите это как личный вызов и найдите ответ. Поспрашивайте у других, поищите ответ в Интернете, причем не только в бытовых, но и в академических кругах.

Если вы не можете найти ответ сами, найдите того, кто *способен* это сделать. Не оставляйте заданный вам вопрос без ответа. Общение с другими людьми поможет вам создать свой личный круг знакомств, где в ходе расспросов вы можете неожиданно для себя найти ответы и на другие, не связанные с данным вопросы. И при этом багаж ваших прежних знаний станет пополнятся новыми…

На все это чтение и изучение вам, конечно, требуется время, которого и так не хватает, поэтому подобные занятия необходимо планировать заранее. У вас всегда должно быть что-нибудь почитать в те моменты, когда вам нечего делать. Так, временем ожидания приема у доктора, в очереди или в городском транспорте можно выгодно воспользоваться для чтения, поэтому берите всегда с собой электронную книгу. Хотя вы, может быть, предпочитаете по старой привычке мусолить пальцами печатные издания вроде зачитанной до дыр газеты 1973 года о положении в Папуа Новой Гвинее.

Критическое мышление

Наконец, очень важно уметь *критически* осмысливать то, что вы читаете и слышите. Вам следует позаботиться о том, чтобы знания в вашем багаже были точными, непредубежденными и не поддающимися влиянию поставщиков или

шумихе, поднятой в средствах массовой информации. Берегитесь страстных приверженцев, настаивающих на том, что *только их догма* дает правильный ответ — это может быть и неприемлемо для вас, и вашего проекта.

Ни в коем случае не следует недооценивать силу коммерциализации. Если что-то оказывается в первых строчках результатов поиска в Интернете, то это совсем не означает, что оно самое лучшее. Ведь за то, чтобы занять ведущее положение в результатах поиска, могли просто заплатить. И если какая-нибудь книга выставлена на заметное место в книжной лавке, то это совсем не означает, что она хорошая или даже популярная. Ведь издательство могло заплатить за то, чтобы его книгу выставили на видном месте.

Совет 10 Критически анализируйте то, что вы читаете и слышите

Критическое мышление само по себе является отдельной дисциплиной, поэтому мы рекомендуем вам изучать ее и читать все, что только можно найти о ней. Между тем, вы можете начать с нескольких перечисленных ниже вопросов и обдумать их.

- **Пять вопросов "Почему?".** Задать хотя бы пять вопросов "Почему?" — излюбленный прием консультантов. Задайте сначала один такой вопрос и получите ответ. Затем копните глубже, задав еще один вопрос "Почему?". Повторяйте эту процедуру словно нетерпеливый, но вежливый четырехлетний ребенок. Подобным образом вам, возможно, удастся приблизиться к коренной причине.
- **Кому это выгодно?** Такой вопрос может показаться циничным, но *следование за деньгами* — это очень полезный путь для анализа. Выгоды для кого-нибудь другого лица или другой организации могут совпадать или не совпадать с вашими личными выгодами.
- **Каков контекст?** Все происходит в своем контексте, именно поэтому универсальные решения годятся не на все случаи жизни. Так, если вы читаете статью или книгу, где пропагандируется "норма наилучшей практики", вам уместно задаться следующими вопросами. Наилучшая практика для кого? Каковы предпосылки и последствия в краткосрочной и долгосрочной перспективе?
- **Когда и где это будет действовать?** При каких обстоятельствах? Не слишком ли поздно или же слишком рано? Не останавливайтесь на мышлении первого порядка: *что произойдет дальше?* Но перейдите далее к мышлению второго порядка: *что произойдет после этого?*
- **Почему это вызывает трудности?** Есть ли какая-то базовая модель и как она действует?

К сожалению, на этом простые вопросы исчерпываются. Но, пополняя багаж своих знаний и применяя в какой-то степени критический анализ к целой лавине читаемых вами технических статей, вы сможете уяснить и *сложные* вопросы.

Другие разделы, связанные с данной темой

- **Тема 1.** Это ваша жизнь.
- **Тема 22.** Технические дневники, **глава 3** "Основные инструментальные средства".

Задачи

- Начните на этой неделе изучение нового языка. Если вы привыкли программировать на одном и том же старом языке, опробуйте Clojure, Elixir, Elm, F#, Go, Haskell, Python, R, ReasonML, Ruby, Rust, Scala, Swift, TypeScript или любой другой язык, который может привлечь ваше внимание или просто понравиться вам[4].
- Начните читать новую книгу (как только дочитаете эту!). Если вы занимаетесь подробной реализацией и кодированием, прочитайте книгу по проектированию и архитектуре программного обеспечения. А если вы занимаетесь высокоуровневым или архитектурным проектированием, то прочитайте книгу по технологиям кодирования.
- Обсудите какую-нибудь технологию с людьми, не занятыми в вашем текущем проекте или работающими в других организациях. Познакомьтесь с такими людьми в кафетерии вашей организации или на местном собрании приверженцев данной технологии.

Тема 7 Общайтесь!

"Я считаю, что лучше быть проигнорированной вовсе, чем недооцененной".

Мэй Уэст,[5] фильм "Красавица 90-х" (Belle of the Nineties), 1934 г.

[4] Если вы никогда не слышали ни одном из этих языков, вспомните, что знание — исчерпывающийся ресурс, и то же самое можно сказать о распространенной технологии. Перечень новейших и экспериментальных языков сильно изменился после первого издания данной книги, а возможно, он уже будет иным к тому времени, когда вы будете читать эти строки. И это служит еще одним веским основанием для того, чтобы не переставать учиться.

[5] Mae West — актриса, драматург, сценарист и секс-символ первой половины XX века в Америке — *Примеч. пер.*

Вполне возможно, что мы в состоянии усвоить урок, преподанный мисс Уэст. Ведь дело не только в том, что у вас есть, а в том, как вы это преподнесете. Имея самые лучшие идеи, самый изящный код или самое прагматичное мышление, вы в конечном счете окажетесь в полной изоляции, если на сумете общаться с другими людьми. Хорошая идея без эффективного общения осиротеет.

Как разработчики, мы должны общаться на самых разных уровнях. Мы проводим немало часов на совещаниях, слушая и говоря, общаемся с конечными пользователями, пытаясь уяснить их потребности, пишем код, передающий наши намерения вычислительной машине и документирующий наше мышление для будущих поколений разработчиков, а также пишем предложения и памятные записки, требующие и обосновывающие ресурсы, составляем отчеты о текущем состоянии дел и предлагаем новые подходы. Наконец, мы работаем каждый день в своих командах, проповедуя собственные идеи, видоизменяя существующие нормы практики и предлагая новые. Большую часть своего рабочего дня мы проводим в общении, а следовательно, нам нужно делать это умело.

Рассматривайте свой родной язык как очередной язык программирования. Пишите на естественном языке так, как вы пишете код, учитывая принципы DRY и ETC, рассматриваемые в следующей главе, возможности автоматизации и пр.

Совет 11 Родной язык — это просто еще один язык программирования

В последующих разделах перечислены дополнительные идеи по поводу общения, которые, на наш взгляд, кажутся полезными.

Знайте, с кем вы общаетесь

Общение полноценно лишь в том случае, если вы передаете собеседнику то, что подразумеваете, а для этого недостаточно просто говорить. Для этого необходимо знать потребности, интересы и способности тех, с кем вы общаетесь. Всем нам не раз приходилось присутствовать на совещаниях, где некоторые разработчики с остекленевшим взглядом слушали длинный монолог вице-президента по сбыту о преимуществах какой-нибудь мудреной технологии. Это не общение, а просто говорильня, навевающая скуку.

Допустим, вам требуется внести изменения в свою систему дистанционного текущего контроля, чтобы воспользоваться сторонним брокером сообщений для распространения уведомлений о состоянии данной системы. С этой целью вы можете представить данное обновление системы самыми разными способами в зависимости от того, с кем именно вы общаетесь. Так, конечные пользователи по достоинству оценят такое обновление, поскольку их системы теперь смогут взаимодействовать с другими службами, пользующимися предлагаемым броке-

ром сообщений. Отдел сбыта вашей организации сможет воспользоваться этим фактом, чтобы увеличить объемы продаж, а руководители отделов разработки и эксплуатации просто обрадуются, поскольку заботы по сопровождению этой части системы лягут на чужие плечи. Наконец, разработчикам будет, возможно, интересно приобрести опыт работы с новыми прикладными интерфейсами и даже попробовать найти брокеру сообщений новые области применения. Сделав соответствующий краткий доклад для каждой из этих групп в отдельности, вы сможете заинтересовать их всех своим проектом.

Как и во всех остальных формах общения, самое главное здесь — обратная связь. Следует не просто ждать вопросов, а попросить их задавать. Обращайте внимание на движения тел и выражения лиц тех, кто вас слушает. Одна из исходных предпосылок в нейролингвистическом программировании гласит: значение вашего общения состоит в реакции, которую вы получаете. Постоянно совершенствуйте свои знания тех, с кем общаетесь.

Знайте, что вам требуется сказать

Едва ли не самой трудной частью более формальных стилей общения, применяемых в деловой сфере, является умение подобрать именно те слова, которые требуется сказать. Писатели нередко составляют планы своих книг, прежде чем приступить к их написанию, а составители технической документации просто садятся за клавиатуру, набирают, например, приведенный ниже заголовок, а затем все, что следует после него.

1. Введение

Планируйте то, что хотите сказать. Сделайте сначала набросок своей речи, а затем спросите себя: "Передает ли эта речь тем, кто меня слушает, то, что я хочу выразить, именно так, как им было бы полезно?" Уточняйте свою речь до тех пор, пока не достигнете желанной цели.

Такой подход пригоден не только для составления речей и документов. Когда вы встречаетесь с главным заказчиком на важном совещании или дискуссии, набросайте те идеи, которые хотите донести до него, и запланируйте пару стратегий ясного их изложения. Теперь, когда вы знаете, что именно вам требуется сообщить слушателям, останется лишь правильно подать все это.

Выбирайте удобный момент

Представьте, что сейчас конец рабочего дня в пятницу недели, когда проходила аудиторская проверка. Младший ребенок вашей начальницы попал в больницу, на дворе льет дождь как из ведра, а дорога за город домой сулит превратиться в сущий кошмар. Это, вероятно, не самый подходящий момент попросить у начальницы разрешение увеличить оперативную память на вашем компьютере.

Чтобы понять, что именно готовы услышать те, с кем вы общаетесь, необходимо, в частности, выяснить их приоритеты. Поймайте своего руководителя сразу после нелегкого разговора с высшим начальством по поводу утери некоторой части исходного кода, и вы не найдете более восприимчивого слушателя к вашим идеям относительно хранилищ исходного кода. Старайтесь высказывать свои идеи в подходящий момент, облекая их в удобную для восприятия форму. Иногда для этого достаточно спросить: "Удобно ли сейчас поговорить о...?"

Выбирайте стиль общения

Подбирайте стиль донесения своих мыслей под тех, кому вы их адресуете. Одним требуется формальное краткое сообщение "самих только фактов", а другим — длинная, обширная дискуссия, прежде чем перейти к делу. Поэтому необходимо знать уровень их квалификации и опыта в данной области, т.е. являются ли они знатоками или новичками. Требуется ли им подробная сопроводительная записка или же краткое изложение самой сути в сообщении по электронной почте? Если сомневаетесь, спрашивайте.

Но не забывайте, что вы — лишь половина двухстороннего общения. Если кто-нибудь скажет вам, что ему требуется лишь абзац с кратким описанием вашего предложения, а вы не видите способа изложить его короче, чем на нескольких страницах, скажите об этом откровенно. Не забывайте о том, что ответная реакция, какой бы она ни была, также является формой общения.

Подавайте свои идеи в привлекательной форме

Ваши идеи, безусловно, важны и поэтому заслуживают того, чтобы сообщить их в привлекательной форме. Составляя письменные документы, слишком многие разработчики (или их руководители) сосредоточивают основное внимание только на их содержании. Мы считаем это ошибкой. Любой повар (или менеджер сети ресторанов) скажет вам, что вы можете часами работать как каторжный на кухне лишь для того, чтобы кто-то испортил все ваши труды плохой подачей блюда.

Ныне оформлению малопривлекательных документов нет оправдания — современное программное обеспечение способно на поразительные результаты подготовки документов, будь они созданы с использованием языка разметки или в текстовом редакторе. Для этого достаточно выучить всего лишь несколько команд. Так, если вы пользуетесь текстовым редактором, подберите подходящие таблицы стилей и шаблоны. (Вполне возможно, что они уже определены в вашей организации, а следовательно, вы можете воспользоваться уже готовыми шаблонами.) Научитесь оформлять верхние и нижние колонтитулы страниц. Чтобы получить представление о стилях оформления и компоновке докумен-

тов, обратитесь к образцам, входящим в состав пакета текстового редактора. Выполните *проверку орфографии* сначала автоматически, а затем вручную. Ведь в тексте могут быть ошибки, которые не удастся уловить средству проверки — человеку свойственно ошиваться. :)

Привлекайте тех, с кем общаетесь

Нередко оказывается, что составляемые вами документы в конечном счете оказываются менее важными, чем сам процесс их подготовки. Поэтому привлекайте по возможности тех, кому эти документы адресованы, уже на ранних стадиях их составления в черновом варианте. Получите их отзывы и воспользуйтесь их советами. Таким образом вы наладите хорошие рабочие отношения и при этом сможете улучшить свой документ.

Учитесь слушать

Если хотите, чтобы вас услышали другие, научитесь *слушать их*. Даже в том случае, если вы полностью информированы, и даже тогда, когда стоите на формальном совещании перед двадцатью руководителями, они не услышат вас, если вы не слышите их.

Пригласите людей к дискуссии, задавая вопросы, или же попросите их сформулировать обсуждаемую тему своими словами. Превратите совещание в диалог, и тогда вы сможете донести свою точку зрения более четко и эффективно. Кто знает, может быть, вы чему-то в итоге научитесь.

Отвечайте людям

Если вы задаете кому-нибудь вопрос и человек вам не отвечает — вам кажется, что он невежлив. Но как часто вы сами не отвечали людям, когда они присылали вам сообщение по электронной почте или памятную записку, прося сообщить какую-нибудь информацию или предпринять некоторое действие? В суматохе повседневной жизни очень легко об этом забыть. Поэтому старайтесь всегда отвечать на сообщения по электронной почте, даже если это будет простая фраза: "Я отвечу вам позже". Если держать людей в курсе, они будут готовы простить вам случайные оплошности, чувствуя, что вы о них не забыли.

Совет 12 Важно не только то, что вы говорите, но и как вы это говорите

Вы должны непременно общаться, если только не работаете в полной изоляции. Чем эффективнее будет ваше общение, тем больше станет ваше влияние.

Документация

Наконец, осталось рассмотреть вопрос об общении через документацию. Как правило, разработчики мало задумываются о документации. В лучшем случае они считают ее досадной необходимостью, а в худшем — низкоприоритетной задачей, надеясь, что начальство забудет о ней под конец проекта.

Программисты-прагматики воспринимают документацию как неотъемлемую часть общего процесса разработки. Составление документации можно упростить, не дублируя усилия, не тратя зря время и держа документацию под рукой и, в частности, непосредственно в исходном коде. На самом деле к документации можно применить *все* те же прагматические принципы, что и к исходному коду.

Совет 13 Создавая документацию, не фиксируйте ее навечно

Качественную документацию можно получить из комментариев в исходном коде, и поэтому модули и экспортируемые функции рекомендуется снабжать комментариями, чтобы облегчить участь других разработчиков, когда им придется пользоваться этими программными компонентами. Но это совсем не означает, что мы согласны с теми, кто говорит, что *каждую* функцию, структуру данных, объявление типа данных и т.д. следует непременно снабдить комментариями. Такой механичный подход к составлению комментариев на самом деле усложняет сопровождение кода, поскольку при внесении изменений обновлять приходится не только исходный код, но и документацию. В связи с этим рекомендуется ограничить комментирование исходного кода, не относящегося к прикладному интерфейсу API, указанием причин, назначения и целей написания кода. Ведь сам исходный код уже ясно показывает, *как* он написан, а следовательно, комментировать его излишне, поскольку это нарушает принцип "не повторяться" (DRY).

Комментирование исходного кода дает идеальную возможность задокументировать те трудноуловимые особенности проекта, которые нельзя задокументировать где-нибудь еще: инженерные компромиссы, причины принятия конкретных решений и отвержения альтернативных решений и т.д.

Краткие итоги

- Знайте, что вам требуется сказать.
- Знайте, с кем вы общаетесь.
- Выбирайте удобный момент.

- Выбирайте стиль общения.
- Подавайте свои идеи в привлекательной форме.
- Привлекайте тех, с кем общаетесь.
- Учитесь слушать.
- Отвечайте людям.
- Храните исходный код и документацию вместе.

Другие разделы, связанные с данной темой

- **Тема 15.** Оценивание, **глава 2** "Прагматичный подход".
- **Тема 18.** Эффективное редактирование, **глава 3** "Основные инструментальные средства".
- **Тема 45.** Западня требований, **глава 8** "До начала проекта".
- **Тема 49.** Прагматичные команды, **глава 9** "Прагматичные проекты".

Задачи

- Имеется несколько хороших книг, в которых описывается общение в командах разработчиков, в том числе следующие:
  - *The Mythical Man-Month: Essays on Software Engineering* [Bro96].
  - *Peopleware: Productive Projects and Teams* [DL13].
- Постарайтесь прочитать эти книги в ближайшие 18 месяцев. Кроме того, обратите внимание на книгу *Dinosaur Brains: Dealing with All Those Impossible People at Work* [BR89], в которой обсуждается эмоциональный багаж, с которым все мы приходим в новую рабочую среду.
- Когда будете в следующий раз делать презентацию или писать памятную записку, чтобы донести свою точку зрения, постарайтесь проработать советы из этого раздела, прежде чем приступить к делу. Ясно определите круг тех лиц, с которыми вам предстоит общаться. Пообщайтесь с ними, если это возможно, впоследствии, чтобы выяснить, насколько точно вы оценили их потребности.

Общение в оперативном режиме

Все, что упоминалось выше об общении в письменной форме, в равной степени применимо и к общению по электронной почте, в социальных сетях, блогах и т.д. В частности, возможности электронной почты развились до такой степени, что она стала оплотом корпоративной связи, применяется для обсуждения контрактов, организации диспутов и даже служит в качестве свидетельств в суде. Но по какой-то причине люди, никогда не посылавшие ни жалкого листка документации по обычной почте, с удовольствием рассылают электронной почтой скверного вида неуместные сообщения по всему миру.

Наши рекомендации по общению в оперативном режиме довольно просты и перечислены ниже.

- Тщательно выверяйте написанное, прежде чем щелкнуть на кнопке ОТПРАВИТЬ.
- Выполняйте орфографическую проверку, чтобы обнаружить оплошности автоматической корректуры.
- Придерживайтесь простого и понятного формата.
- Старайтесь как можно меньше цитировать исходное сообщение при ответе. Никому не нравится получать обратно свое сообщение по электронной почте с присоединенным к нему единственным словом "Согласен".
- Если цитируете чужое сообщение в электронной почте, укажите его автора и процитируйте его в теле своего сообщения, а не в виде вложения. Это же относится и к общению в социальных сетях.
- Избегайте перепалок или троллинга, если не хотите преследований, которые непременно последуют за таким поведением. Если не решаетесь высказать кому-нибудь в лицо все, что о нем думаете, — не делайте этого и в сети.
- Проверяйте список получателей ваших сообщений, прежде чем отравлять их. Уже стало едва ли не нормой критиковать начальство по электронной почте, не замечая, что оно присутствует в качестве получателя сообщения в списке рассылки. А еще лучше — отказаться от критики начальства по электронной почте.

Как считает бесчисленное множество организаций и политиков, электронная почта и социальные сети — это уже навсегда. Поэтому постарайтесь уделить электронной почте столько же внимания, сколько и любой другой памятной записке или отчету.

Глава 2

Прагматичный подход

Имеются определенные рекомендации и приемы, применяемые на всех уровнях разработки программного обеспечения, совершенно универсальные процессы и почти аксиоматические идеи. Но такие подходы редко документируются, а вместо этого они зачастую обнаруживаются в виде отрывочных записей дискуссий о проектировании, управлении проектом или программировании. Но ради вашего удобства мы постараемся здесь собрать все эти идеи и процессы вместе.

Первая и, может быть, самая важная тема касается самой сути разработки программного обеспечения и раскрывается в разделе "Сущность качественного проектирования". Из нее следует все остальное. Далее следуют разделы "DRY — пороки дублирования" и "Ортогональность", в которых раскрываются две тесно связанные темы. В первом из них содержится предупреждение не дублировать знания по системам, а во втором — не разделять никакие фрагменты знаний по многим компонентам системы.

По мере увеличения темпов изменений становится все труднее и труднее сохранять приложения в должном состояния. Поэтому в разделе "Обратимость" мы рассмотрим некоторые методики, помогающие ограждать проекты от их изменяющейся среды.

Два последующих раздела также взаимосвязаны. Так, в разделе "Трассирующие пули" речь пойдет о стиле разработки, позволяющем одновременно собирать требования, проверять проектные решения и реализовывать код. И это единственный способ идти в ногу с современной жизнью. А в разделе "Прототипы и памятные записки" будет показано, каким образом прототипирование применяется для проверки архитектур, алгоритмов, интерфейсов и идей. В современном мире крайне важно проверять идеи и получать ответную реакцию, прежде чем безоговорочно принимать их.

По мере созревания вычислительной техники разработчики во все большем количестве создают языки высокого уровня. И хотя еще не изобретен компилятор, способный принимать команду "сделай это вот так", в разделе "Предметно-ориентированные языки" представлены более скромные рекомендации, которые вы можете реализовать самостоятельно.

Наконец, все мы работаем в условиях ограниченного времени и ресурсов. Нехватку таких ресурсов можно перенести легче (и принести большее удовлетворение начальству или клиентам), если постараться выяснить, как долго они будут затребованы. Именно об этом и пойдет речь в разделе "Оценивание".

Упомянутые выше принципы следует непременно иметь в виду во время разработки, чтобы писать более совершенный, быстродействующий и устойчивый код. Вы можете даже сделать его более удобочитаемым.

Тема 8 Сущность качественного проектирования

В мире полно умников и знатоков проектирования программного обеспечения, жаждущих передать свою мудрость, приобретенную тяжкими трудами. Для этого существуют особые сокращения, списки (почему-то обычно из пяти пунктов), шаблоны, диаграммы, видеоматериалы, беседы, а возможно, и увлекательные сериалы (Интернет есть Интернет), поясняющие закон Деметры с помощью интерпретирующего танца.

И мы, авторы этой книги, грешим этим. Но нам хотелось бы внести поправки в пояснение того, что лишь стало для нас очевидным. Прежде всего сделаем заявление общего характера.

Совет 14 Удачное проектное решение легче изменить, чем неудачное

Вещь считается удачно спроектированной, если она приспосабливается к тем, кто ею пользуется. Для кода это означает, что он должен приспосабливаться к изменениям. Поэтому мы верим в принцип *легкости изменения* (Easier to Change — ETC). Вот и все.

Насколько мы можем судить, всякий принцип проектирования является частным случаем принципа ETC. Чем, например, хорошо уменьшение степени связывания? Оно хорошо тем, что разделяет обязанности, чтобы их легче было изменить. А это и есть принцип ETC. В чем польза принципа единственной ответственности? Она состоит в том, что изменения в требованиях зеркально отображаются в изменениях лишь в одном модуле. И это соответствует принципу ETC. Почему так важно именование? А потому, что удачные имена делают исходный код более удобочитаемым, а ведь его приходится читать, чтобы внести в него изменения. И в этом случае соблюдается принцип ETC!

Принцип ETC — это ценность, а не правило

Ценности — это сущности, помогающие принять решение сделать то или другое. Что касается осмысления программного обеспечения, то ETC является

руководящим принципом, помогающим выбрать правильный путь. Как и все остальные ценности, этот принцип должен плавно следовать за здравой мыслью, мягко подталкивая в правильном направлении.

Но как этого добиться? Как показывает наш опыт, для этого требуется первоначальное сознательное подкрепление. Вам, возможно, придется потратить около недели, сознательно задавая себе вопрос: "Облегчило или же затруднило изменение всей системы в целом то, что я только что сделал?" Делайте это, когда сохраняете файл, пишете тест или исправляете ошибку в программе.

В принципе ETC присутствует неявное предположение. Он предполагает, что человек способен выбрать из многих путей именно тот, который будет легче сменить впоследствии. Зачастую здравый смысл позволяет выбрать правильный путь и дает возможность сделать обоснованное предположение. Но иногда на это нет и намека. Что ж, бывает и так. Мы считаем, что в подобных случаях можно сделать две вещи.

Во-первых, если вы не знаете, какую конечную форму примут вносимые вами изменения, то всегда можете вернуться к пути "легкости изменений": попытаться сделать заменяемым фрагмент кода, который вы пишете. В таком случае, что бы ни произошло впоследствии, этот фрагмент кода не станет неодолимой преградой на вашем пути. Такая мера кажется крайней, но на самом деле вы должны так или иначе прибегать к ней постоянно. Это всего лишь забота о сохранении кода не связанным и согласованным.

Во-вторых, рассматривайте это как способ развития инстинктов. Запишите данную ситуацию в своем журнале технического учета, упомянув выбранные вами варианты и некоторые предположения об изменениях. Оставьте метку в исходном коде. В дальнейшем, когда этот фрагмент кода придется изменить, вы сможете оглянуться назад и предоставить отчет самому себе. Это может помочь, когда вы в очередной раз окажетесь на аналогичной развилке.

В остальных разделах этой главы рассматриваются конкретные идеи по поводу проектирования. Но все они вызваны описанным здесь одним и тем же принципом легкости изменения.

Другие разделы, связанные с данной темой

- **Тема 9**. DRY — пороки дублирования.
- **Тема 10**. Ортогональность.
- **Тема 11**. Обратимость.
- **Тема 14**. Предметно-ориентированные языки.
- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 30**. Преобразовательное программирование, **глава 5** "Гибкость или ломкость".
- **Тема 31**. Налог на наследование, **глава 5** "Гибкость или ломкость".

Задачи

- Обдумайте принцип проектирования, которым пользуетесь регулярно. Направлен ли он на то, чтобы сделать что-то легко изменяемым?
- Подумайте также о языках и парадигмах программирования: объектно-ориентированного, функционального и т.п. Какие главные положительные, отрицательные или те и другие стороны они имеют для написания кода в соответствии с принципом ETC?
- Что вы можете сделать при программировании для того, чтобы исключить отрицательные и выделить положительные стороны[1]?
- Во многих текстовых редакторах имеется (встроенная или через расширения) поддержка команд, выполняемых при сохранении файла. Настраивайте текстовый редактор на вывод сообщения "ETC?" всякий раз[2], когда сохраняете изменения в файле. Пользуйтесь такой возможностью в качестве напоминания о необходимости обдумать написанный вами код. Насколько легко его изменить?

Тема 9 DRY — пороки дублирования

Капитан Джеймс Тиберий Кирк[3] предпочитал снабжать компьютер двумя противоречащими друг другу фрагментами знаний, чтобы нейтрализовать враждебный искусственный интеллект. К сожалению, тот же самый принцип может очень легко погубить *ваш* код.

Программисты обычно собирают, организуют, хранят и используют знания. Они документируют знания в спецификациях, возрождают их в выполняемом коде и пользуются ими для проверок во время тестирования. К сожалению, знания непостоянны. Они изменяются, и зачастую очень быстро. Так, ваше представление о каком-нибудь требовании может измениться после совещания с клиентом. Стоит правительству изменить какой-нибудь нормативный акт, и соответствующая бизнес-логика устаревает. Тесты могут показать, что выбранный алгоритм неработоспособен. Все это непостоянство означает, что нам приходится большую часть времени работать в режиме сопровождения, реорганизации и преобразования знаний в своих системах.

---

[1] Перефразируя старую песню *Accentuate The Positive* (Делайте акцент на положительном) Джонни Мерсера (Johnny Mercer) — американского исполнителя, популярного в 1940-х годах.

[2] Или хотя бы каждый десятый раз, чтобы оставаться в здравом уме...

[3] Персонаж научно-фантастического телевизионного сериала *Звёздный путь: Оригинальный сериал* (Star Trek: The Original Series). — *Примеч. пер.*

Многие полагают, что сопровождение приложения начинается после его выпуска и означает устранение программных ошибок и совершенствование функциональных средств. Мы считаем, что они не правы. Программисты постоянно работают в режиме сопровождения, поскольку их представления изменяются день за днем, когда поступают новые требования, а уже имеющиеся требования развиваются по мере сосредоточения усилий на проекте. Может измениться и сама среда. Но, независимо от конкретной причины, сопровождение является не каким-то отдельным видом деятельности, а обыкновенной стадией всего процесса разработки.

Когда выполняется сопровождение, приходится искать и изменять представления вещей, т.е. те крупицы знаний, которые вложены в приложение. Но дело в том, что знания очень легко продублировать в спецификациях, процессах и разрабатываемых программах. И делая это, разработчики навлекают сущий кошмар сопровождения, который начинается еще до выпуска приложения. Мы считаем, что единственный способ надежно разрабатывать программное обеспечение и упростить понимание и сопровождение разработок — следовать принципу "не повторяться" (DRY):

> *У каждого фрагмента знаний должно быть единственное, недвусмысленное, непререкаемое представление в системе.*

А почему этот принцип называется DRY? А вот почему:

Совет 15 DRY — Don't Repeat Yourself, т.е. "не повторяйся"

Альтернатива состоит в присутствии чего-то одного в двух или более местах. Если в таком случае изменить что-то одно, то нужно не забыть изменить и все остальное, а иначе программа будет загнана противоречием в угол, как компьютеры пришельцев. И вопрос не в том, чтобы вспомнить об изменениях, а в том, когда они забудутся.

Принцип DRY будет еще не раз упоминаться на страницах этой книги, и зачастую в таком контексте, который не имеет ничего общего с программированием. Мы считаем, что это одно из самых важных инструментальных средств в арсенале программиста-прагматика. И в этом разделе мы вкратце изложим трудности, возникающие в связи с дублированием, а также предложим общие стратегии их преодоления.

Принцип DRY не только для кодирования

Прежде всего устраним некоторое недоразумение. В первом издании этой книги мы поступили неверно, объяснив лишь, что мы подразумевали под выражением "не повторяться". И многие посчитали, что принцип DRY имеет отношение только к коду, думая, что он означает "не выполнять копирование и вставку

строк исходного кода". Но это лишь *часть* принципа DRY, причем мелкая и самая простая. Принцип DRY имеет отношение к дублированию *знаний*, а также *намерений*, т.е. к выражению одного и того же в двух разных местах, кроме того, возможно, разными способами.

В качестве пробного камня попробуйте ответить на следующие вопросы: если должно быть изменено какое-нибудь свойство кода, то не придется ли вносить изменения во многих местах и в самых разных форматах? Не придется ли изменять исходный код и документацию, схему базы данных и хранящуюся в ней структуру и т.д.? Если это именно так, то такой код не соответствует принципу DRY. Рассмотрим некоторые типичные примеры дублирования.

Дублирование в исходном коде

Каким бы банальным это ни показалось, но дублирование кода — довольно частое явление. Ниже приведен характерный тому пример.

```
def print_balance(account)
    printf "Debits: %10.2f\n", account.debits
    printf "Credits: %10.2f\n", account.credits
    if account.fees < 0
        printf "Fees: %10.2f-\n", -account.fees
    else
        printf "Fees: %10.2f\n", account.fees
    end
    printf " ----\n"
    if account.balance < 0
        printf "Balance: %10.2f-\n", -account.balance
    else
        printf "Balance: %10.2f\n", account.balance
    end
end
```

Пренебрежем пока что следствиями типичной для новичков ошибки, работая с денежными суммами в числовом формате с плавающей точкой. Вместо этого попробуем выявить дублирование в приведенном выше фрагменте кода. (Мы обнаружили три таких места, но вы можете найти и больше.)

Что же мы нашли? А вот что.

Прежде всего, в рассматриваемом здесь коде явно проявляется дублирование обработки отрицательных чисел методом копирования и вставки. Этот недостаток можно устранить, введя еще одну функцию, как показано ниже.

```
def format_amount(value)
    result = sprintf("%10.2f", value.abs)
    if value < 0
        result + "-"
    else
        result + " "
    end
end
```

```
def print_balance(account)
    printf "Debits: %10.2f\n", account.debits
    printf "Credits: %10.2f\n", account.credits
    printf "Fees: %s\n", format_amount(account.fees)
    printf " ———-\n"
    printf "Balance: %s\n", format_amount(account.balance)
end
```

Еще одно дублирование проявляется в повторении ширины поля во всех вызовах функции `printf()`. Такое дублирование *можно было бы* устранить, введя константу и передавая ее при каждом вызове данной функции, но почему бы не воспользоваться уже имеющейся функцией? Ниже показано, как это делается.

```
def format_amount(value)
    result = sprintf("%10.2f", value.abs)
    if value < 0
        result + "-"
    else
        result + " "
    end
end

def print_balance(account)
    printf "Debits: %s\n", format_amount(account.debits)
    printf "Credits: %s\n", format_amount(account.credits)
    printf "Fees: %s\n", format_amount(account.fees)
    printf " ———-\n"
    printf "Balance: %s\n", format_amount(account.balance)
end
```

Еще что-нибудь? А что если клиент попросит ввести дополнительные пробелы между метками и числами? Тогда придется изменять пять строк кода. Устраним и это дублирование:

```
def format_amount(value)
    result = sprintf("%10.2f", value.abs)
    if value < 0
        result + "-"
    else
        result + " "
    end
end

def print_line(label, value)
    printf "%-9s%s\n", label, value
end

def report_line(label, amount)
    print_line(label + ":", format_amount(amount))
end

def print_balance(account)
    report_line("Debits", account.debits)
```

```
    report_line("Credits", account.credits)
    report_line("Fees", account.fees)
    print_line("", "----")
    report_line("Balance", account.balance)
end
```

Если потребуется изменить форматирование денежных сумм, то придется внести соответствующие коррективы в функцию `format_amount()`. А если потребуется изменить формат меток — внести коррективы в функцию `report_line()`.

Тем не менее неявное нарушение принципа DRY все еще остается, поскольку количество дефисов в разделительной линии связано с шириной поля денежной суммы. Хотя совпадение неточное: в данном случае поле короче на один символ, поэтому любые конечные дефисы выступают за пределы столбца. Но это намерение клиента, отличающееся от фактического форматирования денежных сумм.

Не всякое дублирование кода является дублированием знаний

В приложении для заказа вин через Интернет требуется зафиксировать и проверить возраст клиента, а также количество заказанного вина. По требованию владельца веб-сайта и то и другое должно быть представлено в виде положительных чисел. Поэтому соответствующие проверки достоверности реализуются в коде следующим образом:

```
def validate_age(value):
    validate_type(value, :integer)
    validate_min_integer(value, 0)

def validate_quantity(value):
    validate_type(value, :integer)
    validate_min_integer(value, 0)
```

Во время проверки исходного кода местный всезнайка может отбраковать приведенный выше фрагмент, заявив, что он нарушает принцип DRY — ведь тела обеих функций одинаковы. Но он не прав! Код действительно одинаковый, но знания, которые он представляет, — разные. Ведь обе функции проверяют на достоверность разные данные, просто в данном случае это делается по одинаковым правилам. Это всего лишь совпадение, а не дублирование.

Дублирование в документации

Неизвестно откуда, но некогда родился миф, что все функции должны быть снабжены комментариями. Те, кто верят в эту нелепость, создают нечто похожее на приведенный ниже код.

```
# Вычислить комиссионные по этому счету.
#
# * Каждый возвращаемый чек стоит $20.
# * Если перерасход денег на счете длится больше 3 дней,
```

```
#   насчитать пеню $10 за каждый день.
# * Если средний остаток на счете превышает $2,000,
#   сократить комиссионные на 50%.

def fees(a)
    f = 0
    if a.returned_check_count > 0
        f += 20 * a.returned_check_count
    end
    if a.overdraft_days > 3
        f += 10*a.overdraft_days
    end
    if a.average_balance > 2_000
        f /= 2
    end
    f
end
```

Назначение приведенной выше функции указано дважды: один раз — в комментариях к ней, второй раз — в ее исходном коде. Если заказчик изменит порядок начисления комиссионных, обновить придется как исходный код, так и комментарии. И можно гарантировать, что со временем комментарии окажутся несогласованными с исходным кодом.

Непременно задайтесь вопросом: какие комментарии вводятся в исходный код? На наш взгляд, комментарии лишь возмещают неудачное именование и компоновку. Так, в приведенном ниже фрагменте кода имена ясно указывают назначение элементов кода. И если кому-нибудь понадобятся подробности, то он обнаружит их в исходном тексте. Именно так и соблюдается принцип DRY!

```
def calculate_account_fees(account)
    fees = 20*account.returned_check_count
    fees += 10*account.overdraft_days  if account.overdraft_days > 3
    fees /= 2                          if account.average_balance > 2_000
    fees
end
```

Нарушение принципа DRY в данных

Структуры данных представляют знания, и они тоже могут противоречить принципу DRY. Рассмотрим в качестве примера следующий класс, пре-дставляющий прямую линию:

```
class Line {
    Point start;
    Point end;
    double length;
};
```

На первый взгляд, такое определение класса может показаться вполне обоснованным. В нем явно определены поля для начальной и конечной точек пря-

мой линии, а также ее длины, даже если она нулевая. Но в нем все равно присутствует дублирование. Ведь длина прямой линии определяется ее начальной и конечной точками. Достаточно изменить одну из них, чтобы изменилась длина прямой линии. Поэтому поле длины лучше сделать вычисляемым, как показано ниже.

```
class Line {
    Point start;
    Point end;
    double length() { return start.distanceTo(end); }
};
```

На последующей стадии процесса разработки, возможно, придется пойти на нарушение принципа DRY из соображений производительности. Зачастую это происходит в том случае, когда требуется кешировать данные, чтобы исключить повторение затратных операций. И здесь самое главное — локализовать отрицательное влияние, чтобы оно не распространялось наружу. Отличия должны быть заметны лишь в методах класса, как показано ниже.

```
class Line {
    private double length;
    private Point start;
    private Point end;

    public Line(Point start, Point end) {
        this.start = start;
        this.end = end;
        calculateLength();
    }

    // Открытые методы
    void setStart(Point p)  { this.start = p; calculateLength(); }
    void setEnd(Point p)    { this.end = p; calculateLength(); }

    Point getStart()        { return start; }
    Point getEnd()          { return end; }
    double getLength()      { return length; }

    private void calculateLength() {
        this.length = start.distanceTo(end);
    }
};
```

В данном примере также наглядно демонстрируется следующее важное положение: всякий раз, когда модуль раскрывает структуру данных, весь использующий ее код привязывается к реализации этого модуля. Поэтому для чтения и записи содержимого свойств или полей объекта следует всегда пользоваться функциями доступа там, где это возможно. Благодаря этому впоследствии упрощается внедрение дополнительных функциональных возможностей.

Такое употребление функций доступа тесно связано с *принципом единообразия доступа*, описанным Мейером в книге *Object-Oriented Software Construction* [Mey97] и гласящим следующее:

> *Все службы, предоставляемые модулем, должны быть доступны с помощью единообразной системы обозначений, которая не зависит от того, как они реализуются: через хранение в памяти или вычисление.*

Представительное дублирование

Написанный вами код взаимодействует с внешним миром: с другими библиотеками — через интерфейсы прикладного программирования (API), с другими службами — через удаленные вызовы, с данными из внешних источников и т.д. И всякий раз, когда вы организуете такое взаимодействие, так или иначе нарушаете принцип DRY. Ведь в вашем коде требуются знания, которые присутствуют во внешнем объекте, в том числе API, схема базы данных, коды ошибок и пр. Дублирование в данном случае состоит в том, что обеим сторонам взаимодействия (вашему коду и внешнему объекту) необходимо знать представление их интерфейсов. Достаточно изменить его у одной стороны, и на другой стороне все будет нарушено. И хотя такое дублирование неизбежно, его все же можно умерить. Ниже описывается ряд стратегий, позволяющих добиться этого.

Дублирование через внутренние API

Что касается внутренних API, то следует найти такие инструментальные средства, которые позволяют указать интерфейс прикладного программирования в некоторого рода нейтральном формате. Как правило, такие инструментальные средства автоматически генерируют документацию, функциональные тексты, имитирующие API и их клиентов, причем последних — на самых разных языках. В идеальном случае такое инструментальное средство способно сохранять внутренние API в центральном хранилище, делая их общедоступными для команд разработчиков.

Дублирование через внешние API

Все чаще можно обнаружить, что открытые API формально документированы с использованием таких средств, как, например, OpenAPI[4]. Это дает возможность импортировать спецификации API в локальные инструментальные средства и более надежно интегрировать их с соответствующими службами.

Если вам не удастся найти подходящую спецификацию, рассмотрите возможность создать и опубликовать ее самостоятельно. Это не только принесет пользу другим, но и поможет ее сопровождению.

[4] См. по адресу `https://github.com/OAI/OpenAPI-Specification`.

Дублирование через источники данных

Многие источники данных позволяют выполнять самоанализ своих схем данных. Этим обстоятельством можно воспользоваться, чтобы по большей части исключить дублирование между источниками данных и прикладным кодом. Вместо того чтобы писать вручную код для работы с данными из подобных источников, можно сформировать контейнеры непосредственно из схемы данных. Многие каркасы могут сделать эту рутинную работу вместо вас.

Имеется и другая, на наш взгляд, более предпочтительная возможность. Вместо того чтобы писать код, представляющий внешние данные в фиксированной структуре (например, экземпляре структуры или классе), достаточно ввести их в структуру данных в виде пар "ключ–значение", которая может называться отображением, хешем, словарем или даже объектом, в зависимости от конкретного языка программирования.

Само по себе это рискованно, поскольку необходимо знать, какие именно данные следует обрабатывать, а от этого существенно снижается уровень безопасности. Поэтому рекомендуется вводить в такое решение второй уровень в виде простого табличного набора тестов для проверки достоверности, чтобы выяснить, содержит ли созданное отображение требующиеся данные в нужном формате. Вполне возможно, что имеющемуся у вас инструментальному средству документирования прикладных интерфейсов API удастся сформировать такой набор тестов.

Дублирование среди разработчиков

Едва ли не самый трудный для обнаружения и устранения вид дублирования происходит между разными разработчиками в проекте. Неумышленно могут быть сдублированы целые наборы функциональных возможностей, годами оставаясь необнаруженными и существенно затрудняя сопровождение. Нам приходилось слышать, как государственные вычислительные системы США обследовались на соблюдение требований в связи с наступлением 2000 года. Как показала проверка, более 10 тысяч программ содержали разные версии кода проверки номера социального обеспечения.

На самом общем уровне с дублированием можно справиться, организовав сильную команду, сплоченную хорошо налаженным общением. А вот на уровне модуля данная проблема проявляется не так явно. Ведь зачастую требуются функциональные возможности или данные, которые не входят в очевидные границы ответственности, и поэтому они могут быть многократно реализованы. Для борьбы с дублированием на этом уровне, на наш взгляд, лучше всего поощрять активное и частое общение разработчиков друг с другом.

Возможно, стоит проводить ежедневные "летучки" по методике Scrum или организовать форумы (например, через каналы корпоративного обмена сообще-

ниями типа Slack). Подобным образом обеспечивается ненавязчивое общение (даже во многих местах) с постоянным хранением архивов всего сказанного.

Одного из членов команды можно назначить в качестве библиотекаря проекта, вменив ему в обязанность содействовать обмену знаниями. Выделите центральное место в дереве исходного кода, где можно было бы размещать служебные программы и сценарии, и возьмите себе за правило читать исходный код и документацию других разработчиков как неформально, так и во время просмотров исходного кода. Ведь так вы сами учитесь у них, а не подглядываете за ними. И не забывайте, что доступ к исходному коду должен быть взаимным — не смущайте тех, кто внимательно изучает *ваш* исходный код, а возможно, и копается в нем.

Совет 16 Упрощайте повторное использование исходного кода

Старайтесь взрастить такую среду, где было бы проще находить и повторно использовать существующий исходный код, чем писать его самому. *Если это нелегко сделать, люди вообще не будут так поступать.* Если же вы не умеете повторно пользоваться исходным кодом, то рискуете дублировать знания.

Другие разделы, связанные с данной темой

- **Тема 8**. Сущность качественного проектирования.
- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 32**. Конфигурирование, **глава 5** "Гибкость или ломкость".
- **Тема 38**. Программирование по совпадению, **глава 7** "По ходу кодирования".
- **Тема 40**. Рефакторинг, **глава 7** "По ходу кодирования".

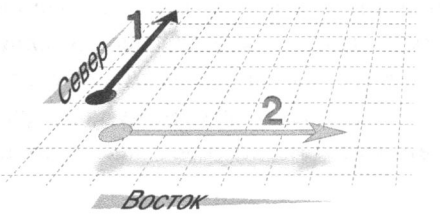
Тема 10 Ортогональность

Понятие ортогональности имеет решающее значение для создания таких систем, которые легко проектировать, строить, тестировать и расширять. Но это понятие редко изучают непосредственно. Зачастую оно считается неявным свойством совсем других изучаемых приемов и методов, а это ошибка. Научившись применять принцип ортогональности непосредственно, вы сразу же заметите, насколько повысится качество создаваемой вами системы.

Что такое ортогональность

Термин *ортогональность* заимствован из геометрии, где две линии называются ортогональными, если они пересекаются под прямым углом (например,

оси на графике). А в терминологии векторов две подобные линии считаются *независимыми*. Так, линия №1 на приведенной ниже диаграмме направлена на север, и она никоим образом не изменяет положение на восток или запад. А линия №2 направлена на восток, а не на север или юг.



В вычислительной технике термин *ортогональность* стал означать независимость, или развязывание. Два или более компонента считаются ортогональными (не связанными), если изменения в одном из них не оказывают никакого влияния на остальные компоненты системы. В грамотно спроектированной системе исходный код базы данных будет ортогональным по отношению к пользовательскому интерфейсу. Это означает, что пользовательский интерфейс можно изменить, не оказывая никакого влияния на базу данных, а последнюю — заменить, не меняя первый. Но прежде чем обсуждать преимущества ортогональных систем, рассмотрим систему, которая не является ортогональной.

Неортогональная система

Представьте, что вы совершаете вертолетную экскурсию по Большому Каньону, шт. Колорадо, и вдруг пилот, съевший, очевидно, по ошибке рыбу на обед, застонал и потерял сознание. К счастью или несчастью, но он оставил вертолет без управления зависшим на высоте около 30 м над землей.

По счастливой случайности вам довелось прочитать прошлым вечером страницу Википедии о вертолетах, и поэтому вам известно, что у вертолета имеются четыре основных рычага управления. В частности, вы берете ручку продольно-поперечного управления *циклическим шагом* несущего винта в правую руку и перемещаете ее так, чтобы вертолет двигался в нужном направлении. А в левую руку берете рычаг управления *общим шагом* несущего винта и тянете его на себя, чтобы увеличить шаг на всех лопастях и, таким образом, набрать высоту. На конце рычага управления общим шагом находится рукоятка управления *газом*. Наконец, *двумя педалями* вы можете изменить силу тяги хвостового винта, чтобы повернуть вертолет.

"Легко! — подумаете вы. — Плавно опусти ручку продольно-поперечного управления циклическим шагом несущего винта и аккуратно приземлишься, герой". Но, попытавшись это сделать, вы обнаружите, что не все так просто. Вертолет нырнет носом вниз и начнет передвигаться по спирали влево. И неожиданно

для вас выяснится, что каждое управляющее воздействие на систему, в которой вы летите, имеет побочные эффекты. Стоит опустить рычаг в левой руке, и вам придется добавить уравновешивающее обратное движение ручкой в правой руке и нажатием правой педали. Но и тогда каждое из этих изменений снова окажет влияние на все остальные органы управления. Таким образом, вы вдруг обнаружите, что пытаетесь манипулировать невероятно сложной системой, где каждое изменение оказывает влияние на все остальные входные воздействия. В итоге на вас падает необычайная нагрузка, поскольку ваши руки и ноги постоянно двигаются, пытаясь уравновесить все взаимодействующие силы. И это означает, что органы управления вертолетом решительно неортогональны.

Преимущества ортогональности

Как демонстрирует приведенный выше пример вертолета, неортогональным системам присущи большая сложность при внесении изменений и управлении. Если компоненты какой-нибудь системы в высшей степени взаимозависимы, то говорить о локальном исправлении ошибок не приходится вовсе.

Совет 17 Исключайте взаимное влияние несвязанных компонентов системы

Компоненты системы следует разрабатывать таким образом, чтобы они были самостоятельными, независимыми и имели единственное, вполне определенное назначение, т.е. то, что Юрдон (Yourdon) и Константин (Constantine) называют в своей книге *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* [YC79] *сцеплением* (cohesion). Если компоненты обособлены один от другого, то один из них можно свободно изменить, не опасаясь затронуть все остальные. При этом вы можете быть уверены, что не внесете никаких осложнений, способных распространиться по всей системе, если только не изменяете внешние интерфейсы компонента. Проектируя ортогональные системы, вы получаете два основных преимущества: повышение производительности и снижение рисков.

Повышение производительности

Это преимущество означает следующее.

- Изменения локализуются, поэтому время разработки и тестирования сокращается. Относительно небольшие, самостоятельные компоненты написать проще, чем один крупный блок кода. Простые компоненты можно спроектировать, запрограммировать, протестировать, а затем забыть о них, поскольку не нужно постоянно изменять уже существующий код, когда вводится новый.

- Ортогональный метод проектирования способствует повторному использованию. Так, если на компоненты возлагаются конкретные, вполне определенные обязанности, их можно объединять с новыми компонентами даже так, как того не предвидели те, кто их первоначально реализовывал. Чем более слабо связаны системы, тем легче они поддаются перекомпоновке и перепроектированию.
- Объединение ортогональных компонентов дает едва заметный выигрыш в производительности. Допустим, один компонент выполняет M отдельных операций, а другой компонент — N операций. Если оба эти компонента ортогональны и объединяются, то в итоге получается система, выполняющая $M \times N$ операций. Но если оба компонента не ортогональны, то происходит наложение, а в итоге выполняется меньше операций. Таким образом, объединяя ортогональные компоненты, можно получить больше функциональных возможностей на единицу объема выполненных работ.

Сокращение риска

При ортогональном подходе сокращаются риски, присущие любой разработке и состоящие в следующем.

- Неисправные разделы кода изолируются. Если какой-нибудь модуль поврежден, вероятность, что признаки его ущербности распространятся на остальную систему, становится меньше при ортогональном подходе. Кроме того, такой модуль проще вычленить и заменить каким-нибудь новым и вполне работоспособным модулем.
- Конечная система менее уязвима. А мелкие изменения и исправления в конкретной области или любые вносимые вами осложнения ограничиваются данной областью.
- Ортогональная система лучше поддается тестированию, поскольку ее проще спроектировать и провести тесты ее компонентов.
- Вас ничто не будет тесно связывать с конкретным поставщиком, программным продуктом или платформой, поскольку интерфейсы с этими сторонними компонентами ограничиваются небольшими частями общей разработки.

А теперь рассмотрим некоторые способы применения принципа ортогональности на практике.

Проектирование

Большинству разработчиков хорошо известна потребность в проектировании ортогональных систем, хотя для описания данного процесса они могут пользоваться такими терминами, как *модульное*, *компонентное* и *многоуровневое*

проектирование. Системы должны состоять из ряда взаимодействующих модулей, каждый из которых реализует независимые друг от друга функциональные возможности. Иногда такие компоненты организуются в отдельные уровни, каждый из которых обеспечивает определенный уровень абстракции. Такой многоуровневый подход оказывается весьма эффективным для проектирования ортогональных систем. А поскольку на каждом уровне используются только те абстракции, которые обеспечиваются нижележащими уровнями, то появляется возможность очень гибко изменять базовые реализации, не оказывая никакого влияния на исходный код. Многоуровневое представление позволяет также снизить риск появления бесконтрольных зависимостей между модулями. Зачастую многоуровневое представление системы выражается в виде диаграмм, аналогичных приведенной ниже.

Проверить проектирование системы на ортогональность совсем не трудно. Как только вы наметите компоненты системы, задайте себе следующий вопрос: *если я внесу коренные изменения в требования к конкретной функции, на какое количество модулей это повлияет*? В ортогональной системе ответ на этот вопрос должен быть "один"[5]. Так, перемещение экранной кнопки в графическом пользовательском интерфейсе не должно требовать изменений в схеме базы данных, а добавление контекстно-зависимой справки — изменений в подсистеме выписки счетов.

Рассмотрим в качестве примера сложную систему управления и текущего контроля нагревательной установки. В первоначальном требовании было указано обязательное наличие пользовательского интерфейса, но затем это требование было дополнено интерфейсом для мобильных устройств, позволяющим инженерам постоянно контролировать основные параметры процесса нагрева. Чтобы

[5] На практике не все оказывается так просто. Лишь при очень счастливом стечении обстоятельств большинство изменений в реальных требованиях не окажет влияние на многие функции в системе. Но если анализировать изменения по функциям, то каждое функциональное изменение в идеальном случае должно все же оказывать влияние лишь на один модуль.

удовлетворить этим требованиям, в ортогонально спроектированной системе придется внести изменения лишь в модули, которые непосредственно связаны с пользовательским интерфейсом, тогда как базовая логика управления нагревательной установки останется без изменения. В самом деле, если структура такой системы тщательно спроектирована, то обеспечить поддержку обоих интерфейсов удастся с помощью одной и той же кодовой базы.

Спросите себя: насколько структура вашей системы отделена от изменений в реальном мире? Так, если вы используете номер телефона в качестве идентификатора клиента, то что произойдет, если телефонная компания переназначит телефонные коды городов, областей и регионов? Почтовые индексы, номера социального обеспечения или паспортов, адреса электронной почты и домены Интернета — все это внешние идентификаторы, которые вам не подвластны и могут измениться в любой момент по какой угодно причине. *Поэтому ни в коем случае не полагайтесь на свойства объектов, неподвластные вашему контролю.*

Инструментальные средства и библиотеки

Старайтесь сохранять ортогональность своей системы, внедряя сторонние инструментальные средства и библиотеки. Благоразумно выбирайте подходящие технологии.

Внедряя набор инструментальных средств (или даже библиотеку от других членов команды), спросите себя: какие нежелательные изменения в исходном коде это может повлечь? Так, если схема сохраняемости объектов оказывается прозрачной, то она спроектирована ортогонально. Если же требуется создавать объекты (или получать к ним доступ) каким-то особым образом, то схема их сохраняемости неортогональна. Отделение таких подробностей реализации от вашего кода дает дополнительное преимущество, упрощающее поставщикам внедряемых средств внесение изменений в будущем.

Система EJB (Enterprise Java Beans — корпоративные компоненты Java Beans) служит интересным примером ортогональности. В большинстве ориентированных на транзакции систем на прикладной код возлагается обязанность обозначить начало и конец каждой транзакции. В системе EJB такая информация выражается декларативно в виде аннотаций и за пределами методов, выполняющих всю необходимую работу. Один и тот же прикладной код может быть выполнен без всяких изменений в разных средах транзакций EJB.

В известном смысле система EJB служит наглядным примером применения проектного шаблона "Декоратор" (Decorator), по которому функциональные возможности вводятся в компоненты системы без их изменения. Такой стиль программирования может быть реализован практически в любом языке программирования и совсем не обязательно требует наличия каркаса или библиотеки. Он требует лишь соблюдать определенную дисциплину в процессе программирования.

Кодирование

Всякий раз, когда вы пишете код, вы рискуете убавить ортогональность своего приложения. И если не контролировать постоянно не только свои действия, но и более крупный контекст приложения, то можно неумышленно сдублировать функциональные возможности в каком-нибудь другом модуле или выразить существующие знания дважды.

Ниже вкратце описываются некоторые методики, с помощью которых можно поддерживать ортогональность.

- *Поддерживайте свой код развязанным.* Пишите код экономно, создавая модули, не раскрывающие ничего лишнего другим модулям и не опирающиеся на их реализации. Старайтесь соблюдать закон Деметры, поясняемый в разделе "Тема 28. Развязывание" главы 5. Если требуется изменить состояние объекта, добейтесь того, чтобы он делал это автоматически. Подобным образом ваш код останется обособленным от реализации другого кода и тем самым повысятся шансы сохранить ортогональность того, что вы проектируете.
- *Избегайте глобальных данных.* Всякий раз, когда в вашем коде выполняется обращение к глобальным данным, он привязывается к другим компонентам, совместно использующим эти же данные. Даже глобальные переменные, предназначенные только для чтения, могут привести к осложнениям (например, в том случае, если срочно потребуется сделать ваш код многопоточным). В общем, если вы явно передаете любой требующийся контекст своим модулям, ваш код будет легче понять и сопровождать. В объектно-ориентированных приложениях контекст обычно передается в виде параметров конструкторам объектов. А в другом коде можно создать структуры, содержащие контекст, и передавать ссылки на них.
- Проектный шаблон Синглтон (Singleton), описанный в книге *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95], — это средство, гарантирующее существование лишь одного экземпляра объекта конкретного класса. Многие разработчики пользуются полученными таким образом одиночными объектами в качестве глобальных переменных (особенно в таких языках, как Java, где глобальные переменные иным образом не поддерживаются). Пользуйтесь, однако, синглтонами аккуратно, поскольку они также могут привести к ненужному связыванию.
- *Избегайте сходных функций.* Зачастую встречается ряд функций, очень похожих, например, тем, что они совместно используют общий код в начале и в конце, хотя имеют разные алгоритмы в центре. Такое дублирование кода служит явным признаком структурных недостатков. Более хорошую реализацию может предоставить проектный шаблон "Стратегия" (Strategy), [GHJV95].

Приобретите привычку постоянно осматривать свой код критическим взглядом. Ищите возможности улучшить его структуру и ортогональность. Такой процесс называется *рефакторингом* и настолько важен, что мы посвятили ему отдельный раздел "Тема 40. Рефакторинг" в главе 7.

Тестирование

Ортогонально спроектированную и реализованную систему легче тестировать. Взаимодействия компонентов такой системы формализованы и ограничены, и поэтому на уровне отдельных модулей можно произвести бо́льшую часть тестирования системы. И это хорошо, поскольку определить и выполнить модульное (или блочное) тестирование намного проще, чем комплексное. При этом предполагается, что подобные тесты должны проводиться автоматически как часть обычного процесса сборки (см. раздел "Тема 41. Тестировать, чтобы кодировать" главы 7).

Написание модульных тестов само по себе является интересной проверкой ортогональности. Что же требуется для составления и выполнения модульного теста? Приходится ли для этого импортировать львиную долю остального кода системы? Если да, то можно считать, что обнаружен модуль, который недостаточно отвязан от остальной системы.

Но устранение программных ошибок также является удобным моментом для оценивания ортогональности всей системы в целом. Когда вы обнаружите ошибку, оцените, насколько локализовано ее исправление. Изменяется ли при этом лишь один модуль, или же изменения распространяются по всей системе? Если вы вносите изменение, исправляет ли оно все ошибки или же загадочным образом возникают другие ошибки? Это удобная возможность задействовать автоматизацию. Если вы пользуетесь системой контроля версий исходного кода (а вы ею все равно станете пользоваться, как только прочитаете раздел "Тема 19. Контроль версий" главы 3), отмечайте исправления ошибок, когда будете снова проверять код после тестирования. И тогда вы сможете составлять и просматривать ежемесячные отчеты, анализируя тенденции в целом ряде исходных файлов, на которые повлияло исправление каждой ошибки.

Документация

Как ни странно, принцип ортогональности распространяется и на документацию. В данном случае ортогональность проявляется в плоскости координат содержания и представления. Если документация истинно ортогональная, ее внешний вид удается изменить коренным образом, совершенно не меняя содержание. Для этой цели в текстовых редакторах предоставляются таблицы стилей и макрокоманды. Мы лично предпочитаем пользоваться такой системой

разметки, как Markdown. При составлении документации мы уделяем основное внимание только содержанию, оставляя представление тому инструментальному средству, которое воспроизводит документацию[6].

Как уживаться с ортогональностью

Ортогональность тесно связана с принципом DRY, описанным ранее в разделе "Тема 9. DRY — пороки дублирования". Соблюдая принцип DRY, вы ищете возможность свести к минимуму дублирование в системе, а соблюдая принцип ортогональности — сократить взаимозависимость компонентов системы. Несмотря на то что само слово "ортогональность" как-то не укладывается в обычные представления программиста, придерживаясь принципа, который это слово обозначает, наряду с принципом DRY, вы непременно обнаружите, что разрабатываете более гибкую, понятную и простую для отладки, тестирования и сопровождения системы.

Если вас привлекли к работе над проектом, где остальные его участники отчаянно борются за то, чтобы внести изменения, и где каждое изменение может привести к тому, что дело не заладится и в четырех других местах, вспомните об описанном ранее кошмаре с управлением вертолетом. Вероятнее всего, такой проект спланирован и запрограммирован неортогонально.

И, если вы пилотируете вертолет, не ешьте рыбу...

Другие разделы, связанные с данной темой

- **Тема 3**. Программная энтропия, **глава 1** "Философия прагматизма".
- **Тема 8**. Сущность качественного проектирования.
- **Тема 11**. Обратимость.
- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 31**. Налог на наследование, **глава 5** "Гибкость или ломкость".
- **Тема 33**. Разрывание временного связывания, **глава 6** "Параллельность".
- **Тема 34**. Общее состояние — неверное состояние, **глава 6** "Параллельность".
- **Тема 36**. Классные доски, **глава 6** "Параллельность".

Задачи

- Выясните, чем инструментальные средства с графическим пользовательским интерфейсом отличаются от небольших, но легко объединяемых в

[6] На самом деле английское издание этой книги было написано в системе разметки Markdown и напечатано в типографии непосредственно из исходного файла Markdown.

командной строке утилит. Какая из этих разновидностей инструментальных средств оказывается более ортогональной и почему? Какой из них удобнее пользоваться в тех целях, для которых она непосредственно предназначена? Какую из них легче объединить с другими инструментальными средствами для решения новых задач? И какую из них легче изучить?

- В языке C++ поддерживается множественное наследование, тогда как в языке Java допускается реализация в классе нескольких интерфейсов, а в языке Ruby имеются миксины. Какое влияние оказывает применение этих языковых средств на ортогональность? Имеются ли какие-нибудь отличия в применении множественного наследования и многих интерфейсов, делегирования и наследования?

Упражнения

1. Допустим, вам было предложено реализовать построчное чтение исходного файла. Каждую строку вы должны разбить на поля. Какой из приведенных ниже определений псевдоклассов, по вашему мнению, является более ортогональным?[7]

```
class Split1 {
    constructor(fileName)   # открывает файл для чтения
    def readNextLine()      # переходит к следующей строке
    def getField(n)         # возвращает n-е поле из текущей строки
}
```

или

```
class Split2 {
    constructor(line)       # разбивает строку на поля
    def getField(n)         # возвращает n-е поле из текущей строки
}
```

2. Чем объектно-ориентированные и функциональные языки программирования отличаются в смысле ортогональности? Присущи ли эти отличия самим языкам или же только тому способу, каким люди ими пользуются?

[7] Возможные ответы на это и все остальные упражнения в данной книге см. в приложении B.

Тема 11 Обратимость

"Нет ничего опаснее идеи, если это все, что у вас есть".

— Эмиль-Огюст Шартье, псевдоним "Ален",
О религии (Propos sur la religion), 1938 г.

Инженеры предпочитают простые, однозначные решения задач. Контрольные работы по математике, позволяющие с полной уверенностью заявить, что $x = 2$, намного более удобны, чем неясные, хотя и страстные рассуждения о многих причинах Французской революции. Начальство готово согласиться с инженерами, что однозначные, простые вопросы изящно вписываются в электронные таблицы и проектные планы.

Если бы реальный мир пошел таким же образом навстречу! К сожалению, если x сегодня равно 2, то завтра x может быть равно 5, а на следующей неделе — 3. Ведь ничто не вечно, и если вы сильно полагаетесь на какой-то факт, то можете быть уверены, что он *все равно* изменится.

Реализовать что-нибудь можно всегда не одним, а несколькими способами. Кроме того, предоставить сторонний программный продукт может не один, а несколько поставщиков. Если вы вступаете в проект, недальновидно считая, что разработать его можно только *одним* способом, то будете неприятно удивлены. Многие проектные команды просто вынуждены пристально вглядываться в будущее по мере разворачивания событий. И вот тому наглядный пример:

> *"Но вы же сказали, чтобы мы пользовались базой данной XYZ! Мы запрограммировали проект уже на 85%, и теперь уже нельзя ничего изменить! — запротестовал программист.*
>
> *— Извините, но наша компания решила перевести все наши проекты на базу данных PDQ как на стандартную. И я не могу с этим ничего поделать. Нам просто придется перепрограммировать проект, а всем вам — работать и в выходные, впредь до особого распоряжения".*

Изменения совсем не обязательно должны быть до такой степени коренными или даже немедленными. Но со временем, по мере развития проекта, вы можете оказаться в безвыходном положении, если с каждым очень важным решением проектная команда берет на себя обязательство достичь более мелкой цели, воплощающей суженную версию реально с меньшим числом вариантов выбора.

И к тому времени, когда многие важные решения будут сделаны, цель окажется настолько мелкой, что вы непременно промахнетесь, если она сместится,

изменится направление ветра или японская бабочка махнет в Токио крылышками[8]. А промахнуться вы можете очень сильно.

Все дело в том, что важные решения не так легко повернуть вспять. Как только вы решите воспользоваться базой данных конкретного поставщика, архитектурным шаблоном или определенной моделью развертывания, тем самым вы возьмете на себя обязательство выполнить порядок действий, который нельзя отменить, разве что ценой немалых затрат.

Обратимость

Многие темы в этой книге посвящены особенностям производства гибкого, адаптируемого программного обеспечения. Если придерживаться приведенных в них рекомендаций, особенно принципа DRY, описанного ранее в разделе "Тема 9. DRY — пороки дублирования", развязывания, обсуждаемого в разделе "Тема 28. Развязывание" главы 5, а также применения внешней конфигурации, как поясняется в разделе "Тема 32. Конфигурирование" главы 5, то не придется принимать много важных решений, которые могут оказаться необратимыми. И это совсем неплохо, поскольку мы не всегда принимаем наилучшие решения с первого раза. Придерживаясь определенной технологии, мы лишь впоследствии обнаруживаем, что не можем нанять достаточно людей с требующимися навыками. Мы замыкаемся на определенном поставщике стороннего программного обеспечения, после чего его перекупает конкурент. Требования, пользователи и аппаратные средства меняются быстрее, чем нам удается разработать программное обеспечение.

Допустим, на ранней стадии работы над проектом вы решили воспользоваться базой данных от поставщика A. Гораздо позже, на стадии тестирования производительности, вы обнаруживаете, что выбранная вами база данных работает слишком медленно, тогда как документная база данных от поставщика B работает быстрее. Выходит, что вам просто не повезло, как это бывает в большинстве обыкновенных проектов. Зачастую обращения к сторонним программным продуктам глубоко вплетены в исходный код. Но если вы *действительно* обобщите саму идею базы данных, причем до такой степени, чтобы она просто предоставляла услуги хранения данных в качестве службы, то у вас появится удобная возможность поменять лошадей на переправе.

Аналогично допустим, что ваш проект начинается как браузерное приложение, но на более поздней его стадии в отделе сбыта решают, что им на самом деле требуется мобильное приложение. Насколько трудно вам будет перестро-

[8] Попробуйте внести небольшое изменение в одно из входных воздействий нелинейной или хаотической системы. В итоге вы можете получить существенно иной и зачастую непредсказуемый результат. Обыкновенная японская бабочка, машущая крылышками в Токио, может стать началом целой цепочки событий, которая закончится торнадо в штате Техас. Не напоминает ли это какой-нибудь известный вам проект?

иться по ходу дела? В идеальном случае это не должно особенно повлиять на ваш проект — по крайней мере, на его серверную сторону. Вам придется лишь удалить HTML-разметку воспроизведения, заменив ее соответствующим API.

Ошибка кроется в допущении, что любое решение принимается окончательно, а следовательно, в неготовности к непредвиденным обстоятельствам, которые могут так или иначе возникнуть. Вместо того чтобы принимать решения бесповоротно, как высеченные на камне, рассматривайте их как начертанные на береговом песке. Ведь большая волна может нахлынуть в любой момент и смыть их без следа.

Совет 18 Окончательных решений не существует

Гибкая архитектура

Несмотря на то что многие программисты стараются сохранять как можно более гибким сам *код*, необходимо также подумать и об удобстве сопровождения на уровне архитектуры, развертывания и интеграции сторонних программных продуктов. С начала нынешнего столетия и вплоть до момента написания этих строк появились следующие "нормы наилучшей практики" построения серверных архитектур.

- Большая груда железа.
- Образования больших груд железа.
- Кластеры недорогого стандартного оборудования, выровненные по нагрузке.
- Облачные виртуальные машины, выполняющие приложения.
- Облачные виртуальные машины, выполняющие службы.
- Контейнеризованные варианты облачных виртуальных машин.
- Бессерверные приложения с облачной поддержкой.
- И как неизбежность, вполне очевидный возврат к большим грудам железа для решения некоторых задач.

Можете дополнить этот список самыми последними веяниями и благоговейно считаться с ними как с каким-то чудом, благодаря которому что-то вообще работает. Можно ли вообще запланировать все это многообразие архитектурных решений? Никак нельзя.

Единственное, что можно действительно сделать, — это упростить изменения. В частности, скрывайте сторонние API за собственными уровнями абстракции. Разбивайте свой код на компоненты, даже если их придется развертывать на одном крупном сервере. Ведь это намного проще, чем разбивать на части одно монолитное приложение. Мы убедились в этом на собственном горьком

опыте. Наконец, приведем еще один полезный совет, хотя он и не имеет непосредственного отношения к рассматриваемому здесь вопросу обратимости.

Совет 19 Остерегайтесь увлекаться новомодными веяниями

Никто не знает, что сулит нам будущее! Поэтому заставляйте свой код "трястись", когда это возможно, и живо "крутиться", когда это необходимо, как в танце рок-н-рол.

Другие разделы, связанные с данной темой

- **Тема 8**. Сущность качественного проектирования.
- **Тема 10**. Ортогональность.
- **Тема 19**. Контроль версий, **глава 3** "Основные инструментальные средства".
- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 45**. Западня требований, **глава 8** "До начала проекта".
- **Тема 51**. Начальный набор инструментальных средств программиста-прагматика, **глава 9** "Прагматичные проекты".

Задачи

- А теперь уделите немного времени квантовой механике, проведя мысленный эксперимент, называемый "котом Шрёдингера". Представьте, что вы закрыли кота в ящике с радиоактивной частицей, которая распадется на две другие частицы с вероятностью 50%. Если она распадется, то убьет кота, а если не распадется, то кот останется живым. Так погибнет кот или выживет? По мнению Шрёдингера, верно и то и другое — по крайне мере, до тех пор, пока ящик закрыт. Всякий раз, когда происходит субъядерная реакция с двумя возможными исходами, мир распадается на два других: один, где событие произошло, а другой, где оно не произошло. Кот выживет в одном мире и погибнет в другом. И вы узнаете, в каком *именно* мире находитесь сами, лишь тогда, когда откроете ящик.

  Не удивительно, что программировать на будущее так трудно.

  Но подумайте об эволюции своего кода как о ящике, в котором полно котов Шрёдингера, когда всякое решение может привести к разным версиям кода в будущем. Сколько возможных вариантов будущего развития способен поддерживать ваш код? Какие из них наиболее вероятны? Насколько трудно будет их поддерживать, когда настанет время? Отважитесь ли вы открыть ящик?

Тема 12 Трассирующие пули

"Заряжай! Огонь! Цельсь!..."

Неизвестный сержант

При разработке программного обеспечения нередко речь заходит о попадании в цель. И хотя мы не стреляем по мишени на огневом рубеже, такая метафора все же оказывается очень удобной. В частности, любопытно выяснить, *как* попасть в цель в сложном и постоянно меняющемся мире.

Ответ на этот вопрос, безусловно, зависит от характера того устройства, которое вы нацеливаете. И у вас лишь один шанс прицелиться и лишь затем убедиться, попали ли вы в яблочко, удовлетворив своего клиента. Но есть и лучший способ.

Вам, вероятно, приходилось не раз видеть в кино, по телевизору или в видеоиграх, как люди стреляют из пулеметов? В подобных сценах траектории полета пуль нередко наблюдаются как яркие полосы света в атмосфере. Такие полосы оставляют за собой трассирующие пули.

Трассирующие пули устанавливаются в пулеметную ленту через определенные промежутки вместе с обычными боеприпасами. Когда они выстреливаются, содержащийся в них фосфор загорается, оставляя огненный след, тянущийся от пулемета к тому месту, куда они попадают. Если трассирующие пули попадут в цель, то в нее попадут и обычные пули. Солдаты делают очереди трассирующими пулями для того, чтобы уточнить прицеливание, поскольку они дают вполне прагматичную, оперативную реакцию в условиях реальной стрельбы.

Аналогичный принцип можно применить и к проектам, особенно в том случае, если приходится проектировать нечто совершенно новое, не имея прежнего опыта. Мы пользуемся термином *разработка методом трассирующих пуль*, чтобы наглядно показать потребность в немедленной реакции в действительных условиях с подвижной конечной целью.

Проектируя совершенно новую систему, вы, подобно пулеметчикам, пытаетесь попасть в цель, находясь в полной темноте. Ваши пользователи вообще не сталкивались прежде с такой системой, поэтому их требования не совсем ясны. А вам приходится решать уравнение со многими неизвестными, поскольку вы можете пользоваться незнакомыми прежде алгоритмами, методиками, языками или библиотеками. Для завершения подобных проектов требуется время, и поэтому можно смело предположить, что ваша рабочая среда изменится задолго до того, как вы завершите работу над своим проектом.

Классический подход к проектированию системы состоит в том, чтобы составить ее спецификацию во всех подробностях, исписав стопки бумаги, аккуратно составив перечень всех требований, связав все неизвестные и наложив

ограничения на рабочую среду. При этом стрельба по конечной цели ведется с точным расчетом: сначала делается один крупный предварительный расчет, а затем производится выстрел с надеждой попасть в цель. Но программисты-прагматики предпочитают в подобных случаях пользоваться программным эквивалентом трассирующих пуль.

Код, сверкающий в темноте

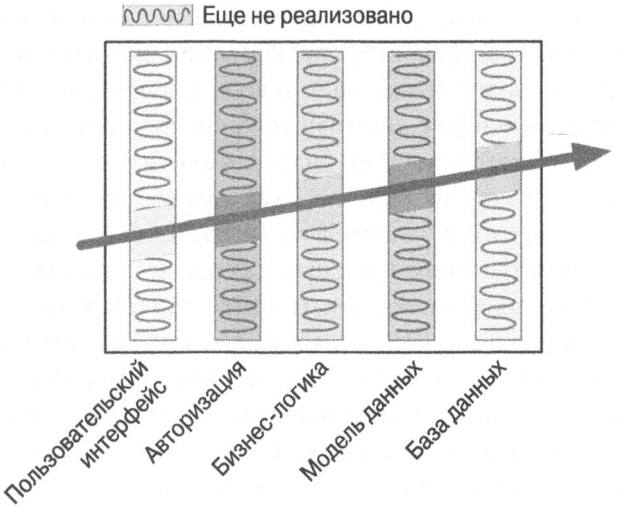
Трассирующие пули удобны тем, что они действуют в тех же условиях и при тех же самых ограничениях, что и настоящие пули. Они быстро достигают цели, так что стрелок сразу же получает ответную реакцию. А с практической точки зрения они являются относительно недорогим решением. Чтобы добиться аналогичного результата в коде, мы ищем нечто, быстро наглядно и неоднократно приводящее нас от исходного требования к некоторому свойству конечной системы.

Ищите сначала самые важные требования, определяющие систему, а затем те участки, где у вас имеются сомнения и где вы видите наибольшие риски. Расставьте далее приоритеты в своей разработке, чтобы они указывали на те участки, с которых следует в первую очередь начать программирование.

Совет 20 Пользуйтесь методом трассирующих пуль для отыскания цели

В действительности, принимая во внимание сложность организации проекта в современных условиях с массой внешних зависимостей и инструментальных средств, метод трассирующих пуль приобретает еще большее значение. Для разработчиков первая трассирующая пуля просто состоит в том, чтобы *создать* проект, *добавить* в него код, выводящий пробное сообщение типа "Здравствуй, мир!", а затем *убедиться*, что этот код компилируется и выполняется. Далее следует найти участки неопределенности во всем приложении и добавить скелетный вариант, требующийся для того, чтобы привести его в работоспособное состояние.

Рассмотрим приведенную ниже диаграмму, где показаны пять архитектурных уровней проектируемой системы. Нас интересует, каким образом они интегрируются, поэтому найдем простое функциональное средство, позволяющее испытать их вместе. Диагональная линия на этой диаграмме показывает путь, который некоторое функциональное средство проходит через исходный код. Чтобы воплотить все это в жизнь, достаточно реализовать окрашенные сплошным цветом участки диаграммы на каждом уровне, тогда как все, что обозначено на ней завитками, будет реализовано в дальнейшем.



Однажды нам пришлось вести сложный маркетинговый проект с базой данных архитектуры "клиент–сервер". Часть требований к данному проекту состояла в предоставлении возможности составлять и выполнять временные запросы. На серверах был установлен целый ряд реляционных и специализированных баз данных, а клиентский пользовательский интерфейс был написан на произвольно выбранном языке A, в котором применялся ряд библиотек, написанных на другом языке для обеспечения интерфейса с серверами. Пользовательский запрос сохранялся на сервере в виде Lisp-подобной записи, прежде чем быть преобразованным в оптимизированный запрос SQL непосредственно перед его выполнением. В данном проекте было много неизвестных и самых разных сред, причем никто не знал точно, как должен себя вести пользовательский интерфейс.

Это была прекрасная возможность воспользоваться трассирующим кодом. С этой целью мы разработали каркас клиентской части системы, библиотеки для представления запросов и структуру для преобразования хранимых запросов в фактические запросы конкретной базы данных. Затем мы собрали все это вместе и проверили на работоспособность. И хотя мы могли направить в этом первоначально собранном варианте лишь запрос, в котором перечислялись все строки в таблице базы данных, мы все же смогли убедиться, что пользовательский интерфейс способен взаимодействовать с библиотеками, сами библиотеки — выполнять сериализацию и десериализацию запроса, а сервер — формировать запрос SQL из полученного результата. В течение последующих месяцев мы постепенно наполнили деталями эту основную структуру, добавили новые функциональные возможности, параллельно нарастив каждый компонент трассирующего кода. По мере внедрения нового типа запроса в пользовательский интерфейс библиотека разрасталась, а формирование запроса SQL усложнялось.

Трассирующий код пишется не для одноразового применения, а надолго. Он содержит все проверки ошибок, структурирование, документирование и самопроверку, как и любой фрагмент выходного кода. Он просто не полностью функционален. Но как только будет достигнуто сквозное соединение компонентов вашей системы, вы сможете проверить, насколько близки вы к желанной цели, и вносить коррективы по мере надобности. Как только цель будет достигнута, добавить требующиеся функциональные возможности не составит особого труда.

Разработка методом трассирующих пуль согласуется с представлением о том, что проект никогда не завершается. В нем всегда найдется место для изменений и добавлений новых функций. Это постепенный подход. Обычно его альтернативой является тяжеловесный подход, при котором исходный код разбивается на модули, программируемые по отдельности. Эти модули затем объединяются в сборки, а те, в свою очередь, объединяются дальше до тех пор, пока однажды не получится готовое приложение. И лишь тогда приложение может быть представлено как единое целое пользователям для последующего тестирования.

У подхода к разработке с помощью трассируемого кода имеются следующие преимущества.

- **Пользователи могут увидеть нечто работоспособное уже на ранней стадии разработки.** Если вы своевременно известите пользователей о том, что вы делаете (см. раздел "Тема 52. Доставляйте своим пользователям удовольствие" главы 9), они будут заранее знать, что увидят нечто еще не зрелое. Следовательно, их не разочарует нехватка функциональных возможностей, и они будут восторженно приветствовать видимый прогресс в разработке создаваемой для них системы. Они также смогут внести свой вклад в проект по ходу работы над ним, повысив тем самым свою заинтересованность в нем. Те же самые пользователи, вероятнее всего, подскажут вам, насколько вы приблизились к желанной цели на каждом последующем шаге.
- **Разработчики создают структуру, чтобы работать в ней.** Больше всего пугает пустой лист бумаги, на котором вообще ничего не написано. Если вы выяснили все сквозные взаимодействия в своем приложении и воплотили их в коде, то вашей команде не придется ничего особенного брать с потолка. Это способствует повышению общей производительности труда и согласованности действий в команде.
- **У вас есть платформа для интеграции.** Как только вы осуществите сквозное соединение компонентов системы, вы получите в свое распоряжение такую среду, в которую можно будет благополучно внедрить новые фрагменты кода, как только они пройдут модульное тестирование. Вместо того чтобы пытаться интегрировать все одним махом, вы можете выполнять интеграцию каждый день, а то и несколько раз в день. В этом случае влияние каждого нового изменения становится более очевидным, тогда как

взаимодействия — более ограниченными, а следовательно, отладка и тестирование выполняются быстрее и точнее.

- **Вам есть что продемонстрировать.** Спонсоры проекта и высшее начальство обычно требуют, чтобы им показали демонстрационные версии в самые неподходящие для этого моменты. Благодаря трассирующему коду у вас всегда будет что предъявить.
- **Вы лучше ощущаете прогресс.** Работая над проектом методом трассирующих пуль, разработчики занимаются прецедентами использования по очереди. Справившись с одним прецедентом использования, они переходят к другому. Благодаря этому оказывается намного проще количественно оценить производительность и продемонстрировать пользователям прогресс в работе над проектом. А поскольку каждая индивидуальная разработка ведется в более мелких масштабах, то можно избежать создания монолитных блоков кода, о завершенности которых на 95% разработчики отчитываются с недели на неделю.

Трассирующие пули не всегда попадают в цель

Обычно трассирующие пули показывают, куда вы попали. Но это не всегда может быть желанная цель. В таком случае вам придется поправлять прицел до тех пор, пока вы не попадете точно в цель. И в этом основная идея трассирующих пуль.

То же самое относится и к трассирующему коду. Вы пользуетесь данным методом в обстоятельствах, когда не знаете точно, куда движетесь. И поэтому не удивляйтесь, если, сделав две первые попытки, вы промахнетесь, а пользователь скажет: "Это совсем не то, что я имел в виду", нужные вам данные окажутся недоступны именно тогда, когда они вам потребуются, или же упадет производительность. В таком случае измените результат, которого вы уже добились, чтобы перенести его поближе к цели, и будьте благодарны за то, что вы воспользовались данной методикой разработки. Ведь небольшой фрагмент кода обладает малой инертностью, так что его можно очень быстро и легко изменить. Помимо этого, вы сумеете собрать отзывы о своем приложении, а затем быстро и дешево сгенерировать новую, более точную его версию. А поскольку все основные компоненты вашего приложения представлены в трассирующем коде, то пользователи могут быть уверены, что все, что они увидят, основывается на чем-то реальном, а не только на бумажной спецификации.

Трассирующий код в сравнении с прототипированием

Можно решить, что рассматриваемый здесь метод трассирующего кода оказывается не более, чем прототипированием, только более воинственно названным. Ведь с помощью прототипа вы нацеливаетесь на исследование конкретных

свойств конечной системы. А настоящий прототип помогает вам отбросить все, что вы связали вместе, когда опробовали какой-нибудь принцип, а затем перепрограммировать, руководствуясь извлеченными вами уроками.

Допустим, вы разрабатываете приложение, помогающее грузоотправителям выяснить, как упаковать ящики нестандартных размеров в контейнеры. Среди прочих трудностей, пользовательский интерфейс такого приложения должен быть интуитивным, тогда как алгоритмы, применяемые вами для определения оптимального варианта упаковки, довольно сложные.

Вы могли бы, конечно, создать с помощью специального инструментального средства прототип пользовательского интерфейса для его демонстрации конечному пользователю, а также запрограммировать его настолько, чтобы он реагировал на действия пользователя. И как только пользователь согласился бы с компоновкой прототипа, вы могли бы отбросить его и перепрограммировать пользовательский интерфейс — на этот раз уже с бизнес-логикой, используя целевой язык. Аналогично вы могли бы прототипировать целый ряд алгоритмов, выполняющих конкретную задачу упаковки. Кроме того, вы могли бы написать функциональные тесты на таком неприхотливом языке высокого уровня, как, например, Python, а тесты производительности на низком уровне — на каком-нибудь простом языке, близком к машинному. Но в любом случае, как только вы приняли бы решение, вам пришлось бы начинать все заново, программируя алгоритмы в среде их конечного применения, взаимодействующей с реальным миром. В этом, собственно, и состоит *прототипирование*, польза от которого несомненна.

Метод трассирующего кода направлен на разрешение другого затруднения, когда вам требуется знать, насколько удачно все приложение связано вместе. Ведь пользователям вашего приложения нужно показать, каким образом происходит его взаимодействие с ними на практике, а разработчикам — предоставить скелетный вариант его архитектуры, который можно было бы наполнить конкретным кодом. В таком случае вы можете написать трассирующий код, состоящий из тривиальной реализации алгоритма упаковки контейнера (возможно, действующего по принципу "первым пришел, первым обслужен"), а также простой, но работоспособный пользовательский интерфейс. И как только вы скомпонуете вместе все компоненты своего приложения, то получите каркас, который можно показать как пользователям, так и разработчикам. Со временем вы дополните этот каркас новыми функциональными возможностями, заменив ими суррогатные процедуры. Но сам каркас останется нетронутым, а вы будете уверены, что система и дальше поведет себя таким же образом, как и по завершении первого трассирующего кода.

Рассмотренное выше отличие настолько важно, что его стоит еще раз повторить. Прототипирование приводит к генерации одноразового кода. А трассирующий код — скудный, но завершенный, образующий скелетный вариант

конечной системы. Прототипирование можно рассматривать как своего рода разведку местности и добывание разведывательных данных, прежде чем будет выпущена первая трассирующая пуля.

Другие разделы, связанные с данной темой

- **Тема 13**. Прототипы и памятные записки.
- **Тема 27**. Не опережайте свет фар вашего автомобиля, **глава 4** "Прагматичная паранойя".
- **Тема 40**. Рефакторинг, **глава 7** "По ходу кодирования".
- **Тема 49**. Прагматичные команды, **глава 9** "Прагматичные проекты".
- **Тема 50**. Кокосами не обойтись, **глава 9** "Прагматичные проекты".
- **Тема 51**. Начальный набор инструментальных средств программиста-прагматика, **глава 9** "Прагматичные проекты".
- **Тема 52**. Доставляйте своим пользователям удовольствие, **глава 9** "Прагматичные проекты".

Тема 13 Прототипы и памятные записки

Прототипы применяются во многих отраслях промышленности для опробования конкретных идей, ведь создавать прототипы намного дешевле, чем производить полномасштабные образцы продукции. Например, автомобилестроители могут создавать самые разные прототипы новой конструкции, используя каждый из них для испытания конкретного свойства автомобиля: аэродинамики, стилевого оформления, конструктивных характеристик и т.д. А приверженцы старых традиций пользуются моделью из глины для испытаний в аэродинамической трубе, тогда как для отдела дизайна может вполне подойти пробковое дерево и клейкая лента. Менее романтичным выглядит моделирование на экране компьютерного монитора или в виртуальной реальности, хотя оно и обходится намного дешевле. В последнем случае рискованные и ненадежные элементы конструкции можно испытать, не прибегая к построению настоящего образца.

Мы создаем прототипы программ аналогичным образом и в тех же целях: чтобы проанализировать и выявить риск, а также дать возможность внести коррективы, существенно снизив затраты. Как и автомобилестроители, мы можем нацелить прототип на проверку одного или нескольких конкретных аспектов проекта.

Мы привыкли считать, что прототипы основываются на коде, хотя это не всегда именно так. Как и автомобилестроители, мы можем построить прототипы из различных материалов. В частности, памятные записки на клейких лис-

точках бумаги как нельзя лучше подходят для прототипирования таких динамических элементов, как рабочий поток и прикладная логика. А пользовательский интерфейс можно прототипировать в виде рисунка на белой доске или нефункционального макета, нарисованного в программе раскраски или в конструкторе интерфейса.

Прототипы служат для поиска ответов лишь на немногие вопросы, поэтому они обходятся намного дешевле и создаются быстрее, чем приложения, передаваемые в эксплуатацию. В коде прототипа можно пренебречь подробностями, которые не важны в данный момент, но впоследствии могут стать очень важными для пользователей. Так, если создается прототип пользовательского интерфейса, в таком случае можно довольствоваться и неточными результатами или данными. А если исследуются вычислительные или производительные особенности приложения, то можно довольствоваться и весьма скромным пользовательским интерфейсом, а возможно, и вообще обойтись без него.

Но если вы окажетесь в такой среде, где *нельзя* пренебречь подробностями, то спросите себя: а стоит ли вообще создавать прототип? В таком случае вам, может быть, больше подойдет разработка методом трассирующих пуль (см. выше раздел "Тема 12. Трассирующие пули").

Что подлежит прототипированию

Что можно выбрать для исследования с помощью прототипа? Все, что несет в себе риск, все, что еще не было опробовано, или то, что крайне важно для конечной системы, а также все, что не проверено, экспериментально, сомнительно или то, в чем нет никакой уверенности. Итак, прототипированию подлежит следующее.

- Архитектура.
- Новые функциональные возможности уже имеющейся системы.
- Структура или содержание внешних данных.
- Сторонние инструментальные средства или компоненты.
- Вопросы производительности.
- Оформление пользовательского интерфейса.

Прототипирование означает опыт познания. Его ценность состоит не в производимом коде, а в извлекаемых уроках. И в этом заключается весь смысл прототипирования.

Совет 21 Создавайте прототипы для обучения

Как пользоваться прототипами

Ниже перечислены подробности, которыми можно пренебречь, создавая прототипы.

- **Правильность.** Там, где это уместно, возможно, удастся воспользоваться фиктивными данными.
- **Завершенность.** Прототип может функционировать лишь в весьма ограниченном смысле, а возможно, и с единственным предварительно выбранным фрагментом входных данных и одним пунктом меню.
- **Надежность.** Проверка ошибок вряд ли должна быть полной, а возможно, и вообще должна отсутствовать. Если отклониться от заранее определенного пути, прототип может потерпеть крах и сгореть ярким пламенем. И это нормально.
- **Стиль программирования.** Исходный код прототипа вряд ли следует снабжать комментариями или документировать, хотя и можно написать целые стопки документации, опираясь на свой опыт работы с прототипом.

В прототипах подробности не принимаются во внимание, которое сосредоточено в основном на конкретных свойствах исследуемой системы, и поэтому прототипы можно реализовать на языке сценариев (например, Python или Ruby), который по своему уровню выше, чем другие языки, применяемые в остальной части проекта, поскольку они могут лишь стать помехой. Продолжить дальнейшую разработку можно как на языке, выбранном для создания прототипа, так и перейти на другой язык. Ведь в конечном итоге прототип все равно будет отброшен за ненадобностью.

Для прототипирования пользовательского интерфейса можно воспользоваться инструментальным средством, позволяющим сосредоточиться на внешнем виде и/или взаимодействии интерфейса с пользователем, не особенно беспокоясь о коде или разметке. Языки сценариев вполне пригодны и в качестве связующего звена для объединения низкоуровневых фрагментов в новые комбинации. Применяя такой подход, можно очень быстро собрать существующие компоненты в новые конфигурации, чтобы проверить их на работоспособность.

Прототипирование архитектуры

Многие прототипы создаются для моделирования всей рассматриваемой системы. В отличие от метода трассирующих пуль, индивидуальные модули совсем не обязательно должны как-то по-особому функционировать в прототипной системе. В действительности для прототипирования архитектуры можно даже не программировать, поскольку это можно сделать на белой доске, листках для памятных записок или карточках для записей. В данном случае требуется выяснить, каким образом система связана в единое целое, не обращая, опять же,

внимания на подробности. Ниже перечислен ряд конкретных вопросов, которые требуется выяснить, рассматривая архитектурный прототип.

- Вполне ли определены и приемлемы области ответственности основных компонентов системы?
- Вполне ли определены взаимодействия основных компонентов системы?
- Сведено ли к минимуму связывание?
- Можно ли выявить потенциальные источники дублирования?
- Насколько приемлемы определения интерфейсов и ограничения?
- Имеется ли у каждого модуля путь доступа к данным, которые требуются ему во время выполнения? Имеет ли каждый модуль доступ *именно тогда*, когда он требуется?

Последний ряд вопросов приносит больше всего неожиданностей и самых ценных результатов из опыта прототипирования.

Как *не* следует пользоваться прототипами

Прежде чем приступить к любому прототипированию на основе кода, убедитесь, что все заинтересованные лица ясно отдают себе отчет, что вы собираетесь писать одноразовый код. Ведь прототипы могут быть обманчиво привлекательны для тех людей, которые не знают, что это всего лишь прототипы. Поэтому вы должны *очень* ясно дать понять, что данный код одноразовый, незавершенный и непригодный для завершения.

Кажущаяся завершенность демонстрируемого прототипа может легко ввести в заблуждение, а попечители проекта или руководство могут даже настаивать на развертывании прототипа (или его детища), если вы не дадите ясно понять, чего именно следует ожидать от него. Напомните им, что вы можете построить отличный прототип нового автомобиля из пробкового дерева и клейкой ленты, но поехать на нем в часы пик вряд ли удастся!

Если вы остро ощущаете, что назначение прототипного кода в вашей среде или культуре может быть понято превратно, в таком случае вам лучше выбрать метод трассирующих пуль. В конечном счете у вас будет прочный каркас в качестве основания для последующей разработки.

Надлежащим образом применяемые прототипы позволяют сэкономить немало времени, средств и трудов, выявляя и корректируя потенциальные проблемные места на ранней стадии цикла разработки, когда исправление ошибок делается просто и обходится недорого.

Другие разделы, связанные с данной темой

- **Тема 12.** Трассирующие пули.
- **Тема 14.** Предметно-ориентированные языки.

- **Тема 17**. Игры в скорлупки, **глава 3** "Основные инструментальные средства".
- **Тема 27**. Не опережайте свет фар вашего автомобиля, **глава 4** "Прагматичная паранойя".
- **Тема 37**. Прислушивайтесь к своим инстинктам, **глава 7** "По ходу кодирования".
- **Тема 45**. Западня требований, **глава 8** "До начала проекта".
- **Тема 52**. Доставляйте своим пользователям удовольствие, **глава 9** "Прагматичные проекты".

Упражнения

3. В отделе сбыта предложили обсудить вместе с вами несколько вариантов оформления веб-страниц, в том числе карты ссылок, выбираемые щелчком кнопкой мыши для перехода на другие страницы, и т.д. Но они не могут выбрать модель для изображения на карте ссылок из нескольких вариантов: автомобиля, телефона и дома. У вас имеется список целевых страниц и их содержимого, и поэтому им хотелось бы посмотреть несколько прототипов. Кстати, вам на это дается 15 минут. Какими инструментальными средствами вы могли бы для этого воспользоваться?

Тема 14 Предметно-ориентированные языки

"Границы моего языка означают границы моего мира".

Людвиг Витгенштейн[9]

Языки программирования компьютеров оказывают влияние на то, *как* мы осмысливаем задачу и как понимаем общение. У каждого такого языка имеется перечень средств, означаемых специальными терминами вроде статической или динамической типизации, раннего или позднего связывания, функционального или объектно-ориентированного программирования, моделей наследования, примесей, макрокоманд, причем все они могут предложить или затруднить определенные решения. Выработка решения, принимая во внимание особенности программирования на языке C++, может привести к иным результатам, чем решение, основанное на мышлении в стиле языка Haskell, и наоборот. Но в то же время и язык предметной области может предложить решение для программирования, что, на наш взгляд, важнее.

[9] Ludwig Wittgenstein — австрийский философ и логик XX века. — *Примеч. пер.*

Мы всегда пытаемся писать код, пользуясь словарем из прикладной области (см. раздел "Ведение словаря" главы 8 "До начала проекта"). Иногда программисты-прагматики могут перейти на следующий уровень и сразу приступить к программированию, пользуясь словарем, синтаксисом и семантикой (т.е. языком) предметной области.

Совет 22 Программируйте близко к предметной области

Некоторые предметно-ориентированные языки

Итак, рассмотрим несколько примеров, в которых было сделано именно то, что и рекомендуется в приведенном выше совете.

RSpec

RSpec — это библиотека для тестирования кода[10], написанного на языке Ruby. Она вдохновила на создание версий большинства других современных языков. Тест в RSpec преследует цель отразить ожидаемое поведение проверяемого кода, как показано в приведенном ниже примере.

```
describe BowlingScore do
    it "totals 12 if you score 3 four times" do
        score = BowlingScore.new
        4.times { score.add_pins(3) }
        expect(score.total).to eq(12)
    end
end
```

Cucumber

Cucumber — это язык программирования[11], позволяющий составлять тесты нейтральным способом. Для выполнения тестов используется версия Cucumber, подходящая для применяемого языка. Чтобы поддерживать такой же синтаксис, как и у нейтрального языка, придется написать также конкретные сопоставители, распознающие фразы и извлекающие параметры для тестов. Ниже приведен характерный тому пример.

```
Feature: Scoring
Background:
    Given an empty scorecard
Scenario: bowling a lot of 3s
    Given I throw a 3
    And I throw a 3
    And I throw a 3
    And I throw a 3
    Then the score should be 12
```

[10] См. по адресу `https://rspec.info`.

[11] См. по адресу `https://cucumber.io/`.

Тесты составлены на языке Cucumber таким образом, чтобы их могли прочитать клиенты программного обеспечения (хотя такое очень редко происходит на практике; возможные причины этого поясняются в приведенной ниже врезке).

Почему многие корпоративные пользователи не читают функции Cucumber

Одна из причин, по которым классический подход "собирание требований, проектирование, программирование, выпуск" оказывается недейственным, состоит в том, что он зиждется на понятии о том, что требования известны, хотя такое бывает редко. Корпоративные пользователи весьма смутно представляют, чего они хотят достичь, но им неизвестны, да и не важны подробности. И в этом отчасти состоит ценность разработчиков, поскольку они постигают внутренним чутьем намерения своих клиентов и воплощают их в коде.

Следовательно, когда вы настоятельно просите корпоративного пользователя окончательно утвердить требования или согласовать их с рядом функций Cucumber, это все равно, что заставить его проверить правописание в сочинении, написанном на шумерском языке. Он внесет ряд случайных изменений в требования, чтобы не уронить свое лицо, и утвердит их, чтобы побыстрее выпроводить вас из своего учреждения.

Лучше предоставьте корпоративным клиентам работоспособный код, чтобы они могли опробовать его на практике. Именно тогда и проявятся их реальные потребности.

Маршруты Phoenix

Во многих каркасах веб-приложений имеется средство маршрутизации, преобразующее входящие HTTP-запросы в функции их обработки в коде. Ниже приведен характерный тому пример из Phoenix[12].

```
scope "/", HelloPhoenix do
    pipe_through :browser # использовать стандартный стек браузера

    get "/", PageController, :index
    resources "/users", UserController
end
```

[12] См. по адресу `https://phoenixframework.org/`.

В данном примере запросы, начинающиеся со знака "/", должны пройти через ряд фильтров, пригодных для браузеров. Сам запрос ресурса "/" будет обработан функцией `index()` из модуля `PageController`. А в модуле `UsersController` реализуются все функции, требующиеся для управления ресурсом, доступным по следующему URL: `/users`.

Ansible

Ansible — это инструментальное средство[13], предназначенное для конфигурирования программного обеспечения, как правило, на целом ряде удаленных серверов. С этой целью оно читает предоставляемую ему спецификацию, а затем делает все необходимое, чтобы зеркально отобразить ее на них. Спецификация может быть написана на YAML[14] — языке, предназначенном для построения структур данных из текстовых описаний. Так, в приведенном ниже примере обеспечивается установка веб-сервера nginx на удаленных серверах, его запуск по умолчанию, а также применение предоставляемого файла конфигурации.

```
---
- name: install nginx
  apt: name=nginx state=latest

- name: ensure nginx is running (and enable it at boot)
  service: name=nginx state=started enabled=yes

- name: write the nginx config file
  template: src=templates/nginx.conf.j2 dest=/etc/nginx/nginx.conf
  notify:
  - restart nginx
```

Характеристики предметно-ориентированных языков

Рассмотрим приведенные выше примеры более подробно. В частности, примеры теста RSpec и маршрутизатора Phoenix написаны на их базовых языках (Ruby и Elixir). В них употребляется довольно замысловатый код, включающий в себя метапрограммирование и макрокоманды, но в конечном счете они компилируются и выполняются как обычный код.

Тесты Cucumber и конфигурации Ansible написаны на их собственных языках. В частности, тест Cucumber преобразуется в выполняемый код или в структуру данных, тогда как спецификации Ansible всегда преобразуются в структуру данных, выполняемую непосредственно в Ansible.

В итоге код теста RSpec и маршрутизатора Phoenix внедряется в выполняемый прикладной код в виде подлинных расширений его словаря. А тесты Cucumber и конфигурации Ansible *читаются* прикладным кодом и преобра-

[13] См. по адресу `https://www.ansible.com/`.

[14] См. по адресу `https://yaml.org/`.

зуются в некоторую форму, пригодную для употребления в прикладном коде. Тест RSpec и маршрутизатор Phoenix можно назвать примерами *внутренних* предметно-ориентированных языков, тогда как тест Cucumber и конфигурацию Ansible — примерами *внешних* предметно-ориентированных языков.

Компромисс между внутренними и внешними предметно-ориентированными языками

В общем случае во внутреннем предметно-ориентированном языке могут выгодно использоваться средства его базового языка, поскольку создаваемый предметно-ориентированный язык обладает бо́льшими функциональными возможностями, которые даются даром. Например, для автоматического создания набора тестов RSpec можно воспользоваться некоторым кодом Ruby. В этом случае можно проверить очки на промахи или попадания:

```
describe BowlingScore do
  (0..4).each do |pins|
    (1..20).each do |throws|
      target = pins * throws

      it "totals #{target} if you score #{pins} #{throws} times" do
            score = BowlingScore.new
            throws.times { score.add_pins(pins) }
            expect(score.total).to eq(target)
      end
    end
  end
end
```

В итоге было написано 100 тестов, так что оставшаяся часть дня свободна!

Недостаток внутренних предметно-ориентированных языков заключается в необходимости привязываться к их синтаксису и семантике. И хотя некоторые языки данной категории проявляют в этом отношении замечательную гибкость, все равно приходится искать компромисс между языком, который требуется, и языком, который можно реализовать. И все, что в конечном счете будет достигнуто, должно быть по-прежнему согласовано с синтаксисом целевого языка. Языки с макрокомандами (например, Elixir, Clojure и Crystal) предоставляют немного больше удобств, но синтаксис все равно остается синтаксисом.

На внешние предметно-ориентированные языки подобные ограничения не накладываются. Если имеется возможность написать синтаксический анализатор для такого языка, то именно так и следует поступить. Иногда можно воспользоваться сторонним синтаксическим анализатором, как это сделано в Ansible с помощью YAML, но и тогда приходится искать компромисс.

Написание синтаксического анализатора, вероятнее всего, означает внедрение новых библиотек (а возможно, и инструментальных средств) в разрабатываемое приложение. А ведь написать качественный синтаксический анализа-

тор — дело непростое. Но если отважиться, то можно поискать и применить подходящие генераторы синтаксических анализаторов (например, bison или ANTLR) или каркасы для синтаксического анализа (например, многочисленные реализации анализаторов PEG — грамматики, разбирающей выражения).

Наша рекомендация довольно проста: не тратить усилий больше, чем можно сэкономить. Написание предметно-ориентированного языка так или иначе увеличивает затраты на проект, и поэтому необходимо убедиться, что они окупятся (потенциально — в долгосрочной перспективе).

В общем, пользуйтесь по возможности готовыми внешними предметно-ориентированными языками (например, YAML, JSON или CSV), а иначе — поищите подходящий внутренний язык. Пользоваться внешними языками рекомендуется только в тех случаях, когда выбранный вами язык будет написан пользователями вашего приложения.

Внутренний предметно-ориентированный язык почти даром

Наконец, если просачивание синтаксиса базового языка не имеет особого значения, можно прибегнуть к одной хитрости, чтобы создать внутренний предметно-ориентированный язык. В частности, вместо усердного метапрограммирования достаточно написать функции, выполняющие все необходимые действия. На самом деле нечто подобное делается в RSpec, как показано ниже.

```
describe BowlingScore do
    it "totals 12 if you score 3 four times" do
        score = BowlingScore.new
        4.times { score.add_pins(3) }
        expect(score.total).to eq(12)
    end
end
```

В приведенном выше фрагменте кода `describe`, `it`, `expect`, `to` и `eq` — это всего лишь методы языка Ruby. И хотя в нем скрыт внутренний механизм передачи объектов, это все же код. Более подробно этот вопрос рассматривается в последующих упражнениях.

Другие разделы, связанные с данной темой

- **Тема 8**. Сущность качественного проектирования.
- **Тема 13**. Прототипы и памятные записки.
- **Тема 32**. Конфигурирование, **глава 5** "Гибкость или ломкость".

Задачи

- Можно ли выразить какое-нибудь требование к вашему текущему проекту на предметно-ориентированном языке? И можно ли написать компилятор или транслятор, способный сгенерировать бо́льшую часть требующегося кода?

- Если вы решаете принять мини-языки как средство программирования поближе к предметной области, то тем самым признаете, что для их реализации придется приложить некоторые усилия. Можете ли вы найти пути для повторного применения в других проектах того каркаса, который разрабатываете в текущем проекте?

Упражнения

4. Требуется реализовать мини-язык для управления простой системой черепашьей графики. Такой язык должен состоять из однобуквенных команд и одиночных цифр, указываемых после некоторых из них. В качестве примера применения такого языка ниже приведены входные команды для рисования прямоугольника.

```
P 2 # выбрать перо 2
D   # опустить перо
W 2 # нарисовать линию длиной 2 см в западном направлении
N 1 # затем нарисовать линию длиной 2 см в северном направлении
E 2 # далее нарисовать линию длиной 2 см в восточном направлении
S 1 # и, наконец, провести линию обратно в южном направлении
U   # поднять перо
```

   Напишите код, реализующий синтаксический анализ команд этого мини-языка. Он должен быть разработан таким образом, чтобы вводить новые команды было просто.

5. В предыдущем упражнении вам было предложено реализовать синтаксический анализатор для языка рисования, и это был внешний предметно-ориентированный язык. Теперь реализуйте его снова, но для внутреннего языка. Постарайтесь не делать ничего заумного, а просто напишите функцию для синтаксического анализа каждой команды. Вполне возможно, что вам придется изменить имена команд, обозначив их строчными буквами, а может быть, и заключить их в какую-нибудь оболочку, чтобы предоставить определенный контекст.

6. Разработайте грамматику BNF (запись Бэкуса–Наура) для синтаксического анализа формата времени. Для такого анализа должны быть приемлемы все приведенные примеры формата времени.

```
4pm, 7:38pm, 23:42, 3:16, 3:16am
```

7. Реализуйте на избранном вами языке синтаксический анализатор для грамматики BNF из предыдущего упражнения, используя генератор анализаторов PEG. Такой анализатор должен выводить целое число, обозначающее количество минут, прошедших после полуночи.

8. Реализуйте синтаксический анализатор формата времени, используя язык сценариев и регулярные выражения.

Тема 15 Оценивание

В Библиотеке Конгресса США в Вашингтоне хранится около 75 терабайт цифровой информации, доступной в оперативном режиме. Любопытно, сколько времени потребуется, чтобы переслать всю эту информацию по сети на скорости 1 Гбит/с? А сколько памяти потребуется для хранения миллиона Ф.И.О. и адресов? Сколько времени отнимет уплотнение 100 Мбайт текста? Наконец, сколько месяцев вам потребуется, чтобы выпустить свой проект? Отвечайте, живо!

В какой-то степени все эти вопросы бессмысленны, поскольку они содержат недостаточно информации. И тем не менее на все эти вопросы вы можете найти ответы, если умеете выполнять оценки. Более того, в процессе оценивания вы сможете лучше понять среду, в которой обитают ваши программы.

Научившись оценивать и развивать в себе этот навык до такой степени, чтобы чувствовать величины оцениваемых вещей, вы сможете проявить кажущуюся, на первый взгляд, волшебной способность определять их осуществимость. Так, если кто-нибудь скажет: "Мы отправим резервную копию по сетевому соединению в центральное хранилище", вы должны быть в состоянии интуитивно почувствовать, насколько это практично. А когда вы программируете, вы должны понимать, какой именно системе требуется оптимизация и какие системы можно оставить в покое.

Совет 23 Выполняйте оценку, чтобы исключить неожиданности

В качестве поощрения приведем в конце этого раздела единственно правильный ответ, который следует дать, если вас попросят что-нибудь оценить.

Какой точности оценки достаточно?

Все ответы являются в той или иной мере оценками. Просто одни из них более точные, чем другие. Поэтому первый вопрос, который вы должны выяснить, когда вас попросят оценить что-нибудь, связан с контекстом, в котором вы должны дать свою оценку. В частности, требуется ли оценка с большой точностью, или просящего о ней вполне удовлетворит приблизительное значение?

Оценивание любопытно, в частности, тем, что интерпретация его результатов зависит от выбранных единиц измерения. Так, если вы скажете, что на какой-нибудь проект потребуется около 130 рабочих дней, то можно вполне предположить, что такая оценка достаточно точная. Но если вы скажете, что для этого потребуется около шести месяцев, то такая оценка будет воспринята как весьма приблизительная — в пределах от пяти до семи месяцев. Обе упомянутые

числовые оценки обозначают один и тот же срок, но первая из них, вероятно, подразумевает бо́льшую точность, чем вы считаете. Поэтому мы рекомендуем пользоваться приведенной ниже шкалой временны́х оценок.

Продолжительность	Оценка (порядок величин)
1–15 дней	Дни
3–6 недель	Недели
8–20 недель	Месяцы
20 и больше недель	Прежде чем оценивать, стоит хорошенько подумать

Итак, если вы, проделав всю необходимую работу, решите, что выполнение проекта отнимет 125 рабочих дней (25 недель), то вам придется дать оценку "около шести месяцев". По тому же принципу оцениваются любые другие величины: выберите единицы измерения, отражающие точность, с которой вы намерены дать свою оценку.

Откуда берутся оценки

Все оценки основываются на моделях решаемой задачи. Но прежде чем перейти к подробному описанию методов построения таких моделей, следует упомянуть простой прием оценивания, всегда дающий хорошие результаты: спросите того, кто уже это делал. Следовательно, прежде чем строить модель, поищите того, кто уже бывал в подобном положении, и выясните, как он решил свою задачу. Вы вряд ли найдете решение, точно соответствующее вашей задаче, но будете удивлены, как часто вы сможете обращаться к чужому опыту.

Понимание того, что спрашивается

Первая часть любого упражнения в оценивании состоит в понимании того, о чем именно спрашивают. Помимо рассмотренных выше вопросов, необходимо уяснить пределы предметной области. Нередко это подразумевается в самом вопросе, но необходимо выработать в себе привычку обдумывать пределы, прежде чем строить догадки. Зачастую выбранные пределы составляют часть ответа, например: "Если не произойдет никаких дорожных происшествий и топлива в автомобиле окажется достаточно, я должен добраться до места за 20 минут".

Построение модели системы

Это самая любопытная часть оценивания. Понимая то, о чем спрашивают, можно приступить к построению приближенной элементарной модели. Так, если вы оцениваете время реакции системы, ваша модель может включать в себя сервер и определенного рода входящий трафик. Для проекта моделью могут стать те стадии разработки, которые принято использовать в вашей организации, а

также весьма приблизительное общее представление о том, каким образом система может быть реализована.

Построение модели может стать как творческим, так и полезным в долгосрочной перспективе процессом. Нередко процесс построения модели приводит к открытиям подспудных шаблонов и процессов, которые не были видны на поверхности. Возможно, даже придется проанализировать первоначальный вопрос заново, например: "Вы просили оценить компонент *X*. Но похоже, что компонент *Y* как вариант компонента *X* можно сделать в два раза быстрее, потеряв лишь одно свойство".

Построение модели вносит неточности в процесс оценивания, и это неизбежно, хотя и приносит известную пользу. Вам приходится искать компромисс между простотой и точностью модели. Так, удвоив усилия для проработки модели, можно добиться лишь незначительного повышения ее точности. И здесь опыт должен подсказать вам, когда следует остановиться, уточняя модель.

Разделение модели на компоненты

Как только модель будет построена, ее можно разделить на компоненты. Для этого придется обнаружить математические правила, описывающие взаимодействие компонентов. Иногда компонент привносит единственное значение, добавляемое к конечному результату. Одни компоненты могут привносить множественные факторы, тогда как другие могут быть более сложными, как, например, те, которые имитируют поступление трафика в узел сети. Как правило, у каждого компонента имеются свои параметры, оказывающие влияние на его участие в общей модели. В таком случае на данной стадии следует просто определить каждый параметр.

Присваивание значения каждому параметру

Как только будут определены все параметры, можно пройтись по ним, присвоив каждому соответствующее значение. На этом этапе можно ожидать внесения некоторого количества ошибок, поэтому следует выбрать те параметры, которые оказывают наибольшее влияние на конечный результат, и сосредоточить все внимание на том, чтобы они получили правильные значения. Как правило, те параметры, значения которых добавляются к конечному результату, менее важны, чем те, значения которых умножаются или делятся. Так, удвоение скорости передачи данных в канале связи может привести к удвоению объема данных, принимаемых в час, тогда как добавление транзитной задержки на 5 мс не окажет заметного влияния.

Для расчета этих очень важных параметров следует иметь вполне обоснованный способ. Например, для оценивания проектируемой системы массового обслуживания, возможно, придется измерить фактическую частоту поступления транзакций в существующей системе или же найти аналогичную систему, чтобы измерить данный параметр. Аналогично можно измерить текущее время

обслуживания запроса или же оценить его, используя методы, описываемые в этом разделе. В действительности свои оценки нередко приходится основывать на других подоценках. Именно здесь и вкрадываются самые крупные ошибки.

Вычисление ответов

Лишь в самых простых случаях оценивание дает единственный ответ. Так, вы можете, не моргнув глазом, заявить, что способны пройти пять городских кварталов за 15 минут. Но по мере усложнения систем вам, вероятно, придется давать более осторожные ответы. Проведите многократные вычисления, варьируя значения самых важных параметров до тех пор, пока не получите такие значения, которые действительно приводят модель в действие. И здесь неоценимую помощь могут оказать электронные таблицы. Затем сформулируйте свой ответ, исходя из этих параметров, например: "Время реакции составляет приблизительно три четверти секунды, если в системе применяются твердотельные накопители и оперативная память объемом 32 Гбайт, и одну секунду, если объем памяти 16 Гбайт". (Обратите внимание, насколько иное ощущение точности вызывает формулировка "три четверти секунды" по сравнению с обозначением 750 мс.)

На стадии вычисления вы можете получить ответы, которые покажутся вам странными, но не спешите их отвергать. Если ваши арифметические расчеты правильны, вероятно, вы неверно понимаете стоящую перед вами задачу; возможно также, что неверна ваша модель. И это весьма ценная информация для оценивания.

Прослеживание своих способностей оценивать

Мы считаем, что вам стоит вести учет своих оценок, чтобы иметь возможность выяснить, насколько они точны. Так, если для общей оценки требуется вычислить подоценки, проследите и их. Зачастую оценки будут казаться вам довольно хорошими, и со временем они, действительно, станут такими, как вы ожидали.

Если оценка окажется неверной, не отмахивайтесь от нее и не уходите от ответа, а лучше найдите причину. Вполне возможно, что вы выбрали такие параметры, которые не соответствуют действительности решаемой задачи, а может быть, ваша модель оказалась неверной. Какова бы ни была причина, уделите время, чтобы разобраться в произошедшем. И тогда следующая оценка окажется лучше.

Оценивание сроков выполнения проектов

Как правило, вас могут попросить оценить, сколько времени отнимет работа над чем-то конкретным. И если это нечто сложное, дать ему оценку, возможно, будет нелегко. В этом разделе мы рассмотрим два метода, позволяющих уменьшить неопределенность, проявляющуюся в подобных случаях.

Покраска ракеты

> *— Сколько времени потребуется, чтобы покрасить дом?*
>
> *— Ну, если все будет нормально и расход этой краски окажется таким, как заявлено производителем, то может потребоваться не меньше 10 часов, хотя это маловероятно. Думаю, что более реалистичная оценка ближе к 18 часам. И, конечно, если погода испортится, то работа может растянуться до 30 часов, а то и больше.*

Именно таким образом люди обычно оценивают окружающую их реальность, т.е. целым рядом возможных вариантов, а не одним числом, если только не принудить их дать именно такую оценку. Когда ВМС США потребовалось спланировать проект баллистической ракеты "Поларис" для атомных подводных лодок, был принят именно такой порядок оценивания по так называемой методологии *PERT* (Program Evaluation and Review Technique — Метод оценки и анализа проектов). По методологии PERT каждая задача получает *оптимистическую*, *наиболее вероятную* и *пессимистическую* оценки. Все задачи организованы в граф зависимостей, а для выявления самых оптимистичных и пессимистичных сроков выполнения работ во всем проекте используется простая статистика.

Используя такую область значений, удается благополучно избежать самой распространенной причины ошибок при оценивании: раздувания цифр из-за неуверенности. Вместо этого статистика по методологии PERT распределяет неопределенность автоматически, позволяя точнее оценить проект в целом.

Тем не менее мы не являемся страстными поклонниками такой методологии. Ведь разработчики привыкли рисовать едва ли не на всю стену диаграммы всех задач в проекте, неявно полагая, что имеют его точную оценку просто потому, что они воспользовались какой-то *формулой*. И если они никогда не делали этого прежде, то, скорее всего, точную оценку они не получат.

Поедание слона

Мы считаем, что составить календарный план проекта можно лишь на основании опыта работы над тем же самым проектом. И в этом не будет никакого парадокса, если практиковать поэтапную разработку, повторяя шаги перечисленной ниже процедуры, на которых функциональные возможности внедряются очень тонкими слоями.

- Проверить требования.
- Проанализировать риски (начав с самых рискованных элементов).
- Спроектировать, реализовать, интегрировать.
- Проверить правильность с помощью пользователей.

Поначалу у вас может сложиться лишь смутное представление, сколько раз придется повторить данную процедуру и как долго это может продолжаться.

И хотя некоторые методы требуют закрепить ее как часть первоначального плана, на самом деле это будет ошибкой во всех проектах, кроме самых тривиальных. Ведь вам придется просто гадать, если только вы не разрабатывали прежде аналогичное приложение с той же самой командой.

Итак, завершив кодирование и тестирование первоначальных функциональных возможностей, отметьте это как окончание первого шага повторяющейся процедуры. Основываясь на приобретенном опыте, вы сможете теперь уточнить первоначальное предположение о количестве последующих шагов и о том, что следует включить в каждый из них. С каждым разом уточнение становится все лучше и вместе с тем растет уверенность в правильности графика выполнения работ. Такого рода оценивание нередко выполняется во время подведения командой итогов в конце повторяющегося цикла. И это еще раз подтверждает старую шутку: слона нужно есть по кусочкам.

Совет 24 Повторно уточняйте график выполнения работ по мере написания кода

Такой метод оценивания может и не найти признания у руководства, которому, как правило, требуется точное число еще до начала проекта. Поэтому вы должны объяснить руководству, что именно состав команды, ее производительность и среда будут определять график выполнения работ. Формализуя это положение и уточняя график на каждом шаге повторяющейся процедуры, вы сумеете дать руководству самую точную оценку планирования работ, какую только сможете.

Что ответить на просьбу что-нибудь оценить

На это следует ответить: "Вернемся к этому позже". Если вы замедлите процесс, уделив время прохождению всех шагов описанной выше процедуры, то почти всегда сможете добиться лучших результатов. А оценки, данные мимоходом за чашкой кофе, будут еще не раз навещать и тревожить вас.

Другие разделы, связанные с данной темой

- **Тема 7**. Общайтесь!, **глава 1** "Философия прагматизма".
- **Тема 39**. Быстродействие алгоритма, **глава 7** "По ходу кодирования".

Задачи

- Начните вести журнал своих оценок. Проследите за тем, насколько точной оказалась каждая из них. Если допущенная вами ошибка превысила 50%, постарайтесь выяснить, в чем именно она оказалась неверной.

Упражнения

9. Представьте, что к вам обратились со следующим вопросом: "Чья пропускная способность выше: сетевого соединения на скорости 1 Гбит/с или человека, перемещающегося между двумя компьютерами с запоминающим устройством емкостью 1 Тбайт в своем кармане?" Какие ограничения вы наложите на свой ответ, чтобы дать его в правильных пределах? (Вы можете, например, сказать, что временем доступа к запоминающему устройству можно пренебречь.)

10. Так чья же пропускная способность выше?

Глава 3

Основные инструментальные средства

Каждый мастер начинает свое ремесло с основного набора качественных инструментов. В частности, столяру могут потребоваться линейки, уровни, пилы, качественные рубанки, тонкие долота, сверла и скрепы, киянки и струбцины. Эти инструменты тщательно подбираются, чтобы служить долго для выполнения особых работ вместе с другими инструментами, а самое, возможно, главное — чтобы столяру было удобно работать с ними, держа их в своих пока еще неумелых руках.

Затем начинается процесс обучения работе с инструментами и приспосабливания к ним. Ведь у каждого инструмента свои свойства и особенности, а следовательно, он требует особого обращения. Каждый инструмент нужно по-особому заточить и соответственно держать в руках. Со временем все инструменты изнашиваются, вплоть до того, что их рукоятка напоминает отпечаток руки столяра, а режущая кромка идеально совпадает с углом, под которым он держит свой инструмент. И в этот момент инструменты превращаются в проводников замыслов, возникающих в уме мастера, доводя их до готового изделия и становясь как бы продолжением его рук. Постепенно столяр пополняет свой арсенал новыми инструментами, в том числе ламельными фрезерными станками, торцовочными пилами с лазерным наведением, шаблонами под "ласточкин хвост" и прочими чудесами техники деревообработки. Но можно не сомневаться, что столяру будет намного приятнее работать с одним из своих первых инструментов, держа его в руках и чувствуя, например, как рубанок буквально поет, скользя по дереву.

Инструменты развивают ваши способности. Чем они лучше, и чем лучше вы знаете, как ими пользоваться, тем выше может стать производительность вашего труда. Начните с простого набора инструментальных средств, подходящих в общем случае. По мере обретения опыта обращения с ними и появления каких-то особых требований вы будете постепенно пополнять этот простой набор новыми средствами. Как и любому другому мастеру, вам придется делать это регу-

лярно, поэтому старайтесь всегда искать лучшие способы делать свое дело. Если же возникнет ситуация, с которой текущие инструментальные средства, на ваш взгляд, не справятся, сделайте себе заметку, чтобы поискать какое-нибудь другое или более эффективное инструментальное средство, которое поможет вам справиться с данной ситуацией. Пусть вашими приобретениями правит нужда.

Многие начинающие программисты совершают ошибку, останавливая свой выбор лишь на одном эффективном инструментальном средстве (например, на конкретной интегрированной среде разработки), так и не покидая его удобный интерфейс. А вы должны чувствовать себя уверенно и за пределами конкретной интегрированной среды разработки. И достичь этого можно единственным способом, храня свой набор простых инструментальных средств в постоянной готовности к применению.

В этой главе речь пойдет об инвестировании в собственный набор простых инструментальных средств. Как и любое грамотное обсуждение инструментальных средств, мы начнем (в следующем разделе) с рассмотрения исходных материалов, которые требуется облечь в нужную форму. А далее мы перейдем к верстаку (в нашем случае — к компьютеру), чтобы выяснить, как пользоваться им, чтобы извлечь наибольшую пользу из инструментальных средств. Подробнее об этом — в разделе "Игры в скорлупки". И как только в нашем распоряжении окажется сырьевой материал и станок для его обработки, мы обратимся к одному из наиболее употребительных инструментов (в данном случае — к текстовому редактору). Так, в разделе "Эффективное редактирование" предлагаются способы, повышающие эффективность труда программиста.

Чтобы не утратить ничего из своей ценной работы, мы должны всегда пользоваться *системой контроля версий* — даже для таких личных документов, как рецепты или заметки. А поскольку Мерфи был все же оптимистом, несмотря на сформулированные им пессимистические законы, то нельзя стать грамотным программистом до тех пор, пока не приобретешь превосходные навыки *отладки* исходного кода.

Чтобы скрепить все чудесное вместе, потребуется нечто вроде клея. Некоторые возможности добиться этого рассматриваются в разделе "Манипулирование текстом". Наконец, самые бледные чернила лучше, чем самая хорошая память. Поэтому ведите учет своих мыслей и предыстории, как поясняется в разделе "Технические дневники".

Уделяя время изучению всех упомянутых выше инструментальных средств, вы однажды с удивлением обнаружите, как ваши пальцы перемещаются по клавиатуре, безотчетно манипулируя текстом. И это будет означать, что применяемые вами инструменты стали продолжением ваших рук.

Тема 16 Сила простого текста

Будучи программистами-прагматиками, мы выбираем в качестве исходного материала не дерево и не железо, а знания. Сначала мы собираем требования как знания, а затем выражаем эти знания в своих проектных решениях, реализациях, тестах и документах. И мы считаем, что наилучшим форматом для постоянного хранения данных является *простой текст*. Благодаря простому тексту мы получаем возможность манипулировать знаниями как вручную, так и программно, пользуясь буквально каждым инструментальным средством, имеющимся в нашем распоряжении.

Но трудности обращения с большинством бинарных форматов состоят в том, что контекст, требующийся для понимания данных, отделен от самих данных, и тогда они искусственно отторгаются от своего смыслового значения. Кроме того, данные могут быть зашифрованы, и поэтому анализировать их без прикладной логики совершенно бессмысленно. Тем не менее простой текст дает возможность получить поток самостоятельно описываемых данных независимо от того приложения, в котором он создан.

Что такое простой текст

Простой текст состоит из печатных символов в форме, передающей информацию. Это может быть, например, простой список покупок, аналогичный приведенному ниже, или же сложный текст, подобный рукописи данной книги. (Она действительно набрана простым текстом к явному неудовольствию издателя, которому хотелось бы, чтобы мы пользовались для этого специализированным текстовым редактором.)

```
* молоко
* сахар
* кофе
```

И здесь важна информационная часть. Так, следующая строка простого текста вряд ли принесет какую-то пользу:

```
hlj;uijn bfjxrrctvh jkni'pio6p7gu;vh bjxrdi5rgvhj
```

Как, впрочем, и эта строка:

```
Field19=467abe
```

Ведь читающий ее понятия не имеет, в чем смысл значения `467abe`. Поэтому мы стараемся сделать набираемый нами простой текст *удобочитаемым*.

Совет 25 Храните свои знания в виде простого текста

В чем сила простого текста

Простой текст совсем не означает, что сам он не структурирован. HTML, JSON, YAML и все аналогичные форматы служат примерами простого текста, как, впрочем, и большинство основных сетевых протоколов, в том числе HTTP, SMTP, IMAP и т.д. И для этого имеется ряд веских оснований, включая следующие.

- Гарантия от устаревания.
- Эффективное использование инструментальных средств.
- Упрощение тестирования.

Гарантия от устаревания

Удобочитаемые формы данных и описывающие сами себя данные, несомненно, переживут все остальные формы данных и те приложения, в которых они сформированы. А раз они уцелеют, то есть шанс воспользоваться ими даже через много времени после того, как перестанет функционировать первоначальная версия приложения, в котором они были сформированы. Такого рода файл данных можно проанализировать, зная лишь частично его формат. Что же касается большинства бинарных файлов, то для успешного их анализа необходимо знать во всех подробностях их формат.

Рассмотрим в качестве примера файл данных из устаревшей системы[1], которую вам дали сопровождать. Исходно вам мало что известно об ее первоначальной версии, помимо того, что в ней ведется список номеров социального обеспечения клиентов, который требуется найти и извлечь. Среди прочих данных вы обнаруживаете следующее:

```
<FIELD10>123-45-6789</FIELD10>
...
<FIELD10>567-89-0123</FIELD10>
...
<FIELD10>901-23-4567</FIELD10>
```

Распознав формат номера социального обеспечения, вы можете быстро написать небольшую программу для извлечения данных из файла, даже если вам больше ничего не известно о другом его содержимом.

Но представьте, что файл отформатирован иначе — например, так, как показано ниже.

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

[1] Все программное обеспечение устаревает в момент его написания.

Вполне возможно, что вам не удастся легко распознать смысловое значение приведенных выше чисел. В этом, собственно, и состоит отличие *удобочитаемого* содержимого файлов от *неудобочитаемого*. Тем не менее содержимое поля `FIELD10` в рассматриваемом здесь файле мало чем помогает уяснить его смысловое значение. А вот приведенный ниже формат делает его намного более понятным, гарантируя, что такие данные переживут тот проект, в котором они сформированы.

```
<SOCIAL-SECURITY-NO>123-45-6789</SOCIAL-SECURITY-NO>
```

Эффективное использование инструментальных средств

Буквально каждое инструментальное средство в области вычислений (от системы контроля версий до утилит командной строки) может оперировать простым текстом. Допустим, вы развертываете крупное приложение в условиях его эксплуатации с помощью специального файла конфигурации. Если этот файл содержит простой текст, вы можете заменить его в системе контроля версий (см. далее раздел "Тема 19. Контроль версий"), чтобы автоматически отслеживать предысторию всех изменений. С помощью таких средств сравнения файлов, как, например, утилиты `diff` и `fc`, вы можете сразу выяснить, какие именно изменения были внесены, тогда как утилита `sum` поможет сформировать контрольную сумму, чтобы проверить файл на предмет случайной (или злонамеренной) модификации.

Философия Unix

Как известно, операционная система Unix спроектирована по принципу утилит — небольших заточенных инструментов, каждый из которых предназначен для выполнения одной конкретной операции. Этот основной принцип позволяет использовать общий базовый формат файлов с простым построчно расположенным текстом. Все базы данных, используемые в Unix для системного администрирования (пользователей и паролей, конфигурации сети и т.д.), организованы в виде файлов с простым текстом. (Ради оптимизации в некоторых системах поддерживается также бинарная форма определенных баз данных. При этом версия базы данных в простом тексте хранится для сопряжения с ее бинарной версией.)

Если система выйдет из строя, для ее восстановления, возможно, придется иметь дело с минимальной средой, не имея, например, доступа к графическим драйверам. В подобных случаях можно по достоинству оценить несложность простого текста.

Кроме того, простой текст проще для поиска. Так, если вы не помните, какой именно файл конфигурации управляет созданием резервных копий системы, команда **`grep -r backup /etc`** поможет вам быстро найти его.

Упрощение тестирования

Если вы пользуетесь простым текстом, чтобы сформировать синтетические данные для тестирования системы, в таком случае вам достаточно ввести, обновить или модифицировать тестовые данные, *даже не создавая* для этого специальные инструментальные средства. Аналогично результаты регрессионного тестирования, выведенные простым текстом, совсем не трудно проанализировать с помощью команд оболочки или простого сценария.

Наименьший общий знаменатель

Вездесущие текстовые файлы понадобятся даже в будущих интеллектуальных агентах на основе блокчейна, автономно перемещающихся в дикой и опасной среде Интернета, для согласования обмена данными. В действительности преимущества простого текста в гетерогенных средах перевешивают все их недостатки. Нужно лишь обеспечить общение всех сторон по общему стандарту. И таким стандартом является простой текст.

Другие разделы, связанные с данной темой

- **Тема 17**. Игры в скорлупки.
- **Тема 21**. Манипулирование текстом.
- **Тема 32**. Конфигурирование, **глава 5** "Гибкость или ломкость".

Задача

- Спроектируйте на избранном вами языке небольшую базу данных для хранения адресной книги (Ф.И.О., номера телефонов и т.д.), используя простое бинарное представление. Сделайте это, прежде чем читать остальную часть этой задачи.
  - Преобразуйте данный формат в простой текст формата XML или JSON.
  - Добавьте для каждой версии новое поле переменной длины `directions`, в котором можно было бы ввести направления к дому каждого лица, указанного в адресной книге.

  Какие вопросы, связанные с контролем версий и расширяемостью, у вас возникнут? Какую форму легче видоизменить? А как насчет преобразования имеющихся данных?

Тема 17 Игры в скорлупки

Каждому столяру требуется хороший, прочный и надежный верстак, где можно было бы закрепить заготовки на такой высоте, чтобы их было удобно обрабатывать. Таким образом, верстак становится центром столярной мастерской, в которую мастер будет периодически возвращаться по мере того, как заготовка станет приобретать нужную форму.

Для программиста, манипулирующего текстовыми файлами, таким верстаком служит командный процессор, иначе называемый просто оболочкой. По приглашению из командной строки оболочки можно вызывать весь арсенал инструментальных средств, объединяя их с помощью конвейеров так, как это и не снилось их разработчикам. Из оболочки можно также запускать приложения, отладчики, браузеры, редакторы и утилиты, искать файлы, запрашивать состояние систем и фильтровать выводимые результаты. Программируя на уровне оболочки, можно строить сложные макрокоманды для автоматизации часто выполняемых действий.

Для программистов, привыкших работать с использованием графических интерфейсов пользователя и интегрированных сред разработки, такой вариант может показаться крайним. Поэтому у них может возникнуть вполне резонный вопрос: нельзя ли одновременно пользоваться как оболочкой для ввода команд, так и графическим интерфейсом пользователя для манипулирования мышью?

Самый простой ответ на этот вопрос отрицательный. GUI, безусловно, примечательны тем, что позволяют быстрее и удобнее выполнять простые операции. Перемещение файлов, чтение и запись сообщений электронной почты, построение и развертывание проекта — все эти операции обычно требуется выполнять в графической среде. Но если делать всю свою работу только в графическом интерфейсе пользователя, то вряд ли удастся раскрыть полностью истинный потенциал своей рабочей среды и, в частности, автоматизировать типичные задачи или использовать весь потенциал доступных инструментальных средств, а также объединять их, создавая специализированные *макрокоманды*. Главное преимущество графического интерфейса пользователя заключается в том, что он действует по принципу WYSIWYG (what you see is what you get — что видишь, то и получишь). Но главным его недостатком является принцип WYSIAYG (what you see is *all* you get — получишь *лишь* то, что видишь).

Возможности графических сред, как правило, ограничиваются теми целями, которые преследовали их разработчики. Если же требуется выйти за переделы той модели поведения графической среды, которая была предусмотрена ее разработчиками (что требуется чаще, чем хотелось бы), то сделать это вряд ли удастся. Программисты-прагматики не просто специализируются на написании кода, разработке объектных моделей или составлении документа, а делают *все* это сами. Область действия любого инструмента обычно ограничивается теми

задачами, для выполнения которых он предназначен. Допустим, в интегрированную среду разработки требуется встроить препроцессор кода, чтобы реализовать принципы проектирования по контракту или многопоточной обработки. Сделать это не удастся, если разработчик интегрированной среды разработки не предоставил явно соответствующие функциональные возможности.

Совет 26 Используйте всю мощь командных оболочек

Освойте оболочку как следует, и производительность вашего труда не замедлит резко возрасти. Так, если требуется составить список всех однозначных имен пакетов, явно импортируемых в коде Java, его можно сохранить в файле `list` с помощью следующей команды, вызываемой из оболочки:

sh/packages.sh

```
grep '^import ' *.java |
   sed -e's/.*import *//' -e's/;.*$//' |
   sort -u >list
```

Такая команда может выглядеть пугающе, если вы не уделяли должного времени изучению функциональных возможностей командных оболочек тех систем, которыми пользуетесь. Все же немного потрудитесь, чтобы поближе ознакомиться с доступной вам командной оболочкой, и вскоре все станет на свои места. Поэкспериментировав с ней, вы с удивлением обнаружите, насколько возрастет производительность вашего труда.

Ваша собственная оболочка

Подобно тому, как столяр упорядочивает свое рабочее место и подгоняет его под себя, разработчик должен специально настроить свою оболочку. Как правило, это подразумевает также изменение конфигурации применяемой терминальной программы. Ниже перечислены типичные изменения, которые требуется внести при этом.

- **Установка цветовых тем.** Опробование каждой такой темы, доступной в оперативном режиме для конкретной оболочки, может отнять немало времени.
- **Настройка приглашения.** Приглашение, которое сообщает о готовности оболочки к вводу команд, можно настроить на отображение лишь самой необходимой информации, поскольку остальная его часть вряд ли вообще понадобится. И здесь явно проявляются личные предпочтения. Нам, например, больше нравятся простые приглашения, где в укороченной форме указано имя текущего каталога, текущая версия оболочки и время.

- **Псевдонимы и функции оболочки.** Упростите свой рабочий процесс, преобразовав в простые псевдонимы те команды, которыми вы часто пользуетесь. Так, если вы регулярно обновляете свою платформу Linux, но в то же время можете забыть, в какой именно последовательности следует это делать: обновлять и модернизировать или же модернизировать и обновлять, создайте для данной цели псевдоним, как показано ниже.

  ```
  alias apt-up='sudo apt-get update && sudo apt-get upgrade'
  ```

  Если вы случайно удалили файлы командой `rm`, отметив, что делаете это слишком часто, создайте на будущее следующий псевдоним, чтобы всегда выводить соответствующее приглашение:

  ```
  alias rm ='rm -iv'
  ```

- **Автозавершение команд.** В большинстве оболочек поддерживается автозавершение команд и файлов: достаточно набрать несколько первых символов, нажать клавишу табуляции, и оболочка постарается автоматически завершить ввод той команды, которую вы подразумеваете. Но можно пойти еще дальше, настроив оболочку таким образом, чтобы распознавать вводимые команды, предлагая контекстно-зависимые варианты их завершения, а иногда даже делая это в зависимости от текущего каталога.

Вам придется немало времени работать в одной из таких оболочек[2]. Будьте как рак-отшельник, сделав оболочку своим обиталищем.

Другие разделы, связанные с данной темой

- **Тема 13**. Прототипы и памятные записки, **глава 2** "Прагматичный подход".
- **Тема 16**. Сила простого текста.
- **Тема 21**. Манипулирование текстом.
- **Тема 30**. Преобразовательное программирование, **глава 5** "Гибкость или ломкость".
- **Тема 51**. Начальный набор инструментальных средств программиста-прагматика, **глава 9** "Прагматичные проекты".

Задачи

- Есть ли какие-нибудь операции, которые вы в настоящее время выполняете в GUI? Даете ли вы своим коллегам какие-то инструкции, включающие в себя такие действия, как, например, "щелкни на этой кнопке" или "выбери этот элемент"? Можно ли их как-то автоматизировать?
- Всякий раз, когда вы переходите в новую среду, постарайтесь выяснить, какие в ней имеются командные оболочки и не удастся ли вам перенести свою текущую оболочку в новую среду.

[2] Игра слов: shell (англ.) — оболочка, ракушка. — *Примеч.пер.*

- Изучите альтернативы своей текущей оболочке. Если окажется, что ваша командная оболочка не справляется со своими обязанностями, выясните, сможет ли справиться с ними лучше альтернативная оболочка.

Тема 18 Эффективное редактирование

Как упоминалось ранее, рабочие инструменты являются продолжением рук мастера. И это относится к текстовым редакторам даже в большей степени, чем к любым другим программным средствам. Ведь у вас должна быть возможность манипулировать текстом, прилагая как можно меньше усилий, поскольку текст служит сырьевым материалом для программирования.

В первом издании этой книги мы настоятельно рекомендовали пользоваться одним и тем же редактором для выполнения самых разных операций: программирования, документирования, памятных записок, системного администрирования и т.д. Но теперь мы немного смягчили свою позицию, рекомендуя использовать сколько угодно редакторов, поскольку сами стремимся умело работать с текстом в каждом из них.

Совет 27 Стремитесь свободно владеть редактором

Почему это так важно? Не хотим ли мы просто сказать о том, что это сэкономит нам массу времени? По сути, да: ведь за год вы сможете сэкономить целую неделю, повысив эффективность редактирования текста лишь на 4% (если, конечно, вам приходится делать это по 20 часов в неделю).

Но настоящая выгода состоит даже не в этом, а в том, чтобы, научившись свободно владеть редактором, вы больше не задумывались о самом механизме редактирования. Благодаря этому сокращается промежуток между осмыслением чего-нибудь и появлением результата в буфере редактора. От этого ваше программирование только выигрывает, поскольку поток ваших мыслей не прерывается. (Если вам приходилось учить кого-нибудь вождению автомобиля, то вы хорошо знаете, насколько начинающий водитель, обдумывающий каждое свое действие, отличается от опытного водителя, интуитивно управляющего автомобилем.)

Что означает свободное владение редактором

Ниже перечислены задачи редактирования текста, которые проясняют, что значит свободно владеть редактором.

- Перемещение и выделение теста по отдельным символам, словам, строкам и абзацам.

- Перемещение различных синтаксических единиц (совпадающих ограничителей, функций, модулей и т.д.)
- Повторное форматирование исходного кода с отступами после внесения изменений.
- Комментирование и раскомментирование блоков кода одной командой.
- Отмена и повтор изменений.
- Разбиение окна редактора на несколько панелей и перемещение между ними.
- Перемещение на строку с конкретным номером.
- Сортировка выбранных строк.
- Поиск как по символьным строкам, так и по регулярным выражениям и повторение предыдущего поиска.
- Временное создание нескольких курсоров по выделенному тексту или совпадению с шаблоном и параллельное редактирование текста в позиции каждого курсора.
- Отображение ошибок компиляции в текущем проекте.
- Выполнение тестов в текущем проекте.

Сможете ли вы сделать все это, не пользуясь мышью, сенсорной панелью или иным координатно-указательным устройством? Конечно, вы можете сказать, что текущий редактор не позволяет выполнить некоторые из перечисленных выше задач. В таком случае вам, может быть, стоит сменить редактор?

Стремление к свободному владению редактором

Мы не сомневаемся, что найдется больше десятка людей, знающих *все* команды в конкретном эффективном редакторе, но в то же время готовы предположить, что вы к их числу не относитесь. Поэтому предлагаем более прагматичный подход: изучить те команды редактора, которые упрощают работу с текстом.

Рецепт этого довольно прост. Сначала проанализируйте свои действия во время редактирования. Всякий раз, когда вы ловите себя на том, что делаете что-нибудь повторяющееся, выработайте в себе привычку обдумать и найти лучший способ сделать это.

Как только вы обнаружите новое полезное средство, непременно освойте его вплоть до автоматизма, чтобы пользоваться им, не задумываясь. Этого можно добиться лишь многократным повторением. Сознательно ищите возможности применить свои вновь приобретенные сверхнавыки (в идеальном случае — много раз в течение рабочего дня). И через неделю или около того вы сможете пользоваться ими, даже не задумываясь.

Расширение возможностей редактора

Большинство эффективных редакторов исходного кода построены на ядре, наращиваемом посредством расширений. Некоторые расширения поставляются вместе с редактором, тогда как другие могут быть внедрены впоследствии.

Если вы натолкнетесь на какое-то очевидное ограничение применяемого вами редактора, поищите его расширение, позволяющее преодолеть это ограничение. Вероятнее всего, такие функциональные возможности требуются не только вам, и вполне возможно, что кто-нибудь уже опубликовал свое решение данной задачи.

Не останавливаясь на этом, можете даже освоить язык расширений своего редактора. Научитесь пользоваться им для автоматизации некоторых часто повторяемых вами операций. Зачастую для этого потребуется лишь одна или две строки кода.

Иногда можно пойти еще дальше, самостоятельно написав полноценное расширение. В таком случае опубликуйте его. Ведь если оно понадобилось вам, то, скорее всего, понадобится и другим.

Другие разделы, связанные с данной темой

- **Тема 7**. Общайтесь!, **глава 1** "Философия прагматизма".

Задачи

- Откажитесь от автоповторов. Все привыкли делать следующее: нажимать клавишу <Backspace>, ожидая автоповтора, чтобы удалить последнее набранное слово. Можем побиться об заклад, что вы уже поднаторели делать это в уме настолько, что можете точно рассчитать, когда отпустить данную клавишу. Поэтому отключите режим автоповтора, а вместо него освойте комбинации клавиш для перемещения, выбора и удаления символов, слов, строк и блоков.
- Этот способ дается нелегко. Отставьте мышь или иное координатно-указательное устройство в сторону, чтобы целую неделю редактировать текст только с помощью клавиатуры. В итоге вы обнаружите, что немало операций манипулирования текстом вы в состоянии выполнять, не наводя курсор и не щелкая кнопкой мыши. Но этому нужно учиться, поэтому записывайте комбинации клавиш, которые осваиваете, старым и надежным способом, используя карандаш и бумагу.
- В первые дни это заметно скажется на производительности вашего труда. Но, учась манипулировать текстом так, чтобы не убирать свои руки из исходного положения на клавиатуре, вы постепенно обнаружите, что редактируете текст быстрее и свободнее, чем раньше.

- Ищите пути для интеграции. Когда Дэйв писал эту главу, его посетила мысль: а нельзя ли предварительно просмотреть окончательную ее компоновку (файл формата PDF) в буфере редактора. И как только он нашел и загрузил подходящее расширение, предварительно просматриваемый вид окончательной компоновки тотчас расположился рядом с текстом рукописи в том же самом текстовом редакторе. Итак, составьте список функциональных возможностей, которыми вы хотели бы расширить свой редактор, а затем поищите их.
- Если вам не удастся найти подходящее расширение или подключаемый модуль, напишите его, хотя это более честолюбивая задача. Энди нравится создавать специальные, локальные файловые модули, подключаемые к его излюбленным редакторам для составления вики-страниц. Если вам не удастся найти такой подключаемый модуль, создайте его сами!

Тема 19 Контроль версий

"Прогресс не проявляется лишь в изменениях и зависит от памятливости. Те, кто не учится на своих ошибках, обречены повторять их".

Джордж Сантаяна,[3] ***Жизнь разума*** *(Life of Reason)*

К числу самых востребованных элементов пользовательского интерфейса относится кнопка *отмены* — как единственная возможность исправления ошибки. Еще лучше, если в рабочей среде поддерживается несколько уровней отмены и повтора, чтобы можно было вернуться к тому, что было сделано пару минут назад.

Но что, если ошибка была совершена на прошлой неделе, и с тех пор компьютер неоднократно включался и выключался? Именно в таких случаях и проявляется одно из многих преимуществ системы контроля версий как гигантской кнопки *отмены*, которая, словно машина времени, позволяет вернуться к той благополучно завершенной на прошлой неделе стадии проекта, когда исходный код компилировался и выполнялся.

Многие разработчики ограничиваются именной этой возможностью, пользуясь системой контроля версий. Они упускают из виду всю намного более обширную область — сотрудничества, конвейерного развертывания, отслеживания ошибок и общего взаимодействия членов команды. Поэтому рассмотрим систему контроля версий сначала как хранилище изменений, а затем как центральное место встреч, где члены команды встречаются со своим исходным кодом.

[3] George Santayana — американский философ и писатель XX века испанского происхождения. — *Примеч. пер.*

Совместно используемые каталоги — это *не* система контроля версий

Нам до сих пор встречаются иногда такие команды, которые обмениваются исходными файлами своих проектов по сети: как внутренним образом, так и с помощью определенного рода облачного хранилища. Но это непрактично.

Команды, поступающие подобным образом, приводят работу друг друга в полный беспорядок, теряют изменения и нарушают сборки, словно идя стенка на стенку за место на парковке автомобилей. Это можно сравнить с написанием параллельного кода с совместно используемыми данными и без механизма синхронизации. Вместо этого лучше пользоваться системой контроля версий.

Более того, некоторые разработчики *пользуются* системой контроля версий, организуя ее основной репозиторий в сети или в облачном хранилище. Они считают такое решение обоюдовыгодным, поскольку их файлы доступны отовсюду и резервируются в удаленном месте (если речь идет об облачном хранилище).

На самом же деле это еще хуже, поскольку есть риск потерять все. В системе контроля версий применяется ряд взаимодействующих файлов и каталогов, и если изменения вносятся одновременно в два экземпляра, то общее их состояние может быть нарушено, а о степени нанесенного ущерба ничего не сообщается. Вряд ли кому-нибудь будет приятно видеть, как разработчики воют от отчаяния.

Все начинается с исходного кода

Системы контроля версий следят за каждым изменением, вносимым в исходный код и документацию. Настроив такую систему надлежащим образом, можно *всегда* вернуться к предыдущей версии своего программного обеспечения.

Но возможности системы контроля версий далеко не ограничиваются отменой ошибок. Хорошая система контроля версий позволяет отслеживать вносимые изменения, отвечая на такие вопросы, как, например, следующие: кто внес изменения в конкретной строке кода, чем текущая версия отличается от той, что была на прошлой неделе, какие файлы изменяются чаще всего? Такого рода сведения бесценны для целей отслеживания программных ошибок, ревизии, повышения производительности и качества.

Кроме того, система контроля версий позволяет идентифицировать выпуски программного обеспечения. Указав конкретную выпускаемую версию, можно всегда вернуться и сформировать ее снова, сделав это независимо от внесенных впоследствии изменений. Системы контроля версий позволяют отслежи-

вать файлы, хранящиеся в центральном репозитории, вполне подходящем для архивирования.

Наконец, системы контроля версий дают двум или большему количеству пользователей возможность работать параллельно над одним и тем же рядом файлов и даже вносить одновременные изменения в один и тот же файл. Система контроля версий сама объединит эти изменения, когда файлы будут отправлены обратно на хранение в репозиторий. Несмотря на кажущийся риск, такие системы вполне пригодны для практического применения в проектах любых масштабов.

Совет 28 Всегда пользуйтесь системой контроля версий

Делайте это всегда, даже если работаете над индивидуальным или краткосрочным проектом, "одноразовым" прототипом или иным материалом, не являющимся исходным кодом. При этом *все* версии должны быть под полным контролем, будь то документация, списки номеров телефонов, памятные записки для поставщиков, сборочные файлы, процедуры сборки и выпуска, небольшой сценарий оболочки, упорядочивающий журнальные файлы, словом — *все*. Мы регулярно пользуемся контролем версий практически для всего, что набираем сами, включая текст этой книги. И даже если мы не работаем над конкретным проектом, то все равно сохраняем для надежности свою повседневную работу в репозитории.

Ветвление

В системах контроля версий не только отслеживается предыстория конкретного проекта. К числу самых эффективных и полезных свойств таких систем относится возможность выделять отдельные участки разработки в так называемые *ветви*. Создать ветвь можно в любой момент предыстории проекта, и тогда любая работа, выполняемая в данной ветви, будет обособлена от всех остальных ветвей. А в какой-то последующий далее момент свою рабочую ветвь можно *объединить* обратно с другой ветвью, чтобы целевая ветвь содержала в итоге изменения, внесенные в рабочей ветви. Более того, в одной ветви могут работать несколько человек, и такие ветви становятся похожими в какой-то степени на небольшие клонируемые проекты.

Ветви выгодно отличаются, во-первых, тем, что позволяют работать обособленно. Так, если вы разрабатываете функциональное средство A в одной ветви, а ваш коллега — функциональное средство B в другой ветви, вы не мешаете друг другу. И во-вторых, ветви, как ни странно, нередко оказываются в центре процесса работы команды над проектом.

Именно здесь и возникает небольшая путаница. У ветвей контроля версий и организации тестов есть нечто общее: и в тех и в других участвуют тысячи людей, подсказывающих, как нужно делать. Но такие советы по большей части лишены смысла, поскольку они, по существу, означают прием, который оказался пригодным для советчика в конкретной ситуации.

Итак, пользуйтесь контролем версий в своем проекте, и если в процессе работы над ним у вас возникнут какие-нибудь затруднения, то поищите возможные пути их разрешения. И не забывайте просматривать и корректировать все, что вы делаете, постепенно набираясь необходимого опыта.

Мысленный эксперимент

Вылейте чашку вашего любимого чая на клавиатуру своего компьютера. Отнесите его в мастерскую по ремонту вычислительной техники, задав им работу. Если они не сумеют починить ваш компьютер, приобретите новый.

Сколько времени потребуется, чтобы вернуть компьютер в прежнее состояние, включая установку и настройку операционной системы, сетевого протокола, текстового редактора, приложений и т.д., когда вы только подняли злосчастную чашку чая? Подобный вопрос пришлось недавно решать одному из авторов этой книги.

Практически все, что определяло конфигурацию и применение первоначального компьютера, хранилось в системе контроля версий, в том числе следующее:

- Все пользовательские настройки и файлы конфигурации.
- Параметры настройки текстового редактора.
- Список программного обеспечения, установленного с помощью утилиты командной строки типа Homebrew.
- Сценарий Ansible для конфигурирования приложений.
- Все текущие проекты.

К вечеру компьютер был восставлен в прежнее рабочее состояние.

Контроль версий как центральный узел проекта

Несмотря на то что контроль версий невероятно удобен для работы над индивидуальными проектами, истинный его потенциал проявляется в коллективной работе. И наибольшая его ценность заключается в том, как организовано размещение репозитория.

Многим современным системам контроля версий вообще не требуется никакого размещения. Они полностью децентрализованы, и поэтому все разработчики могут на равных работать над совместным проектом. Но даже в таких системах стоит все же организовать центральный репозиторий. Ведь в этом случае можно многократно выполнять интеграцию, упрощая ход работы над проектом.

Многие репозиторные системы доступны в виде открытого исходного кода, что дает возможность установить их и работать с ними в своей организации. Но для программистов-прагматиков мы все же рекомендуем размещать репозиторий системы контроля версий на третьей стороне, где следует искать перечисленные ниже возможности.

- Хорошая защита и управление доступом.
- Интуитивный пользовательский интерфейс.
- Возможность выполнять все операции из командной строки на тот случай, если потребуется автоматизировать их.
- Автоматическое построение и тестирование.
- Хорошая поддержка объединения ветвей, иногда еще называемого запросами на включение изменений.
- Разрешение затруднений (в идеальном случае интегрировано в операции фиксации изменений и объединения ветвей для сохранения количественных показателей).
- Хорошая отчетность (может оказаться удобным отображение неразрешенных вопросов и задач).
- Удобное общение в команде: уведомления об изменениях по электронной почте или иными способами, вики-страницы и пр.

Многие команды настраивают свои системы контроля версий таким образом, чтобы отправка изменений в конкретную ветвь автоматически приводила к построению системы, выполнению тестов, а после успешного завершения — к развертыванию нового кода в условиях эксплуатации. На первый взгляд такая автоматизация операций вызывает опасение, но в ней нет ничего опасного, если ясно осознавать сущность контроля версий: любые изменения можно всегда откатить назад.

Другие разделы, связанные с данной темой

- **Тема 11**. Обратимость, **глава 2** "Прагматичный подход".
- **Тема 49**. Прагматичные команды, **глава 9** "Прагматичные проекты".
- **Тема 51**. Начальный набор инструментальных средств программиста-прагматика, **глава 9** "Прагматичные проекты".

Задачи

- Одно дело — знать, как произвести откат в предыдущее состояние, используя систему контроля версий, а другое — уметь это делать. Можете ли вы произвести такую операцию и знаете ли подходящие для этой цели команды? Овладейте ими сейчас, а не тогда, когда нагрянет беда и придется принимать срочные меры.
- Обдумайте, как вы будете восстанавливать рабочую среду на переносном компьютере после аварийной ситуации. Что именно вам придется восстанавливать? Иногда требуется восстановить лишь текстовые файлы. Если они не находятся в системе контроля версий, размещаемой в удаленном месте, найдите способ ввести их в эту систему. Затем подумайте о другом материале: установленных приложениях, конфигурации системы и т.д. Как выразить весь этот материал в текстовых файлах, чтобы сохранить и его? Добившись некоторого прогресса, попробуйте в качестве любопытного эксперимента отыскать старый компьютер, которым вы больше не пользуетесь, и выяснить, можно ли настроить его, используя новую систему контроля версий.
- Сознательно подойдите к исследованию тех различных возможностей вашей системы контроля версий и хостинга, которыми вы в настоящее время не пользуетесь. Так, если в вашей команде не применяются функциональные ветви, поэкспериментируйте с их внедрением. Это же относится к запросам на извлечение или включение изменений, непрерывной интеграции, сборочным конвейерам и даже к непрерывному развертыванию. Поищите средства, подходящие для общения в команде, в том числе вики-страницы, доски по методу канбан и пр. Пользоваться всем этим совсем не обязательно, но все же нужно знать, чем они могут помочь в принятии решений.
- Пользуйтесь контролем версий и для работы с другим материалом вне своего проекта.

Тема 20 Отладка

"Как больно глядеть на свои страданья и знать, что причинил их ты сам, и никто другой".

Софокл, ***Аякс***

Слово *bug* (жучок) служит в английском языке для описания "ужасного объекта" еще с XIV века. Контр-адмирал д-р Грэйс Хоппер, создатель языка COBOL, была первым наблюдателем *компьютерного "жучка"*, а по существу, моли, попав-

шей в одно из реле, на которых были построены первые ЭВМ. Когда техника из обслуживающего персонала попросили объяснить, почему машина ведет себя не так, как следует, он ответил, что обнаружен "жучок в системе", и с сознанием выполненного долга приклеил насекомое (вместе с крылышками и всем остальным) клейкой лентой к странице регистрационного журнала.

К сожалению, "жучки" до сих пор попадают в вычислительные системы, хотя они и не летают. Впрочем, их лучше было бы назвать бытующим с XIV века словом *bogeyman* и означающим "призрак", поскольку теперь оно уместно в гораздо большей степени, чем на заре вычислительной техники. Изъяны проявляются в программном обеспечении по-разному: от превратно понятых требований до ошибок программирования. И, к сожалению, современные вычислительные системы способны делать лишь то, что мы им *прикажем*, но совсем не обязательно то, что нам *хотелось* бы, чтобы они делали.

Никому не удается писать идеальные программы, а следовательно, их отладка занимает львиную долю рабочего времени программиста. Рассмотрим некоторые вопросы, касающиеся отладки, а также ряд общих стратегий обнаружения едва уловимых программных ошибок.

Психология отладки программ

Для многих разработчиков отладка является довольно щепетильной, чувствительной темой. Вместо желания приступить к ней как к требующей разгадки головоломке, можно встретить отказ, взаимные упреки, слабые отговорки и просто безразличие. Следует, однако, иметь в виду, что отладка — это всего лишь средство для разрешения затруднений и их устранения.

Обнаружив чью-нибудь программную ошибку, можно затратить немало времени и труда, возлагая всю вину на того нечестивца, который ее совершил. В некоторых ситуациях и коллективах это оказывается (очистительной) частью общей культуры. Но в области техники требуется сосредоточиться на устранении самого *затруднения*, а не винить других в его возникновении.

Совет 29 Устраните затруднение, а не вините в нем других

И совсем не важно, является ли программная ошибка вашей или чужой оплошностью. Она все равно остается вашей проблемой.

Мысленная установка на отладку

"Обманывать легче всего самого себя".

Эдвард Булвер-Литтон[4], ***Отвергнутый*** *(The Disowned)*

[4] Edward Bulwer-Lytton — английский писатель XIX века. — *Примеч. пер.*

Прежде чем приступить к отладке, очень важно принять соответствующий образ мыслей. Для этого необходимо устранить многие преграды, защищающие ваше самолюбие, избавиться от любого возможного нажима со стороны проекта и почувствовать себя в уютной обстановке. А самое главное — не забывать первое правило отладки, которое гласит:

Совет 30 Не паникуй!

Впасть в панику очень легко, особенно если поджимают сроки или нервничающий начальник либо клиент дышит вам в затылок в то время, как вы пытаетесь найти причину программной ошибки. И здесь очень важно отступить на шаг и тщательно обдумать возможную причину тех симптомов, которые, на ваш взгляд, указывают на программную ошибку.

Если вашей первой реакцией на свидетельство или сообщение о программной ошибке станет фраза "это невозможно", значит, вы совершенно не правы. Не тратьте ни одного нейрона на ход рассуждений, начинающийся с того, что этого просто не может быть. Ведь совершенно очевидно, что это *может* быть, и так оно и есть.

Стерегитесь близорукости во время отладки. Противьтесь сильному желанию устранить лишь видимые симптомы, ведь фактическая неисправность может отстоять на несколько шагов от того, что вы наблюдаете, и быть связанной с целым рядом других неполадок. Старайтесь всегда обнаружить корневую причину возникшего затруднения, а не только его конкретное внешнее проявление.

С чего начинать отладку

Прежде чем приступить к анализу программной ошибки, убедитесь, что работаете над кодом, при построении которого не было никаких предупреждений компилятора. Нет никакого смысла тратить время, пытаясь выявить неполадку, которую компьютер может обнаружить автоматически! Вам нужно сосредоточиться на решении более трудных насущных задач.

Пытаясь разрешить любое затруднение, необходимо собрать все уместные данные. К сожалению, сообщения о программных ошибках для этого не совсем годятся из-за своей неточности. Ведь совпадения могут очень легко ввести в заблуждение, а тратить время на отладку совпадений непозволительно. Прежде всего необходимо быть точным в своих наблюдениях.

Точность сообщений о программных ошибках снижается еще больше, когда они проходят через стороннее программное обеспечение. Чтобы добиться достаточного уровня детализации, возможно, придется *понаблюдать* за действиями пользователя, сообщившего о программной ошибке.

Однажды Энди работал над крупным графическим приложением. Ближе к выпуску этого приложения тестировщики сообщили, что оно завершалось аварийно всякий раз, когда они наносили мазок конкретной кистью. Программист, отвечавший за эту часть приложения, возразил, что в таком поведении нет ничего неверного. Он попробовал раскрасить участок изображения данной кистью, и она действовала как следует. Эта полемика продолжалась в течение нескольких дней, и всякий раз на быстро повышавшихся тонах.

Наконец, мы собрались вместе в одном помещении. Тестировщик выбрал инструмент кисти и нанес мазок от правого верхнего до нижнего левого угла экрана, в результате чего приложение перестало работать. Программист удивленно воскликнул сдавленным голосом и смущенно признался, что наносил пробные мазки кистью только от нижнего левого до правого верхнего угла экрана, что не приводило к появлению ошибки.

Из этой истории можно сделать следующие выводы.

- С пользователем, сообщившим о программной ошибке, возможно, придется побеседовать, чтобы получить больше сведений, чем удалось собрать первоначально.
- Искусственных тестов вроде единственного мазка, проведенного кистью снизу вверх, недостаточно для проверки приложения. Необходимо подвергнуть безжалостному тестированию как граничные условия, так и реалистичные образцы его употребления конечным пользователем. И делать это следует систематически (см. раздел "Строгое и непрерывное тестирование" главы 9 "Прагматичные проекты").

Стратегии отладки

Как только *вы* определите, что, собственно, происходит, выясните, как это представляется в самой *программе*.

Воспроизведение ошибок

Нет, наши ошибки в действительности не размножаются (хотя некоторые из них, вероятно, достаточно стары, чтобы это было законным). Речь здесь идет о другом способе размножения.

Чтобы приступить к устранению программной ошибки, ее нужно сделать воспроизводимой. Ведь если воспроизвести ошибку нельзя, то как выяснить, была ли она вообще устранена?

Но ведь нам требуется выяснить нечто большее, чем программная ошибка, которая может быть воспроизведена путем выполнения некоторой длинной последовательности шагов. Мы хотим, чтобы можно было воспроизвести программную ошибку с помощью *единственной команды*. Ведь устранить программную ошибку намного сложнее, если, чтобы дойти до того места, где она проявляется, придется выполнить 15 шагов. Поэтому самое важное правило отладки гласит:

Совет 31 Вылавливающий ошибку тест должен предшествовать ее исправлению

Иногда, пытаясь выделить обстоятельства, при которых проявляется ошибка, можно даже получить ясное представление, как ее устранить. Сам процесс написания теста подсказывает нужное решение.

Программист в чужой стране

Все сказанное выше о локализации программной ошибки, конечно, замечательно. Но что делать бедному программисту, если требуется отладить 50 тысяч строк кода, когда времени явно не хватает?

Прежде всего рассмотрите проблему. Идет ли речь об аварийном завершении приложения? Когда мы преподаем на курсах, включающих программирование, нас всегда удивляет, как много разработчиков, увидев появление исключения в красном поле результатов тестирования, сразу же переходят к исходному коду.

Совет 32 Внимательно читайте сообщение об ошибке, каким бы отвратительным оно ни было

К этому добавить нечего.

Плохие результаты

А что, если это не аварийное завершение, а всего лишь плохой результат? В таком случае перейдите в отладчике к месту его формирования и выполните тест, чтобы выявить ошибку.

Прежде чем делать что-нибудь другое, непременно выявите неверное значение в отладчике. Мы оба часами пытались отследить программную ошибку лишь затем, чтобы обнаружить, что данное конкретное выполнение кода завершилось удачно.

Иногда ошибка вполне очевидна. Например, значение переменной `interest_rate` равно `4.5`, а должно быть равно `0.045`. Но чаще всего ошибку приходится искать глубже, чтобы выяснить, прежде всего, причину, по которой значение оказалось неверным. Поэтому убедитесь сначала, что знаете, как перемещаться вверх и вниз по стеку вызовов, а затем проанализируйте окружение стека.

Нередко полезно держать под другой лист бумаги и ручку, чтобы делать заметки. В частности, мы часто находим ключ к разгадке и, идя по следу, не достигаем в конечном итоге успеха. И если не записать место, с которого мы начали свой поиск, то можно потерять много времени, возвращаясь назад.

Иногда приходится анализировать результаты трассировки стека, прокрутка которых кажется бесконечной. В таком случае *бинарный поиск* нередко оказы-

вается более быстрым способом выявить неполадку, чем анализ всех фреймов стека подряд. Но, прежде чем обсуждать этот способ, рассмотрим две другие общие ситуации возникновения программных ошибок.

Чувствительность ко входным данным

Эта ситуация должна быть хорошо вам знакома. Сначала ваша программа нормально работает со всеми тестовыми данными и с честью переживает первую неделю своей эксплуатации, а затем неожиданно терпит крах, когда в нее вводится некоторый конкретный массив данных.

Вы можете попытаться проанализировать свою программу в месте ее аварийного завершения и продвигаться далее в обратном направлении. Сделайте копию массива данных и введите его в локальную копию своей программы, убедившись, что она по-прежнему терпит крах. Затем выполняйте бинарный поиск до тех пор, пока не дойдете до конкретного места, где вводимые значения приводят к аварийному завершению.

Регрессии по выпускам

Представьте, что вы работаете в хорошей команде, выпускающей программное обеспечение в эксплуатацию. В какой-то момент в исходном коде, который нормально работал еще неделю назад, обнаруживается программная ошибка. Не стоит ли в таком случае выявить конкретное изменение, внедрившее данную ошибку? И, как вы, вероятно, уже догадались, что самое время для бинарного поиска, о котором речь пойдет далее.

Бинарный поиск

Всякому студенту, изучавшему вычислительную технику, приходилось программировать бинарный поиск, иначе называемый делением пополам. В его основу положена довольно простая идея: найти конкретное значение в отсортированном массиве. Для этого можно было бы, конечно, проанализировать поочередно каждое значение, но тогда пришлось бы перебрать в среднем около половины элементов массива до тех пор, пока не будет найдено требуемое или большее значение. Последнее означает, что искомое значение отсутствует в массиве.

Но найти нужное значение можно намного быстрее, воспользовавшись методом "разделяй и властвуй". С этой целью выбирается значение посредине массива. Если это искомое значение, то поиск прекращается, а иначе массив делится надвое. Если найденное значение больше искомого, то становится ясно, что оно должно быть в первой половине массива, а иначе — во второй. Данная процедура повторяется в соответствующем подмассиве, и очень скоро достигается искомый результат. (Как поясняется в разделе "Асимптотическое обозначение" главы 7 "По ходу кодирования", линейный поиск имеет сложность $O(n)$, а бинарный — $O(\log n)$).

Таким образом, бинарный поиск позволяет намного быстрее решить любую задачу приличных масштабов. А теперь посмотрим, как применить его к отладке.

Когда вы анализируете результаты массивной трассировки стека, пытаясь выяснить, какая именно функция ошибочно исказила значение, то выполняете бинарный поиск, выбрав кадр где-то посредине стека, чтобы посмотреть, не проявилась ли ошибка до этого места. Если она проявилась, то становится ясно, что основное внимание следует сосредоточить на предыдущих фреймах; в противном случае следует искать неполадку в последующих фреймах стека. Затем бинарный поиск повторяется, и даже если в результатах трассировки стека присутствуют 64 кадра, подобным методом можно найти причину ошибки, сделав не более шести попыток.

Если вы ищете программные ошибки, проявляющиеся в определенных массивах данных, то можете поступить таким же образом, как описано выше. В частности, разделите массив данных надвое и выясните, проявится ли ошибка, если ввести одну или другую его половину в отлаживаемое приложение. Продолжайте далее разделять массив данных до тех пор, пока не получите минимальный массив значений, в котором проявляется искомая ошибка.

Если ваша команда внесла программную ошибку давно, и с тех пор прошло уже несколько выпусков, то и в этом случае можно воспользоваться тем же самым методом. С этой целью создайте тест, который не проходит в текущем выпуске, а затем выберите промежуточный вариант между текущей и последней версией, о которой заведомо известно, что она рабочая. Выполните тест снова, чтобы решить, каким образом следует сузить поиск. Возможность такого решения — одно из многих преимуществ, которые дает наличие качественного контроля версий в ваших проектах. Безусловно, многие системы контроля версий позволяют пойти еще дальше и автоматизировать данный процесс, самостоятельно выбирая выпуски по результатам тестирования.

Протоколирование и/или трассировка

Как правило, отладчики сосредоточиваются на *текущем* состоянии программы. Но иногда требуется нечто большее: наблюдать состояние программы или структуры данных во времени. Просмотр результатов трассировки стека позволяет лишь выяснить, как непосредственно дойти до текущего состояния. Он, как правило, не позволяет выяснить, что именно было сделано до данной цепочки вызовов, особенно в событийно-управляемых системах[5].

Операторы трассировки являются небольшими диагностическими сообщениями, выводимыми на экран или в файл и уведомляющими, например, о том, что выполнение программы дошло до конкретного места или значение переменной *x* стало равным 2. И хотя это довольно примитивный метод по сравнению с возможностями отладчиков в интегрированных средах разработки, он все же

[5] Хотя в языке Elm имеется отладчик, допускающий перемещение во времени.

особенно эффективен для диагностики нескольких классов ошибок, выявить которые отладчики неспособны. Трассировка неоценима в любой системе, где важным фактором в одновременно выполняющихся процессах, системах реального времени и событийно-управляемых приложениях является само время. Используя операторы трассировки, можно подробно анализировать исходный код вглубь, т.е. вводить их, продвигаясь вниз по дереву вызовов.

Сообщения трассировки должны быть представлены в обычном согласованном формате, чтобы их можно было анализировать автоматически. Так, если требуется проследить утечку ресурсов (например, неравномерность открытий и закрытий файлов), с этой целью можно протрассировать каждый оператор `open` и `close` в журнальном файле. Обрабатывая журнальный файл инструментальными средствами для обработки текста или по командам оболочки, можно легко определить место, где появился оператор `open`, нарушивший равномерность открытий и закрытий файлов.

Метод резинового утенка

Самый простой, но особенно полезный способ найти причину неполадки — просто объяснить ее кому-то другому. Он должен глядеть на экран через ваше плечо, постоянно кивая головой, как резиновый утенок, ныряющий и всплывающий в ванной. Не говоря ни слова, а просто показывая шаг за шагом, что именно должен делать код, нередко можно дойти до того, что неполадка вынырнет на экран и проявится сама[6].

Казалось бы, все просто, но, объясняя возникшее затруднение другому человеку, вам приходится растолковывать ему то, что может быть само собой разумеющимся для вас, когда вы просматриваете исходный код сами. Выражая словами свои предположения, вы можете неожиданно получить новое представление о возникшем затруднении. А если вам некому объяснить свое затруднение, то вполне подойдет и резиновый утенок, плюшевый медвежонок или растение в горшке.

Процесс исключения

В большинстве проектов отлаживаемый прикладной код может состоять из фрагментов, написанных вами и другими членами команды, работающей над проектом, сторонних программных продуктов (базы данных, средств подключения к сети, каркаса веб-приложений, специальных протоколов связи или ал-

[6] Почему данный метод носит название "резинового утенка"? Когда Дэйв учился в Имперском колледже Лондона, он много работал с младшим научным сотрудником Грегом Пафом (Greg Pugh), одним из самых лучших из всех известных ему разработчиков. Грег целыми месяцами носил с собой небольшого желтого резинового утенка и ставил его на свой терминал, когда программировал. И лишь некоторое время спустя Дэйв отважился спросить, зачем Грег это делает...

горитмов и т.д.), а также платформенного окружения (операционной системы, системных библиотек и компиляторов).

Вполне возможно, что программная ошибка присутствует в операционной системе, компиляторе или стороннем программном продукте, хотя это и не должно быть вашей первой мыслью. Ведь намного более вероятно, что программная ошибка присутствует в разрабатываемом прикладном коде. Как правило, предположить, что в прикладном коде неверно вызывается библиотека, целесообразнее, чем допустить, что нарушена сама библиотека. И даже если неполадка кроется в стороннем программном продукте, то нужно все равно устранить неполадки в своем коде, прежде чем предоставить отчет о программных ошибках.

Нам как-то пришлось работать над проектом, где старший инженер был убежден, что системный вызов `select` нарушался в самой операционной системе Unix. И никакие убедительные доводы или логические обоснования не могли изменить его мнение, а тот факт, что все остальные сетевые приложения работали на данной платформе нормально, не принимался во внимание. Он потратил не одну неделю, изобретая обходные пути, которые по какой-то странной причине все же не устраняли неполадку. И когда ему пришлось в конечном счете сесть и прочитать документацию на системный вызов `select`, он обнаружил и устранил неполадку в считанные минуты. С тех пор мы пользуемся выражением "системный вызов `select` нарушен" в качестве осторожного напоминания, как только один из нас начинает винить в сбое, который, вероятнее всего, происходит по его вине, саму систему.

Совет 33 Системный вызов **`select`** работает нормально

Увидев следы копыт, не забывайте подумать прежде всего о конях, а не зебрах. Операционная система скорее всего работает нормально, и системный вызов `select` действует как следует.

Если вы внесли изменение лишь в какой-то один компонент и система перестала работать, то, вероятнее всего, именно этот компонент, прямо или косвенно, служит тому причиной, какой бы неправдоподобной она ни казалась. Иногда измененный компонент оказывается вне вашего поля зрения. Это может быть, например, новая версия операционной системы, компилятор или другое стороннее программное обеспечение, способное повредить код, который раньше работал нормально. Могут проявиться новые программные ошибки, нарушающие обходные пути, устраняющие обнаруженные ранее ошибки. Изменяются также API, функциональные возможности. Короче говоря, возникает совсем новая ситуация, вынуждающая снова проверять всю систему в этих новых условиях. Поэтому внимательно следите за графиком выполнения работ, если планируете модернизацию, которую, возможно, придется отложить до момента *после* следующего выпуска.

Элемент удивления

Если программная ошибка приведет вас в недоумение, а возможно, даже и к бормотанию шепотом "этого просто не может быть" так, чтобы никто не услышал, — в таком случае вам придется переоценить те истины, которыми вы так дорожите. Выполняя этот алгоритм учета поправки на преувеличение того, что казалось вам вполне надежным и не могло вызвать эту программную ошибку, спросите себя: проверили ли вы *все* граничные условия? И не кроется ли до сих пор программная ошибка в том фрагменте кода, которым вы пользуетесь годами? Возможно ли такое вообще?

Конечно, возможно. Степень удивления, которое вы испытываете, когда что-нибудь не ладится, пропорциональна степени доверия и веры в непогрешимость выполняемого кода. Именно поэтому, сталкиваясь с "удивительным" сбоем, вы должны признать, что одно или несколько ваших предположений неверны. Не обходите стороной подпрограмму или фрагмент кода, связанный с ошибкой, лишь потому, что твердо уверены в его работоспособности. Убедитесь в том, что в *таком* контексте, с *такими* данными и при *таких* граничных условиях проверяемый код действительно работает.

Совет 34 Не предполагайте, а доказывайте

Обнаружив удивительную программную ошибку, необходимо не только устранить ее, но и установить причину, по которой ее не удалось перехватить. Выясните, следует ли исправить модульные или другие тесты, чтобы они могли выявлять данную ошибку.

А если программная ошибка возникает в результате распространения некачественных данных через пару уровней, прежде чем привести к сбою, то выясните, можно ли выделить ее раньше, улучшив проверку параметров в соответствующих подпрограммах. (Подробнее о досрочном аварийном завершении и утверждениях см. соответственно в разделе "Принцип "поймал-отпустил" — только для ловли рыбы" и в разделе "Тема 25. Утвердительное программирование" главы 4 "Прагматичная паранойя".)

Проверьте также, есть ли какие-нибудь другие места в исходном коде, которые можно подозревать на проявление той же самой ошибки? Самое время найти их и устранить в них эту ошибку. Убедитесь, что чтобы ни случилось, если это произойдет вновь, вы будете об этом знать.

Если устранение ошибки отнимает много времени, выясните, почему происходит именно так и можно ли сделать что-нибудь, чтобы упростить устранение данной ошибки в следующий раз. Возможно, стоит встроить более совершенные перехватчики для тестирования или написать анализатор журнальных файлов.

Наконец, обсудите возникшее затруднение со всей командой, если программная ошибка является результатом чьего-то неверного предположения. Ведь если заблуждается один человек, то могут заблуждаться и многие люди. Если вы сделаете все написанное выше, то можно надеяться, что в следующий раз возникающие программные ошибки удивят вас гораздо меньше.

Контрольный список вопросов по отладке

Ниже перечислены вопросы, которые следует задавать в процессе отладки программ.

- Является ли рассматриваемая неполадка непосредственным результатом или только симптомом скрытой программной ошибки?
- Присутствует ли программная ошибка в используемом вами каркасе, операционной системе или же в вашем коде?
- Если бы вам довелось подробно объяснять возникшее затруднение коллеге, то что бы вы ему сказали?
- Если подозрительный код проходит свои модульные тесты, то насколько они полноценны? Что произойдет, если выполнить эти тесты с *конкретными* данными?
- Существуют ли условия, при которых возникает данная программная ошибка, где-нибудь еще в системе? Находятся ли какие-нибудь другие скрытые ошибки, ожидающие возможности своего появления на свет?

Другие разделы, связанные с данной темой

- **Тема 24.** Мертвые программы не лгут, **глава 4** "Прагматичная паранойя".

Задача

- Отладка — довольно трудная задача.

Тема 21 Манипулирование текстом

Программисты-прагматики манипулируют текстом таким же образом, как и столяры, обрабатывающие древесину, придавая ей нужную форму. В предыдущих разделах упоминался ряд конкретных инструментальных средств (оболочки, редакторы, отладчики), которыми пользуются программисты-прагматики. Их можно сравнить со стамесками, пилами и рубанками, применяемыми для выполнения тех или иных столярных работ. Но иногда требуется выполнить преобразование, на которое неспособен основной набор инструментальных средств. И для этой цели требуется универсальное инструментальное средство манипулирования текстом.

В программировании языки манипулирования текстом выполняют те же функции, что и фрезерные станки в столярном деле. Это шумные, тяжелые инструменты, применяющие грубую силу. Стоит совершить ошибку, пользуясь ими, как вся заготовка разлетается на куски. Некоторые клянутся, что таким инструментам нет места в их арсенале. Но в хороших руках как фрезерные станки, так и языки манипулирования текстом могут оказаться невероятно эффективными и универсальными инструментами. С их помощью можно быстро обрезать заготовку, выполнить соединение и вырезать нужную форму. Но для овладения ими требуется время.

Правда, существует целый ряд отличных языков манипулирования текстом. Так, разработчики приложений для Unix, включая и пользователей macOS, нередко любят использовать весь потенциал своих командных оболочек, подкрепленных такими инструментальными средствами, как `awk` и `sed`. А те, кому больше нравятся структурированные инструментальные средства, могут отдать предпочтение таким языкам, как Python или Ruby.

Такие языки являются важными базовыми технологиями. С их помощью можно быстро проработать утилиты и прототипные идеи, т.е. решить те задачи, на решение которых может потребоваться в пять–десять раз больше времени, чем с помощью обычных языков. А ведь такая многократная экономия времени очень важна для проведения экспериментов. В самом деле, потратить всего лишь 30 минут на опробование какой-нибудь безумной идеи намного привлекательнее, чем израсходовать на это целых пять часов. Автоматизацией важных компонентов проекта вполне приемлемо заниматься целый рабочий день, но не целую неделю. В своей книге *Практика программирования* [KP99] Керниган и Пайк приводят пример создания одной и той же программы на пяти разных языках. Самой краткой оказалась версия на языке Perl (она состояла из 17 строк кода по сравнению со 150 строками кода на языке C). Используя Perl, можно манипулировать текстом, взаимодействовать с программами, обмениваться сообщениями по сети, вести веб-страницы, выполнять арифметические операции с произвольной точностью и писать программы, похожие на символические реплики Снупи[7].

Совет 35 Изучите язык манипулирования текстом

Чтобы продемонстрировать применимость языков манипулирования текстом в обширных пределах, приведем ниже ряд примеров приложений, разработан-

[7] Snoopy (любопытный) — персонаж в виде песика породы бигль из популярной серии комиксов *Peanuts* (Арахис), созданных американским художником Чарльзом М. Шульцем в 1950-е годы. — *Примеч. пер.*

ных нами на языках Ruby и Python и непосредственно связанных с написанием этой книги.

- **Составление книг.** Система составления книг для издательства Pragmatic Bookshelf написана на языке Ruby. Авторы, редакторы, верстальщики и обслуживающий персонал пользуются задачами Rake для координации процессов составления текста в форматах PDF и электронных книг.
- **Включение и выделение исходного кода.** Мы считаем очень важным проверить сначала любой фрагмент кода, представленный на страницах книги, что и было проделано со всеми примерами кода из этой книги. Но, следуя принципу DRY (см. раздел "Тема 9. DRY — пороки дублирования" главы 2 "Прагматичный подход"), нам не хотелось бы копировать и вставлять строки кода из проверенных программ в текст книги. Ведь это означало бы дублирование кода и буквально гарантировало бы, что мы забудем обновить пример кода, когда соответствующая программа изменится. В некоторых примерах отсутствует каркасный код, требующийся для компиляции и выполнения этих примеров, и поэтому мы обратились к услугам Ruby. В частности, для форматирования книги вызывается простой сценарий, извлекающий именованные сегменты из исходного файла, производится синтаксический анализ выделенного сегмента, выделяется синтаксис и полученный результат преобразуется в синтаксис применяемого языка типографского набора.
- **Обновление веб-сайта.** Для этой цели у нас имеется простой сценарий, частично составляющий книгу, извлекающий ее содержание, а затем обновляющий его на странице нашего веб-сайта, посвященной данной книге. Кроме того, у нас имеется сценарий, извлекающий отдельные разделы книги и выгружающий их на соответствующую страницу нашего веб-сайта в качестве образцов текста книги.
- **Включение уравнений.** Для этой цели имеется сценарий, преобразующий разметку математических формул, набранных в редакторе LaTeX, в аккуратно отформатированный текст.
- **Формирование предметного указателя.** Большинство предметных указателей к книгам создаются как отдельные документы, что затрудняет их сопровождение при внесении изменений в книгу. Составленные нами предметные указатели размечены в тексте самой книги, а по сценарию Ruby они автоматически составляются и форматируются по отдельным статьям.

Эти примеры можно продолжить. На практике издания Pragmatic Bookshelf построены на манипулировании текстом. И если вы последуете нашему совету хранить данные в виде простого текста, то, пользуясь языками манипулирования текстом, сможете извлечь для себя немало выгод.

Другие разделы, связанные с данной темой

- **Тема 16.** Сила простого текста.
- **Тема 17.** Игры в скорлупки.

Упражнения

11. Допустим, вы переписываете приложение, в котором раньше для конфигурирования использовался формат YAML. А теперь ваша организация перешла на стандартизированный формат JSON, и поэтому вам придется преобразовать целый ряд файлов с расширением `.yaml` в файлы с расширением `.json`. Напишите с этой целью сценарий, принимающий каталог и преобразующий каждый файл с расширением `.yaml` в файл с расширением `.json` таким образом, чтобы файл `database.yaml` стал, например, файлом `database.json`, а его содержимое — достоверно преобразовано в формат JSON.
12. Допустим, ваша команда первоначально решила обозначать имена переменных в `ГорбатомРегистре`, а затем изменила свое коллективное решение, перейдя на `змеиный_регистр`. Напишите сценарий, просматривающий имена в `ГорбатомРегистре` во всех исходных файлах и сообщающий о них.
13. Продолжая предыдущее упражнение, добавьте в написанный сценарий возможность автоматически изменять имена переменных в одном или нескольких исходных файлах. Не забудьте создать резервные копии оригиналов на тот случай, если что-то пойдет совсем не так, как предполагалось.

Тема 22 Технические дневники

Дэйв как-то работал в небольшой компании, производившей вычислительную технику. Это означало, что ему пришлось работать вместе с инженерами-электронщиками, а иногда и с инженерами-механиками. Многие из них ходили с бумажной записной книжкой и ручкой, прикрепленной к переплету такой книжки. И когда они общались с Дэйвом, то доставали свою записную книжку, открывали ее и что-то записывали в нее.

В конечном счете Дэйв задал им вполне очевидный запрос, и оказалось, что они приучены вести *технические дневники*, т.е. своего рода журналы повседневного учета, в которых они записывают все, что сделали, чему научились, наброски идей, показания измерительных приборов, а по существу, все, что связано с их работой. Как только подобный дневник заполнялся, они надписывали на корешке интервалы дат ведения дневника и клали его на полку рядом с предыдущими дневниками. Между инженерами могло даже возникнуть соревнование, чьи дневники займут больше места на полке.

Мы пользуемся техническими дневниками на совещаниях, чтобы записать в них то, над чем работаем, занести значения переменных во время отладки, оставить памятки, что и где разместить, набросать самые безумные идеи, а иногда и просто рисовать машинально "чертиков"[8].

Техническим дневникам присущи следующие достоинства.

- Они более надежны, чем память. Так, если вас спросят: "Как называлась компания, к которой вы обращались на прошлой неделе по поводу энергоснабжения?", вы, пролистав свой дневник на несколько страниц назад, сможете сообщить название и номер телефона данной компании.
- В них есть место для хранения идей, которые не сразу оказываются пригодными для решения насущной задачи. Подобным образом вы можете и дальше сосредоточиться на том, что делаете, зная, что ваша замечательная идея не будет забыта.
- Они действуют подобно резиновому утенку, описанному ранее в разделе "Метод резинового утенка". Как только вы перестанете что-то записывать в дневник, ваш ум может переключиться на другие мысли, как если бы во время беседы с кем-нибудь у вас появилась возможность поразмыслить над сказанным. Вы можете начать запись в дневнике и вдруг осознать, что сделанная только что запись просто неверна.

У технических дневников имеется еще одно преимущество. Время от времени вы можете просматривать свои записи, сделанные много лет назад, вспоминая людей, проекты, ужасные одежды и прически.

Итак, старайтесь вести свой технический дневник, пользуясь бумажной записной книжкой, а не файлом или вики-страницей, поскольку в записывающем действии есть нечто особенное по сравнению с набором текста. Через месяц ведения такого дневника вы почувствует все его преимущества, перечисленные выше. Во всяком случае, это упростит написание воспоминаний, когда вы разбогатеете и добьетесь известности.

Другие разделы, связанные с данной темой

- **Тема 6**. Ваш багаж знаний, **глава 1** "Философия прагматизма".
- **Тема 37**. Прислушивайтесь к своим инстинктам, **глава 7** "По ходу кодирования".

[8] Имеется ряд свидетельств, что машинальное рисование "чертиков" и прочих каракулей помогает сосредоточиться и совершенствует познавательные навыки. Примеры тому см. в книге *What does doodling do?* [And10].

Глава 4

Прагматичная паранойя

Совет 36 Написать идеальную программу нельзя

Задевает? А не должно бы. Примите этот совет как аксиому, оценив и приветствовав его, поскольку идеального программного обеспечения не существует. Никому за краткую историю вычислительной техники еще не удавалось написать идеальную программу, и вы вряд ли будете первым. Если же вы не примете это как данность, то в конечном счете потратите зря время и силы, преследуя неосуществимую мечту. Так как же программисту-прагматику извлечь выгоду из этой гнетущей реальности? Именно данной теме и посвящена эта глава.

Каждый считает, что он является самым лучшим водителем на свете, а всем остальным далеко до такого лихача, проскакивающего запрещающие знаки, лавирующего между дорожными полосами, не подавая сигналы поворота, набирающего текст сообщений на мобильном телефоне и вообще живущего не по правилам. А мы вот ездим осторожно, остерегаясь беды еще до того, как она случится, предвидя неожиданное и никогда не ставя себя в такое положение, из которого нельзя выпутаться.

Аналогия с программированием вполне очевидна. Мы постоянно имеем дело с чужим кодом, который может и не соответствовать высоким стандартам, а также с входными данными, которые могут и не быть достоверными. Поэтому мы приучены программировать, будучи начеку. И если у нас возникает какое-нибудь сомнение, то мы проверяем всю данную нам информацию. Для выявления некачественных данных мы пользуемся проверками, не доверяя данным, получаемым от потенциальных атакующих злоумышленников или троллей. Мы выполняем проверку на согласованность, накладываем ограничения на столбцы в таблице базы данных и, в общем, чувствуем себя прекрасно.

Но программист-прагматик идет дальше, *не доверяя даже себе*. Отлично зная, что никому не дано написать идеальный код, включая и нас самих, программисты-прагматики выстраивают бастионы защиты против собственных ошибок.

Первая защитная мера описывается в разделе "Проектирование по контракту", когда клиенты и поставщики должны согласовать права и обязанности.

Затем в разделе "Мертвые программы не лгут" поясняется, что обработка программных ошибок не наносит никакого вреда. Поэтому следует стараться производить проверки как можно чаще и прерывать программу, если дело примет скверный оборот. А в разделе "Утвердительное программирование" описывается простой метод оперативной проверки, который состоит в написании кода, активно проверяющего предположения программиста.

По мере того как ваша программа становится все более динамичной, вы обнаруживаете, что манипулируете системными ресурсами (оперативной памятью, файлами, устройствами и т.п.), словно жонглируя шарами. Поэтому в разделе "Как сбалансировать ресурсы" предлагаются способы не потерять ни одного такого шара. А самое главное — продвигаться к цели мелкими шагами, чтобы не свалиться в пропасть с края утеса, как поясняется в разделе "Не опережайте свет фар вашего автомобиля".

В мире несовершенных систем, нелепых временных шкал, забавных инструментов и невыполнимых требований приходится действовать осторожно. Как однажды сказал известный кинорежиссер Вуди Аллен: "Когда все фактически строят вам козни, единственная хорошая мысль — это паранойя".

Тема 23 Проектирование по контракту

"Ничто так не поражает людей,
как здравый смысл и откровенность".

Ральф Уолдо Эмерсон[1], ***Очерки*** *(Essays)*

Обращаться с вычислительными системами нелегко, а иметь дело с людьми еще труднее. Но как биологический вид мы с давних пор выясняли вопросы человеческих взаимоотношений. Некоторые решения, к которым мы пришли за последние несколько тысячелетий, могут пригодиться и для написания программ. И одним из самых лучших решений, гарантирующих прямоту отношений, является *контракт*. Контракт определяет права и обязанности обеих заключивших его сторон. Кроме того, в нем предусмотрено соглашение сторон, касающееся последствий, если одна из них не выполнит контракт.

Так, вы могли заключить контракт о найме, в котором указано сколько часов вы должны работать, а также правила, которые вы обязаны соблюдать. В свою очередь, работодатель платит вам зарплату и остальные надбавки. Если каждая сторона выполняет свои обязательства, то выигрывают от этого все.

[1] Ralph Waldo Emerson — американский эссеист, поэт, философ, пастор, общественный деятель XIX века. — *Примеч. пер.*

Идея контракта используется во всем мире (формально или неформально) с целью помочь людям наладить взаимоотношения. А может ли тот же самый принцип оказать помощь в налаживании взаимодействий между программными модулями? Да, может.

Принцип проектирования по контракту

Бертран Мейер (Bertrand Meyer; *Object-Oriented Software Construction* [Mey97]) разработал принцип *проектирования по контракту* для языка Eiffel[2]. Это простая, но эффективная методика, направленная на документирование (и согласование) прав и обязанностей программных модулей, чтобы обеспечить правильность программы. А что такое правильная программа? Это такая программа, которая выполняет только то, что от нее требуется, — ни больше и ни меньше. Документирование и верификация такого требования и составляет саму суть *проектирования по контракту* (Design by Contract, DBC).

Каждая функция и метод в программной системе *выполняет* какое-то *действие*. Прежде чем начать это *действие*, функция может предполагать какое-то состояние окружающего мира, а по завершении она может сделать заявление о состоянии окружающего мира. Эти предположения и требования Мейер описывает следующим образом.

- **Предусловия.** Это требования подпрограммы, которые определяют, что должно быть истинным для ее вызова. Подпрограмма вообще не должна вызываться, если ее предусловия будут нарушены. На вызывающий код возлагается ответственность за передачу качественных данных (см. врезку "Кто несет ответственность" далее в этой главе).
- **Постусловия.** Это состояние окружающего мира по завершении подпрограммы, т.е. то, что ею гарантируется. Наличие постусловия у подпрограммы подразумевает, что она непременно *завершится*, а следовательно, бесконечные циклы не допускаются.
- **Инварианты класса.** Класс гарантирует, что данное условие для вызывающего кода всегда истинно. В процессе внутренней обработки в подпрограмме инвариант может и не соблюдаться, но к моменту выхода из подпрограммы и передачи управления вызывающему коду инвариант должен быть непременно истинным. (Следует, однако, иметь в виду, что класс не может предоставить неограниченный доступ для записи к любому члену данных, принимающему участие в инварианте.)

Таким образом, контракт между подпрограммой и любым потенциально вызывающим кодом может быть составлен следующим образом.

---

[2] Частично на основании прежних трудов Дейкстры (Dijkstra), Флойда (Floyd), Хоара (Hoare), Вирта (Wirth) и других.

- Если все предусловия подпрограммы удовлетворяются вызывающим кодом, то подпрограмма гарантирует истинность всех постусловий и инвариантов при своем завершении.

Если же одна из сторон контракта не выполняет его условия, то вызывается (предварительно согласованное) средство защиты нарушенных прав, например: возникает исключение или прерывание программы. Что бы ни случилось, не сомневайтесь, что несоблюдение условий контракта считается программной ошибкой. Нет никакой гарантии, что этого вообще не произойдет, именно поэтому предусловия и нельзя использовать для выполнения таких операций, как проверка вводимых пользователем данных на достоверность.

В одних языках эти принципы поддерживаются лучше, чем в других. Например, в языке Clojure поддерживаются предусловия и постусловия, а также более обширный инструментарий, предоставляемый по *спецификации*. Ниже приведен пример функции для открытия вклада в банке с помощью пред- и постусловий.

```
(defn accept-deposit [account-id amount]
  { :pre [ (> amount 0.00)
           (account-open? account-id) ]
    :post [ (contains? (account-transactions account-id) %) ] }
  "Accept a deposit and return the new transaction id"
  ;; Здесь выполняется остальная обработка...
  ;; Возвратить вновь созданную транзакцию:
  (create-transaction account-id :deposit amount))
```

Для функции `accept-deposit()` здесь заданы два предусловия. Первое состоит в том, что сумма вклада должна быть больше нуля, а второе — в том, что счет должен быть открыт и действителен. Последнее определяется путем вызова функции `account-open?()`. Имеется также следующее постусловие: функция гарантирует, что новая транзакция (возврат из данной функции значения, представленного в процентах `"%"`) может находиться среди прочих транзакций для данного счета.

Если вызвать функцию `accept-deposit()` с положительной суммой для открытия вклада и действительным счетом в банке, то она создаст далее транзакцию подходящего типа и выполнит всю остальную требующуюся обработку. Но если в программе имеется ошибка и данной функции каким-то образом передана отрицательная сумма для открытия вклада, то во время выполнения возникнет следующее исключение:

```
Exception in thread "main"...
Caused by: java.lang.AssertionError:
           Assert failed: (> amount 0.0)
```

Кроме того, данная функция требует, чтобы указанный счет был открыт и действителен. В противном случае появится такое исключение:

```
Exception in thread "main"...
Caused by: java.lang.AssertionError:
          Assert failed: (account-open? account-id)
```

В других языках имеются средства, дающие неплохой результат, несмотря на то, что они не относятся непосредственно к проектированию по контракту. Например, в языке Elixir применяются *операторы защиты* для диспетчеризации вызовов функции по нескольким имеющимся ее телам:

```
defmodule Deposits do
  def accept_deposit(account_id, amount) when (amount > 100000) do
    # Вызвать диспетчер!
  end
  def accept_deposit(account_id, amount) when (amount > 10000) do
    # Дополнительные федеральные требования для отчетности
    # Здесь выполняется некоторая обработка...
  end
  def accept_deposit(account_id, amount) when (amount > 0) do
    # Здесь выполняется некоторая обработка...
  end
end
```

В данном случае вызов функции `accept_deposit()` с крупной суммой может потребовать дополнительных шагов и обработки. Но если попытаться вызвать данную функцию с суммой, меньшей или равной нулю, то возникнет приведенное ниже исключение, которое извещает, что сделать такой вклад нельзя.

```
** (FunctionClauseError) no function clause
   matching in Deposits.accept_deposit/2
```

Это лучше, чем просто проверять входные данные. В данном случае функцию `accept_deposit()` просто нельзя вызвать, если ее аргументы указаны вне заданных пределов.

Совет 37 Проектируйте по контракту

В разделе "Тема 10. Ортогональность" главы 2 "Прагматичный подход" мы рекомендовали писать "скромный" код. А здесь акцент делается на "ленивый" код, предусматривающий строго относиться к тому, что принимается, прежде чем начинать что-нибудь делать, а также обещать как можно меньше. Не следует, однако, забывать, что если в контракте вы согласились принимать все, что угодно, и наобещали на выходе целый мир, то для соблюдения такого контракта вам придется написать немало кода! При программировании на любом языке, будь то функциональном, объектно-ориентированном или процедурном, проектирование по контракту заставляет вас *думать*.

Инварианты класса и функциональные языки

Все дело в названиях. Язык Eiffel является объектно-ориентированным, и поэтому Мейер назвал данное понятие "инвариантом класса". Но на самом деле это более общее понятие, обозначающее *состояние*. В объектно-ориентированном языке состояние связано с экземплярами классов, но ведь состояние имеется и в других языках. Так, в функциональном языке состояние, как правило, передается функции и принимается обновленным в результате ее выполнения. Понятия инвариантов оказываются столь же полезными и в этих обстоятельствах.

Проектирование по контракту и разработка на основе тестирования

Есть ли потребность в проектировании по контракту там, где разработчики практикуют модульное тестирование, разработку на основе тестирования (Test-Driven Development — TDD), тестирование на основе свойств или защитное программирование? Конечно, есть.

Проектирование по контракту и тестирование — это разные подходы к более обширной теме правильности программ. Оба подхода ценны и пригодны в разных ситуациях. Проектирование по контракту дает ряд преимуществ по сравнению с иными подходами к тестированию.

- Оно не требует никакой подготовки или имитации.
- Определяет параметры удачного или неудачного исхода во *всех* случаях, тогда как тестирование может быть одновременно нацелено лишь на один конкретный случай.
- Разработка на основе тестирования и другие разновидности тестирования происходят в цикле сборки лишь во время тестирования, а проектирование по контракту и утверждения — постоянно: во время проектирования, разработки, развертывания и сопровождения.
- Разработка на основе тестирования не сосредоточена на проверке внутренних инвариантов в тестируемом коде, а действует в большей степени по принципу "черного ящика" для проверки открытого интерфейса.
- Проектирование по контракту более эффективно и соблюдает принцип DRY точнее, чем защитное программирование, где *все* должны проверить данные на достоверность, если этого никто больше не делает.

Разработка на основе тестирования — отличная методика, но, как и многие другие методики разработки, она может привести к сосредоточению на удачном пути, а не на реальном мире, где полно некачественных данных, плохих исполнителей, неудачных версий и скверных спецификаций.

Реализация проектирования по контракту

Простое перечисление пределов области входных значений, граничных условий и того, что подпрограмма обещает (а еще важнее то, что она *не* обещает) предоставить, перед тем, как писать код, — это огромный шаг вперед в разработке более качественного программного обеспечения. Не сформулировав все эти условия, вы, по существу, возвращаетесь к *программированию по совпадению*, рассматриваемому в разделе "Тема 38. Программирование по совпадению" главы 7 "По ходу кодирования". Именно с этого начинаются, этим заканчиваются и терпят крах многие проекты.

В тех языках, где проектирование по контракту в исходном тексте не поддерживается, это может быть все, чего удастся достичь, но и это не так уж плохо. Ведь проектирование по контракту — это всего лишь методика *проектирования*. Даже не прибегая к автоматической проверке, можно внедрить контракт в исходный текст в виде комментариев или как модульные тесты и, тем не менее, извлечь из этого весьма реальную выгоду.

Утверждения

Несмотря на то что документирование подобных предположений служит хорошим началом, можно извлечь еще бо́льшую выгоду, заставив компилятор автоматически проверять контракт. Такую проверку можно частично сэмулировать в некоторых языках с помощью *утверждений* (assertions) — проверки логических условий во время выполнения (см. далее раздел "Тема 25. Утвердительное программирование"). А почему лишь частично? Нельзя ли с помощью утверждений добиться всего, на что только способно проектирование по контракту?

К сожалению, нельзя. Прежде всего, в объектно-ориентированных языках может не поддерживаться распространение утверждений по иерархии наследования. Это означает, что если переопределить в базовом классе метод, имеющий контракт, то утверждения, реализующие этот контракт, не будут корректно вызваны, если только не продублировать их вручную в новом коде. И нужно не забыть вызвать инвариант класса (и все инварианты базового класса) вручную перед выходом из каждого метода. Главная трудность состоит в том, что контракт не выполняется автоматически. А в других средах исключения, генерируемые из утверждений, составленных в стиле проектирования по контракту, могут быть отключены глобально или полностью проигнорированы исходным текстом.

Кроме того, не предусмотрено такое понятие, как "старые" значения, т.е. значения, которые существовали на момент входа в метод. Если вы пользуетесь утверждениями для проверки контрактов, то должны ввести код в предусловие, чтобы сохранить любую информацию, которая потребуется в постусловии, если язык вообще допускает такое. Так, в языке Eiffel, где, собственно, зародилось проектирование по контракту, для этой цели достаточно воспользоваться *выражением* `old`.

Наконец, обычные системы времени выполнения и библиотеки не предназначены для поддержки контрактов, поэтому подобные вызовы не проверяются. Это немалая потеря, поскольку на границе между прикладным кодом и применяемыми в нем библиотеками обнаруживается большинство затруднений (подробнее об этом — далее в разделе "Тема 24. Мертвые программы не лгут").

Проектирование по контракту и аварийное завершение

Проектирование по контракту изящно вписывается в понятие досрочного аварийного завершения (см. далее в разделе "Тема 24. Мертвые программы не лгут"). Используя механизм утверждений или проектирования по контракту для проверки достоверности предусловий, постусловий и инвариантов, можно добиться досрочного аварийного завершения программы и сообщить более точные сведения о возникшей неполадке.

Допустим, имеется метод, вычисляющий квадратные корни. Этому методу требуется предусловие проектирования по контракту, ограничивающее область его действия положительными числами. Если передать данному методу отрицательное значение параметра `sqrt` в тех языках, где поддерживается проектирования по контракту, то в конечном итоге будет получено сообщение об ошибке наподобие `sqrt_arg_must_be_positive` (аргумент для извлечения квадратного корня должен быть положительным) вместе с результатами трассировки стека.

Это, конечно, лучше, чем альтернатива, имеющаяся в других языках наподобие Java, C и C++, где в результате передачи методу отрицательного значения параметра `sqrt` возвращается специальное значение `NaN` (не число). И если некоторое время спустя попытаться выполнить в программе какие-нибудь математические операции над значением `NaN`, то в итоге будут получены совсем неожиданные результаты.

Намного проще найти и диагностировать неполадку, досрочно достигнув аварийного завершения в месте возникновения этой неполадки.

Семантические инварианты

Семантические инварианты могут быть использованы для выражения неизменных требований, нечто наподобие "философского контракта". Нам как-то пришлось разрабатывать программный коммутатор транзакций по дебетовым банковским карточкам. Главное требование состояло в том, чтобы пользователь дебетовой карточки не мог дважды выполнить одну и ту же транзакцию на своем счету. Иными словами, какого бы рода режим отказа ни возник, ошибка должна вызывать запрет обработки транзакции, но только не обработку дублированной транзакции. Это простое правило, вытекающее непосредственно из требований, оказывается очень полезным, когда приходится разбираться со сложными случаями устранения ошибок. И оно направляет на путь подробного проектирования и реализации во многих областях.

Кто несет ответственность

Кто же отвечает за проверку предусловия: вызывающий код или вызываемая подпрограмма? Если проектирования по контракту реализуется как часть языка, то за такую проверку не отвечает ни то ни другое. Ведь предусловие проверяется подспудно после того, как подпрограмма будет вызвана в вызывающем коде, но до входа в саму подпрограмму. Так, если требуется выполнить явную проверку параметров, это должно быть сделано в *вызывающем коде*, поскольку самой подпрограмме вообще недоступны параметры, нарушающие предусловие. (В тех языках, где отсутствует встроенная поддержка проектирования по контракту, чтобы проверить подобные утверждения, *вызываемая* подпрограмма должна иметь для этой цели преамбулу и/или заключительную часть.)

Рассмотрим в качестве примера программу, вводящую число с консоли, извлекающую из него квадратный корень с помощью функции `sqrt()` и выводящую результат. У функции `sqrt()` имеется следующее предусловие: ее аргумент не должен быть отрицательным. Если пользователь введет в консоли отрицательное число, то вызывающий код обязан принять меры, чтобы оно не было передано функции `sqrt()`. У этого вызывающего кода имеется много вариантов действий: прервать выполнение, выдать предупреждение и потребовать ввести другое число в консоли, сделать число положительным, или добавить к результату, возвращаемому функцией `sqrt()`, символ мнимой единицы (*i*). Но какой бы выбор ни был сделан, это совершенно не касается самой функции `sqrt()`.

Выражая область действия функции `sqrt()`, извлекающей квадратный корень, в ее предусловии, вы фактически переносите бремя ответственности за правильность на вызывающий код, где ему и место. И тогда вы можете благополучно проектировать функцию `sqrt()`, твердо зная, что ее входные данные окажутся в пределах, заданных контрактом.

Ни в коем случае не путайте требования, являющиеся фиксированными и неизменными правилами, с требованиями, представляющими собой лишь стратегии, которые могут измениться вместе с новым режимом управления. Именно поэтому мы пользуемся здесь термином *семантические инварианты*, поскольку ему принадлежит центральное место в определении самой *сути* предмета, и он не должен подчиняться прихотям стратегии, которой в большей степени отвечают более динамичные бизнес-правила.

Когда вы обнаруживаете требование, которое может уточняться, примите меры, чтобы оно стало хорошо известной частью любой составляемой документации, будь то маркированный список в документе требований, который под-

писывается в трех экземплярах, или же крупная заметка на общей белой доске, видимой всем. Попробуйте сформулировать это требование ясно и однозначно. Так, если вернуться к примеру с дебетовыми карточками, то такое требование можно было бы сформулировать следующим образом:

Ошибка — в пользу потребителя.

Это ясная, краткая, однозначная формулировка, применимая на самых разных участках системы. Это наш контракт со всеми пользователями системы и наша гарантия ее поведения.

Динамические контракты и агенты

До сих пор речь шла о контрактах как фиксированных, неизменяемых спецификациях. Но в области автономных агентов это совсем не обязательно должно быть именно так. По определению *автономные агенты* вольны отвергать те запросы, которые они не желают принимать во внимание. Они вольны согласовывать контракт заново, заявляя: "Я не могу это предоставить, но если вы дадите мне то и это, тогда я мог бы предоставить вам что-нибудь другое".

Безусловно, любая система, опирающаяся на технологию агентов, находится в *решающей* зависимости от договорных отношений, даже если они генерируются динамически.

Представьте: при достаточном количестве компонентов и агентов, способных согласовывать между собой контракты, чтобы достичь поставленной цели, мы могли бы просто дать программному обеспечению возможность разрешить кризис его производительности вместо нас.

Но если мы не можем пользоваться контрактами вручную, то не сможем воспользоваться ими и автоматически. Поэтому в следующий раз, когда будете проектировать какое-нибудь программное обеспечение, спроектируйте для него контракт.

Другие разделы, связанные с данной темой

- **Тема 24**. Мертвые программы не лгут.
- **Тема 25**. Утвердительное программирование.
- **Тема 38**. Программирование по совпадению, **глава 7** "По ходу кодирования".
- **Тема 42**. Тестирование на основе свойств, **глава 7** "По ходу кодирования".
- **Тема 43**. Будьте осторожны, **глава 7** "По ходу кодирования".
- **Тема 45**. Западня требований, **глава 8** "До начала проекта".

Задачи

- Рассмотрите следующие вопросы для размышления: если проектирование по контракту настолько эффективно, то почему оно не находит более широкого применения? Трудно ли сформулировать контракт? Заставляет ли это вас задуматься над теми вопросами, которые вы иначе проигнорировали бы? Заставляет ли это вас вообще ДУМАТЬ?! Очевидно, что это опасное инструментальное средство!

Упражнения

14. Разработайте интерфейс для кухонного блендера. В конечном итоге это должен быть ориентированный на веб и Интернет вещей блендер, но пока что требуется лишь интерфейс для управления им. В нем должны быть средства для установки десяти скоростей, причем 0 означает выключение блендера. Работать с блендером вхолостую нельзя, а его скорость можно изменять лишь по очереди и на единицу, т.е. от 0 до 1 и от 1 до 2, но не от 0 до 2.

    Ниже приведены соответствующие методы. Добавьте соответствующие пред- и постусловия и инвариант.

    ```
    int getSpeed()
    void setSpeed(int x)
    boolean isFull()
    void fill()
    void empty()
    ```

15. Сколько всего чисел в ряду 0, 5, 10, 15, ..., 100?

Тема 24 Мертвые программы не лгут

Приходилось ли вам замечать, что иногда другие люди могут раньше вас обнаружить, что с вами что-то не так? То же самое случается и с кодом. Так, если с одной из ваших программ случится что-то неладное, то неполадка иногда выявляется в сторонней библиотеке или каркасе. Причина может быть, например, в том, что в библиотечную функцию было передано пустое значение `nil` или же пустой список. Возможно, в хеше отсутствует ключ или значение, которое, как нам казалось, содержит хеш, на самом деле содержит список. А может быть, в сети или файловой системе произошла ошибка, которая не была перехвачена, в результате чего были получены пустые или испорченные данные. Логическая ошибка, совершенная пару миллионов операций назад, означает, что в селекторе оператора `case` больше не ожидается значение `1`, `2` или `3`, и поэтому неожиданно происходит переход к ветви `default`. И это еще одна причина, по которой в каждой инструкции `switch` непременно должна присутствовать ветвь `default` на тот случай, если произойдет нечто "невозможное".

Очень легко впасть в ошибку "этого не может произойти". Большинству из нас приходилось писать код, в котором не проверялось успешное закрытие файла и надлежащее написание оператора трассировки. И при прочих равных условиях нам это, вероятнее всего, было не нужно, поскольку мы считали, что наш код не потерпит крах при любых нормальных условиях. Но мы используем защитное программирование. В частности, проверяем данные на достоверность, рабочий код — на работоспособность, а версии загруженных зависимостей — на правильность.

Все ошибки дают какую-то информацию для размышления. Можно, конечно, убедить себя, что ошибка не случится и просто проигнорировать ее. Вместо этого программисты-прагматики говорят себе: если возникла ошибка, значит, произошло что-то очень и очень скверное. Поэтому не забывайте читать все эти отвратительные сообщения об ошибках (см. раздел "Программист в чужой стране" главы 3).

Принцип "поймал–отпустил" — только для ловли рыбы

Некоторые разработчики считают хорошим стилем программирования перехватывать все исключения или восстанавливать нормальную работу программы после их возникновения, а также повторно генерировать их после вывода какого-нибудь сообщения. В их коде можно найти немало таких мест, как в приведенном ниже фрагменте, где пустой оператор `raise` повторно генерирует текущее исключение.

```
try do
    add_score_to_board(score);
rescue InvalidScore
    Logger.error("Can't add invalid score. Exiting");
    raise
rescue BoardServerDown
    Logger.error("Can't add score: board is down. Exiting");
    raise
rescue StaleTransaction
    Logger.error("Can't add score: stale transaction. Exiting");
    raise
end
```

А вот что написал бы программист-прагматик:

```
add_score_to_board(score);
```

Мы предпочитаем именно такой стиль по двум причинам. Во-первых, прикладной код не затмевается обработкой ошибок. Во-вторых, что, вероятно, еще важнее, прикладной код оказывается менее связанным. В приведенном выше многословном примере приходится перечислять каждое исключение, которое может быть сгенерировано в методе `add_score_to_board()`. Если автор этого метода введет еще одно исключение, то его код едва заметно устареет. А в

более прагматичном втором примере новое исключение распространяется автоматически.

Совет 38 Пользуйтесь досрочным аварийным завершением программы

Аварийное завершение вместо отправки на свалку

Одно из преимуществ как можно более раннего выявления неполадок состоит в возможности досрочного аварийного завершения. И зачастую это самое лучшее, что можно делать. В качестве альтернативы можно продолжить, записав испорченные данные в некоторую актуальную базу данных или дав стиральной машине команду на двадцатом по счету цикле вращения ее барабана с бельем.

Именно такой принцип поддерживается в языках Erlang и Elixir. Джо Армстронг, изобретатель языка Erlang и автор книги *Programming Erlang: Software for a Concurrent World* [Arm07], часто говорил следующее: "Защитное программирование — это напрасная трата времени. Допускайте аварийное завершение!" В таких средах предусматривается, что программы могут отказывать, но отказ находится под управлением *супервизора*, отвечающего за выполнение кода и знающего, что именно следует делать, если код откажет. К подобным мерам относится очистка после сбоя, перезапуск кода и т.д. А что, если откажет сам супервизор? Этим событием управляет его собственный супервизор, что приводит к архитектуре, состоящей из *деревьев супервизоров*. Такая методика довольно эффективна и способствует применению упомянутых выше языков в высоконадежных, отказоустойчивых системах.

В других средах простой выход из работающей программы может оказаться неприемлемым. Ведь в ней могли быть затребованы ресурсы, которые могут быть и не освобождены. А возможно, придется написать протокольные сообщения, очистить открытые транзакции или организовать взаимодействие с другими процессами.

Тем не менее основной принцип остается прежним: если в прикладном коде обнаруживается, что произошло нечто, предполагавшееся невозможным, такой код больше не считается жизнеспособным. И все, что в таком коде делается дальше, становится подозрительным, а следовательно, его выполнение следует прервать как можно скорее. Мертвая программа, как правило, наносит намного меньше вреда, чем испорченная.

Другие разделы, связанные с данной темой

- **Тема 20**. Отладка, **глава 3** "Основные инструментальные средства".
- **Тема 23**. Проектирование по контракту.

- **Тема 25**. Утвердительное программирование.
- **Тема 26**. Как сбалансировать ресурсы.
- **Тема 43**. Будьте осторожны, **глава 7** "По ходу кодирования".

Тема 25 Утвердительное программирование

"В самобичевании есть свое наслаждение.
И когда мы виним себя сами, то чувствуем,
что никто другой не вправе винить нас".

Оскар Уайльд, ***Портрет Дориана Грея***

По-видимому, есть такая мантра, которую каждый программист должен запомнить, едва начав свою карьеру. Это основополагающий постулат, убеждение, которое мы учимся применять к требованиям, проектам, исходному коду, комментариям и практически ко всему, что мы делаем. И он гласит:

Это никогда не может случиться...

Примерами тому служат следующие рассуждения: "Это приложение никогда не будет применяться за рубежом, так зачем его интернационализировать?", "Переменная `count` не может принимать отрицательное значение" или "Протоколирование не может дать сбой". Старайтесь избегать в своей практике подобного самообольщения, особенно во время программирования.

Совет 39 Пользуйтесь утверждениями, чтобы предотвратить невозможное

Всякий раз, когда вы ловите себя на мысли "но этого, конечно, никогда не сможет случиться", вводите код для ее проверки. И сделать это проще всего с помощью утверждений. В реализациях многих языков можно обнаружить некоторую форму утверждений в виде оператора `assert`, где проверяется логическое условие[3]. Такие проверки могут оказаться неоценимыми. Так, если параметр или результат вообще не должен быть пустым (`null`), его следует проверить явным образом, как показано ниже.

```
assert (result != null);
```

[3] В языках C и C++ утверждения обычно реализуются в виде макрокоманд, а в языке Java они запрещены по умолчанию. Чтобы разрешить их, следует вызвать виртуальную машину Java с параметром `-enableassertions` из командной строки, и оставить их разрешенными.

В реализации Java можно (и нужно) ввести в утверждение описательную строку следующим образом:

```
assert result != null && result.size() > 0 : "Empty result from XYZ";
```

Утверждения полезны также для проверки функционирования алгоритмов. Так, если написать алгоритм изощренной сортировки под названием `my_sort`, проверить его работоспособность можно, например, таким образом:

```
books = my_sort(find("scifi"))
assert(is_sorted?(books))
```

Ни в коем случае не пользуйтесь утверждениями вместо реальной обработки ошибок. Утверждения позволяют проверить то, что никогда не должно случиться. Вряд ли вам захочется написать код, аналогичный приведенному ниже.

```
puts("Enter 'Y' or 'N': ")
ans = gets[0]                          # взять первый символ из ответа
assert((ch == 'Y') || (ch == 'N'))     # Явно неудачная идея!
```

То, что большинство реализаций оператора `assert` просто завершают процесс, если утверждение не выполняется, не является причиной для того, чтобы это происходило и в написанных вами версиях программного обеспечения. Так, если вам требуется освободить используемые ресурсы, перехватите исключение, возникшее в утверждении, или прервите выход из программы, а затем выполните свой обработчик ошибок. Нужно лишь сделать так, чтобы код, выполняемый в течение этих истекающих миллисекунд, прежде всего не опирался на информацию, послужившую причиной того, что утверждение не подтвердилось.

Утверждения и побочные эффекты

Если код, вводимый для обнаружения ошибок, фактически приводит к новым ошибкам, то положение только усугубляется. Такое может случиться и с утверждениями, если проверка условия имеет побочные эффекты. Например, написание следующего фрагмента кода вряд можно считать удачной идеей:

```
while (iter.hasMoreElements()) {
    assert(iter.nextElement() != null);
    Object obj = iter.nextElement();
    // ....
}
```

Вызов `.nextElement()` в утверждении имеет побочный эффект, проявляющийся в том, что итератор пропускает извлекаемый элемент, и поэтому в цикле `while` будет обработана лишь половина элементов, находящихся в коллекции. В таком случае было бы лучше написать данный цикл следующим образом:

```
while (iter.hasMoreElements()) {
  Object obj = iter.nextElement();
  assert(obj != null);
  // ....
}
```

Это разновидность неопределенной, едва уловимой программной ошибки под названием *гейзенбаг*[4], которая изменяет поведение отлаживаемой системы. Мы также считаем, что теперь, когда в большинстве языков имеется приличная поддержка функций, циклически перебирающих коллекции, такого рода явный цикл не нужен и является неудачной разновидностью их обработки.

Оставляйте включенным режим утверждений

В отношении утверждений существует весьма распространенное заблуждение:

> *Утверждения вносят некоторые издержки в код. Они проверяют то, что никогда не должно случиться, и поэтому приводятся в действие только программной ошибкой в коде. И как только код будет проверен и доставлен потребителю, они больше не потребуются, а следовательно, режим утверждений можно отключить, чтобы код выполнялся быстрее. Утверждения являются средством отладки исходного кода.*

В приведенном выше рассуждении есть два заведомо ложных допущения. Во-первых, в нем предполагается, что во время тестирования обнаруживаются все программные ошибки. В действительности в любой сложной программе вряд ли удастся проверить даже мельчайшую долю всех перестановок, сделанных в исходном коде. И во-вторых, оптимисты забывают, что их программы выполняются в опасной среде. Во время тестирования крысы вряд ли перегрызут кабель связи, игрок исчерпает оперативную память по ходу компьютерной игры, а журнальные файлы заполнят весь раздел запоминающего устройства. Подобные события скорее могут произойти при выполнении программы в реальной эксплуатационной среде. Поэтому вашей первой линией обороны должна стать проверка любых возможных ошибок, а второй линией — применение утверждений, чтобы попытаться обнаружить пропущенные ошибки.

Выключение режима утверждений при запуске программы в эксплуатацию можно сравнить с хождением по натянутой под куполом цирка проволоке без страховочной сетки только потому, что это уже однажды удалось сделать. В этом есть драматичность и зрелищность, но не страховка от смертельного исхода.

Даже если вам приходится решать вопросы производительности, выключайте лишь те утверждения, которые действительно отрицательно влияют на производительность. Так, приведенный выше пример сортировки может быть крайне важной частью приложения, которая должна выполняться побыстрее. Добавление в нее проверки означает еще один проход данных, что может оказаться неприемлемым. Поэтому сделайте эту конкретную проверку необязательной, но оставьте остальные проверки без изменений.

---

[4] Название гейзенбаг (Heisenbug) происходит от принципа неопределенности Гейзенберга из квантовой механики и, собственно, программной ошибки (bug). Подробнее об этом см. по адресу `http://www.eps.mcgill.ca/jargon/jargon.html#heisenbug` или `https://ru.wikipedia.org/wiki/Гейзенбаг`.

Применение утверждений в условиях реальной эксплуатации приносит немалую прибыль

Бывший сосед Энди руководил небольшой начинающей компанией, производившей сетевые устройства. Одним из секретов их успеха было решение оставить утверждения на месте в рабочих версиях. Эти утверждения были достаточно грамотно составлены, чтобы сообщать всю информацию, имеющую отношение к сбоям, а также представлены конечному пользователю через изящного вида интерфейс. Такой уровень организации откликов реальных пользователей в конкретных условиях эксплуатации позволял разработчикам затыкать дыры в защите и устранять едва заметные, трудно воспроизводимые программные ошибки, а следовательно, производить необыкновенно устойчивое, надежное программное обеспечение. Эта небольшая, малоизвестная компания производила настолько добротную продукцию, что вскоре была приобретена за сотни миллионов долларов. Но это так, к слову.

Другие разделы, связанные с данной темой

- **Тема 23**. Проектирование по контракту.
- **Тема 24**. Мертвые программы не лгут.
- **Тема 42**. Тестирование на основе свойств, **глава 7** "По ходу кодирования".
- **Тема 43**. Будьте осторожны, **глава 7** "По ходу кодирования".

Упражнения

16. Быстрая проверка на чувство реальности. Какие из перечисленных ниже "невозможных" событий могут все же произойти?

- Месяц, в котором меньше 28 дней.
- Код ошибки, возвращаемый из системного вызова и обозначающий невозможность доступа к текущему каталогу.
- В языке C++: `a = 2, b = 3`, но `(a + b)` не равно `5`.
- Треугольник, сумма углов которого не равна 180°.
- Минута, не насчитывающая 60 секунд.
- `(a + 1) <= a`

Тема 26 Как сбалансировать ресурсы

"Зажечь свечу — отбросить тень..."

Урсула К. Ле Гуин[5], ***Волшебник Земноморья*** *(A Wizard of Earthsea)*

Когда мы программируем, то так или иначе манипулируем ресурсами: оперативной памятью, транзакциями, потоками выполнения, сетевыми соединениями, файлами, таймерами, т.е. всеми средствами с ограниченной доступностью. И чаще всего ресурсы используются по вполне предсказуемому шаблону: сначала ресурс выделяется, затем используется и, наконец, освобождается.

Тем не менее у многих разработчиков нет постоянного плана по выделению и освобождению ресурсов. Поэтому дадим в этой связи следующий совет:

Совет 40 Завершайте то, что начали

Этот совет легко применить в большинстве случаев. Он просто означает, что функция или объект, выделяющие ресурс, должны отвечать за его освобождение. Рассмотрим его применение на конкретном примере неудачно написанного на Ruby фрагмента кода, открывающего файл, читающего из него сведения о клиенте, обновляющего отдельное поле данных и записывающего результат обратно в файл. Ради простоты и большей ясности данного примера из него исключена проверка ошибок, как показано ниже.

```
def read_customer
    @customer_file = File.open(@name + ".rec", "r+")
    @balance = BigDecimal(@customer_file.gets)
end

def write_customer
    @customer_file.rewind
    @customer_file.puts @balance.to_s
    @customer_file.close
end

def update_customer(transaction_amount)
    read_customer
    @balance = @balance.add(transaction_amount,2)
    write_customer
end
```

На первый взгляд, подпрограмма `update_customer()` из приведенного выше фрагмента кода выглядит вполне обоснованно. По-видимому, для реализации ее логики потребуется прочитать запись, обновить баланс на текущем счете и записать обновленную запись обратно. Тем не менее за такой опрят-

[5] Ursula K. Le Guin — современная американская писательница и литературный критик, автор романов и повестей в жанре фэнтези. — *Примеч. пер.*

ностью скрывается главная трудность. Подпрограммы `read_customer()` и `write_customer()` тесно связаны[6], поскольку совместно пользуются общей переменной экземпляра `customer_file`. Сначала подпрограмма `write_customer()` открывает файл и сохраняет ссылку на него в переменной `customer_file`, а затем подпрограмма `read_customer()` использует сохраненную ссылку, чтобы по ее завершении закрыть файл. Но эта общая переменная даже не упоминается в подпрограмме `update_customer()`.

Почему это плохо? Рассмотрим пример незадачливого программиста, получившего задание внести следующее изменение в спецификацию сопровождаемого им приложения: баланс на счете должен обновляться лишь в том случае, если новое значение не является отрицательным. С этой целью он обращается к исходному коду и вносит следующие изменения в подпрограмму `update_customer()`:

```
def update_customer(transaction_amount)
    read_customer
    if (transaction_amount >= 0.00)
        @balance = @balance.add(transaction_amount,2)
        write_customer
    end
end
```

Во время тестирования этот код вроде бы ведет себя как следует. Но когда этот код запускается в эксплуатацию, он терпит крах несколько часов спустя, завершаясь предупреждением о том, что открыто *слишком много* файлов. И оказывается, что при определенных обстоятельствах подпрограмма `write_customer()` не вызывается. И когда этого происходит, файл не закрывается.

Весьма *неудачным* решением данной задачи была бы попытка реализовать особый случай в подпрограмме `update_customer()`:

```
def update_customer(transaction_amount)
    read_customer
    if (transaction_amount >= 0.00)
        @balance += BigDecimal(transaction_amount, 2)
        write_customer
    else
        @customer_file.close # Плохое решение!
    end
end
```

И хотя такое решение устраняет затруднение, поскольку файл теперь будет закрыт независимо от нового баланса на счете, тем не менее, теперь данное исправление означает, что *все* три подпрограммы связаны общей переменной `customer_file`, а слежение за тем, открыт ли файл или закрыт, начинает при-

[6] Подробнее об опасностях, кроющихся в связанном коде, см. в разделе "Тема 28. Развязка" главы 5.

ходить в полный беспорядок. В итоге наш программист попадает в ловушку, и все дело начнет быстро катиться под откос, если продолжить его в том же духе. Это и есть признак несбалансированности!

Совет "Завершайте то, что начали" предписывает, что в идеальном случае подпрограмма, выделяющая ресурс, должна его освободить. Чтобы воспользоваться этим советом, реорганизуем немного рассматриваемый здесь код, как показано ниже.

```
def read_customer(file)
    @balance=BigDecimal(file.gets)
end
def write_customer(file)
    file.rewind
    file.puts @balance.to_s
end
def update_customer(transaction_amount)
    file=File.open(@name + ".rec", "r+")              # >--
    read_customer(file)                               #    |
    @balance = @balance.add(transaction_amount,2)     #    |
    file.close                                        # <--
end
```

Вместо того чтобы сохранять ссылку на файл, мы изменили в данном случае исходный код таким образом, чтобы передавать ее в качестве параметра[7]. И теперь вся ответственность за манипулирование файлом возлагается на подпрограмму `update_customer()`, которая открывает файл и (по завершении того, с чего она начинается) закрывает его перед самым возвратом. Таким образом, в данной подпрограмме балансируется пользование файлом как внешним ресурсом, поскольку операции его открытия и закрытия происходят в одном и том же месте, и вполне очевидно, что на каждую операцию открытия файла приходится соответствующая операция его закрытия. Кроме того, благодаря реорганизации кода выводится из употребления скверная общая переменная.

В рассматриваемый здесь код можно внести еще одно важное усовершенствование. Во многих современных языках срок действия ресурса можно заключить в пределы некоторого замкнутого блока. Так, в языке Ruby имеется разновидность оператора `open`, передающего ссылку на открытый файл такому блоку, как показано ниже в промежутке между операторами `do` и `end`.

```
def update_customer(transaction_amount)
    File.open(@name + ".rec", "r+") do |file|         # >--
        read_customer(file)                           #    |
        @balance = @balance.add(transaction_amount,2) #    |
        write_customer(file)                          #    |
    end                                               # <--
end
```

[7] См. совет 50 в главе 5.

В данном случае переменная `file` выходит в конце блока из области своего действия, и на этом внешний файл закрывается. А программист избавляется от необходимости помнить, что следует непременно закрыть файл и тем самым освободить ресурс, поскольку это гарантированно будет сделано автоматически. Если же гложут сомнения, то всегда стоит сократить область действия.

Совет 41 Действуйте локально

Вложенное выделение ресурсов

Основной шаблон выделения ресурсов может быть расширен для тех подпрограмм, которым одновременно требуется не один ресурс. И для этого имеются лишь следующие два предложения.

- Освободить ресурсы в порядке, противоположном тому, в каком они были выделены. Подобным образом ресурсы не останутся зависшими, если один из них содержит ссылки на другие.
- Если один и тот же ряд ресурсов выделяется в разных местах исходного кода, они должны выделяться в том же самом порядке. Благодаря этому снижается вероятность взаимной блокировки. Так, если процесс A затребует ресурс 1 и готов востребовать ресурс 2, тогда как процесс B уже затребовал ресурс 2 и пытается заполучить ресурс 1, то оба процесса перейдут в состояние бесконечного ожидания.

Независимо от того, какого рода ресурсы используются, будь то транзакции, сетевые соединения, оперативная память, файлы, потоки выполнения или окна, применяется следующий основной шаблон: тот, кто выделяет ресурс, должен отвечать за его освобождение. Тем не менее этот принцип может быть развит далее в некоторых языках.

Объекты и исключения

Баланс между выделением и освобождением ресурсов напоминает конструктор и деструктор класса в объектно-ориентированном программировании. В частности, класс представляет собой ресурс, конструктор представляет собой выделение конкретного объекта данного ресурса, а деструктор удаляет его из области видимости.

Если вы программируете на объектно-ориентированном языке, то можете найти удобной инкапсуляцию ресурсов в классах. Всякий раз, когда вам потребуется ресурс конкретного типа, вы получаете экземпляр объекта данного класса. А когда объект выходит из области своего действия или утилизируется сборщиком "мусора", деструктор этого объекта освобождает заключенный в нем ресурс. Такой подход дает особые преимущества при программировании на тех языках, где исключения могут мешать освобождению ресурсов.

Баланс во времени

В данной теме рассматриваются в основном эфемерные ресурсы, используемые для выполнения определенного процесса. Но вам, возможно, придется рассмотреть и другие сложные вопросы, которые могли быть оставлены без внимания.

Как, например, обрабатывать журнальные файлы? Так, если вы, формируя данные, исчерпаете все свободное место на запоминающем устройстве, то следует ли вместо этого организовать ротацию и очистку журнальных файлов? А как насчет удаления неофициальных файлов отладки? Если вы вводите протокольные записи в базу данных, то есть ли аналогичный процесс, определяющий окончание срока их действия? Рассмотрите возможность сбалансировать все, что ни создается и требует конечного ресурса.

И, наконец, не осталось ли еще что-нибудь вне вашего внимания?

Баланс исключений

В языках, поддерживающих исключения, освобождение ресурсов может быть затруднено. Так, если генерируется исключение, то как гарантировать очистку всего, что было выделено до этого исключения? Ответ на этот вопрос зависит от поддержки этой возможности языком. Как правило, для этого имеются две возможности.

1. Воспользоваться областью видимости переменной (например, стековыми переменными в C++ или Rust).
2. Воспользоваться оператором `finally` в блоке операторов `try/catch`.

По обычным правилам соблюдения области видимости в таких языках, как C++ или Rust, память, выделяемая для переменной, будет утилизирована, как только переменная выйдет из области своей видимости посредством возврата из функции, завершения блока кода или исключения. Но для очистки любых внешних ресурсов можно также осуществить привязку к деструктору переменной. В приведенном ниже примере кода на языке Rust файл автоматически закроется, как только переменная `accounts` выйдет из области своей видимости.

```
{
    let mut accounts = File::open("mydata.txt")?;     // >--
    // Пользуемся переменной 'accounts'               //    |
    ..                                                //    |
}                                                     // <--
// Теперь переменная 'accounts' находится вне области
// своей видимости, а файл автоматически закрывается
```

Другая возможность, если только она поддерживается в конкретном языке, состоит в применении оператора `finally`, как показано ниже. Этот оператор гарантирует, что указанный код будет выполнен независимо от того, будет ли сгенерировано исключение в блоке операторов `try/catch`.

```
try
    // нечто не внушающее доверия
catch
    // сгенерировано исключение
finally
    // очистить в любом случае
```

Но здесь есть одна хитрость.

Антишаблон исключений

Нам часто приходилось наблюдать, как программисты пишут следующий код:

```
begin
    thing = allocate_resource()
    process(thing)
finally
    deallocate(thing)
end
```

Можете ли вы обнаружить ошибку в этом коде? Что, если выделить ресурс не удастся и возникнет исключение? Оно будет перехвачено в операторе `finally`, который попытается освободить ресурс `thing`, который не был выделен.

Правильный шаблон для освобождения ресурсов в среде с исключениями выглядит следующим образом:

```
thing = allocate_resource()
begin
  process(thing)
finally
  deallocate(thing)
end
```

Когда нельзя сбалансировать ресурсы

Иногда основной шаблон для выделения ресурсов оказывается просто непригодным. И обычно это бывает в тех программах, в которых применяются динамические структуры данных. Так, одна подпрограмма может выделить область памяти и связать ее с какой-нибудь более крупной структурой, где она может оставаться некоторое время.

Выход из этого положения состоит в том, чтобы установить семантический инвариант для выделения памяти. При этом необходимо решить, кто должен отвечать за данные в агрегированной структуре данных. Что произойдет, если

освободить структуру верхнего уровня? Для этого имеются три основные возможности.

- Структура верхнего уровня отвечает за освобождение любых подструктур, которые она содержит. Из этих структур затем удаляются находящиеся в них данные и т.д.
- Структура верхнего уровня просто освобождается. А любые структуры, на которые она указывает и которые нигде больше не делаются ссылки, "зависают".
- Структура верхнего уровня отказывается освобождаться, если она содержит любые подструктуры.

Выбор конкретной возможности зависит от обстоятельств каждой структуры данных в отдельности. Тем не менее ее необходимо выразить явно для каждой структуры данных и согласованно реализовать свое решение. Реализация любой из этих возможностей в таких процедурных языках, как C, может оказаться затруднительной, поскольку сами структуры данных неактивны. В таких случаях мы предпочитаем написать для каждой из основных структур данных модуль, обеспечивающий стандартное выделение и освобождение ресурсов, требующихся данной структуре. (Такой модуль может также предоставить такие ресурсы, как вывод на печать при отладке, сериализация и десериализация, а также перехватчики обхода.)

Проверка баланса

Программисты-прагматики никому не доверяют, в том числе и самим себе, и поэтому мы считаем, что всегда стоит писать такой код, в котором проверяется, действительно ли ресурсы освобождены должным образом. В большинстве приложений это, как правило, означает создание оболочек для каждого типа ресурса, а также их применение для отслеживания всех операций выделения и освобождения ресурсов. В некоторых местах кода логика программы будет диктовать нахождение ресурсов в определенном состоянии, поэтому для его проверки следует воспользоваться оболочками. Например, в верхней части главного цикла обработки данных в долго выполняемой программе, обслуживающей запросы, вероятно, имеется единственная точка, где ожидается поступление очередного запроса. Это удобное место, где можно гарантировать, что использование ресурса не возросло с момента последнего выполнения данного цикла. На более низком, хотя и менее полезном уровне можно вложить средства в инструментальные средства, которые, среди прочего, проверяют выполняющиеся программы на наличие утечки памяти.

Другие разделы, связанные с данной темой

- **Тема 24**. Мертвые программы не лгут.
- **Тема 30**. Преобразовательное программирование, **глава 5** "Гибкость или ломкость".
- **Тема 33**. Разрывание временного связывания, **глава 6** "Параллельность".

Задачи

- В отсутствие способов, всегда гарантирующих освобождение ресурсов, может помочь последовательное применение некоторых методик. Ранее в этом разделе пояснялось, каким образом установка семантического инварианта для основных структур данных может привести к правильным решениям по поводу освобождения оперативной памяти. Выясните, как это делается. Для этого вам, возможно, придется вернуться к материалу раздела "**Тема 23**. Проектирование по контракту".

Упражнения

17. Некоторые программирующие на C и C++, как правило, устанавливают значение указателя равным `NULL` после освобождения области памяти, на которую он ссылается. Чем хороша эта идея?
18. Некоторые программирующие на Java устанавливают значение `NULL` для объектной переменной по окончании использования объекта. Чем хороша эта идея?

Тема 27 Не опережайте свет фар вашего автомобиля

"Трудно делать предсказания, особенно о будущем".

Лоуренс "Йоги" Берра[8], из датской поговорки

Представьте темную дождливую ночь. Двухместный автомобиль резко поворачивает на крутых поворотах узкой извилистой горной дороги, едва вписываясь в них. И вдруг под колесо автомобиля попадает шпилька для волос, в результате чего он ударяется в слабое дорожное ограждение, стремительно падает вниз и разбивается вдребезги о землю в лежащей ниже долине. Сотрудники местной полиции прибывают на место происшествия, осматривают его, и старший офицер, печально кивая головой, говорит: "Видно, опередили свет фар своего автомобиля".

[8] Lawrence "Yogi" Berra — известный американский бейсболист. — *Примеч. пер.*

Неужели двухместный автомобиль мог ехать быстрее скорости света? Нет, конечно, поскольку превысить этот предел скорости не позволяют законы физики. Офицер имел в виду способность водителя вести машину и вовремя остановиться, реагируя на дорожную обстановку, освещаемую светом фар.

Свет фар распространяется в ограниченных пределах, называемых *проекционным расстоянием*, а дальше свет фаз становится слишком рассеянным, чтобы эффективно освещать дорогу. Кроме того, фары проецируют свет по прямой осевой линии, ничего не освещая за ее пределами, в том числе повороты, подъемы и впадины на дороге. Согласно данным Национального управления безопасностью движения на трассах (National Highway Traffic Safety Administration — NHTSA) среднее расстояние, освещаемое фарами ближнего света, составляет около 50 м. Но, к сожалению, безопасный тормозной путь на скорости 65 км/час составляет 57 м, тогда как на скорости 110 км/час — уже около 140 м.[9] Следовательно, опередить свет фар своего автомобиля очень легко.

В разработке программного обеспечения дальность света наших "передних фар" столь же ограничена. Мы не можем заглянуть слишком далеко в будущее, и чем дальше мы отклоняем свой взгляд от основной оси, тем темнее становится. Поэтому программист-прагматик должен придерживаться следующего твердого правила:

Совет 42 Всегда предпринимайте небольшие шаги

Старайтесь всегда предпринимать небольшие, обдуманные шаги, проверяя ответную реакцию и корректируя свои действия, прежде чем продолжить дальше. И ни в коем случае не предпринимайте слишком большой шаг и не беритесь за слишком крупную задачу.

А что имеется конкретно в виду под ответной реакцией? Все, что независимо подтверждает или опровергает ваше действие. Например:

- результаты в цикле "чтение–вычисление–вывод" (REPL) обеспечивают ответную реакцию на ваше понимание API и алгоритмов;
- модульные тесты обеспечивают ответную реакцию на последнее изменение кода;
- демонстрация пользователям и ее обсуждение обеспечивают ответную реакцию на функциональные средства, удобство и простоту использования.

А какую задачу следует считать слишком крупной? Любую задачу, которая требует "предсказания будущего". Подобно ограниченной дальности света пере-

[9] В соответствии со следующей формулой NHTSA: безопасный тормозной путь = расстояние, проходимое автомобилем за время реакции водителя + путь торможения, при условии что среднее время реакции составляет 1,5 с, а ускорение — 5,19 м/с^2.

дних фар автомобиля, мы можем также заглянуть в будущее, возможно, на один или два шага, а, может быть, самое большее — на несколько часов или дней. Выйдя за эти пределы, можно очень быстро перейти от *обоснованного предположения* к *досужим домыслам*. Легко впасть в грех предсказания будущего, когда приходится:

- оценивать даты завершения работ на месяцы вперед;
- планировать расширяемость сопровождения в перспективе;
- предугадывать будущие потребности пользователей;
- прогнозировать техническую пригодность в будущем.

Но мы уже слышим ваши громкие возражения: "Разве мы не проектируем с учетом будущего сопровождения?" Да, конечно, но лишь до определенного момента, т.е. настолько, насколько мы в состоянии смотреть вперед. Чем больше вам приходится предсказывать будущее, тем больше риск, что вы ошибетесь. И вместо того чтобы тратить усилия на проектирование для неопределенного будущего, можно всегда прибегнуть к разработке заменяемого кода. Старайтесь облегчить отказ от непригодного кода и его замену более пригодным кодом. Возможность заменить код помогает также его сцеплению, связыванию и соблюдению принципа DRY, что приводит к более качественному проектированию. И хотя вы можете быть уверены в будущем, всегда существует вероятность, что не за горами окажется нечто совсем негаданное, словно черный лебедь.

Черные лебеди

В своей книге *The Black Swan: The Impact of the Highly Improbable* [Tal10] Нассим Наколас Талеб утверждает, что все значительные события в истории произошли от широко известных, трудно предсказуемых и редких событий, выходящих за пределы обычных ожиданий. Эти невероятные события имеют несоразмерный эффект, несмотря на свою статистическую редкость. Кроме того, наши когнитивные отклонения приводят к тому, что мы не замечаем, как перемены незаметно проникают в нашу работу, доходя до краев (см. раздел "Тема 4. Суп из камней и вареные лягушки" главы 1 "Философия прагматизма").

Ко времени выхода в свет первого издания этой книги в компьютерных журналах и оперативно доступных форумах разгорелась полемика по следующему вопросу: "Кто одержит верх в войнах настольных GUI между стилями Motif и OpenLook?"[10] Этот вопрос был поставлен неверно. Вполне вероятно, что вы вообще не слышали об этих технологиях, поскольку ни одна из них не одержала верх, и в данной области быстро возобладала веб-технология, ориентированная на браузеры.

[10] Motif и OpenLook были стандартными графическими интерфейсами для системы X-Window на рабочих станциях Unix.

Совет 43 Избегайте предсказания будущего

Завтра зачастую выглядит очень похожим на сегодня. Но на это не следует особенно полагаться.

Другие разделы, связанные с данной темой

- **Тема 12**. Трассирующие пули, **глава 2** "Прагматичный подход".
- **Тема 13**. Прототипы и памятные записки, **глава 2** "Прагматичный подход".
- **Тема 40**. Рефакторинг, **глава 7** "По ходу кодирования".
- **Тема 41**. Тестировать, чтобы кодировать, **глава 7** "По ходу кодирования".
- **Тема 48**. Сущность гибкости, **глава 8** "До начала проекта".
- **Тема 50**. Кокосами не обойтись, **глава 9** "Прагматичные проекты".

Глава 5

Гибкость или ломкость

Жизнь не стоит на месте, как, впрочем, и код, который мы пишем. Чтобы поспевать за бешеным темпом нынешних перемен, нам приходится каждый раз прилагать усилия для написания кода, который должен быть как можно более слабо связанным, т.е. гибким. В противном случае наш код может быстро устареть или оказаться слишком хрупким для исправления и в конечном счете может быть оставлен без внимания в безумном стремлении в будущее.

В разделе "Тема 11. Обратимость" главы 2 "Прагматичный подход" упоминалось о рисках необратимых решений. В этой главе речь пойдет о том, как принимать *обратимые* решения, чтобы сохранить свой код гибким и приспосабливаемым к неопределенному окружающему миру.

Сначала в этой главе будет рассмотрено *связывание*, т.е. создание зависимостей между фрагментами кода, а затем *развязывание*, позволяющий сохранить отдельные части кода разделенными, тем самым убавив связывание.

Далее в этой главе будут рассмотрены методики, которые можно применять при *манипулировании реальным миром*. В соответствующем разделе будут исследованы различные стратегии, помогающие управлять событиями и реагировать на них, что является очень важной особенностью современной разработки приложений.

Традиционный процедурный и объектно-ориентированный код может быть тесно связан в определенных целях. Поэтому в разделе "Преобразовательное программирование" поясняется, как воспользоваться с выгодой более гибким и ясным стилем программирования, предлагаемым конвейерами функций, даже если они не поддерживаются непосредственно в конкретном языке.

Обычный стиль объектно-ориентированного программирования может завести в еще одну ловушку. Не попадайтесь в нее, иначе вам придется заплатить непомерный *налог на наследование*. В соответствующем разделе будут рассмотрены более подходящие альтернативы, позволяющие сохранить гибкость кода и упростить его изменение.

И, разумеется, чтобы сохранить гибкость кода, лучше писать его *меньше*. Изменение кода оставляет возможность вносить новые программные ошибки. В разделе "Конфигурирование" поясняется, как вынести все подробности

из кода туда, где их можно более безопасно и легко изменить. Все рассматриваемые в этой главе методики помогают писать код, который сгибается, но не ломается.

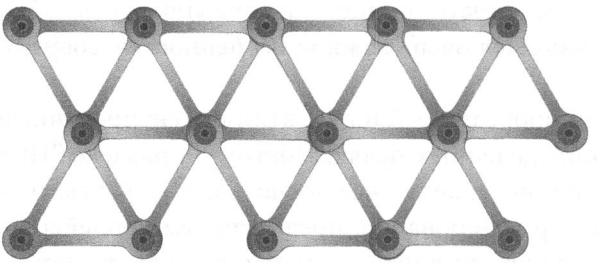
Тема 28 Развязывание

"Когда мы пытаемся выбрать что-нибудь само по себе, то обнаруживаем, что оно зацепляется за все остальное в мире".

Джон Мьюр[1], ***Мое первое лето в Сьерре*** *(My First Summer in the Sierra)*

В разделе "Тема 8. Сущность качественного проектирования" главы 2 "Прагматичный подход" утверждалось, что, применяя подходящие принципы проектирования, можно упростить внесение изменений в исходный код. Связывание враждебно изменению, поскольку связывает вместе компоненты, которые должны изменяться параллельно. И это затрудняет внесение изменений, поскольку приходится тратить время на отслеживание всех частей, которые должны быть изменены, или же на выяснение причин, по которым все сломалось, когда изменился всего лишь один компонент, а не другие компоненты, с которыми он был связан.

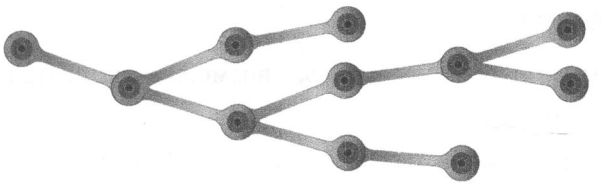
Когда вы разрабатываете что-нибудь такое, что должно быть таким же прочным, как мост или башня, тогда вам, возможно, придется связать вместе два компонента, как показано ниже.



Совместно связи делают структуру прочной. Сравните ее с чем-то наподобие приведенной ниже структуры, где структурная прочность напрочь отсутствует, поскольку одни связи могут измениться, а другие просто приспособиться к этому изменению.

---

[1] John Muir — американский естествоиспытатель, писатель XIX века, и защитник дикой природы, один из инициаторов создания в США национальных парков и заповедных территорий. — *Примеч. пер.*



При проектировании мостов требуется сохранить их форму, помимо того, что их конструкции должны быть жесткими и прочными. При проектировании программного обеспечения, которое предполагает внесение изменений, требуется совершенно противоположное: оно должно быть гибким. А для этого одни компоненты должны быть связаны с как можно меньшим числом других компонентов.

И хуже того, связывание транзитивно. Так, если компонент A связан с компонентами B и C, компонент B — с компонентами M и N, компонент C — компонентами X и Y, то компонент A оказывается связанным с компонентами B, C, M, N, X и Y. Это означает, что необходимо следовать приведенному ниже простому принципу.

Совет 44 Развязанный код проще изменить

Если принять во внимание, что для конструирования кода, как правило, не применяются стальные балки и заклепки, то что же означает развязывание кода? В этом разделе будут рассмотрены следующие вопросы.

- Крушения поездов — цепочки вызовов методов.
- Глобализация — опасности статических элементов.
- Наследование — причины опасностей, таящихся в создании подклассов.

В какой-то степени перечень этих вопросов искусственный. В частности, связывание может происходить практически в любой момент, когда два фрагмента кода совместно используют нечто общее. Поэтому, читая приведенный ниже материал, не упускайте из виду те базовые шаблоны, которые вы можете применять в своем коде. А кроме того, постоянно ищите следующие признаки связывания.

- Причудливые зависимости между несвязанными модулями или библиотеками.
- "Простые" изменения в одном модуле, распространяющиеся через несвязанные модули или вызывающие нарушения где-нибудь в системе.
- Разработчики, боящиеся вносить изменения в код, поскольку они не знают, на что это может повлиять.
- Совещания, которые должны посещать все участники проекта, поскольку никто не знает, на кого окажут влияние планируемые изменения.

Крушения поездов

Всем нам не раз приходилось видеть, а возможно, и писать код, аналогичный приведенному ниже.

```
public void applyDiscount(customer, order_id, discount) {
    totals = customer
            .orders
            .find(order_id)
            .getTotals();
    totals.grandTotal = totals.grandTotal - discount;
    totals.discount   = discount;
}
```

В этом коде сначала получается ссылка на некоторые заказы из объекта заказчика, затем используется этот заказ, а далее подводится ряд итогов по данному заказу. Наконец, используя эти итоги, из общего итога вычитается скидка, после чего они обновляются с учетом данной скидки.

В рассматриваемом здесь фрагменте кода производится обход пяти переходных уровней абстракции: от заказчика до итоговых сумм. В конечном счете в коде верхнего уровня должно быть известно, что объект заказчика раскрывает заказы, в заказах имеется метод `find()`, принимающий идентификатор заказа и возвращающий заказ, а у объекта заказа имеется объект `totals` с методами получения и установки общих итогов и скидок. И все это во многом становится известным неявно. Но еще хуже, что многое не *сможет* измениться в будущем, если данный код должен продолжать свою работу и дальше. Таким образом, все методы и свойства оказываются связанными вместе, подобно тому, как в поезде, терпящем крушение, сцеплены все вагоны.

Допустим, в компании принимается следующее решение: скидка ни на один из заказов не должна превышать 40%. Спрашивается: где разместить код, соблюдающий это правило? Можно предположить, что он относится к телу упомянутой выше функции `applyDiscount()`. Но это, конечно, лишь часть ответа на поставленный вопрос, и, учитывая то состояние, в каком теперь находится прикладной код, нельзя с уверенностью сказать, что такой ответ окажется *полным*. Ведь в любом фрагменте кода, т.е. где угодно, можно установить поля в объекте `totals`, и если не уведомить заранее (памятной запиской) сопровождающего данный код, то он не проверит, соблюдается ли в этом коде новое правило.

На данный вопрос можно, например, взглянуть с точки зрения обязанностей. Очевидно, что объект `totals` должен отвечать за подведение итогов. Но он пока еще этого не делает и на самом деле служит лишь контейнером для целого ряда полей, которые доступны любому для запроса и обновления. Чтобы исправить этот недостаток, можно воспользоваться следующим принципом:

Совет 45 Указывай, а не спрашивай

Этот принцип гласит: решения нельзя принимать на основании внутреннего состояния объекта, а затем обновлять данный объект. Ведь это полностью нивелирует преимущества инкапсуляции и распространяет знание о реализации по всему коду. Поэтому первой мерой, предотвращающей крушение поезда, является передача скидки объекту итогов, как показано ниже.

```
public void applyDiscount(customer, order_id, discount) {
    customer
       .orders
       .find(order_id)
       .getTotals()
       .applyDiscount(discount);
}
```

Затруднение, подобное несоблюдению принципа "указывай, а не спрашивай" (tell-don't-ask, TDA), возникает в связи с объектом заказчика и его заказами. В частности, нельзя извлекать список заказов и искать их. Вместо этого следует получить нужный заказ непосредственно от заказчика, как показано ниже.

```
public void applyDiscount(customer, order_id, discount) {
    customer
       .findOrder(order_id)
       .getTotals()
       .applyDiscount(discount);
}
```

Это же относится и к объекту заказа и его итогов. Зачем внешнему миру знать, что в реализации заказа используется отдельный объект для хранения его итогов?

```
public void applyDiscount(customer, order_id, discount) {
    customer
       .findOrder(order_id)
       .applyDiscount(discount);
}
```

И на этом можно бы остановиться. В данный момент можно было бы подумать, что принцип TDA позволяет ввести метод `applyDiscountToOrder(order_id)` в объекты заказчиков. И это в самом деле возможно, если слепо следовать данному принципу.

Но принцип TDA нельзя считать законом природы. Это всего лишь шаблон, помогающий распознать затруднения. В данном случае удобно выявить тот факт, что у заказчика имеются заказы и один из них можно найти, запросив его у объекта заказчика. И это прагматичное решение.

Во всяком приложении имеются определенные понятия верхнего уровня, которые являются универсальными. В рассматриваемом здесь приложении эти понятия включают в себя *заказчиков* и *заказы*. Нет никакого смысла полностью скрывать заказы в объектах заказчиков, поскольку они существуют самостоятельно. Таким образом, можно без особых трудностей создавать API, через которые становятся доступными объекты заказов.

Закон Деметры

Разработчики нередко упоминают в своих беседах о связывании так называемый *закон Деметры*, который устанавливает руководящие принципы[2], сформулированные в конце 1980-х годов Ианом Холландом. Он сформулировал их для того, чтобы помочь разработчикам проекта "Деметра" уточнить свои функции и развязать их.

Закон Деметры гласит, что в методе, определенном в классе C, можно обращаться лишь к следующему:

- другим методам экземпляра из класса C;
- его параметрам;
- методам из объектов, которые в нем создаются и находятся как в стеке, так и в динамической области памяти (так называемой "куче");
- глобальным переменным.

В первом издании данной книги мы уделили немного времени описанию закона Деметры. Но за прошедшие с тех пор двадцать лет цвет этой благоухающей розы несколько увял. Нам теперь не по душе само понятие "глобальная переменная" по причинам, поясняемым в следующем разделе. Кроме того, мы обнаружили, что пользоваться этим законом на практике нелегко. Это все равно, что синтаксически анализировать корректный документ всякий раз, когда вызывается метод.

Тем не менее положенный в его основу принцип все еще остается правильным. Мы лишь рекомендуем более простой способ выражения почти того же самого.

Совет 46 Не связывайте вызовы методов в цепочку

Старайтесь связывать не более двух вызовов методов через точку, получая доступ к какому-нибудь объекту. Такой доступ охватывает также те случаи, когда используются промежуточные переменные, как в следующем фрагменте кода:

[2] На самом деле это не закон, а, скорее, счастливая мысль Деметры.

```
# Это весьма скверный стиль
amount = customer.orders.last().totals().amount;

# Как, впрочем, и этот:
orders = customer.orders;
last = orders.last();
totals = last.totals();
amount = totals.amount;
```

Из данного правила имеется следующее существенное исключение: это правило не применяется, если маловероятно, что все связываемое в цепочку вообще изменится. На практике же в приложении может вполне измениться все, что угодно. Так, все, что находится в сторонней библиотеке, следует считать непостоянным, особенно если сопровождающие данной библиотеки знают об изменениях, вносимых в API между ее последовательными выпусками. Но библиотеки, входящие в состав языков, вероятнее всего, окажутся довольно устойчивыми, а следовательно, нас вполне удовлетворит код, аналогичный приведенному ниже.

```
people
.sort_by {|person| person.age }
.first(10)
.map {| person | person.name }
```

Этот код был вполне работоспособным двадцать лет назад, когда мы написали его на языке Ruby в качестве примера для первого издания данной книги. И он, вероятнее всего, таковым и останется, если теперь обнаружить его в наработках старых программистов.

Цепочки и конвейеры

В разделе "Преобразовательное программирование" далее в этой главе речь пойдет о составлении конвейеров из отдельных функций. В таких конвейерах данные преобразуются и передаются от одной функции к другой. Хотя это и не то же самое, что и крушение поезда из вызовов методов, поскольку в данном случае не делается опора на скрытые подробности реализации.

Нельзя сказать, что конвейеры не вносят никакого связывания. Формат данных, возвращаемых одной функцией в конвейере, должен быть совместим с форматом данных, принимаемых следующей функцией. Как показывает наш опыт, такая форма связывания ставит намного меньше препятствий изменению кода, чем форма, внедряемая через крушения поездов.

Изъяны глобализации

Глобально доступные данные служат коварным источником связывания компонентов приложения. Каждый фрагмент глобальных данных действует так, как будто каждый метод в приложении неожиданно получает дополнительный параметр. Ведь эти глобальные данные доступны буквально в *каждом* методе.

Глобальные данные связывают код по многим причинам. И самая очевидная из них состоит в том, что изменение реализации глобальных данных потенциально воздействует на весь код в системе. На практике, однако, такое воздействие, безусловно, весьма ограничено. Затруднение на самом деле сводится к знанию всех мест, в которые требуется внести изменения.

Глобальные данные образуют связывание и в том случае, если дело доходит до разделения прикладного кода.

Из повторного использования кода было извлечено немало выгод. Как показывает наш опыт, повторное использование вряд ли станет основной заботой при создании кода, но осмысление возможности сделать код повторно используемым должно стать важной частью процедуры программирования. Делая свой код повторно используемым, вы снабжаете его ясным интерфейсом, развязывая его с остальным кодом. Это дает возможность извлекать метод или модуль, не таща за собой все остальное. А если в вашем коде применяются глобальные данные, то становится трудно отделять их от всего остального. Такое затруднение можно обнаружить при написании модульных тестов для проверки кода, в котором применяются глобальные данные. Чтобы создать глобальную среду лишь с целью выполнить тест, придется написать немало подготовительного кода.

Совет 47 Избегайте глобальных данных

Синглтоны также являются глобальными данными

В предыдущем разделе мы осторожно вели речь о *глобальных данных*, а не о *глобальных переменных*. Дело в том, что разработчики нередко заявляют нам следующее: "Послушайте! Нет никаких глобальных переменных — я обернул их в данные экземпляра в объекте синглтона или глобальном модуле".

Но если у вас имеется синглтон с набором экспортируемых переменных экземпляра, то это все равно будут глобальные данные — просто с более длинными именами.

Так что ваши коллеги возьмут этот синглтон и скроют все данные за методами. Вместо того чтобы писать `Config.log_level`, они организуют вызов метода `Config.log_level()` или `Config.getLogLevel()`. Это, конечно, лучше, поскольку означает, что ваши глобальные данные скрывают информацию. И если вы решите изменить представление уровней протоколирования, то сможете сохранить совместимость, взаимно преобразуя новые и прежние уровни в API конфигурирования приложения. Но множество данных конфигурации у вас все равно останется лишь одно.

Глобальные данные включают в себя внешние ресурсы

Любой изменяемый внешний ресурс считается глобальными данными. Если в вашем приложении применяется база данных, хранилище информации, файловая система, API службы и т.д., вы рискуете попасть в ловушку глобализации. И в этом случае решение состоит в том, чтобы заключать такие ресурсы в оболочку кода, который вы контролируете.

> **Совет 48** Если данные настолько важны, чтобы быть глобальными, заключите их в оболочку API

Наследование усугубляет связывание

Злоупотребление наследованием, когда один класс наследует состояние и поведение другого класса, оказывается настолько важным вопросом, что ему посвящен отдельный раздел "Тема 31. Налог на наследование" далее в этой главе.

Все дело в изменениях

Изменить связанный код трудно. Ведь изменения в одном месте кода могут иметь побочные эффекты где-нибудь в другом месте кода, а зачастую — в трудно обнаруживаемых местах, и выходят на свет лишь через месяц эксплуатации.

Сохранение кода скромным, чтобы он оперировал только тем, что ему известно непосредственно, помогает поддерживать приложения развязанными, а следовательно, поддающимися изменениям в большей степени.

Другие разделы, связанные с данной темой

- **Тема 8**. Сущность качественного проектирования **глава 2**, "Прагматичный подход".
- **Тема 9**. DRY — пороки дублирования, **глава 2** "Прагматичный подход".
- **Тема 10**. Ортогональность, **глава 2** "Прагматичный подход".
- **Тема 11**. Обратимость, **глава 2** "Прагматичный подход".
- **Тема 29**. Манипулирование реальным миром.
- **Тема 30**. Преобразовательное программирование.
- **Тема 31**. Налог на наследование.
- **Тема 32**. Конфигурирование.
- **Тема 33**. Разрывание временного связывания, **глава 6** "Параллельность".
- **Тема 34**. Общее состояние — неверное состояние, **глава 6** "Параллельность".

- **Тема 35**. Акторы и процессы, **глава 6** "Параллельность".
- **Тема 36**. Классные доски, **глава 6** "Параллельность".
- Принцип "указывай, а не спрашивай" подробно рассматривается в нашей статье "The Art of Enbugging" (Искусство внесения программных ошибок), опубликованной в *Software Construction*[3] за 2003 г.

Тема 29 Манипулирование реальным миром

"События не происходят просто так: они созданы для этого".

Джон Ф. Кеннеди

В прежние времена, когда мы выглядели еще очень молодо, компьютеры были не особенно удобны в применении. Поэтому мы, как правило, организовывали свое взаимодействие с ними, исходя из их ограничений.

Ныне мы ожидаем от компьютеров намного большего: они должны быть интегрированы в наш мир, и никак иначе. А ведь наш мир довольно беспорядочен. В нем постоянно что-нибудь происходит, предметы перемещаются, а наше мнение меняется, поэтому разрабатываемым нами приложениям приходится каким-то образом согласовывать свои действия.

В этом разделе речь пойдет о таких оперативно реагирующих на внешний мир приложениях. И начнем мы с понятия *события*.

События

Событие представляет собой доступность некоторой информации. С одной стороны, оно может возникнуть во внешнем мире, когда пользователь щелкает мышью на экранной кнопке или обновляется котировка акций на бирже. С другой стороны, оно может быть внутренним, представляя собой результат завершенного вычисления или поиска. Оно может быть даже чем-то тривиальным вроде извлечения следующего элемента из списка.

Независимо от конкретного источника событий, приложения будут работать в реальном мире лучше, если разрабатывать их таким образом, чтобы они реагировали на события. Пользователи найдут такие приложения более интерактивными, а сами приложения будут лучше пользоваться ресурсами.

Но как разрабатывать такие приложения? В отсутствие определенного рода стратегии мы быстро окажемся в затруднительном положении, а наши приложения будут состоять из массы тесно связанного кода.

[3] См. по адресу `https://media.pragprog.com/articles/jan_03_enbug.pdf`.

Рассмотрим четыре стратегии, способные оказать в этом помощь.

1. Конечные автоматы.
2. Проектный шаблон "Наблюдатель".
3. Модель "издатель–подписчик".
4. Реактивное программирование и потоки данных.

Конечные автоматы

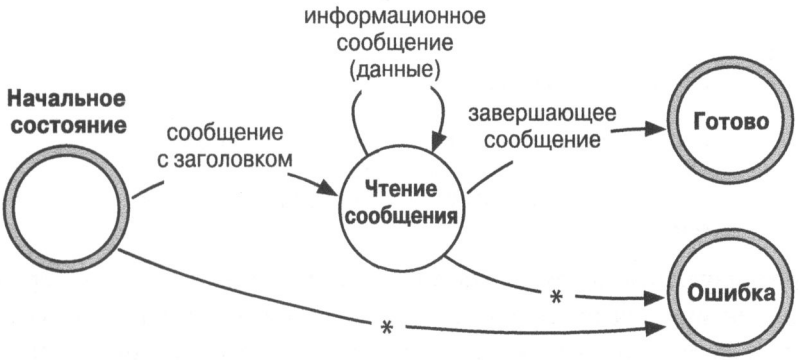
Дэйв считает, что он пишет код с использованием конечных автоматов (КА) почти каждую неделю. Довольно часто реализация КА может состоять из пары строк кода, но эти строки помогают распутать целый клубок потенциальных затруднений.

Пользоваться КА совсем не трудно, и тем не менее, разработчики настроены против них. Вероятнее всего, они считают, что это слишком сложно, что КА применимы только в работе аппаратных средств или что для этого придется воспользоваться малопонятной библиотекой. Ни одно из этих мнений неверно.

Анатомия прагматичного конечного автомата

Конечный автомат, по существу, является всего лишь спецификацией порядка обработки событий. Он состоит из ряда состояний, одно из которых является *текущим*. Для каждого состояния перечисляются события, значимые для данного состояния. И для каждого из этих событий определяется новое текущее состояние системы.

Например, приложение может получать составные сообщения из веб-сокета. Первое сообщение является заголовочным, за ним следует любое количество информационных сообщений, а далее — завершающее сообщение. Такая процедура приема сообщений может быть представлена в виде КА так, как показано ниже.



Все начинается с "начального состояния". Если получено сообщение заголовка, происходит *переход* в состояние "чтение сообщения". Если же в первоначальном состоянии получено еще что-нибудь, то происходит переход (линия с меткой "*" (звездочка)) в состояние "ошибка", и на этом процедура приема сообщений завершается.

В состоянии "чтение сообщения" можно принимать информационные сообщения (в этом случае чтение продолжается в том же самом состоянии) или завершающее сообщение, и тогда процедура переходит в состояние "готово". Все остальное вызывает переход в состояние "ошибка".

КА примечательны тем, что их можно выразить исключительно как данные. В качестве примера ниже приведена таблица, представляющая описанный анализатор сообщений.

Состояние	События			
	Заголовок	Данные	Завершение	Другое
Начальное	Чтение	Ошибка	Ошибка	Ошибка
Чтение сообщения	Ошибка	Чтение	Готово	Ошибка

Строки в этой таблице представляют состояния. Чтобы выяснить, что нужно делать, когда наступает событие, следует найти строку с текущим состоянием, просмотреть столбец, соответствующий событию, и на их пересечении найти ячейку с новым состоянием.

Код, реализующий описанную выше процедуру, в равной степени прост, как показано ниже.

event/simple_fsm.rb

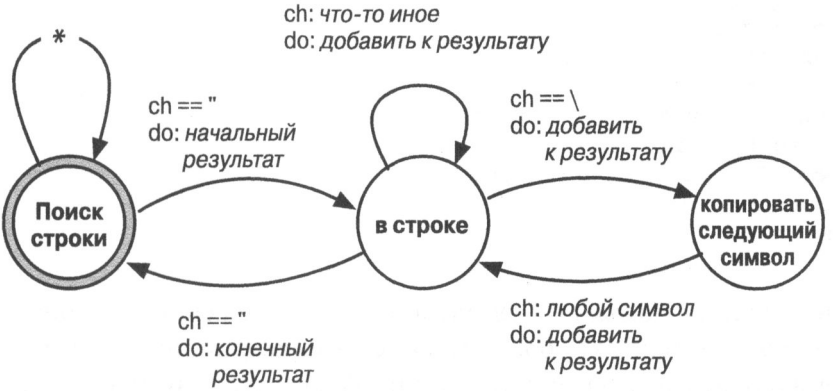
```
Строка 1    TRANSITIONS = {
       -        initial: {header: :reading},
       -        reading: {data: :reading, trailer: :done},
       -    }
       5
       -    state = :initial
       -
       -    while state != :done && state != :error
       -        msg = get_next_message()
      10        state = TRANSITIONS[state][msg.msg_type] || :error
       -    end
```

Код, реализующий переходы между состояниями, находится строке 10, где сначала индексируется таблица переходов — с помощью текущего состояния, а затем переходы из данного состояния индексируются с помощью типа сообщения. Если же соответствующее новое состояние не обнаружено, то происходит переход в состояние ошибки `:error`.

Добавление действий

Чистый КА, наподобие рассматриваемого здесь, является анализатором потока событий, единственным выходом которого является конечное состояние. Его можно наполнить действиями, инициируемыми при определенных переходах.

Например, может возникнуть потребность извлечь все строки из исходного файла. Отдельная строка состоит из текста, заключенного в кавычки, тогда как знак обратной косой черты в строке экранирует следующий символ, а следовательно, `"Ignore \"quotes\""` — единая строка. Ниже приведен КА, выполняющий данное действие.



На этот раз каждый переход помечается двумя метками. Верхняя метка обозначает событие, инициирующее переход, а нижняя — действие, предпринимаемое для смены состояний.

Все это можно выразить в табличном виде, как и в предыдущем примере. Но на этот раз каждая ячейка таблицы содержит двухэлементный список, состоящий из следующего состояния и наименования действия.

event/strings_fsm.rb

```
TRANSITIONS = {
  # текущее состояние   новое состояние      выполняемое действие
  #-------------------------------------------------------------
  look_for_string: {
    '"'             => [ :in_string,          :start_new_string],
    :default        => [ :look_for_string,    :ignore],
  },
  in_string: {
    '"'             => [ :look_for_string,    :finish_current_string],
    '\\'            => [ :copy_next_char,     :add_current_to_string],
    :default        => [ :in_string,          :add_current_to_string],
  },
  copy_next_char: {
    :default        => [ :in_string,          :add_current_to_string],
  },
}
```

Здесь была добавлена возможность указания перехода по умолчанию, предпринимаемого в том случае, если событие не совпадает ни с одним из других переходов из данного состояния. А теперь рассмотрим соответствующий код:

event/strings_fsm.rb

```
state = :look_for_string
result = []

while ch = STDIN.getc
  state, action = TRANSITIONS[state][ch] ||
                  TRANSITIONS[state][:default]
  case action
  when :ignore
  when :start_new_string
    result = []
  when :add_current_to_string
    result << ch
  when :finish_current_string
    puts result.join
  end
end
```

Данный пример похож на предыдущий в том отношении, что события (символы во входных данных) обрабатываются в цикле, инициируя переходы. Но здесь делается нечто большее, чем в предыдущем коде. Результатом каждого перехода является как новое состояние, так и наименование действия. Это наименование используется для того, чтобы выбрать выполняемый код, прежде чем вернуться в начало цикла.

Несмотря на то что рассматриваемый здесь код довольно прост, он исправно выполняет свою функцию. Имеется немало других вариантов: воспользоваться в таблице переходов анонимными функциями или указателями на функции для выполнения отдельных действий; заключить код, реализующий конечный автомат, в отдельный класс со своим состоянием и т.д.

Нечего и говорить, что переходы во все состояния должны быть обработаны одновременно. Так, если требуется пройти процедуру регистрации пользователя в приложении, то по мере ввода им данных, проверки правильности адреса его электронной почты, согласования 107 различных предупреждений, которые приложение должно выдавать в оперативном режиме, и прочего, вероятнее всего, потребуется осуществить целый ряд переходов. Чтобы удовлетворить подобного рода требованиям к последовательности выполняемых действий, целесообразно хранить состояние во внешней памяти и пользоваться им для приведения конечного автомата в действие.

Конечные автоматы как отправная точка

Конечные автоматы находят ограниченное применение у разработчиков, поэтому нам хотелось бы побудить вас искать возможности для их применения.

Тем не менее они не решают все затруднения, возникающие в связи с обработкой событий. Поэтому перейдем к рассмотрению некоторых других способов разрешения затруднений, возникающих при обработке событий.

Проектный шаблон "Обозреватель"

В шаблоне "Обозреватель" (Observer) имеется источник событий, называемый *наблюдаемым объектом*, а также список клиентов, называемых *наблюдателями* и заинтересованных в событиях из этого источника.

Обозреватель регистрирует свой интерес у наблюдаемого объекта, как правило, передавая ссылку на вызываемую функцию. Соответственно, когда наступает событие, наблюдаемый объект обходит список наблюдателей и вызывает функцию, которая была передана ему каждым наблюдателем. При вызове данной функции событие передается ей в качестве параметра.

Ниже приведен пример реализации шаблона "Обозреватель" на языке Ruby. В частности, модуль `Terminator` служит для прерывания работы приложения. Но прежде чем это сделать, он уведомляет всех своих обозревателей о том, что приложение собирается завершиться[4]. Они могут воспользоваться этим уведомлением, чтобы очистить временные ресурсы, зафиксировать данные и т.д.

event/observer.rb

```
module Terminator
  CALLBACKS = []

  def self.register(callback)
    CALLBACKS << callback
  end

  def self.exit(exit_status)
    CALLBACKS.each { |callback| callback.(exit_status) }
    exit!(exit_status)
  end
end

Terminator.register(-> (status) { puts "callback 1 sees #{status}" })
Terminator.register(-> (status) { puts "callback 2 sees #{status}" })

Terminator.exit(99)
```

```
$ ruby event/observer.rb
callback 1 sees 99
callback 2 sees 99
```

Для создания наблюдаемого объекта требуется не так уж и много кода. Для этого достаточно разместить сначала ссылки на функции в списке, а затем вызвать эти функции, когда наступит событие. Это характерный пример того

[4] Нам, конечно, известно, что такая возможность уже реализована в языке Ruby с помощью функции `at_exit()`.

момента, когда *не* следует пользоваться библиотекой. Шаблон наблюдателей и наблюдаемого объекта применялся десятилетиями и сослужил нам добрую службу. Его применение особенно преобладает в системах с пользовательским интерфейсом, где обратные вызовы служат для извещения о том, что в приложении произошло какое-то взаимодействие.

Но шаблону "Обозреватель" присущ следующий недостаток: он вносит связывание, поскольку каждый наблюдатель должен регистрироваться у наблюдаемого объекта. А поскольку в типичной реализации обратные вызовы обрабатываются наблюдаемым объектом синхронно, он может вносить узкие места в производительность системы. Этот недостаток устраняется с помощью рассматриваемой ниже стратегии "издатель–подписчик".

Модель "издатель–подписчик"

Эта модель обобщает шаблон "Обозреватель" и в то же время устраняет недостатки связывания и производительности. Модель "издатель–подписчик" состоит из *издателей* и *подписчиков*, соединяемых вместе через каналы, которые реализуются в отдельном коде: иногда — в библиотеке, порой — в процессе, а иной раз — в распределенной инфраструктуре. Все эти подробности реализации скрыты от прикладного кода.

У каждого канала имеется свое имя. Подписчики регистрируют свой интерес в одном или нескольких этих именованных каналах, куда издатели выводят свои события. В отличие от шаблона "Обозреватель", общение между издателем и подписчиком происходит за пределами прикладного кода и потенциально — асинхронно.

Несмотря на то что самую элементарную систему "издатель–подписчик" можно реализовать самостоятельно, это вряд ли потребуется. Большинство поставщиков облачных услуг предоставляют услуги типа "издатель–подписчик", позволяя подключать приложения по всему миру. В каждом распространенном языке программирования может быть хотя бы одна библиотека типа "издатель–подписчик".

Модель "издатель–подписчик" является удобной технологией развязывания обработки асинхронных событий. Она позволяет добавлять и заменять код, потенциально — во время выполнения конкретного приложения, не изменяя существующий код. Ее недостаток заключается в том, что она затрудняет выяснение происходящего в системе, где интенсивно применяется данная модель. В частности, глядя на издателя, нельзя сразу же увидеть, какие именно подписчики связаны с конкретным сообщением.

В сравнении с шаблоном "Обозреватель" модель "издатель–подписчик" служит характерным примером сокращения нежелательного связывания абстрагированием через общий интерфейс (т.е. канал). Тем не менее она, по сущест-

ву, остается не более чем системой передачи сообщений. Для создания систем, реагирующих на события, возникающие в определенном сочетании, требуется нечто большее, чем передача сообщений. Поэтому рассмотрим далее способы, позволяющие внести измерение времени в обработку событий.

Реактивное программирование, потоки данных и события

Если вам приходилось когда-нибудь пользоваться электронными таблицами, то вы уже в какой-то степени знакомы с *реактивным программированием*. Так, если в одной ячейке электронной таблицы содержится формула, ссылающаяся на другую ячейку, то обновление последней приведет к обновлению первой. Можно сказать, что одни значения *реагируют* на изменения других значений, которые в них используются.

Имеется немало каркасов, способных оказать помощь в организации такого рода реакционности на уровне данных. Так, в области браузеров наиболее распространены каркасы React и Vue.js, реализованные на языке JavaScript, хотя эти сведения могут устареть ко времени выхода настоящего издания в свет.

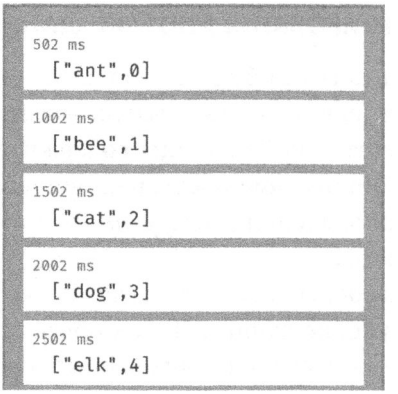
Очевидно, что события могут быть также использованы для запуска реакций в прикладном коде, хотя подключить их не так-то просто. Именно здесь и пригодятся *потоки данных*, которые позволяют трактовать события так, как если бы они были коллекцией данных. Это похоже на список событий, расширяющийся по мере наступления новых событий. Прелесть такого подхода состоит в том, что потоки данных можно обрабатывать таким же образом, как и любую другую коллекцию, манипулируя, комбинируя, фильтруя и делая все остальное, что, как хорошо нам известно, делается с данными. Потоки данных можно даже объединять с обыкновенными коллекциями. Они могут быть асинхронными, а это означает, что в прикладном коде появляется возможность реагировать на события по мере их наступления.

Текущие исходные условия для обработки реактивных событий определяются на веб-сайте по адресу `http://reactivex.io`, где устанавливается ряд не зависящих от конкретного языка принципов и документов для некоторых типичных реализаций. Так, в приведенном ниже примере используется библиотека RxJs для сценария на языке JavaScript.

event/rx0/index.js

```
import * as Observable from 'rxjs'
import { logValues }   from "../rxcommon/logger.js"

let animals  = Observable.of("ant", "bee", "cat", "dog", "elk")
let ticker   = Observable.interval(500)
let combined = Observable.zip(animals, ticker)

combined.subscribe(next => logValues(JSON.stringify(next)))
```

В данном коде применяется простая журнальная функция[5], вводящая элементы в список, отображаемый в окне браузера. Каждый такой элемент снабжается отметкой времени в миллисекундах, прошедших с момента запуска программы. Ниже показано, что именно выводится на экран из данного кода.



Обратите внимание на отметки времени. В частности, через каждые 500 мс из потока данных получается одно событие. Каждое событие содержит порядковый номер, формируемый наблюдаемым объектом `interval`, а также название следующего животного, выбираемое из списка. В окне браузера реально наблюдается появление протокольных строк через каждые полсекунды.

Потоки данных, как правило, заполняются по мере наступления событий, а это подразумевает, что наблюдаемые объекты могут заполнять их параллельно. В приведенном ниже примере кода сведения о пользователях извлекаются из удаленного сайта. Для этой цели используется веб-сайт, общедоступный по адресу `https://reqres.in` и предоставляющий открытый прикладной интерфейс REST API. Частью этого прикладного интерфейса можно воспользоваться, чтобы извлечь данные о конкретном (вымышленном) пользователе, выполнив запрос по методу GET и условию `users/"id"`. В данном коде извлекаются сведения о пользователях с идентификаторами 3, 2 и 1.

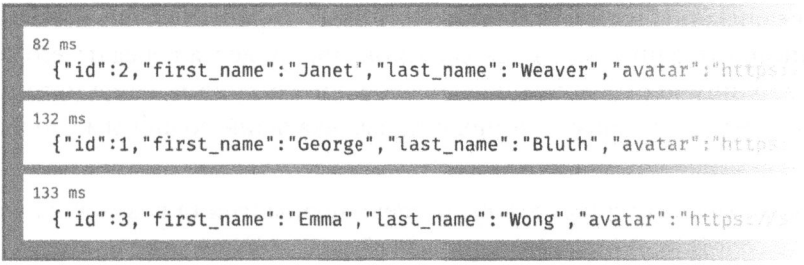
event/rx1/index.js

```
import * as Observable from 'rxjs'
import { mergeMap }    from 'rxjs/operators'
import { ajax }        from 'rxjs/ajax'
import { logValues }   from "../rxcommon/logger.js"

let users = Observable.of(3, 2, 1)
let result = users.pipe(mergeMap((user) =>
           ajax.getJSON('https://reqres.in/api/users/${user}'))
)
```

[5] См. по адресу `https://media.pragprog.com/titles/tpp20/code/event/rxcommon/logger.js`.

```
result.subscribe(
    resp => logValues(JSON.stringify(resp.data)),
    err  => console.error(JSON.stringify(err))
)
```

Внутренние подробности реализации данного кода не так важны. Более привлекателен результат его выполнения:



Обратите внимание на отметки времени. В частности, три запроса или три отдельных потока данных обрабатываются параллельно. Так, первый запрос по идентификатору 2 был обработан через 82 мс, а два последующих запроса — еще через 50 и 51 мс соответственно.

Потоки событий как асинхронные коллекции

В предыдущем примере список идентификаторов пользователей (в наблюдаемом объекте `users`) был статическим, хотя он совсем не обязательно должен быть таковым. Собирать подобные сведения, вероятно, требуется в том случае, когда пользователи регистрируются на веб-сайте. Для этого достаточно инициировать наблюдаемое событие, содержащее идентификатор пользователя, когда создается сеанс его работы на сайте, а затем воспользоваться этим наблюдаемым событием вместо статического. И тогда подробные сведения о пользователях будут извлекаться по мере получения их идентификаторов и предположительно сохраняться где-нибудь.

Это весьма эффективная абстракция, поскольку она избавляет от необходимости думать о времени как о чем-то, чем приходится управлять. Потоки событий обобщают синхронную и асинхронную обработку в общем удобном API.

События вездесущи

События присутствуют везде. Одни из них более очевидны: щелчок кнопкой мыши, истечение срока действия таймера, а другие — менее очевидны: регистрация пользователя, совпадение строки в файле с шаблоном. Но независимо от конкретного источника событий, код их обработки может более оперативно реагировать и стать лучше развязанным, чем более линейный его аналог.

Другие разделы, связанные с данной темой

- **Тема 28.** Развязывание.
- **Тема 36.** Классные доски, **глава 6** "Параллельность".

Упражнения

19. В разделе, посвященном конечным автоматам, упоминалось, что обобщенную реализацию конечного автомата можно переместить в отдельный класс. Для инициализации этого класса, вероятно, придется передать его конструктору таблицу переходов и первоначальное состояние конечного автомата. Попробуйте реализовать компонент, извлекающий подобным способом символьную строку.

20. Какая из рассмотренных в этом разделе технологий (а возможно, их определенное сочетание) окажется пригодной в следующих случаях:

- если в течение пяти минут получены три события о *выходе из строя* сетевого интерфейса, непременно уведомить об этом обслуживающий персонал;
- если после захода солнца обнаруживается движение сначала внизу, а затем вверху лестницы, включить лестничное освещение;
- требуется уведомить различные системы учета о выполнении заказа;
- чтобы выяснить, имеет ли клиент право на заем, приложение должно отправить соответствующие запросы трем серверным службам, ожидая от них ответов.

Тема 30 Преобразовательное программирование

"Если вы не в состоянии описать то, что делаете, как процесс, значит, вы не знаете, что делаете".

У. Эдвардс Деминг[6]

Все программы преобразуют входные данные в выходные. Но когда мы рассуждаем о проектировании, то редко задумываемся об организации преобразований. Вместо этого мы больше беспокоимся о классах и модулях, структурах данных и алгоритмах, языках и каркасах приложений.

Мы считаем, что такая сосредоточенность на коде нередко приводит к тому, что из виду упускается самое главное, поэтому необходимо вернуться к тому, чтобы рассматривать программы как нечто, преобразующее входные данные в выходные. И если поступить именно так, то многие беспокоившие нас ранее подробности просто улетучатся, структуры данных станут более понятными, обработка ошибок — более согласованной, а связывание сведется к минимуму.

[6] W. Edwards Deming — американский ученый-статистик XX века. — *Примеч. пер.*

Чтобы начать исследование данного вопроса, вернемся во времени назад в 1970-е годы, чтобы попросить программирующего в операционной системе Unix написать для нас программу, в которой перечисляются пять самых длинных файлов в дереве каталога, где под словом "самый длинный" подразумевается наличие в файле наибольшего количества строк.

Можно было бы предположить, что этот программирующий откроет текстовый редактор и начнет набирать исходный код на языке C. Но он не сделает этого, поскольку мыслит категориями того, что имеется (дерева каталогов), а также того, что требуется (списка файлов), т.е. входных и выходных данных. И с этой целью он сядет за терминал и наберет на нем последовательность команд, аналогичную приведенной ниже.

```
$ find . -type f | xargs wc -l | sort -n | tail -5
```

В этой последовательности команд выполняются следующие преобразования:

- **`find . -type f`** — направить список всех файлов (параметр **`-type f`**) из текущего каталога (`.`) или его подкаталогов в стандартный поток вывода.
- **`xargs wc -l`** — ввести этот список из стандартного потока ввода и упорядочить их таким образом, чтобы передать в качестве аргументов команде **`wc -l`**, которая подсчитает количество строк в каждом из файлов как своих аргументов и направит полученный результат вместе с именем файла в стандартный поток вывода.
- **`sort -n`** — отсортировать данные из стандартного потока ввода, принимая во внимание, что каждая строка начинается с числа (параметр **`-n`**), направив полученный результат в стандартный поток вывода.
- **`tail -5`** — взять данные из стандартного потока ввода и направить в стандартный поток вывода только последние пять строк.

Если выполнить описанную выше последовательность команд в каталоге с файлами глав данной книги, то в конечном счете будет получен следующий результат:

```
 470 ./test_to_build.pml
 487 ./dbc.pml
 719 ./domain_languages.pml
 727 ./dry.pml
9561 total
```

В последней строке приведенного выше результата указано общее количество строк во всех обработанных, а не только в показанных файлах, поскольку именно таким образом действует команда `wc`. Эту строку можно исключить из результата, запросив еще одну строку из команды `tail`, а затем проигнорировав последнюю строку:

```
$ find . -type f | xargs wc -l | sort -n | tail -6 | head -5
    470 ./debug.pml
    470 ./test_to_build.pml
    487 ./dbc.pml
    719 ./domain_languages.pml
    727 ./dry.pml
```

Проанализируем этот результат с точки зрения данных, проходящих потоком через все стадии процесса их обработки. Первоначальное требование вывести 5 файлов с наибольшим количеством строк превращается в целый ряд преобразований, перечисленных ниже и далее на рис. 5.1.

имя каталога

→ список файлов

→ список с количеством строк

→ отсортированный список

→ пять файлов с наибольшим количеством строк плюс общий итог

→ пять файлов с наибольшим количеством строк

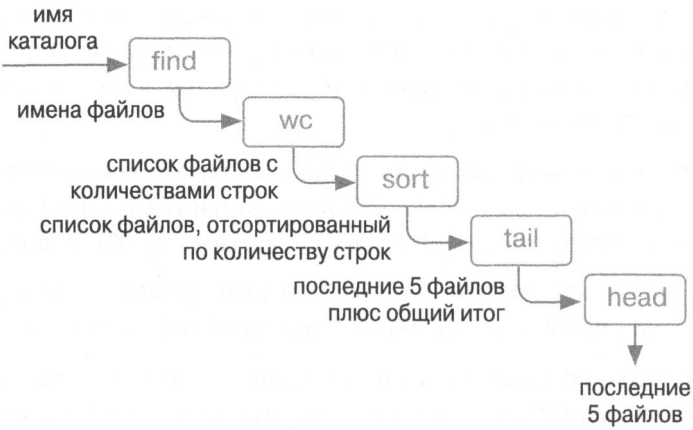


Рис. 5.1. Конвейерное выполнение команды `find` в виде последовательности преобразований

Это очень похоже на промышленный сборочный конвейер, на вход которого подается сырье (исходные данные), а на выходе получается готовая продукция (результаты обработки исходных данных). Именно так мы и предпочитаем рассматривать весь код.

Совет 49 Программирование имеет отношение к коду, а программы — к данным

Обнаружение преобразований

Иногда, чтобы найти преобразования, проще всего начать с требования и определить сначала его входные и выходные данные, а затем функцию, представляющую программу в целом. После этого можно выяснить стадии преобразования входных данных в выходные. Это метод *нисходящего* проектирования.

Допустим, требуется создать веб-сайт для тех, кто увлекается играми в слова, находя среди них все слова, которые могут быть составлены из определенного ряда букв. В качестве входных данных здесь служит ряд букв, а в качестве выходных данных — список слов, составленных из трех, четырех и большего количества букв, как показано ниже.

```
                                          3 => ivy, lin, nil, yin
слово "lvyin" преобразуется в слова →     4 => inly, liny, viny
                                          5 => vinyl
```

(Все это действительно слова — по крайней мере, в соответствии со словарем macOS.)

Вся хитрость в данном примере состоит только в наличии словаря, группирующего слова по *сигнатуре*, выбранной таким образом, чтобы все слова, состоящие из одних и тех же букв, имели одинаковую сигнатуру. Простейшая функция сигнатуры состоит лишь в отсортированном списке букв в слове. И это дает возможность найти входную символьную строку, сформировав сначала для нее сигнатуру, а затем найти в словаре, если удастся, все слова с одинаковой сигнатурой.

Таким образом, *средство поиска анаграмм* (т.е. слов, получаемых путем перестановки букв) можно разбить на процесс, состоящий из четырех отдельных преобразований, как показано ниже.

Стадия	Преобразование	Образец данных
0	Первоначальные входные данные	`"ylvin"`
1	Все сочетания из трех или более букв	**`vin`**, **`viy`**, **`vil`**, **`vny`**, **`vnl`**, **`vyl`**, **`iny`**, **`inl`**, **`iyl`**, **`nyl`**, **`viny`**, **`vinl`**, **`viyl`**, **`vnyl`**, **`inyl`**, **`vinyl`**
2	Сигнатуры сочетаний	**`inv`**, **`ivy`**, **`ilv`**, **`nvy`**, **`lnv`**, **`lvy`**, **`iny`**, **`iln`**, **`ily`**, **`lny`**, **`invy`**, **`ilnv`**, **`ilvy`**, **`lnvy`**, **`ilny`**, **`ilnvy`**
3	Список всех слов в словаре, совпадающих с любыми из указанных сигнатур	**`ivy`**, **`yin`**, **`nil`**, **`lin`**, **`viny`**, **`liny`**, **`inly`**, **`vinyl`**

4	Группировка слов по длине	`3 => ivy, lin, nil, yin`
		`4 => inly, liny, viny`
		`5 => vinyl`

Преобразования по нисходящей

Итак, начнем с рассмотрения первой стадии описываемого здесь процесса, на которой берется исходное слово и из него составляется список всех сочетаний из трех или более букв. Эта стадия выражается в виде приведенного ниже перечня преобразований.

Стадия	Преобразование	Образец данных
1.0	Первоначальные входные данные	`"vinyl"`
1.1	Преобразование символов	`v, i, n, y, l`
1.2	Получение всех подмножеств	`[], [v], [i], ... [v,i], [v,n], [v,y], ... [v,i,n], [v,i,y], ... [v,n,y,l], [i,n,y,l], [v,i,n,y,l]`
1.3	Только те сочетания, которые состоят более чем из трех символов	`[v,i,n], [v,i,y], ... [i,n,y,l], [v,i,n,y,l]`
1.4	Преобразование обратно в символьные строки	`[vin,viy, ... inyl,vinyl]`

Итак, мы достигли момента, когда можно без труда реализовать каждое преобразование непосредственно в коде (используя в данном случае язык Elixir).

function-pipelines/anagrams/lib/anagrams.ex

```
defp all_subsets_longer_than_three_characters(word) do
  word
  |> String.codepoints()
  |> Comb.subsets()
  |> Stream.filter(fn subset -> length(subset) >= 3 end)
  |> Stream.map(&List.to_string(&1))
end
```

Назначение оператора |>

Как и во многих других функциональных языках, в Elixir имеется конвейерный оператор, иногда еще называемый *подающим конвейером* (forward pipe), или просто *конвейером* (pipe)[7]. Его основное назначение — взять значение, указанное слева от него, и вставить его в качестве первого параметра функции справа, так что строка кода

```
"vinyl" |> String.codepoints |> Comb.subsets()
```

аналогична приведенной ниже.

```
Comb.subsets(String.codepoints("vinyl"))
```

[7] По-видимому, первое употребление символов |> для обозначения канала относится к 1994 году, когда велась дискуссия о языке Isobelle/ML, архивные материалы которой см. по адресу https://blogs.msdn.microsoft.com/dsyme/2011/05/17/archeological-semiotics-the-birth-of-the-pipeline-symbol-1994/.

В других языках конвейерное значение может быть вставлено в качестве *последнего* параметра следующей функции. Хотя это зависит, главным образом, от стиля организации встроенных библиотек.

На первый взгляд, это всего лишь синтаксическое удобство, но конвейерный оператор на самом деле является важной возможностью мыслить иначе. Применение конвейера, по существу, означает автоматическое мышление категориями преобразования данных. Всякий раз, когда в исходном коде встречается оператор |>, он обозначает место, где данные перемещаются потоком от одного преобразования к другому.

Нечто подобное конвейерному оператору имеется во многих других языках. Так, в языках Elm, F# и Swift имеется аналогичный оператор |>, а в Clojure — операторы -> и ->>, которые действуют несколько иначе, тогда как в языке R — оператор %>%. В языке Haskell имеются не только готовые конвейерные операторы, но и возможность легко объявлять новые. На момент написания данной книги велась полемика по поводу внедрения конвейерного оператора |> и в языке JavaScript.

Если в применяемом вами языке поддерживается нечто подобное, то вам повезло. В противном случае см. далее врезку "Отсутствие конвейеров в языке X". Впрочем, обратимся непосредственно к коду.

Продолжим преобразование...

Теперь рассмотрим вторую стадию описываемого здесь процесса, реализуемого в основной программе, где подмножества преобразуются в сигнатуры. И на этот раз речь пойдет о простом преобразовании, в результате которого список подмножеств становится списком сигнатур:

Стадия	Преобразование	Образец данных
2.0	Первоначальные входные данные	**vin**, **viy**, ... **inyl**, **vinyl**
2.1	Преобразование в сигнатуры	**inv**, **ivy** ... **ilny**, **inlvy**

Код Elixir, реализующий такое преобразование, достаточно прост, как можно убедиться:

function-pipelines/anagrams/lib/anagrams.ex

```
defp as_unique_signatures(subsets) do
  subsets
  |> Stream.map(&Dictionary.signature_of/1)
end
```

Теперь преобразуем этот список сигнатур, чтобы отобразить каждую из них на список известных слов одинаковой сигнатуры или присвоить пустое значение `nil` в отсутствие таких слов. После этого придется удалить все пустые значения `nil` и свести все вложенные списки в единый список.

function-pipelines/anagrams/lib/anagrams.ex

```
defp find_in_dictionary(signatures) do
  signatures
  |> Stream.map(&Dictionary.lookup_by_signature/1)
  |> Stream.reject(&is_nil/1)
  |> Stream.concat(&(&1))
end
```

На четвертой стадии группирования слов по длине выполняется еще одно простое преобразование списка в отображение, где ключами являются длины слов, а значениями — все слова данной длины:

function-pipelines/anagrams/lib/anagrams.ex

```
defp group_by_length(words) do
  words
  |> Enum.sort()
  |> Enum.group_by(&String.length/1)
end
```

Собирая все вместе

Мы реализовали в коде каждое отдельное преобразование, теперь самое время собрать их все вместе в одной главной функции, приведенной ниже.

function-pipelines/anagrams/lib/anagrams.ex

```
def anagrams_in(word) do
  word
  |> all_subsets_longer_than_three_characters()
  |> as_unique_signatures()
  |> find_in_dictionary()
  |> group_by_length()
end
```

Проверим работоспособность данной функции следующим образом:

```
iex(1)> Anagrams.anagrams_in "lyvin"
%{
    3 => ["ivy", "lin", "nil", "yin"],
    4 => ["inly", "liny", "viny"],
    5 => ["vinyl"]
}
```

Отсутствие конвейеров в языке X

Конвейеры применялись уже давно, но только в узкоспециализированных языках. И лишь недавно они перекочевали в ведущие языки программирования, хотя во многих распространенных языках поддержка данного понятия по-прежнему отсутствует.

Преимущество мышления категориями преобразований заключается в том, что оно не требует знания синтаксиса конкретного языка. Это скорее принцип проектирования. И хотя этот принцип означает конструирование кода в виде преобразований, они записываются в виде последовательности операторов присваивания, как показано ниже.

```
const content = File.read(file_name);
const lines   = find_matching_lines(content, pattern)
const result  = truncate_lines(lines)
```

Несмотря на то что это трудоемкий способ, он все же вполне работоспособен.

В чем же здесь польза

Рассмотрим тело главной функции снова:

```
word
|> all_subsets_longer_than_three_characters()
|> as_unique_signatures()
|> find_in_dictionary()
|> group_by_length()
```

Это всего лишь цепочка преобразований, необходимых для удовлетворения исходного требования, причем каждое из них принимает входные данные из предыдущего преобразования и передает выходные данные следующему преобразованию. И это как можно более точно приводит непосредственно к коду.

Но дело здесь обстоит несколько сложнее, чем кажется на первый взгляд. Так, если у вас есть навыки объектно-ориентированного программирования, то ваши рефлексы требуют скрывать данные, инкапсулируя их в объектах, которые обмениваются ими, изменяя состояние друг друга. А поскольку это приводит к внесению в программу связывания, то служит веским основанием считать объектно-ориентированные системы с трудом поддающимися изменениям.

Совет 50 Не накапливайте состояние, а передавайте его по кругу

В преобразовательной модели дело меняется коренным образом. Вместо небольших массивов, распределенных по всей системе, данные следует уподобить

в рассуждении могучей реке или *потоку*. Таким образом, данные становятся сродни функциональным возможностям, где конвейер — это последовательность код → данные → код → данные... Данные больше не привязаны к конкретной группе функций, как это обычно делается в определении класса. Напротив, они свободно представляют постепенно развертывающийся в приложении процесс преобразования входных данных в выходные. А это означает возможность значительно сократить связывание, поскольку одну функцию можно использовать (и неоднократно) везде, где ее параметры совпадают с выходом из какой-нибудь другой функции.

Впрочем, определенная степень связывания все же существует, но, как показывает наш опыт, она легче поддается управлению, чем при объектно-ориентированном программировании. И если вы программируете на языке с проверкой соответствия типов, то получите предупреждения во время компиляции, если попытаетесь соединить две несовместимые вещи.

А как насчет обработки ошибок

До сих пор рассматриваемые здесь преобразования действовали в такой среде, где ничего неверного не происходит. Но как выполнить их в реальной обстановке? Если можно составить лишь линейные цепочки, то как внедрить всю ту условную логику, которая требуется для проверки ошибок?

Это можно сделать самыми разными способами, но все они основываются на следующем соглашении: значения ни в коем случае нельзя передавать от одного преобразования к другому в исходном виде, а вместо этого их следует заключать в структуру (или тип) данных, где можно проверить достоверность содержащихся в них значений. Например, в языке Haskell такая оболочка называется `Maybe`, а в языках F# и Scala — `Option`.

Особенности применения данного принципа зависят от конкретного языка программирования. Но в общем, реализовать его непосредственно в коде можно двумя способами: проверять ошибки в самих преобразованиях или же за их пределами.

В языке Elixir, которым мы пользовались в приведенных до сих пор примерах кода, отсутствует встроенная поддержка обработки ошибок при преобразованиях. Но это вполне отвечает нашим целям, поскольку нам нужно продемонстрировать реализацию такой обработки ошибок с самого начала. Нечто подобное должно оказаться пригодным и для программирования на других языках.

Сначала выбирается представление

Нам потребуется некоторое представление оболочки, т.е. структуры данных, содержащей значение или признак ошибки. Для этой цели можно воспользоваться структурами, но в языке Elixir уже имеется следующее довольно строгое соглашение: функции преимущественно возвращают кортеж `{:ok, `*значение*`}`

или {:error, *причина*}. Например, функция File.open() возвращает признак :ok и идентификатор процесса ввода-вывода или же признак :error и код причины ошибки, как показано ниже. Кортежами с признаками :ok и :error можно воспользоваться в качестве оболочки, передавая результаты преобразований по конвейеру.

```
iex(1)> File.open("/etc/passwd")
{:ok, #PID<0.109.0>}
iex(2)> File.open("/etc/wombat")
{:error, :enoent}
```

Затем обработка ошибок выполняется в каждом преобразовании

Итак, напишем функцию, возвращающую из файла все строки, содержащие заданную символьную строку, усеченную до 20 первых символов. Эту функцию требуется написать как преобразование с именем файла и сопоставляемой символьной строкой в качестве входных данных, а также с кортежем, содержащим признак :ok и список строк или же признак :error и причину ошибки, в качестве выходных данных. В общих чертах такая функция должна выглядеть следующим образом:

function-pipelines/anagrams/lib/grep.ex

```
def find_all(file_name, pattern) do
  File.read(file_name)
  |> find_matching_lines(pattern)
  |> truncate_lines()
end
```

Здесь ошибки не проверяются явным образом, но если на любой стадии конвейерного процесса возвращается кортеж с признаком ошибки, то обнаруженная ошибка будет возвращена из конвейера, причем последующие функции далее не выполняются[8]. Это делается языковыми средствами Elixir для сопоставления с шаблоном, как показано ниже.

function-pipelines/anagrams/lib/grep.ex

```
defp find_matching_lines({:ok, content}, pattern) do
  content
  |> String.split(~r/\n/)
  |> Enum.filter(&String.match?(&1, pattern))
  |> ok_unless_empty()
end

defp find_matching_lines(error, _), do: error
  # ----------
```

[8] Мы допустили здесь некоторую вольность. Технически последующие функции выполняются. Не выполняется их код.

```
defp truncate_lines({ :ok, lines }) do
  lines
  |> Enum.map(&String.slice(&1, 0, 20))
  |> ok()
end

defp truncate_lines(error), do: error
# ----------

defp ok_unless_empty([]), do: error("nothing found")
defp ok_unless_empty(result), do: ok(result)

defp ok(result), do: { :ok, result }
defp error(reason), do: { :error, reason }
```

Проанализируем функцию `find_matching_lines()`. Если в качестве ее первого параметра указан кортеж с признаком `:ok`, то содержимое этого кортежа используется для поиска строк, совпадающих с заданным шаблоном `pattern`. Но если кортеж с признаком `:ok` *не* указан в качестве первого параметра данной функции, то выполняется ее второй вариант, который просто возвращает этот параметр. Таким образом, данная функция просто передает ошибку дальше по конвейеру. Аналогичным образом действует и функция `truncate_lines()`.

Мы можем поэкспериментировать с этим в консоли, как показано ниже. И, как видите, ошибка, возникающая в любом месте конвейера, сразу же становится его значением.

```
iex> Grep.find_all "/etc/passwd", ~r/www/
{:ok, ["_www:*:70:70:World W", "_wwwproxy:*:252:252:"]}
iex> Grep.find_all "/etc/passwd", ~r/wombat/
{:error, "nothing found"}
iex> Grep.find_all "/etc/koala", ~r/www/
{:error, :enoent}
```

Иначе обработка ошибок выполняется в самом конвейере

Глядя на функции `find_matching_lines()` и `truncate_lines()`, можно подумать, что бремя ответственности за обработку ошибок перенесено на преобразования. Так и есть. В таком языке, как Elixir, где в вызовах функций применяется сопоставление с шаблоном, последствия подобного переноса смягчаются, но они все равно скверные.

Было бы неплохо, если бы в языке Elixir была внедрена такая разновидность конвейерного оператора `|>`, которому было бы известно, как обращаться с кортежами `:ok`/`:error`, чтобы сократить выполнение, когда возникает ошибка[9].

[9] На самом деле такой оператор можно внедрить в Elixir, используя его макросредства. Примером тому служит библиотека Monad в шестнадцатеричной форме. Для этой цели можно также воспользоваться конструкцией `with` языка Elixir, но тогда потеряет всякий смысл программирование преобразований, выполняемых с помощью конвейеров.

Но, к сожалению, ничего подобного нельзя сделать в Elixir, причем так, чтобы это было применимо и в ряде других языков.

Дело в том, что когда возникает ошибка, было бы нежелательно выполнять код дальше по конвейеру, где совсем не обязательно знать о возникшей ошибке. Это означает, что выполнение некоторых функций в конвейере придется отложить до тех пор, пока не станет известно, что предыдущие стадии конвейерного процесса завершились удачно. И для этого *вызовы* функций придется заменить *значениями* функций, которые могут быть вызваны позднее. Ниже приведен один из примеров реализации такого подхода к обработке ошибок в конвейере.

function-pipelines/anagrams/lib/grep1.ex

```
defmodule Grep1 do

  def and_then({ :ok, value }, func), do: func.(value)
  def and_then(anything_else, _func), do: anything_else

  def find_all(file_name, pattern) do
    File.read(file_name)
    |> and_then(&find_matching_lines(&1, pattern))
    |> and_then(&truncate_lines(&1))
  end

  defp find_matching_lines(content, pattern) do
    content
    |> String.split(~r/\n/)
    |> Enum.filter(&String.match?(&1, pattern))
    |> ok_unless_empty()
  end

  defp truncate_lines(lines) do
    lines
    |> Enum.map(&String.slice(&1, 0, 20))
    |> ok()
  end

  defp ok_unless_empty([]),     do: error("nothing found")
  defp ok_unless_empty(result), do: ok(result)

  defp ok(result),    do: { :ok, result }
  defp error(reason), do: { :error, reason }
end
```

Функция `and_then()` служит примером функции *связывания*. Она принимает значение, заключенное в некоторую оболочку, а затем применяет к нему заданную функцию, возвращая новое значение, заключенное в оболочку. Чтобы воспользоваться функцией `and_then()` в конвейере, потребуется дополнительная расстановка знаков препинания, поскольку в языке Elixir требуется явно указывать преобразование вызовов функций в значения функций, хотя эти дополнительные усилия возмещаются тем, что функции преобразования упрощаются. В частности, каждая из них принимает значение (и любые дополнительные параметры) и возвращает кортеж `{:ok, `*новое_значение*`}` или `{:error, `*причина*`}`.

Преобразования преобразуют программирование

Осмысление кода как последовательности (вложенных) преобразований может стать освободительным подходом к программированию. Чтобы привыкнуть к этому, потребуется время, но как только вы выработаете в себе такую привычку, то обнаружите, что ваш код станет более ясным, функции — более краткими, а проекты — более ровными. Попробуйте!

Другие разделы, связанные с данной темой

- **Тема 8**. Сущность качественного проектирования, **глава 2** "Прагматичный подход".
- **Тема 17**. Игры в скорлупки, **глава 3** "Основные инструментальные средства".
- **Тема 26**. Как сбалансировать ресурсы, **глава 4** "Прагматичная паранойя".
- **Тема 28**. Развязывание.
- **Тема 35**. Акторы и процессы, **глава 6** "Параллельность".

Упражнения

21. Можете ли вы выразить перечисленные ниже требования в виде преобразования верхнего уровня? Определите для каждого из них входные и выходные данные.

1) Налоги на поставку и с продаж добавляются в заказ.
2) Приложение загружает конфигурацию из именованного файла.
3) Пользователь входит в веб-приложение, регистрируясь в нем.

22. Допустим, вы выяснили потребность проверить на достоверность и преобразовать символьную строку из поля ввода в целое число, находящееся в пределах от 18 до 150. В общих чертах такое преобразование описывается следующим образом:

```
field contents as string
     → [validate & convert]
        → {:ok, value} | {:error, reason}
```

Напишите код, реализующий отдельные преобразования, включающие проверку на достоверность и преобразование.

23. Ранее во врезке "Отсутствие конвейеров в языке X" был приведен следующий фрагмент кода:

```
const content = File.read(file_name);
const lines   = find_matching_lines(content, pattern)
const result  = truncate_lines(lines)
```

Многие разработчики пишут объектно-ориентированный код, составляя в цепочку вызовы методов. Они могли бы поддаться искушению написать приведенный выше фрагмент кода так, как показано ниже.

```
const result = content_of(file_name)
               .find_matching_lines(pattern)
               .truncate_lines()
```

Чем отличаются оба эти фрагмента кода? Какой из них, на ваш взгляд, является более предпочтительным?

Тема 31 Налог на наследование

"Вам захотелось бананов, но в итоге вы получили гориллу, держащую банан, и целые джунгли".

Джо Армстронг[10]

Программируете ли вы на объектно-ориентированном языке и пользуетесь ли вы наследованием? Если так, то остановитесь! Ведь это, вероятнее всего, совсем не то, что вам требуется. В этом разделе поясняются причины.

Немного предыстории

Наследование впервые появилось в языке Simula 67 в 1969 году. Это было изящное решение задачи упорядочения нескольких типов событий в одном и том же списке. В подходе к наследованию, принятом в языке Simula, принялись так называемые *префиксные классы*. Это давало возможность написать код, аналогичный приведенному ниже.

```
link CLASS car;
... реализация автомобиля
link CLASS bicycle;
... реализация велосипеда
```

В данном коде `link` обозначает префиксный класс, внедряющий функциональные возможности связанных списков. Это позволяет ввести автомобили и велосипеды в список транспортных средств, ожидающих, например, зеленого света дорожного светофора. В современной терминологии `link` обозначает родительский класс.

Мысленная модель, применявшаяся программирующими на Simula, состояла в том, что данным экземпляра и реализации класса `link` предшествовала реализация классов `car` и `bicycle`, а часть `link` этой модели рассматривалась в основном как *контейнер* для хранения автомобилей и велосипедов. Тем самым

[10] Joe Armstrong — создатель языка Erlang. — *Примеч. пер.*

программирующие на Simula получали в свое распоряжение такую форму полиморфизма, в которой автомобили и велосипеды реализовывали интерфейс `link`, поскольку и те и другие содержали код `link`.

Вслед за Simula последовал язык Smalltalk. Алан Кей, один из создателей Smalltalk, отвечает на портале вопросов и ответов *Quora* за 2019 год[11] на вопрос, почему в этом языке имеется наследование, следующим образом:

> *"Когда я разработал язык Smalltalk-72 (а это было сделано шутки ради по зрелом размышлении о Smalltalk-71), я подумал, что было бы забавно воспользоваться Lisp-подобной динамикой для экспериментирования с "дифференциальным программированием". (Имеются в виду различные способы осуществить нечто подобное методом исключения.)"*

Это, в сущности, означает наследование исключительно для поведения.

Обе эти разновидности наследования, у которых было очень много общего, постепенно развивались в течение последующих десятилетий. Принятый в Simula подход, который предполагал в наследовании способ объединения типов данных, был продолжен в таких языках, как C++ и Java. А направление, взятое в Smalltalk, где наследование считалось динамической организацией видов поведения, нашло свое продолжение в таких языках, как Ruby и JavaScript.

С тех пор выросло целое поколение разработчиков объектно-ориентированных приложений, которые пользуются наследованием по одной из двух причин: они не любят набирать код на клавиатуре или же им нравятся типы данных. Те, кто не любит набирать код на клавиатуре, берегут свои пальчики, используя наследование для внедрения общих функциональных возможностей из базового класса в порожденные классы. Например, подклассы `User` и `Product` являются производными от класса `ActiveRecord::Base`. А те, кому нравятся типы данных, пользуются наследованием для того, чтобы выразить отношение между классами. Например, класс `Car` реализует автомобиль как разновидность транспортного средства, реализуемого в классе `Vehicle`. Но, к сожалению, применение обеих упоминаемых здесь разновидностей наследования вызывает определенные трудности.

Трудности применения наследования для совместного использования кода

Наследование означает связывание. При этом связывается не только порожденный класс с родительским, но и родительский класс со своим родителем и т.д. Но исходный код, в котором *применяется* порожденный класс, связывается также со всеми своими предками. Ниже приведен характерный тому пример.

---

[11] См. по адресу `https://www.quora.com/What-does-Alan-Kay-think-about-inheritance-in-object-oriented-programming`.

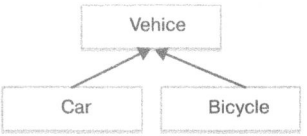
```
class Vehicle
  def initialize
    @speed = 0
  end
  def stop
    @speed = 0
  end
  def move_at(speed)
    @speed = speed
  end
end

class Car < Vehicle
  def info
    "I'm car driving at #{@speed}"
  end
end

# код верхнего уровня
my_ride = Car.new
my_ride.move_at(30)
```

Когда на верхнем уровне делается вызов `my_car.move_at()`, вызывается метод из класса `Vehicle`, являющегося родительским для класса `Car`. Теперь, когда разработчик, отвечающий за класс `Vehicle`, вносит изменения в API, метод `move_at()` превращается в метод `set_velocity()`, а переменная экземпляра `@speed` — в переменную `@velocity`.

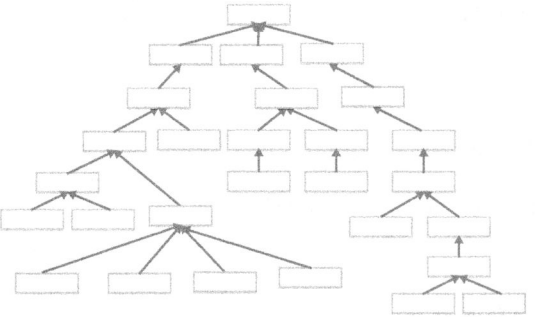
Следует ожидать, что изменения в API нарушат поведение клиентов класса `Vehicle`. Но этого не произойдет на верхнем уровне — по крайней мере, что касается применения в нем класса `Car`. Ведь то, что делается в классе `Car` с точки зрения реализации, не касается кода верхнего уровня, хотя его функционирование все же нарушается. Аналогично имя переменной экземпляра относится исключительно к подробностям внутренней реализации, но когда вносятся изменения в класс `Vehicle`, негласно нарушается и функционирование класса `Car`. И все это объясняется слишком тесным связыванием.

Трудности применения наследования для построения типов данных

Некоторые разработчики усматривают в наследовании способ определения новых типов данных. Их излюбленная проектная диаграмма показывает иерархии классов. Они рассматривают решаемые задачи так, как викторианские благородные ученые рассматривали в свое время природу как нечто, разделяемое на категории.



Но, к сожалению, такие диаграммы вскоре, по мере ввода новых уровней наследования с целью выразить в мельчайших подробностях отличия между классами, вырастают до чудовищных размеров на всю стену. Такое усложнение способно сделать приложение более хрупким, как только вносимые в него изменения станут распространяться вверх и вниз по уровням наследования.



Еще хуже дело обстоит с множественным наследованием. В частности, автомобиль, представленный в классе `Car`, может быть разновидностью не только транспортного средства, представленного в классе `Vehicle`, но и активом, застрахованной или закладываемой под ссуду частной собственностью, представленными классами `Asset`, `InsuredItem` и `LoanCollateral` соответственно. Для правильного моделирования такой иерархии классов потребуется множественное наследование.

В 1990-е годы множественное наследование получило в языке C++ неудачное название в силу некоторой сомнительно разрешающей противоречия семантики. В результате во многих современных объектно-ориентированных языках оно не поддерживается. Это означает, что вам не удастся точно смоделировать предметную область своего приложения, даже если вас вполне удовлетворяют сложные деревья типов данных.

Совет 51 Не платите налог на наследование

Лучшие альтернативы

Рассмотрим три следующие методики программирования, которые обозначают, что вы не должны больше пользоваться наследованием.

- Интерфейсы и протоколы.
- Делегирование.
- Миксины и свойства.

Интерфейсы и протоколы

В большинстве объектно-ориентированных языков можно указать, что в классе реализуется один или несколько видов поведения. Например, в классе `Car` реализуются виды поведения `Drivable` и `Locatable`. Синтаксис, применяемый для этой цели, отличается в разных языках. Так, в языке Java он может выглядеть следующим образом:

```
public class Car implements Drivable, Locatable {
  // Код для класса Car. Этот код должен включать
  // в себя функциональные возможности обоих
  // интерфейсов Drivable и Locatable
}
```

`Drivable` и `Locatable` называются в языке Java *интерфейсами*, а в других языках программирования — *протоколами*, а в некоторых — *свойствами* (traits) (хотя термином "свойство" мы будем в дальнейшем называть нечто иное).

Интерфейсы объявляются, например, так:

```
public interface Drivable {
  double getSpeed();
  void stop();
}

public interface Locatable() {
  Coordinate getLocation();
  boolean locationIsValid();
}
```

В приведенных выше объявлениях не программируется никаких действий, а просто извещается, что в любом классе, реализующем интерфейс `Drivable`, обязаны быть реализованы методы `getSpeed()` и `stop()`, а в классе, реализующем интерфейс `Locatable`, — методы `getLocation()` и `locationIsValid()`. Это означает, что предыдущее определение класса `Car` окажется действительным лишь в том случае, если в него будут включены эти четыре метода.

Сильной стороной интерфейсов и протоколов является возможность пользоваться ими как типами данных, причем любой класс, реализующий соответствующий интерфейс, будет совместим с его типом. Так, если оба класса `Car` и `Phone` реализуют интерфейс `Locatable`, то их объекты можно размещать в списке элементов, определяемых по местоположению, как показано ниже.

```
List<Locatable> items = new ArrayList<>();
items.add(new Car(...));
items.add(new Phone(...));
items.add(new Car(...));
// ...
```

В таком случае можно обработать данный список, зная наверняка, что у каждого его элемента имеются методы `getLocation()` и `locationIsValid()`:

```
void printLocation(Locatable item) {
  if (item.locationIsValid() {
    print(item.getLocation().asString());
  }
}

  // ...

  items.forEach(printLocation);
```

Совет 52 Выражать полиморфизм предпочтительнее с помощью интерфейсов

Интерфейсы и протоколы обеспечивают полиморфизм без наследования.

Делегирование

Наследование побуждает разработчиков создавать классы, у объектов которых имеется большое количество методов. Так, если в родительском классе имеется 20 методов, а в его подклассе требуется воспользоваться лишь двумя из них, то в его объектах все равно будут доступны для вызова 18 остальных методов. Можно сказать, что такой класс потерял контроль над своим интерфейсом, и это типичное затруднение. Так, многие каркасы хранения и пользовательского интерфейса требуют, чтобы компоненты приложения осуществляли наследование подкласса от предоставляемого каркасом базового класса:

```
class Account < PersistenceBaseClass
end
```

Класс `Account` содержит теперь весь прикладной интерфейс API класса `PersistenceBaseClass`. Рассмотрим в качестве альтернативы применение делегирования, как в приведенном ниже примере.

```
class Account
  def initialize(. . .)
    @repo = Persister.for(self)
  end

  def save
    @repo.save()
  end
end
```

Теперь из прикладного интерфейса API ничто *не доступно* клиентам класса `Account`. Более того, мы не ограничены в своих действиях только API применяемого каркаса хранения и можем создать новый API при надобности в таковом. Разумеется, мы могли бы сделать это и ранее, но всегда есть риск, что написанный *нами* интерфейс может быть обойден, а вместо него использован API каркаса хранения. Теперь же мы полностью контролируем ситуацию.

Совет 53 Делегируйте полномочия службам: отношение СОДЕРЖИТ превосходит отношение ЯВЛЯЕТСЯ

На самом деле можно пойти еще дальше. Зачем, например, классу `Account` знать, как сохраняться? Обязан ли он знать и соблюдать бизнес-правила ведения учетной записи?

```
class Account
  # ничего, кроме сведений об учетной записи
end

class AccountRecord
  # заключает в себе учетную запись с возможностями
  # извлечения и сохранения
end
```

Теперь развязывание действительно достигнуто, хотя и не бесплатно. Для этого пришлось написать больше кода, часть которого, как правило, стереотипная. И вероятнее всего, что во всех классах записи потребуется, например, метод `find()`. Правда, для этой цели имеются миксины и свойства.

Миксины, свойства, категории, расширения протоколов и прочее

Те, кто работают в той же отрасли, где и все мы, любят присваивать разным понятиям замысловатые названия. И зачастую им присваиваются многие названия. И чем их больше, тем лучше, не так ли?

Именно так и происходит, когда мы рассматриваем понятие *миксинов* (mixin). В их основу положена довольно простая идея: потребность в возможности расширить классы и объекты новыми функциональными возможностями, не прибегая к наследованию. С этой целью мы создаем ряд функций, присваиваем им имя, а затем определенным образом расширяем с их помощью класс или объект. И в этот момент создается новый класс или объект, сочетающий в себе функциональные возможности исходного класса и всех его миксинов. Зачастую такое расширение удается осуществить, даже если отсутствует доступ к исходному коду расширяемого класса.

Реализация и название этих средств расширения отличается в разных языках. И хотя они называются здесь *миксинами*, их лучше рассматривать как средства, не зависящие от конкретного языка. И самое главное, что все их реализации обладают способностью объединять *существующие* и *новые* функциональные возможности.

Вернемся снова к примеру класса `AccountRecord`. В том виде, в каком мы оставили этот класс, ему нужно было знать как об учетных записях, так и о каркасе хранения. Ему необходимо было также делегировать все методы на уровне хранения, которые требовалось открыть внешнему миру.

Миксины предоставляют альтернативу. Так, сначала мы могли бы написать миксин, реализующий, например, два или три стандартных метода поиска, а затем добавить его в класс AccountRecord. По мере написания новых классов для сохраняемых элементов их также можно дополнять миксинами, как показано ниже.

```
mixin CommonFinders {
  def find(id) { ... }
  def findAll() { ... }
end

class AccountRecord extends BasicRecord with CommonFinders
class OrderRecord   extends BasicRecord with CommonFinders
```

Можно пойти еще дальше. Всем разработчикам, например, известно, что бизнес-объектам требуется код проверки достоверности, чтобы исключить проникновение вредных данных в выполняемые вычисления. Но что именно имеется в виду под *проверкой достоверности*? Например, для учетной записи имеются, вероятно, самые разные уровни проверки достоверности, которые можно было бы применить в следующих целях:

- совпадение хешированного пароля с тем, который ввел пользователь;
- проверка правильности данных, введенных пользователем в форму при создании учетной записи;
- проверка правильности данных, введенных администратором, обновляющим сведения о пользователе;
- проверка правильности данных, введенных в учетную запись другими компонентами системы;
- проверка данных на согласованность перед сохранением.

Типичный и, на наш взгляд, неидеальный подход состоит в том, чтобы увязать все проверки на достоверность в единый класс (бизнес-объект/объект сохраняемости), а затем ввести признаки, чтобы управлять выполнением отдельных проверок в конкретных обстоятельствах.

Лучший, на наш взгляд, способ состоит в применении миксинов для создания специализированных классов, подходящих для конкретных случаев, как показано ниже, где в производные классы включаются проверки достоверности, общие для всех объектов учетных записей.

```
class AccountForCustomer extends Account
     with AccountValidations,AccountCustomerValidations

class AccountForAdmin extends Account
     with AccountValidations,AccountAdminValidations
```

Клиентский вариант класса включает для себя также проверки достоверности, пригодные для API, взаимодействующих с клиентом, тогда как административный вариант класса, вероятно, должен содержать (возможно, менее ограничивающие) проверки достоверности учетных данных администратора.

Теперь путем передачи экземпляров класса `AccountForCustomer` или `AccountForAdmin` наш код *автоматически* обеспечивает правильное применение проверок достоверности.

Совет 54 Пользуйтесь миксинами для совместного использования функциональных возможностей

Наследование редко является ответом

В этом разделе был сделан краткий анализ трех следующих альтернатив традиционному наследованию классов:

- интерфейсов и протоколов;
- делегирования;
- миксинов и свойств.

Каждая из этих методик может подойти вам лучше в тех или иных обстоятельствах, в зависимости от того, преследуете ли вы цель обмениваться информацией о типах данных, внедрять дополнительные функциональные возможности или пользоваться общими методами. Как и везде в программировании, задача заключается в том, чтобы пользоваться той методикой, которая лучше всего выражает ваши намерения.

И постарайтесь вместе с бананом не прихватить заодно целые джунгли.

Другие разделы, связанные с данной темой

- **Тема 8**. Сущность качественного проектирования, **глава 2** "Прагматичный подход".
- **Тема 10**. Ортогональность, **глава 2** "Прагматичный подход".
- **Тема 28**. Развязывание.

Задачи

- В следующий раз, когда вам придется создавать подклассы, уделите время изучению разных альтернатив наследованию. Сможете ли вы достичь желаемой цели с помощью интерфейсов, делегирования и/или миксинов? Удастся ли вам подобным образом уменьшить степень связывания?

Тема 32 Конфигурирование

"Уделяйте всем вашим вещам свои места, а каждой части вашего дела — свое время".

Бенджамин Франклин, автобиография
***Тринадцать добродетелей** (Thirteen Virtues)*

Если прикладной код полагается на значения, которые могут измениться после запуска приложения, храните эти значения вне данного приложения. И если приложение выполняется в разных средах и потенциально работает для разных клиентов, также храните значения для разных сред и клиентов вне данного приложения. Подобным образом вы параметризируете свое приложение, исходный код которого приспосабливается к месту его выполнения.

Совет 55 Параметризируйте свое приложение, используя внешнюю конфигурацию

В данные конфигурации, вероятнее всего, придется включить разнообразные общие сведения.

- Учетные данные для внешних служб (баз данных, сторонних API и т.д.)
- Уровни и места назначения протоколирования.
- Номера портов, IP-адреса, имена вычислительных машин и кластеров, используемые в приложении.
- Параметры проверки достоверности, характерные для конкретной среды.
- Внешне устанавливаемые параметры, например, ставки налогообложения.
- Особенности форматирования для разных сайтов.
- Лицензионные ключи.

Словом, найдите все, что может измениться или быть выражено за пределами основного кода, и вынесите эту информацию в отдельную конфигурационную "корзину".

Статическая конфигурация

Конфигурация многих каркасов и специализированных приложений хранится в обычных текстовых файлах или таблицах базы данных. Если конфигурационные данные хранятся в обычном текстовом файле, то для этой цели, как правило, применяется один из доступных текстовых форматов. В настоящее время для

этой цели чаще всего применяются такие текстовые форматы, как YAML и JSON. Иногда в приложениях, написанных на языках сценариев, применяются файлы исходного кода, специально предназначенные для хранения одной лишь конфигурации. Если подобная информация структурирована и, вероятно, будет изменяться клиентом (например, ставки налога с продаж), ее лучше хранить в таблице базы данных. И, разумеется, вы вольны пользоваться обоими способами, разделяя конфигурационную информацию в соответствии с ее назначением.

Независимо от формы хранения конфигурационной информации она вводится в приложение в виде структур данных — как правило, при запуске выполнения приложения. Эта структура данных обычно делается глобальной, чтобы упростить доступ к хранящимся в ней значениям из любой части прикладного кода.

Однако мы не рекомендуем вам поступать таким образом. Вместо этого вынесите всю конфигурационную информацию в отдельный (тонкий) API. Этим вы развяжете свой код от подробностей представления его конфигурации.

Конфигурация как служба

Несмотря на распространенность статического конфигурирования, в настоящее время мы отдаем предпочтение другому подходу. Нам по-прежнему приходится выносить конфигурационные данные за пределы приложения, но вместо того чтобы хранить их в текстовом файле или базе данных, мы предпочитаем получать их через API отдельной службы. У такого подхода к конфигурированию имеется целый ряд преимуществ, перечисленных ниже.

- Многие приложения могут обмениваться конфигурационной информацией, при этом с помощью аутентификации можно ограничивать доступ, определяя, какую информацию может видеть каждое из них.
- Изменения конфигурации могут выполняться глобально.
- Конфигурационные данные можно сопровождать с использованием специализированного пользовательского интерфейса.
- Конфигурационные данные становятся динамическими.

Последнее из перечисленных выше преимуществ, предполагающее динамический характер конфигурационных данных, имеет решающее значение для перехода к приложениям с высокой степенью доступности. Необходимость остановки и перезапуска приложения для изменения единственного параметра конфигурации безнадежно оторвана от современной действительности. При использовании службы конфигурирования компоненты приложения могут зарегистрироваться для получения уведомлений об обновлении используемых ими параметров, и тогда служба могла бы посылать им сообщения, содержащие новые значения — когда и если они изменятся.

Какую бы форму ни принимали конфигурационные данные, именно они определяют поведение приложения во время выполнения. Перестраивать код приложения, когда конфигурационные значения изменяются, нет необходимости.

Не пишите морально устаревший код

В отсутствие внешней конфигурации прикладной код перестает быть настолько приспосабливающимся и гибким, насколько он мог бы им быть. Плохо ли это? Если обратиться к примеру из природы, то можно заметить, что в ней вымирают те виды, которые не могут приспособиться к реальным условиям существования.

Так, птице дронту так и не удалось приспособиться к присутствию людей и домашнего скота в местах ее обитания на острове Маврикий, и поэтому этот вид быстро вымер[12]. И это был первый задокументированный случай вымирания вида от рук человека. Поэтому не позволяйте своему проекту (или карьере) пойти по пути дронта[13].

Другие разделы, связанные с данной темой

- **Тема 9**. DRY — пороки дублирования, **глава 2** "Прагматичный подход".
- **Тема 14**. Предметно-ориентированные языки, **глава 2** "Прагматичный подход".
- **Тема 16**. Сила простого текста, **глава 3** "Основные инструментальные средства".
- **Тема 28**. Развязывание.

---

[12] И нисколько не улучшило положение то обстоятельство, что поселенцы забивали дубинками безмятежных (а по существу, глупых) птиц ради развлечения.

[13] Изображение дронта взято из серии OpenClipart-Vectors, доступной на веб-сайте Pixabay

Не переусердствуйте

В первом издании данной книги мы предлагали пользоваться конфигурацией вместо кода подобным же образом. Но мы, очевидно, должны были точнее давать свои рекомендации. Ведь любой совет можно довести до крайности или воспользоваться им несообразно. Поэтому ниже даются некоторые предостережения по данному поводу.

Не переусердствуйте. Один из наших прежних клиентов решил, что буквально каждое поле в его приложении должно быть конфигурируемым. В итоге на внесения даже самых мелких изменений уходили целые недели, поскольку приходилось реализовывать как само поле, так и весь административный код для сохранения и редактирования его содержимого. А ведь в этом приложении насчитывалось до 40 тысяч переменных конфигурации, что превращало кодирование в сущий кошмар для разработчиков.

Не принимайте решения относительно конфигурирования из лени. Если возникает реальная дискуссия о том, как должно действовать какое-то функциональное средство и следует ли дать пользователю возможность выбирать режим его работы, попробуйте реализовать один из его вариантов и получить отзывы о нем, чтобы выяснить, насколько верным оказалось данное решение.

Глава 6

Параллельность

Пока мы все находимся на одной и той же странице, а не читаем главу параллельно, начнем с некоторых определений. *Параллельность* (concurrency) означает такое такое поведение двух или большего количества фрагментов кода, *как будто* они выполняются одновременно. А *параллелизм* (parallelism) означает, что они *действительно* выполняются одновременно. Чтобы добиться параллельности, необходимо выполнять код в среде, способной переключать выполнение между разными частями данного кода. Зачастую параллельный режим работы реализуется с помощью таких средств, как нити (fiber), потоки выполнения (thread) и процессы (process).

Для того чтобы добиться параллелизма, требуется оборудование, способное выполнять несколько операций одновременно. Это могут быть многоядерные процессоры, несколько процессоров на одном компьютере или несколько связанных между собой компьютеров.

Все параллельно

Практически невозможно написать код системы приличных масштабов, не прибегая к параллельности. Аспекты используемой параллельности могут быть явно выраженными или скрытыми в библиотеке. Параллельность необходима тогда, когда приложение должно взаимодействовать с реальным миром, где события происходят асинхронно: пользователи взаимодействуют с приложением, извлекаются данные, вызываются внешние службы, и все это происходит одновременно. Если заставить все это выполняться последовательно, чтобы события наступали поочередно, то такая система будет работать медленно и не позволит выгодно использовать истинный потенциал оборудования, на котором она работает. Поэтому в этой главе будут подробно рассмотрены понятия параллельности и параллелизма.

Разработчики нередко рассуждают о связывании, возникающем между фрагментами кода. При этом они ссылаются на зависимости и выясняют, насколько они затрудняют внесение изменений в исходный код. Но ведь имеется и другая форма связывания. В частности, *временное связывание* возникает в том случае,

когда прикладной код накладывает ограничение на последовательность выполнения действий, требуемых для решения текущей задачи. Может ли, например, ваш код зависеть от того, что "тик" происходит раньше "так"? Если вы хотите сохранить его гибкость — он не должен демонстрировать такую зависимость. Следует ли вашему коду обращаться к нескольким серверным службам последовательно, т.е. к одной за другой? Нет, не следует, если вы хотите сохранить своих клиентов. Поэтому в разделе "Разрывание временного связывания" будут рассмотрены способы выявления временного связывания подобного рода.

Почему так трудно писать параллельный код? Объясняется это, в частности, тем, что мы учились программировать с использованием последовательных систем. Применяемые нами языки обладают средствами, которые надежны, когда они используются последовательно, но становятся ненадежны, как только два события могут наступить одновременно. Одним из главных виновников тому служит *общее, или совместно используемое состояние*, которое означает не только глобальные переменные. Такое состояние возникает в любой момент времени, когда два или несколько фрагментов кода содержат ссылки на один и тот же фрагмент изменяемых данных. Но *общее состояние является неверным состоянием*. В соответствующем разделе описывается ряд приемов, позволяющих обойти общее состояние, хотя все они не вполне надежны и непогрешимы.

Если все сказанное выше опечалило вас, не отчаивайтесь! Ведь имеются способы построения параллельных приложений и получше. К их числу относится применение *модели акторов*, где независимые процессы обмениваются данными по каналам, используя вполне определенную простую семантику. Теория и практика такого подхода рассматривается в разделе "Акторы и процессы".

И в завершение этой главы будут рассмотрены *классные доски*, т.е. системы, действующие как комбинация хранилища объектов и интеллектуального брокера "издатель–подписчик". В своей исходной форме они так и не нашли широкого распространения, но в настоящее время наблюдается постепенно растущее количество реализаций на промежуточном уровне программного обеспечения с семантикой, похожей на классные доски. При правильном использовании такие системы обеспечивают серьезную степень развязывания.

Когда-то параллельно выполняющийся код считался экзотикой. Сегодня же он востребован, как никогда ранее.

Тема 33 Разрывание временного связывания

В связи с темой этого раздела невольно возникает вопрос: "А что такое *временное связывание*?" Это связывание, связанное со временем (простите за каламбур).

Время — очень часто игнорируемый аспект в архитектурах программного обеспечения. Единственным временем, которое заботит разработчиков, явля-

ется срок выполнения работ по графику, т.е. время, оставшееся до выпуска. Но здесь речь идет не о самом времени, а о его роли в качестве элемента проектирования самого программного обеспечения. Для нас важны следующие аспекты времени: параллельность (события могут происходить одновременно) и упорядочение (относительное расположение событий во времени).

Обычно мы подходим к программированию, не принимая во внимание ни один из упомянутых выше аспектов. Когда разработчики приступают к проектированию архитектуры или написанию программы, весь этот процесс происходит в линейном порядке. Ведь большинство людей мыслят именно в таком порядке: сделать сначала это, а затем то. Такой порядок мышления приводит к *временному связыванию*, т.е. к такому, которое возникает во времени. Так, метод `A` должен всегда вызываться прежде метода `B`; одновременно может быть составлен лишь один отчет; щелкнув на экранной кнопке, нужно подождать до тех пор, пока не будет перерисовано выводимое на экран графическое изображение; "тик" хронологически должен предшествовать "таку".

Такой подход негибок и не очень реалистичен.

Необходимо допустить параллельность и подумать о развязывании в любой момент времени или упорядочить зависимости. Поступая подобным образом, можно добиться нужной гибкости и снизить любые временны*е* зависимости на многих стадиях разработки: анализа последовательности выполняемых действий, построения архитектуры, проектирования и развертывания. В итоге получаются системы, которые легче осмысливать и которые потенциально быстрее реагируют и более надежны.

В поисках параллельности

Во многих проектах требуется моделировать и анализировать последовательность выполняемых действий в приложении как часть проектирования. При этом нам хотелось бы выяснить, что *может* произойти одновременно и что *должно* произойти в строгом порядке. С этой целью можно, например, зафиксировать последовательность выполняемых действий, используя такое обозначение, как *диаграмма действий*[1].

Совет 56 Анализируйте последовательность выполняемых действий с целью повышения параллельности

[1] Несмотря на то что язык UML постепенно пришел в упадок, многие из его отдельных диаграмм по-прежнему существуют в той или иной форме, включая и очень полезную *диаграмму действий*. Подробнее о типах диаграмм UML см. в книге *UML Distilled: A Brief Guide to the Standard Object Modeling Language* [Fow04].

Диаграмма действий состоит из ряда действий, отображаемых прямоугольниками со скругленными углами. Стрелка, проведенная от одного действия, ведет либо к другому действию (которое может быть начато, как только завершится первое действие), либо же к толстой линии, называемой *чертой синхронизации*. И как только завершатся *все* действия, приводящие к черте синхронизации, построение диаграммы можно продолжить, проводя стрелки от этой черты дальше. Действие без ведущих к нему стрелок может быть начато в любой момент.

С помощью диаграмм действий можно довести параллелизм до максимума, выявляя те действия, которые *могут быть* выполнены параллельно, но таким образом не выполняются. Допустим, разрабатывается программное обеспечение роботизированного аппарата для приготовления коктейлей "пинаколада" из светлого рома, кокосового молока и ананасового сока. С этой целью разработчики получили следующее поэтапное описание процедуры приготовления таких коктейлей.

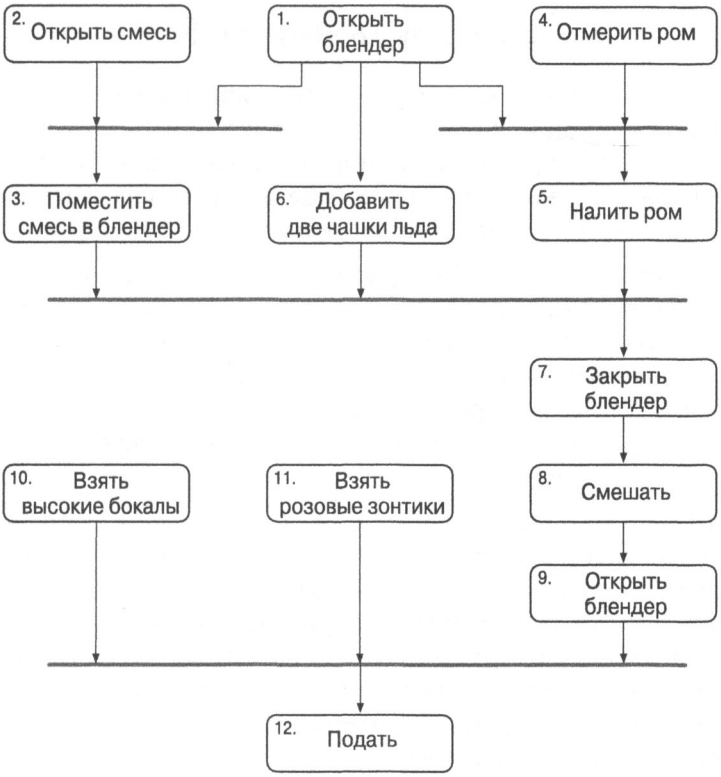
1. Открыть блендер
2. Открыть смесь "пинаколада"
3. Поместить смесь в блендер
4. Отмерить полстакана светлого рома
5. Налить ром
6. Добавить 2 чашки льда
7. Закрыть блендер
8. Взбалтывать в течение 1 минуты
9. Открыть блендер
10. Взять высокие бокалы
11. Взять розовые зонтики
12. Подать

Но бармен потеряет свою работу, если последует этой процедуре поэтапно. Несмотря на то что действия по приготовлению коктейля описаны последовательно, многие из них могут быть выполнены параллельно. Воспользуемся приведенной ниже диаграммой действий, чтобы зафиксировать и обдумать потенциальную параллельность.

Выявление реально существующих зависимостей может стать полным откровением. В данном случае все задачи верхнего уровня (1, 2, 4, 10 и 11) могут быть (одновременно) выполнены в первую очередь, а во вторую очередь — задачи 3, 5 и 6 (также одновременно). Если бы вы принимали участие в конкурсе на лучшего составителя коктейля "пинаколада", подобные оптимизации данной процедуры могли бы оказать вам существенную помощь в достижении желанной цели.

Возможности для достижения параллельности

Диаграммы действий показывают потенциальные участки параллельности, но ничего не говорят о том, стоит ли использовать параллельность. Так, в рассмотренном выше примере приготовления "пинаколады" бармену потребовалось бы пять рук, чтобы выполнить все потенциально первоначальные задачи одновременно.



Именно здесь и наступает стадия проектирования. Глядя на упомянутые выше действия, мы осознаем, что на действие 8 (работа блендера) потребуется одна минута. В течение этого промежутка времени бармен может взять высокие бокалы и розовые зонтики (действия 10 и 11), и, возможно, у него еще останется время, чтобы обслужить другого клиента.

Именно такие моменты мы и ищем, когда выполняем проектирование с учетом параллельности. При этом мы надеемся найти действия, отнимающие время, но не время для выполнения нашего кода. Запрос базы данных, доступ к внешней службе, ожидание ввода пользователем данных — все эти действия, как правило, приостанавливают выполнение программы до тех пор, пока они не завершатся. И все они открывают возможности сделать программу более производительной, не заставляя процессор простаивать в холостую.

Возможности для достижения параллелизма

Напомним главное отличие: параллельность (concurrency) — это программный механизм, а параллелизм (parallelism) — предмет заботы оборудования. Если в нашем распоряжении имеется несколько процессоров, доступных локально или дистанционно, то, распределив работу между ними, мы можем сократить общее время ее выполнения.

Ускоренное форматирование

Эта книга написана простым текстом. Чтобы составить ее версию для печати, электронной книги или иных целей, ее текст был пропущен через конвейер процессоров. Одни из них находили конкретные конструкции (ссылки на литературу, статьи предметного указателя, специальную разметку для советов и т.д.), а другие оперировали рукописью как документом в целом.

Многие процессоры в конвейере должны получать доступ к внешней информации (читать и записывать текст в файлы, обращаться к внешним программам по конвейеру). Вся эта относительно медленно выполняемая работа дает нам возможность поставить себе на службу параллельность: фактически каждый этап конвейера выполняется параллельно, получая результаты работы предыдущего этапа и передавая другие результаты следующему этапу работы конвейера.

Кроме того, на некоторых стадиях процесса требуется относительно интенсивное употребление процессоров. К их числу относится преобразование математических формул. По разным историческим причинам для преобразования каждого уравнения может потребоваться до 500 мс. Чтобы ускорить дело, можно с выгодой воспользоваться параллелизмом. Каждая формула не зависит от других формул, поэтому ее можно преобразовывать параллельно в отдельном процессе, собирая полученные результаты обратно в книгу, как только они становятся доступными.

В итоге составление книги на компьютере с многоядерным процессором намного ускоряется. И, разумеется, по ходу дела выявили целый ряд ошибок параллельности в составленном нами конвейере…

В идеальном случае разделяемые подобным образом части работы являются независимыми, т.е. каждая из них может быть продолжена, ничего не ожидая от других. Типичный шаблон разделяет крупную часть работы на независимые фрагменты, которые обрабатываются параллельно, а затем полученные результаты объединяются.

Интересным практическим примером может служить принцип действия компилятора в языке Elixir. После запуска этот компилятор разделяет проект, который он строит, на модули, компилируя их параллельно. Иногда один модуль зависит от другого, и тогда компиляция приостанавливается до тех пор, пока результаты построения одного модуля не станут доступными для построения другого модуля. Завершение построения модуля верхнего уровня означает, что все его зависимости скомпилированы. В итоге процесс компиляции заметно ускоряется благодаря тому, что задействованы все имеющиеся ядра процессора.

Выявить возможности проще всего

Вернемся, однако, к приложениям. Мы выявили в них места, где можно извлечь выгоду из параллельности и параллелизма. Теперь предстоит самое трудное: решить, как все это надежно реализовать. Данному вопросу и посвящена остальная часть этой главы.

Другие разделы, связанные с данной темой

- **Тема 10**. Ортогональность, **глава 2** "Прагматичный подход".
- **Тема 26**. Как сбалансировать ресурсы, **глава 4** "Прагматичная паранойя".
- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 36**. Классные доски.

Задачи

- Сколько задач вы решаете параллельно, собираясь утром на работу? Не могли бы вы выразить эту процедуру в виде диаграммы действий на языке UML? Можете ли вы найти способ собраться на работу быстрее, увеличив степень параллельности своих действий?

Тема 34 Общее состояние — неверное состояние

Представьте себя за обедом в своем любимом ресторане. Покончив с основным блюдом, вы спрашиваете официанта, остался ли еще яблочный пирог. Глядя через плечо, он замечает один кусок на прилавке-витрине и отвечает утвердительно. Вы делаете заказ и удовлетворенно вздыхаете.

В это время на другой стороне ресторана еще один посетитель задает официанту тот же самый вопрос. Он также смотрит на прилавок-витрину, убеждается в наличии куска пирога, отвечает утвердительно, и тогда другой посетитель делает аналогичный заказ. В итоге один из посетителей ресторана остается разочарованным.

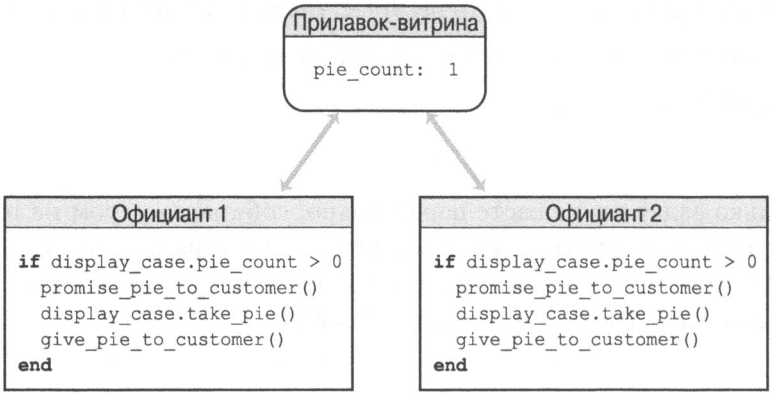
Замените прилавок-витрину совместным банковским счетом, а обслуживающий персонал ресторана — кассовыми терминалами и представьте, что вы и ваш супруг или супруга решили одновременно купить новый телефон, но денег на счете у вас хватит лишь на один. В итоге кто-то (банк, магазин или вы) останетесь весьма недовольны.

Совет 57 Общее состояние — неверное состояние

Все дело в совместно используемом общем состоянии. Каждый из официантов посмотрел на прилавок-витрину в ресторане, но не друг на друга. И каждый кассовый терминал проанализировал остаток на счете безотносительно к другому кассовому терминалу.

Неатомарные обновления

Рассмотрим пример ситуации в ресторане. Ниже показано, как реализовать его непосредственно в коде.



Оба официанта действуют одновременно, а на практике — параллельно. Рассмотрим код, реализующий их поведение:

```
if display_case.pie_count > 0
    promise_pie_to_customer()
    display_case.take_pie()
    give_pie_to_customer()
end
```

В этом коде первый официант получает текущий подсчет количества кусков пирога и обнаруживает, что он равен единице. И тогда он обещает пирог своему клиенту. Но в тот же самый момент действует и второй официант. Он также обнаруживает, что текущий подсчет количества кусков пирога равен единице, и обещает тот же самый пирог своему клиенту. В итоге один из двух официантов берет оставшийся кусок пирога, тогда как другой официант переходит в некоторого рода состояние ошибки, что, вероятнее всего, вызовет немало извинений.

Все дело не в том, что два процесса могут записывать данные в одну и ту же область памяти, а в том, что ни один из них не может гарантировать, что его представление о данной области памяти согласовано с другими. По существу, когда код, реализующий поведение официанта, выполняет метод `display_case.pie_count()`, он копирует значение, определяющее количество кусков пирога, из объекта `display_case` в свою область памяти. Если же

это значение в объекте `display_case` изменится, содержимое данной области памяти, которая используется для принятия решений, окажется неактуальным.

Все это происходит потому, что извлечение и обновление подсчета кусков пирога не является атомарной операцией, поскольку базовое значение может измениться посредине данной операции. Так как же сделать ее атомарной?

Семафоры и другие формы взаимного исключения

Семафор — это просто ресурс, которым может одновременно владеть лишь кто-то один. Семафор можно создать, а затем воспользоваться им для управления доступом к какому-нибудь другому ресурсу. В рассматриваемом здесь примере можно было бы создать семафор для управления доступом к прилавку с пирогом, приняв условие, что всякий, кому требуется обновить содержимое прилавка, может сделать это лишь в том случае, если он владеет данным семафором.

Допустим, проблема пирога в ресторане решается с помощью такого семафора. С этой целью на прилавок-витрину поставили пластмассовую фигурку лепрекона[2]. Прежде чем подать пирог, официант должен взять в руку фигурку лепрекона. Как только он выполнит заказ (что означает доставку пирога на стол посетителя), он может возвратить фигурку лепрекона на место, где тот охраняет пирожные сокровища и готов содействовать следующему заказу.

Теперь выясним, как реализовать такой семафор непосредственно в коде. По классической традиции операция захвата семафора называлась *P*, а операция его освобождения — *V*[3]. Сейчас для этой цели употребляются такие термины, как *блокировка* и *разблокировка*, *захват* и *освобождение* и т.д. Так, в приведенном ниже коде предполагается, что семафор уже создан и хранится в переменной `case_semaphore`.

```
case_semaphore.lock()

if display_case.pie_count > 0
    promise_pie_to_customer()
    display_case.take_pie()
    give_pie_to_customer()
end

case_semaphore.unlock()
```

---

[2] Лепрекон — персонаж ирландского фольклора, волшебник, исполняющий желания и традиционно изображаемый в виде небольшого коренастого бородатого человечка. — *Примеч. пер.*

[3] Названия операций P и V являются производными от начальных букв двух голландских слов, хотя по этому поводу все еще ведется полемика. Изобретатель данного метода, Эдсгер Дейкстра (Edsger Dijkstra), предложил два слова *passering* (передать) и *prolaag* (попытаться понизить уровень) для названия операции P и слово *vrijgave* (снять запрет) и, возможно, слово *verhogen* (повысить уровень) для названия операции V.

Допустим, фрагменты кода, реализующие действия обоих официантов, выполняются одновременно. Они оба пытаются заблокировать семафор, но удается это лишь одному из них. Тот код, который захватывает семафор, продолжает выполняться как обычно, а другой код приостанавливает свое выполнение до тех пор, пока семафор не освободится (т.е. второй официант ожидает своей очереди выполнить заказ). Как только первый официант завершит выполнение своего заказа, он разблокирует семафор, и тогда второй официант продолжит свою работу. Теперь он видит наличие пирога на прилавке-витрине и приносит свои извинения посетителям, попросившим пирог.

Такой подход к разрешению конфликтов при одновременном выполнении операций вызывает некоторые затруднения. И самое, вероятно, существенное из них состоит в том, что такой подход оказывается работоспособным лишь в том случае, если все, кто имеет доступ к прилавку-витрине, соглашаются на условие пользоваться семафором. Если кто-нибудь из них забудет о нем, т.е. если какой-то разработчик напишет код, не соблюдающий данное условие, то снова возникнет беспорядок.

Придание ресурсам транзакционного характера

Рассматриваемое здесь проектное решение оказывается неудачным потому, что ответственность за защиту доступа к прилавку с пирогом возлагается на тех, кто им пользуется. Изменим это решение таким образом, чтобы централизовать управление доступом к прилавку с пирогом. Для этого нам придется внести в API такие изменения, которые позволят официантам проверять подсчет количества кусков пирога и нарезать пирог в течение одного вызова, как показано ниже.

```
slice = display_case.get_pie_if_available()
if slice
    give_pie_to_customer()
end
```

Чтобы сделать приведенный выше код работоспособным, необходимо написать приведенный ниже метод, частично выполняющий функции самого прилавка.

```
def get_pie_if_available()        ####
  if @slices.size > 0                #
    update_sales_data(:pie)          #
    return @slices.shift             #
  else                               # неверный код!
    false                            #
  end                                #
end                                  ####
```

Данный код наглядно показывает типичное заблуждение.

Несмотря на то что доступ к ресурсу перемещен в центральное место, объявляемый метод все равно может быть вызван из нескольких одновременно выполняющихся потоков. Следовательно, он должен быть по-прежнему защищен семафором, как показано ниже.

```
def get_pie_if_available()
  @case_semaphore.lock()
  if @slices.size > 0
    update_sales_data(:pie)
    return @slices.shift
  else
    false
  end
  @case_semaphore.unlock()
end
```

Но даже этот код может оказаться неверным. Так, если в методе `update_sales_data()` будет возбуждено исключение, семафор так и не будет разблокирован, и весь последующий доступ к прилавку с пирогом зависнет на неопределенное время. Этот недостаток придется исправить следующим образом:

```
def get_pie_if_available()
  @case_semaphore.lock()

  try {
    if @slices.size > 0
      update_sales_data(:pie)
      return @slices.shift
    else
      false
    end
  }
  ensure {
    @case_semaphore.unlock()
  }
end
```

Данная ошибка настолько типична, что во многих языках предоставляются отдельные библиотеки, устраняющие ее автоматически, как показано ниже.

```
def get_pie_if_available()
  @case_semaphore.protect() {
    if @slices.size > 0
      update_sales_data(:pie)
      return @slices.shift
    else
      false
    end
  }
end
```

Множественные транзакции ресурсов

В упомянутом выше ресторане только что установили мороженицу. Если посетитель закажет яблочный пирог с мороженым, официанту придется проверить наличие не только пирога, но и мороженого. С этой целью код, реализующий действия официанта, можно было бы изменить следующим образом:

```
slice = display_case.get_pie_if_available()
scoop = freezer.get_ice_cream_if_available()

if slice && scoop
  give_order_to_customer()
end
```

Но так не годится. Что, если посетитель потребует кусок пирога, и когда официант попытается достать шарик мороженого, то его не окажется в наличии? В итоге он останется держать в руках кусок пирога, не зная, что с ним делать, поскольку посетителю следует принести яблочный пирог непременно с мороженым. И тот факт, что официант держит в руках пирог, означает, что его уже нет на прилавке, а следовательно, его не достанется какому-нибудь другому посетителю, из принципа не желающего есть яблочный пирог с мороженым.

Это положение можно было бы исправить, введя в объект, представляющий прилавок с пирогом, метод, позволяющий возвратить кусок пирога. Чтобы не удерживать используемые ресурсы в том случае, если что-нибудь пойдет не так, как предполагалось, в данный метод следует ввести обработку исключений:

```
slice = display_case.get_pie_if_available()

if slice
  try {
    scoop = freezer.get_ice_cream_if_available()
    if scoop
    try {
      give_order_to_customer()
    }
    rescue {
      freezer.give_back(scoop)
    }
    end
  }
  rescue {
    display_case.give_back(slice)
  }
end
```

Но и такое решение отнюдь не идеально. Теперь код выглядит весьма уродливо. Выяснить, что он действительно делает, не так-то просто, поскольку бизнес-логика скрывается в нем за служебными операциями.

Ранее подобный недостаток был устранен перенесением кода, обращающегося с ресурсом, в сам ресурс. Но здесь задействованы два ресурса. Следует ли перенести код в реализацию прилавка с пирогом или мороженицы? Мы считаем, что не следует ни в одно из этих мест. Прагматичный подход подсказывает, что яблочный пирог с мороженым следует рассматривать как отдельный ресурс. Поэтому код лучше перенести в новый модуль, и если посетитель ресторана закажет такой десерт, то выполнить его заказ либо удастся, либо не удастся.

Разумеется, в меню настоящего ресторана может быть немало составных блюд, подобных упомянутому выше, так что вряд ли стоит писать отдельный модуль для каждого из них. Вместо этого, вероятнее всего, придется сделать в каждом составном блюде из меню ссылки на его составляющие, а затем объявить обобщенный метод `get_menu_item()`, выделяющий ресурс по каждой ссылке.

Обновления без транзакций

Общей памяти как источнику осложнений, возникающих в связи с параллельностью, уделяется немало внимания, но на самом деле эти осложнения могут возникнуть *везде*, где прикладной код разделяет общие изменяемые ресурсы: файлы, базы данных, внешние службы и т.д. Всякий раз, когда два или более фрагмента прикладного кода могут одновременно получить доступ к некоторому ресурсу, следует усматривать потенциальное осложнение.

Иногда ресурс не вполне очевиден. Работая над настоящим изданием данной книги, мы обновили набор инструментальных средств, чтобы выполнить больше работы в параллельном режиме с помощью потоков выполнения. Это привело к тому, что сборка завершалась неудачно, но странными способами и в произвольных местах. Общим для всех ошибок оказалось то, что файлы или каталоги не удавалось найти, несмотря на то, что они находились именно там, где им и следовало быть.

Мы проследили эту тенденцию до двух мест в своем коде, где временно изменялся текущий каталог. В нераспараллеленной версии кода обратное восстановление каталога было вполне приемлемым, но в распараллеленной версии каталог изменялся в одном потоке выполнения, а затем начинал выполняться другой поток в том же самом каталоге. В этом потоке предполагалось выполнение в первоначальном каталоге, но этого не происходило, поскольку текущий каталог совместно использовался в обоих потоках выполнения.

Характер подобной ошибки подсказывает еще один совет:

> **Совет 58** Случайные отказы часто вызваны осложнениями, возникающими в связи с параллельностью

Другие виды исключительного доступа

В большинстве языков программирования имеются библиотеки, поддерживающие некоторые виды исключительного доступа к общим ресурсам. Они могут называться мьютексами[4], мониторами или семафорами. Все эти средства поддержки параллельности реализуются в виде библиотек.

Но в некоторые языки поддержка параллельности встроена непосредственно. Например, в языке Rust соблюдается принцип владения данными, в соответствии с которым лишь одна переменная или параметр может одновременно хранить ссылку на любой конкретный фрагмент изменяемых данных.

Можно, конечно, утверждать, что реализовать параллельность в языках функционального программирования проще, поскольку им присуще делать все данные неизменяемыми. Но и в них возникают те же самые трудности, поскольку в какой-то момент им все равно приходится вступать в реальный изменчивый мир.

Доктор, мне больно...

Если вы не вынесете для себя ничего полезного из этого раздела, примите во внимание хотя бы следующее: добиться параллельности в среде с общими ресурсами нелегко, и попытки сделать это самостоятельно сопряжены с серьезными трудностями.

Именно поэтому мы решили завершить этот раздел старой шуткой:

— *Доктор, мне больно, когда я делаю вот так...*

— *Ну так не делайте вот так.*

В двух последующих разделах рассматриваются альтернативные способы безболезненного извлечения выгод из параллельности.

Другие разделы, связанные с данной темой

- **Тема 10**. Ортогональность, **глава 2** "Прагматичный подход".
- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 38**. Программирование по совпадению, **глава 7** "По ходу кодирования".

[4] От анг. *mutual exclusion* — взаимное исключение. — *Примеч. пер.*

Тема 35 Акторы и процессы

"Без писателей никто не напишет истории,
а без актеров никто не воплотит истории в жизнь".

Анджи-Мари Дельсанте[5]

Акторы и процессы предоставляют интересные способы реализации параллельности, избавляя от бремени синхронизации доступа к общей памяти. Но, прежде чем перейти к их рассмотрению, необходимо определить их назначение. И хотя это определение покажется вам несколько академичным, примите его безбоязненно, а мы подробно растолкуем его вскоре.

- *Актор* (actor) — это независимый виртуальный процессор с собственным локальным (или закрытым) состоянием. У каждого актора имеется свой почтовый ящик. Как только в этом ящике появится сообщение, простаивающий в ожидании актор активизируется и обработает его. Завершив его обработку, актор либо переходит к обработке другого сообщения в своем почтовом ящике, либо возвращается в неактивное состояние ожидания, если почтовый ящик пуст. По ходу обработки сообщений актор может создавать других акторов, посылать сообщения другим известным ему акторам или переходить в новое состояние, которое станет текущим после обработки следующего сообщения.
- *Процесс* — это, как правило, более универсальный виртуальный процессор, зачастую реализуемый операционной системой в целях содействия параллельности. На процессы могут условно накладываться ограничения, чтобы они вели себя как акторы. Именно такой тип процесса имеется здесь в виду.

Акторы могут быть только параллельными

В определении акторов имеется ряд перечисленных ниже положений.

- Единого контроля за всем происходящим *не* существует. Что будет выполнено следующим, никак не планируется, а передача информации от исходных данных к окончательным выводимым данным никак не регулируется.
- *Единственное* состояние, имеющееся в системе, содержится в сообщениях и локальном состоянии актора. Сообщения могут быть проанализированы только при прочтении их получателем, а локальное состояние за пределами актора недоступно.

[5] Angie-Marie Delsante — американская писательница в жанре научной фантастики, поборница целостного образа жизни. — *Примеч. пер.*

- Все сообщения являются однонаправленными, и ответы на них не предусмотрены. Если же требуется, чтобы актор возвратил ответ, в посылаемом сообщении следует указать адрес своего почтового ящика, и тогда актор (в конечном итоге) отправит ответ как обычное сообщение в указанный почтовый ящик.
- Актор поочередно полностью обрабатывает каждое сообщение.

Таким образом, акторы выполняются одновременно, асинхронно и не используют совместно ничего общего. Если бы в системе было достаточно физических процессоров, на каждом из них можно было бы выполнять отдельный актор. При наличии единственного процессора требуется соответствующая выполняющая среда, способная переключать контекст с одного актора на другой. Но в любом случае код, выполняемый акторами, остается одним и тем же.

Совет 59 Пользуйтесь акторами, чтобы достичь параллельности без общего состояния

Простой актор

Итак, реализуем рассмотренный ранее пример ситуации в ресторане, используя акторов. В данном случае у нас будут три актора: посетитель, официант и прилавок с пирогом. Общий поток сообщений будет выглядеть следующим образом.

- Мы (подобно некоей внешней богоподобной силе) внушаем посетителю, что он проголодался.
- В ответ посетитель заказывает официанту пирог.
- Официант обращается к прилавку, чтобы взять кусок пирога и принести его посетителю.
- Если на прилавке имеется кусок пирога, он будет отправлен посетителю, а официант уведомлен, что пирог следует внести в счет.
- Если же пирога на прилавке нет, официант будет уведомлен об этом, ему придется извиниться перед посетителем.

Мы решили реализовать описанный выше алгоритм на языке JavaScript, воспользовавшись библиотекой Nact[6]. С этой целью мы ввели в исходный код оболочку, позволяющую реализовать акторов в виде простых объектов, где ключи обозначают типы сообщений, принимаемых актором, а значения — функции, выполняемые при получении конкретного сообщения. (В большинстве систем акторов имеется аналогичная структура, хотя подробности ее реализации зависят от базового языка.)

[6] См. по адресу `https://github.com/ncthbrt/nact`.

Итак, начнем с посетителя. Он может принимать следующие сообщения.

- Он проголодался — это сообщение посылается из внешнего контекста.
- На столе стоит пирог — это сообщение посылается прилавком с пирогом.
- Приносятся извинения в связи с тем, что пирога больше не осталось — это сообщение посылается официантом.

Ниже показано, как все это реализуется непосредственно в коде.

concurrency/actors/index.js

```
const customerActor = {
      'hungry for pie': (msg, ctx, state) => {
        return dispatch(state.waiter,
               { type: "order", customer: ctx.self, wants: 'pie' })
      },
      'put on table': (msg, ctx, _state) =>
          console.log('${ctx.self.name} sees
                    "${msg.food}" appear on the table'),
      'no pie left': (_msg, ctx, _state) =>
         console.log('${ctx.self.name} sulks...')
}
```

Любопытная ситуация возникает в том случае, когда мы получаем сообщение о том, что посетитель проголодался и жаждет пирога, и посылаем сообщение официанту. (Ниже будет показано, откуда актору посетителя известно об акторе официанта.) Ниже приведен код, реализующий актора официанта.

concurrency/actors/index.js

```
const waiterActor = {
  "order": (msg, ctx, state) => {
    if (msg.wants == "pie") {
      dispatch(state.pieCase,
               { type: "get slice", customer: msg.customer,
                 waiter: ctx.self })
    }
    else {
      console.dir('Don't know how to order ${msg.wants}');
    }
  },

  "add to order": (msg, ctx) =>
     console.log('Waiter adds ${msg.food}
                 to ${msg.customer.name}'s order'),

  "error": (msg, ctx) => {
    dispatch(msg.customer, { type: 'no pie left', msg: msg.msg });
    console.log('\nThe waiter apologizes to
                ${msg.customer.name}: ${msg.msg}')
  }
};
```

Получив от посетителя сообщение `'order'` (заказ), официант проверяет, запрошен ли пирог. Если это именно так, он посылает запрос прилавку с пирогом, передавая ссылки как на самого себя, так и на посетителя.

Прилавок с пирогом находится в состоянии, содержащем массив всех находящихся на нем кусков пирога. (Ниже будет показано, как устанавливается это состояние.) Когда прилавок получает сообщение `'get slice'` (взять кусок пирога) от официанта, он выясняет, осталось ли сколько-нибудь кусков пирога. Если осталось, то он передает кусок пирога посетителю, сообщает официанту о необходимости обновить заказ и, наконец, возвращает состояние, содержащее на один кусок пирога меньше. Ниже показано, как все это реализуется непосредственно в коде.

concurrency/actors/index.js

```
const pieCaseActor = {
  'get slice': (msg, context, state) => {
    if (state.slices.length == 0) {
        dispatch(msg.waiter, { type: 'error', msg: "no pie left",
                               customer: msg.customer })
        return state
    }
    else {
      var slice = state.slices.shift() + " pie slice";
      dispatch(msg.customer, type: 'put on table', food: slice });
      dispatch(msg.waiter, { type: 'add to order', food: slice,
                             customer: msg.customer });
      return state;
    }
  }
}
```

Несмотря на то что одни акторы зачастую динамически запускаются другими акторами, в данном случае ради простоты это делается вручную. При этом каждому актору передается некоторое исходное состояние, как поясняется ниже.

- Прилавок получает первоначальный список находящихся на нем кусков пирога.
- Официанту передается ссылка на прилавок с пирогом.
- Посетителям передается ссылка на официанта.

concurrency/actors/index.js

```
const actorSystem = start();

let pieCase = start_actor(
    actorSystem,
    'pie-case',
    pieCaseActor,
    { slices: ["apple", "peach", "cherry"] });
```

```
let waiter = start_actor(
    actorSystem,
    'waiter',
    waiterActor,
    { pieCase: pieCase });

let c1 = start_actor(actorSystem, 'customer1',
                     customerActor, { waiter: waiter });

let c2 = start_actor(actorSystem, 'customer2',
                     customerActor, { waiter: waiter });
```

Наконец, запустим все описанное выше в действие. Посетители ресторана проголодались. Первый посетитель заказывает три куска пирога, а второй — два. Ниже показано, как это реализуется непосредственно в коде.

concurrency/actors/index.js

```
dispatch(c1, { type: 'hungry for pie', waiter: waiter });
dispatch(c2, { type: 'hungry for pie', waiter: waiter });
dispatch(c1, { type: 'hungry for pie', waiter: waiter });
dispatch(c2, { type: 'hungry for pie', waiter: waiter });
dispatch(c1, { type: 'hungry for pie', waiter: waiter });
sleep(500)
    .then(() => {
      stop(actorSystem);
    })
```

Выполнив приведенный выше код, можно увидеть, как акторы обмениваются сообщениями[7]. Хотя порядок их действий у вас может оказаться иным.

```
$ node index.js
customer1 sees "apple pie slice" appear on the table
customer2 sees "peach pie slice" appear on the table
Waiter adds apple pie slice to customer1's order
Waiter adds peach pie slice to customer2's order
customer1 sees "cherry pie slice" appear on the table
Waiter adds cherry pie slice to customer1's order

The waiter apologizes to customer1: no pie left
customer1 sulks...

The waiter apologizes to customer2: no pie left
customer2 sulks...[8]
```

[7] Чтобы выполнить этот код, вам потребуются также функции-оболочки, которые здесь не показаны. Их можно загрузить по адресу https://media.pragprog.com/titles/tpp20/code/concurrency/actors/index.js.

[8] Первый посетитель видит, как на его столе появляется "кусок яблочного пирога"
Второй посетитель видит, как на его столе появляется "кусок персикового пирога"
Официант вносит кусок яблочного пирога в заказ первого посетителя
Официант вносит кусок персикового пирога в заказ первого посетителя
Первый посетитель видит, как на его столе появляется "кусок вишневого пирога"

Отсутствие явной параллельности

В модели акторов отпадает всякая необходимость писать код, поддерживающий параллельность, поскольку отсутствует общее состояние. Нет также необходимости программировать явным образом логику, реализующую всю последовательность действий от начала и до конца, поскольку акторы выясняют эту последовательность сами, исходя из получаемых ими сообщений.

Здесь нет никакого упоминания о базовой архитектуре. Такой набор компонентов может быть в равной степени пригодным для однопроцессорных, многоядерных систем или множества подключенных к сети машин.

Erlang подготавливает почву

Язык Erlang и его выполняющая среда служат характерными примерами реализации акторов, несмотря на то, что изобретатели Erlang не читали оригинальную статью об акторах. И хотя акторы называются в языке Erlang *процессами*, тем не менее их нельзя считать обычными процессами операционной системы. Вместо этого процессы в Erlang, подобно рассмотренным выше акторам, легковесны (на одной машине можно выполнять их миллионами) и обмениваются отправляемыми сообщениями. Каждый такой процесс изолирован от остальных процессов, а следовательно, у них нет общего состояния.

В выполняющей среде языка Erlang реализуется также система контроля над сроками действия процессов, потенциально перезапускающая один процесс или ряд процессов в случае отказа. Кроме того, в языке Erlang поддерживается "горячая" загрузка кода, позволяющая заменять код в системе, не останавливая ее работу. Наконец, система языка Erlang выполняет едва ли не самый надежный в мире код с показателем безотказности порядка 99,9999999%.

Но язык Erlang и производный от него язык Elixir не единственны в своем роде, поскольку акторы реализованы и в большинстве других языков. Далее они будут рассмотрены с точки зрения реализации параллельности.

```
Официант вносит кусок вишневого пирога в заказ первого посетителя

Официант извиняется перед первым посетителем за то, что пирога больше не осталось
Первый посетитель сердится...

Официант извиняется перед вторым посетителем за то, что пирога больше не осталось
Второй посетитель сердится...
```

Другие разделы, связанные с данной темой

- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 30**. Преобразовательное программирование, **глава 5** "Гибкость или ломкость".
- **Тема 36**. Классные доски.

Задачи

- Имеется ли у вас в настоящее время такой код, в котором взаимное исключение применяется для защиты общих данных? Почему бы вам не опробовать прототип того же кода, написанного с помощью акторов?
- В коде, реализующем акторов для контроля над описанной выше ситуацией в ресторане, поддерживаются заказы кусков пирога. Расширьте этот код таким образом, чтобы дать посетителям ресторана возможность заказывать яблочный пирог с мороженым, а отдельным агентам — манипулировать кусками пирога и шариками мороженого. Устройте все так, чтобы обрабатывать ситуацию, когда израсходуется пирог или мороженое.

Тема 36 Классные доски

"И писали напротив лампады на извести чертога царского..."

Книга пророка Даниила, глава 5

Рассмотрим, каким образом детективы могли бы пользоваться *классной доской* для координирования действий в расследовании убийства. Главный инспектор устанавливает для начала крупную черную доску в зале совещаний и пишет на ней единственный вопрос:

Шалтай-Болтай (мужчина, яйцо): несчастный случай или убийство?

Действительно ли Шалтай упал сам или его толкнули? Каждый детектив может внести свой вклад в раскрытие тайны этого потенциального убийства, добавив факты, показания свидетелей, любые данные судебной экспертизы, которые могут появиться, и т.д. По мере накопления данных каждый детектив может заметить во всем этом определенную связь и написать на доске свои наблюдения или предположения. И этот процесс продолжается со всеми отклонениями и участием самых разных людей и посредников до тех пор, пока дело не будет закрыто. Пример классной доски для данного случая приведен на рис. 6.1.

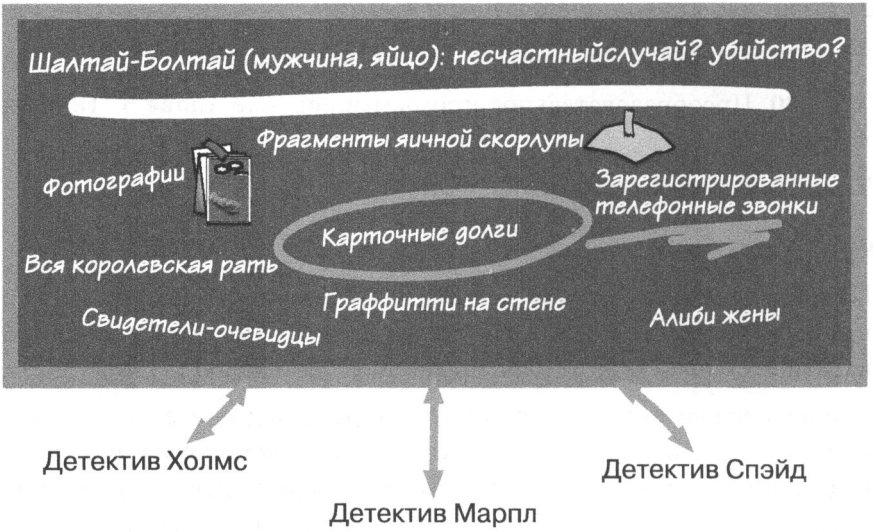


Рис. 6.1. Кто-то обнаружил связь между карточными долгами Шалтая-Болтая и зарегистрированными телефонными звонками. Вероятно, его могли запугивать по телефону

Ниже перечислены основные свойства метода классной доски.

- Ни одному из детективов совсем не обязательно знать о существовании любых других детективов. Они просто следят за новой информацией на классной доске и пишут на ней свои изыскания.
- Детективы могут иметь подготовку в самых разных дисциплинах, разные уровни образования и опыта, а возможно, и вообще не работать на одном и том же участке территории. Они разделяют общее желание раскрыть преступление, и только.
- В ходе данного процесса разные детективы могут приходить и уходить, и работать посменно.
- На информацию, размещаемую на классной доске, не накладывается никаких ограничений. Это могут быть фотографии, свидетельские показания, вещественные доказательства и т.д.

Классная доска служит в данном случае формой *пассивной* параллельности, где детективы являются независимыми процессами, агентами, акторами и т.д. Одни из них размещают добытые факты на классной доске, а другие берут эти факты с доски и, возможно, объединяют или обрабатывают их, а также указывают дополнительные сведения на доске. И постепенно доска помогает им прийти к общему заключению по поводу расследуемого преступления.

Автоматизируемые системы классных досок первоначально применялись в приложениях искусственного интеллекта, где решались крупные и сложные

задачи, включая распознавание речи, системы формирования рассуждений на основе базы данных и т.д. Одна из первых таких систем называлась Linda и была разработана Дэвидом Гелернтером. Факты хранились в ней в виде типизированных кортежей. Приложения могут записывать новые кортежи в систему Linda и запрашивать существующие кортежи, используя определенную форму сопоставления с шаблоном.

Позднее появились распределенные системы типа классных досок, например JavaSpaces и T Spaces. С помощью этих систем можно сохранять на классной доске не только данные, но и активные объекты Java, а также извлекать их по частично совпадающим полям (через шаблоны и метасимволы подстановки) или подтипам. Допустим, имеется тип данных `Author`, являющийся подтипом, производным от типа `Person`. На классной доске, содержащей объекты типа `Person`, можно производить поиск, используя шаблон типа `Author` со значением `Shakespeare` в поле `lastName`. В результате такого поиска будет обнаружен автор Уильям Шекспир, но не садовник Фред Шекспир. Мы считаем, что такие системы не нашли широкого применения отчасти потому, что так и не выработалась потребность в определенного рода параллельной коллективной обработке.

Классная доска в действии

Допустим, требуется написать программу, принимающую и обрабатывающую заявки на залог или заем. Законы, регулирующие данную сферу, одиозно сложны, причем в них имеются расхождения на уровне штатов и федерального правительства США. В частности, заимодавец должен подтвердить, что он владеет некоторыми преданными гласности предметами недвижимости, а также запросить определенные сведения, но *не* задавать ряд других вопросов и т.д.

Помимо проблем текущего законодательства, в данном случае придется решать следующие задачи.

- Ответы могут поступать в любом порядке. Например, обработка запросов на проверку кредитной истории или правового статуса может отнять немало времени, тогда как Ф.И.О. и адрес могут быть сверены немедленно.
- Сбор данных может выполняться разными людьми, рассредоточенными по разным учреждениям и в разных часовых поясах.
- Некоторые данные могут быть собраны автоматически другими системами, а также могут поступать асинхронно.
- Тем не менее одни данные могут быть собраны независимо от других. Например, начать проверку правового статуса владельца автомобиля, возможно, не удастся до тех пор, пока не будут подтверждены права владения или страховки.

- Поступление новых данных может поднимать новые вопросы и правила. Допустим, в результате проверки кредитной истории возвращается отнюдь не блестящий отчет, и поэтому придется заполнить еще пять дополнительных форм и взять образец крови.

Используя систему управления потоком операций, можно попытаться рассмотреть все возможные сочетания и обстоятельства. И хотя имеется немало таких систем, они могут быть довольно сложными для проектирования и программирования. Ведь нормативы изменяются, а поток операций может быть реорганизован. Так, если сменятся технологические процедуры, жестко закодированный код, возможно, придется полностью переписать.

В сочетании с механизмом соблюдения правил, инкапсулирующих законные требования, классная доска является изящным решением обнаруженных здесь трудностей. Порядок поступления данных не имеет особого значения: как только на доске будет размещен некоторый факт, он может привести в действие соответствующие правила. Ответная реакция организуется так же легко: результат соблюдения любого набора правил может быть размещен на классной доске, приводя в действие другие подходящие правила.

Совет 60 Пользуйтесь классными досками для координации потока выполнения

Системы обмена сообщениями могут быть подобны классным доскам

На момент написания второго издания данной книги многие приложения проектировались с помощью небольших развязанных служб, обменивавшихся данными через некоторую форму системы обмена сообщениями. Такие системы (например, Kafka и NATS) способны на гораздо большее, чем просто организовывать обмен сообщениями между сторонами A и B. Они, в частности, обеспечивают сохраняемость (в форме журнала регистрации событий), а также возможность извлекать сообщения посредством определенной формы сопоставления с шаблоном. Это означает, что их можно применять в качестве системы классной доски и/или платформы для выполнения целого ряда акторов.

Но не все так просто...

Подходы к архитектуре с использованием актора, классной доски и/или микрослужбы устраняют целый класс потенциальных затруднений, обусловленных в приложениях параллельностью. Но данное преимущество дается не бесплатно. Такие подходы труднее осмыслить, поскольку они действуют по большей части косвенно. Но при этом можно обнаружить, что они помогают организо-

вать и вести центральное хранилище форматов сообщений и/или прикладные интерфейсы API, особенно если в таком хранилище можно автоматически сгенерировать код и документацию. Потребуется также хороший инструментарий для отслеживания сообщений и фактов по мере их продвижения через систему. (В качестве удобной методики можно ввести однозначный *идентификатор трассировки* при инициализации конкретной прикладной функции, а затем распространить его среди всех заинтересованных акторов. Это позволит восстановить то, что происходит, из файлов регистрации.)

Наконец, развертывание и сопровождение такого рода системы может оказаться затруднительным, поскольку в ней немало подвижных частей. Этот недостаток в какой-то степени уравновешивается тем, что система оказывается более детализированной и может быть обновлена заменой отдельных акторов, а не целиком.

Другие разделы, связанные с данной темой

- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 29**. Манипулирование реальным миром, **глава 5** "Гибкость или ломкость".
- **Тема 33**. Разрывание временного связывания.
- **Тема 35**. Акторы и процессы.

Упражнения

24. Подойдет ли система типа классной доски для перечисленных ниже приложений? Объясните, почему она может подойти или не подойти.

- **Обработка изображений.** Здесь желательно иметь ряд параллельных процессов, которые выбирают фрагменты изображения, обрабатывают их и возвращают готовые фрагменты обратно.
- **Групповое ведение календаря.** Здесь требуется запланировать совещание участников проекта, рассредоточенных по всему миру, в разных часовых поясах и говорящих на разных языках.
- **Инструментальное средство текущего контроля сети.** Здесь система собирает статистические данные производительности и отчеты об отказах, используемые посредниками для обнаружения неисправностей в системе.

Задачи

- Пользуетесь ли вы системами классных досок на практике, в том числе доской семейных объявлений на холодильнике или белой доской на работе? Чем они эффективны? Появляются ли вообще на них сообщения в согласованном формате и насколько это важно?

Глава 7

По ходу кодирования

Житейская мудрость гласит: как только проект доходит до стадии кодирования, работа становится в основном механическим воплощением проектного решения в выполняемые операторы. На наш взгляд, такое отношение к проекту служит самой главной причиной его неудачного окончания, а многие спроектированные подобным образом системы оказываются скверными, неэффективными, плохо структурированными, не поддающимися сопровождению или просто наносящими ущерб.

Кодирование не механично. Если бы это было именно так, то все инструментальные средства CASE, на которые возлагались большие надежды еще в начале 1980-х годов, уже давно заменили бы собой программистов. По ходу работы приходится каждую минуту принимать решения, требующие тщательного обдумывания и оценивания, чтобы написанная в итоге программа работала долго, надежно и продуктивно.

Не все решения даже просто здравы, поэтому лучше обуздывать себя, прислушиваясь к своим инстинктам, как поясняется в соответствующем разделе. В этом разделе будет показано, как прислушиваться к своим инстинктам более осторожно и анализировать способы активной реакции на иногда пустячные мысли.

Но необходимость прислушиваться к инстинктам совсем не означает, что можно просто летать на автопилоте. Те разработчики, которые не обдумывают активно свой код, *программируют по совпадению*. Это означает, что написанный код может работать, но по какой именно причине — непонятно. Поэтому в разделе "Программирование по совпадению" отстаивается более положительный подход к процессу программирования.

Несмотря на то что код, который мы пишем, выполняется быстро, мы время от времени разрабатываем алгоритмы, способные застопорить работу даже самых быстродействующих процессоров. В связи с этим в разделе "Быстродействие алгоритмов" обсуждаются способы оценивания быстродействия написанного кода и даются некоторые рекомендации относительно выявления потенциальных осложнений в нем еще до того, как они возникнут.

Прагматичные программисты умеют критически осмысливать весь код, включая и свой собственный. Они постоянно ищут основания для совершенствования своих программ и проектов. Поэтому в разделе "Рефакторинг" будут рассмотрены методики, помогающие постоянно исправлять существующий код по ходу его написания.

Тестирование означает не столько обнаружение программных ошибок, сколько получение ответной реакции на свой код, включая особенности архитектуры, прикладного интерфейса API, связывания и т.д. Следовательно, основные выгоды из тестирования проявляются тогда, когда тесты сначала обдумываются, а затем пишутся, но не лишь тогда, когда они выполняются. Именно этот принцип и поясняется в разделе "Тестировать, чтобы кодировать".

Но, безусловно, когда вы тестируете свой код, вы можете привнести в решаемую задачу свои пристрастия. Поэтому в разделе "Тестирование на основе свойств" показано, как заставить компьютер выполнять обширное тестирование автоматически и обрабатывать неизбежно возникающие программные ошибки.

Очень важно писать удобочитаемый и понятный код. Окружающий мир полон злоумышленников, активно пытающихся взломать действующую систему и нанести ей ущерб. Поэтому в разделе "Будьте осторожны" рассматриваются самые элементарные методики и подходы, помогающие организовать надежную защиту системы от взлома.

Наконец, одну из самых больших трудностей в разработке программного обеспечения вызывает *именование* элементов кода. Разработчикам приходится именовать немало элементов кода, и зачастую выбираемые ими имена определяют ту реальность, которую они создают. Поэтому по ходу кодирования следует постоянно иметь в виду любые возможные семантические отклонения.

Большинство из нас водят автомобиль в основном машинально, не давая явно команду своей ступне нажать педаль или руке повернуть руль. Мы лишь задумываемся о необходимости притормозить и повернуть направо или налево в нужный момент. Но грамотные и осторожные водители постоянно оценивают ситуацию на дороге, упреждая потенциальные осложнения и тем самым оказываясь в лучшем положении, даже если произойдет что-нибудь непредвиденное. Это же относится и к кодированию, которое в основном оказывается рутинным занятием, но все же не зевайте — это поможет вам предотвратить возможные неприятности.

Тема 37 Прислушивайтесь к своим инстинктам

"Только люди способны непосредственно рассматривать что-то, имея всю необходимую информацию, чтобы сделать точный прогноз, а возможно, даже сделать его моментально и затем сказать, что дело обстоит иначе".

Гэвин де Беккер, "Дар страха"

Труд всей жизни Гэвина де Беккера помогает людям защищать себя. Свои взгляды на вопросы защиты от насилия он изложил в книге *The Gift of Fear: And Other Survival Signals That Protect Us from Violence* [de 98]. Одна из главных тем этой книги касается того обстоятельства, что люди как высокоразвитые существа научились пренебрегать своей животной сущностью, своими инстинктами и первобытными повадками. Он утверждает, что большинство людей, на которых нападают на улице, чувствуют себя неуютно или нервничают перед нападением. И эти люди просто внушают себе, что не стоит вести себя так глупо. И тогда из темного прохода возникает чужая фигура...

Инстинкты являются попросту реакцией на образцы поведения, хранящиеся в нашем бессознательном. Одни из них врожденные, другие приобретены через повторение. По мере накопления опыта программирования в уме незаметно откладываются слоями знания о том, что пригодно и что непригодно, о вероятных причинах типов ошибок и обо всем, что подмечается изо дня в день. Эта часть ума действует подобно экранной кнопке *сохранения файла*, которую вы нажимаете, когда прекращаете интерактивную переписку с кем-нибудь в оперативном режиме, даже не осознавая того, что делаете.

Независимо от происхождения инстинктов, им присуща одна общая особенность: они безмолвны. Инстинкты вызывают ощущение, а не мысль, и поэтому, когда возбуждается инстинкт, вы не видите никакой вспыхивающей лампочки с крупным заголовком вокруг. Вместо этого вы начинаете нервничать, испытываете недомогание и просто чувствуете, что предстоит слишком много работать.

Вся хитрость здесь в том, чтобы сначала заметить, что это происходит, а затем выяснить причину. Рассмотрим для начала пару типичных случаев, когда внутренний инстинкт пытается нам что-то подсказать, а затем обсудим, как вызволить свой инстинктивный ум из его защитной оболочки.

Боязнь пустой страницы

Все испытывают страх перед пустым экраном, где одиноко мигает курсор. Начало нового проекта (и даже нового модуля в уже существующем проекте)

может стоить немалых нервов. Поэтому многие из нас предпочитают отсрочить первоначальное обязательство приступить к делу. На наш взгляд, это связано с двумя затруднениями, для которых имеется одно и то же решение.

Одно затруднение связано с тем, что наш инстинкт пытается нам что-то подсказать. Но под самой личиной восприятия скрывается какое-то сомнение, что очень важно. Всякому разработчику приходится опробовать разные подходы, чтобы выяснить, какие из них годятся, а какие непригодны. И подобным образом накапливаются знания и опыт. Так, если вы чувствуете неотвязное сомнение или испытываете нежелание решать какую-то задачу, то вполне возможно, что это вам подсказывает ваш опыт. Прислушайтесь к нему. Не спешите совершать явно неверный шаг, а подождите, пока ваши сомнения не оформятся в нечто более ощутимое, чтобы их можно было разрешить. Дайте волю вашим инстинктам, чтобы они способствовали вашим поступкам.

Другое затруднение более прозаично. Вы можете просто бояться совершить ошибку. И это вполне обоснованное опасение. Ведь разработчики немало вкладывают в свой код и поэтому могут воспринимать ошибки в нем как отражение их компетентности. В этом, вероятно, есть также элемент *синдрома самозванца*. Разработчикам может показаться, что данный проект им не под силу. Они не в состоянии найти способ довести его до конца. Зайдя слишком далеко, они вынуждены признать, что потерялись.

Борьба с собой

Иногда код просто вылетает из вашего ума прямо в текстовый редактор, а идеи воплощаются в код без всяких усилий.

А порой кодирование воспринимается как ходьба в гору по колено в грязи. Каждый шаг требует громадных усилий, а через каждые три шага вы соскальзываете на два шага назад.

Но, будучи профессионалом, вы упорно продвигаетесь по этой грязи шаг за шагом вперед, поскольку вам нужно делать свое дело. К сожалению, это, вероятно, именно то, чего вам не следует делать.

Ваш код пытается вам что-то подсказать. Он говорит, что его можно было бы написать проще, чем это сделано. Возможно, структура или архитектура была выбрана неверно, а может быть, неверной оказалась сама решаемая задача или же вы просто создаете нечто, подобное муравейнику, достойному обитающих в нем насекомых. Независимо от конкретной причины, ваши инстинкты воспринимают ответную реакцию написанного кода, отчаянно пытающегося обратить на себя ваше внимание.

Как прислушиваться к своим инстинктам

Мы уже не раз подчеркивали, что необходимо прислушиваться к своим инстинктам, к своему бессознательному, первобытному уму. А существующие для этого методы всегда одинаковы.

Совет 61 Прислушивайтесь к своим внутренним чувствам

Прежде всего, приостановитесь в своей работе. Уделите время и место, чтобы позволить вашему мозгу самоорганизоваться. Перестаньте думать о коде и займитесь на время чем-нибудь откровенно бездумным и далеким от мысли о клавиатуре. Прогуляйтесь, пообедайте, побеседуйте с кем-нибудь, а может быть, даже вздремните немного. Дайте мыслям самим непринужденно проникнуть сквозь слои вашего ума. И когда они в конечном счете достигнут уровня вашего сознания, вы переживете один из моментов *озарения.*

Если этот способ не сработает, попытайтесь облечь возникшее затруднение во внешнюю форму. Сделайте зарисовки о написанном вами коде или объясните его своему коллеге (желательно не программисту) либо своему резиновому утенку. Раскройте разные участки своего ума для возникшего затруднения и выясните, какой из них лучше всего подойдет для того, чтобы разрешить данное затруднение. Мы уже потеряли счет беседам, в ходе которых один из нас объяснял другому возникшее у него затруднение и неожиданно восклицал: "Ах, да! Конечно!", сразу прерывая беседу, чтобы немедля устранить обсуждавшееся затруднение.

Вполне возможно, что вы уже опробовали описанные выше способы на практике и все равно не вышли из затруднения. В таком случае время действовать, и мы должны сказать вашему уму, что все, что вы собираетесь делать, не имеет никакого значения. И для этого мы прибегнем к прототипированию.

Время играть!

Авторам этой книги не раз приходилось часами глядеть на чистый экран текстового редактора. Сначала мы набирали какой-нибудь код, затем глядели в потолок, делали очередной глоток кофе, снова набирали немного кода, после этого читали забавный рассказ, например, о коте с двумя хвостами, опять набирали еще немного кода и, наконец, выделив весь исходный текст, удаляли его, начиная все с самого начала. И все это повторялось снова и снова.

С годами у нас выработался вполне работоспособный, на наш взгляд, метод взлома сознания. Внушите себе, что вам требуется создать прототип чего-нибудь. Если перед вами пустой экран, найдите такую особенность своего проекта, которая требует исследования. Это может быть, например, новый каркас,

применяемый в вашем проекте, и вам нужно выяснить, каким образом в нем осуществляется привязка данных; или же новый алгоритм, который требуется испытать на работоспособность в крайних случаях. А возможно, вам требуется опробовать два разных стиля взаимодействия с пользователем. Если, работая с существующим кодом, вы испытываете сопротивление с его стороны, сохраните его где-нибудь и создайте прототип, выполняющий аналогичные функции.

С этой целью выполните следующие действия.

1. Напишите фразу "Я создаю прототип" на клейкой бумажке для заметок и прикрепите ее к боковой стороне экрана своего монитора.
2. Напомните себе, что прототипы обречены на неудачу и выбрасывание, даже если они окажутся удачными. А самое главное, что в этом не следует усматривать никакого недостатка.
3. Создайте в пустом буфере текстового редактора комментарии, описывающие одним предложением, что именно вам требуется изучить или сделать.
4. Начните программировать.

Если вас начнут мучить сомнения, посмотрите на бумажку на мониторе. А если по ходу кодирования неотвязное сомнение неожиданно оформится в ясный вопрос, то разрешите его. Если же, дойдя до конца своего эксперимента, вы все еще будете испытывать неловкость, начните все с самого начала, сделав перерыв на прогулку, беседу и отдых.

Но, как показывает наш опыт, в какой-то момент по ходу создания первого прототипа вы с удивлением обнаружите, что, напевая под музыку, с удовольствием пишете код. Ваша нервозность исчезнет без следа, вытесненная неуемным желанием создать нечто конкретное. На этой стадии вы уже будете знать, что нужно делать. В таком случае — удалите код прототипа, выбросите бумажку для записей в мусорную корзину и заполняйте экран текстового редактора блестящим новым кодом.

Не только *свой* код

Львиная доля обязанностей разработчиков приходится на работу с уже существующим кодом, нередко написанным другими людьми. У этих людей были не такие инстинкты, как у вас, и поэтому их решения были иными. И хотя эти решения совсем не обязательно неудачные, они все же *иные*.

Чужой код можно читать механически, скрупулезно делая заметки, которые кажутся важными. Хотя это и рутинный, но все же работоспособный метод.

С другой стороны, можно попытаться провести эксперимент. Если вы обнаружите, что код кажется странным — сделайте соответствующие заметки. Затем продолжите дальше и поищите возможные шаблоны. Если вам удастся выяс-

нить, чем руководствовались программисты при написании анализируемого кода именно так, а не иначе, — возможно, вам будет легче его понять. И тогда вы сможете вполне осознанно применять шаблоны, которые подразумевались при разработке данного кода. Более того, вы даже можете узнать что-то новое для себя, анализируя чужой код.

Не только код

Умение прислушиваться к своим инстинктам при программировании является важным навыком, который стоит приобрести. Но такой подход применим и к более общей картине. Иногда проектное решение кажется просто неверным, а какое-нибудь требование вызывает ощущение неловкости. В таком случае следует остановиться и проанализировать свои ощущения. Если вы работаете в благоприятной для этого среде, выразите свои ощущения явно и исследуйте их. Скорее всего, что-нибудь да промелькнет в темном углу. Прислушайтесь к своим инстинктам, чтобы избежать осложнений до того, как они на вас навалятся.

Другие разделы, связанные с данной темой

- **Тема 13**. Прототипы и памятные записки, **глава 2** "Прагматичный подход".
- **Тема 22**. Технические дневники, **глава 3** "Основные инструментальные средства".
- **Тема 46**. Решение неразрешимых головоломок, **глава 8** "До начала проекта".

Задачи

- Есть ли что-нибудь такое, что, на ваш взгляд, должно быть сделано, но вы отложили его реализацию на время, поскольку чувствуете боязнь или предчувствуете трудности? В таком случае примените методы, описанные в этом разделе, уделив им один или два часа и дав себе слово, что, когда прозвенит звонок, удалите все, что было сделано за это время. Чему вы в итоге научились?

Тема 38 Программирование по совпадению

Приходилось ли вам когда-нибудь смотреть старые черно-белые фильмы о войне? Усталое лицо солдата осторожно высовывается из-за кустов. Впереди — открытая полянка, но не зарыты ли на ней противопехотные мины или же ее можно пересечь безопасно? Ничто не указывает на минное поле: никаких предупреждающих знаков, колючей проволоки или воронок от взорвавшихся мин. Солдат прощупывает штыком землю перед собой и отшатывается, ожидая взрыва.

А поскольку ничего не происходит, то он шаг за шагом продвигается по полю, протыкая землю штыком и ощупывая местность. В конечном счете он убеждается, что поле безопасно, и тогда выпрямляется и гордо ступает вперед лишь для того, чтобы быть разорванным на куски взорвавшейся миной. Выходит, первоначальные поиски мин ничего не выявили, и солдату просто везло. В конечном счете это привело его к ложному заключению и злополучному исходу.

Разработчики работают в условиях, которые весьма напоминают такие минные поля. Каждый день их ожидают сотни ловушек, в которые они могут попасть. Памятуя о злополучной участи солдата, они должны делать выводы очень осторожно. Им, в частности, следует избегать программирования по совпадению, полагаясь на удачу и случайный успех, а лучше программировать *осторожно*.

Как программировать по совпадению

Допустим, Фред получил определенное задание. Он набирает некоторый код, тестирует его и делает вывод, что код, по-видимому, работоспособен. Далее Фред набирает еще немного кода, опробует его и снова находит его работоспособным. Через несколько недель программирования в таком стиле получившаяся в конечном итоге программа неожиданно перестает работать. Потратив на попытки устранить неполадки не один час, Фред все еще не знает причин, по которым его программа не работает. Он может потратить немало времени на выявление сбойного фрагмента кода, так и не сумев исправить положение. И что бы он ни делал, его программа так и не работает нормально.

Фреду оказались неизвестны причины сбоев в программе потому, что он не выяснил, *почему она работала прежде*. После ограниченного тестирования Фреду показалось, что программа работает нормально, но это произошло лишь по случайному совпадению. Подбодрившись ложной уверенностью, Фред ринулся вперед, забыв об осторожности, и наткнулся на мину. Многим умным людям, вероятно, известны такие личности, как Фред, а нам известно кое-что получше. Мы не полагаемся на случайные совпадения, но так ли это на самом деле? Иногда мы можем поступить таким образом. Ведь часто очень легко спутать счастливое совпадение с преднамеренным планом действий. Поэтому рассмотрим ряд конкретных примеров.

Случайности реализации

Они происходят просто потому, что код написан в данный момент именно так, а не иначе. И в конечном счете приходится полагаться на недокументированную ошибку или граничные условия.

Допустим, подпрограмма вызывается с неверными данными. Подпрограмма дает некоторый ответ, и вызывающий код опирается на него. Но автор подпрограммы не предусмотрел именно такой режим работы подпрограммы и даже не

рассматривал его. Если внести в подпрограмму требующиеся поправки, работа вызывающего кода может просто нарушиться. В самом крайнем случае вызываемая подпрограмма может быть даже не предназначена делать то, что от нее требуется в вызывающем коде. Тем не менее она, *по-видимому*, вполне работоспособна. Аналогичное затруднение возникает при вызовах не в том порядке или же не в том контексте.

Это похоже на отчаянные попытки Фреда вывести что-нибудь на экран, используя конкретный каркас GUI:

```
paint();
invalidate();
validate();
revalidate();
repaint();
paintImmediately();
```

Но перечисленные подпрограммы никогда не создавались для вызова таким образом; хотя они выглядят работоспособными, на самом деле это всего лишь совпадение.

Положение усугубляется еще больше, когда окончательный вывод выглядит нормально, и Фред даже не попытается вернуться назад и устранить ложные вызовы. Он считает, что его программа работает нормально и лучше не трогать то, что и так работает.

Такая нить рассуждений может ввести в заблуждение. В самом деле, зачем рисковать, приводя в беспорядок то, что уже работает? Что ж, рассмотрим несколько следующих причин, дающих ответ на этот вопрос.

- Программа может на самом деле не работать, а только делать вид, что нормально работает.
- Граничное условие, на котором вы основываетесь, может оказаться простой случайностью. Это означает, что при разных обстоятельствах (разном разрешении экрана или количестве ядер процессора) программа может работать по-разному.
- Недокументированное поведение может измениться в следующем выпуске библиотеки.
- Дополнительные и излишние вызовы могут замедлить работу программы.
- Дополнительные вызовы увеличивают риск добавления новых программных ошибок.

Для кода, который пишется с расчетом быть вызванным из другого кода, могут вполне пригодиться основные принципы модуляризации и сокрытия реализации за небольшими хорошо документированными интерфейсами. А грамотно составленный контракт (см. раздел “Тема 23. Проектирование по контракту”

главы 4 "Прагматичная паранойя") может оказать помощь в устранении недоразумений. Что же касается вызываемых подпрограмм, то следует полагаться на документированное поведение. Если же это по какой-нибудь причине невозможно, то следует задокументировать свое допущение.

Недостаток точности

Однажды нам пришлось работать над крупным проектом, выводившим данные из очень большого количества запоминающих устройств непосредственно в поле. Эти устройства были рассредоточены по разным штатам и часовым поясам, и каждое из них по различным логистическим и историческим причинам было настроено на местное время[1]. В результате конфликтов в интерпретации часовых поясов и несогласованности правил перехода на летнее время результаты почти всегда получались неверными, хотя и отличались всего лишь на единицу. Разработчики данного проекта решили просто прибавлять или отнимать единицу, чтобы получить правильный ответ, полагая, что в данной конкретной ситуации он будет отличаться *всего лишь* на единицу. И когда следующая функция обнаружит, что значение сместилось на единицу в другую сторону, она возвратит его обратно.

Но смещение результатов "всего лишь" на единицу оказалось совпадением, за которым скрывался более глубокий и основательный изъян. В отсутствие надлежащей модели интерпретации времени вся крупная кодовая база превратилась со временем в совершенно непригодную массу операторов `+1` и `-1`. И в конечном счете ни один из них не оказался верным, а проект пришлось отправить на свалку.

Ложные закономерности

Человеческие существа предрасположены усматривать закономерности и причины даже в том случае, когда это всего лишь совпадение. Например, в России всегда чередовались то лысые, то волосатые правители: на протяжении 200 лет лысый (или явно лысеющий) государственный деятель сменялся волосатым, т.е. совсем не лысым, и наоборот[2].

Но если никому не придет в голову написать код в зависимости от того, будет ли следующий правитель России лысым или волосатым, то в некоторых предметных областях разработчики все время рассуждают именно таким образом. Игроки выдумывают закономерности в лотерейных номерах, бросаемых костях или крутящихся рулетках, когда на самом деле это статистически независимые события. Аналогично в финансовой сфере биржевые операции с акциями и облигациями полны совпадений, а не вполне различимых закономерностей.

---

[1] Примечание стреляного воробья: используйте всемирное координированное время (UTC).

[2] См. по адресу `https://en.wikipedia.org/wiki/Correlation_does_not_imply_causation`.

Журнальный файл, в котором ошибка появляется через каждые 1000 запросов, может представлять труднодиагностируемое состояние гонки, а возможно, просто результат старой программной ошибки. Тесты, которые вроде бы проходят на одной машине, могут не пройти на сервере, выявив тем самым отличия в обеих средах. А может быть, это простое совпадение. Словом, не допускайте, а убеждайтесь.

Случайности контекста

Вполне возможны и так называемые "случайности контекста". Допустим, требуется написать служебный модуль. Следует ли полагаться на обязательное наличие графического интерфейса пользователя только потому, что этот модуль разрабатывается в настоящий момент для GUI-среды? Рассчитан ли этот модуль на англоязычных пользователей? На обладающих специальной подготовкой? На что еще не стоит полагаться, если оно не гарантируется?

Следует ли полагаться на то, что текущий каталог доступен для чтения? Наличествуют ли определенные переменные окружения или файлы конфигурации? Будет ли сервер со временем работать точно и с каким допуском? Стоит ли полагаться на доступность и быстродействие сети?

Скопировав исходный код, полученный по первому же ответу, найденному в сети, можно ли быть уверенным, что он имеет тот же контекст? Не разрабатывается ли код в стиле карго-культа, просто выполняя имитацию без контекста?[3] Найти подходящий ответ на все эти вопросы еще не означает получить правильный ответ.

Совет 62 Не программируйте по совпадению

Неявные допущения

Совпадения могут вводить в заблуждение на всех уровнях: от составления требований до тестирования. В частности, тестирование чревато ложными причинными зависимостями и случайными следствиями. Ведь очень легко предположить, что *X* приводит к *Y*, но, как упоминалось ранее в совете 34 из раздела "Тема 20. Отладка" главы 3 "Основные инструментальные средства", не предполагайте, а доказывайте.

Люди на всех уровнях оперируют в своем уме многими предположениями, но эти предположения редко документируются и зачастую вызывают споры у разных разработчиков. Предположения, не основывающиеся на точно установленных фактах, являются бедой любых проектов.

[3] См. раздел "Тема 50. Кокосами не обойтись" в главе 9, "Прагматичные проекты".

Как программировать обдуманно

Всем нам хотелось бы уделять меньше времени исправлению кода, выявляя и устраняя ошибки на как можно более ранней стадии разработки, а также внося в код как можно меньше ошибок. Этого можно добиться, программируя обдуманно, следуя приведенным ниже рекомендациям.

- Старайтесь всегда понимать, что вы делаете. Упоминавшийся ранее Фред медленно пустил дело на самотек до тех пор, пока не ошпарился как лягушка, о которой речь шла в разделе "Тема 4. Суп из камней и вареные лягушки" главы 1 "Философия прагматизма".
- Сможете ли вы подробно объяснить код менее опытному программисту? Если вам это не удается, значит вы, вероятнее всего, полагаетесь на совпадения.
- Не программируйте в темноте. Стоит вам написать приложение, в котором вы сами не разобрались до последней буквы, или воспользоваться технологией, которую не полностью понимаете, как вы тотчас будете ужалены совпадениями. И если вы не знаете причин, по которым программа работает, — значит, вы не знаете причин, по которым она может отказать.
- Действуйте на основе плана, будь то составленного в вашем уме, начертанного на салфетке или написанного на доске.
- Полагайтесь только на надежные факты, а не на предположения. Если вы не знаете, надежно ли что-нибудь, — допустите самое худшее.
- Документируйте свои предположения. Материал раздела "Тема 23. Проектирование по контракту" главы 4 "Прагматичная паранойя" поможет вам как самому прояснить свои предположения, так и поделиться ими с другими.
- Не ограничивайтесь только тестированием своего кода, тестируйте и свои предположения. Не гадайте — испытайте их. Напишите утверждение для проверки своих предположений (см. раздел "Тема 25. Утвердительное программирование" главы 4 "Прагматичная паранойя"). Если ваше утверждение окажется верным, значит, вы улучшили документирование своего кода. А если оно окажется неверным, то можете считать себя везунчиком.
- Расставьте приоритеты в своей работе. Уделите время самым важным ее аспектам. Более чем вероятно, что эти аспекты окажутся и самыми трудными. В отсутствие прочных и верных основ или инфраструктуры любые излишества неуместны.
- Не привязывайтесь к истории, позволяя существующему коду диктовать условия будущему коду. Можно заменить весь код, если он больше не го-

дится. И даже в одной программе не позволяйте тому, что уже сделано, накладывать ограничения на то, что предстоит сделать. Будьте готовы к рефакторингу кода (см. далее раздел "Тема 40. Рефакторинг"). Такое решение может повлиять на календарный план работы над проектом. При этом предполагается, что принятие этого решения приведет к меньшим затратам, чем его *не* принятие[4].

Таким образом, в следующий раз, когда что-нибудь покажется вам работоспособным, но вы не знаете, почему — непременно убедитесь, что это не просто совпадение.

Другие разделы, связанные с данной темой

- **Тема 4**. Суп из камней и вареные лягушки, **глава 1** "Философия прагматизма".
- **Тема 9**. DRY — пороки дублирования, **глава 2** "Прагматичный подход".
- **Тема 23**. Проектирование по контракту, **глава 4** "Прагматичная параноя".
- **Тема 34**. Общее состояние — неверное состояние, **глава 6** "Параллельность".
- **Тема 43**. Будьте осторожны.

Упражнения

25. Допустим, данные предоставлены поставщиком в виде массива кортежей, состоящих из пар "ключ-значение". В частности, для ключа `DepositAccount` значение содержит символьную строку с номером счета, как показано ниже.

```
[
  ...
  {:DepositAccount, "564-904-143-00"}
  ...
]
```

Такая структура данных показала себя идеально при проверке, проведенной на компьютерах разработчиков с 4-ядерными процессорами, а также на сборочной машине с 12-ядерным процессором. Но на рабочих серверах, работающих в контейнерах, постоянно получаются неверные номера счетов. Что происходит?

[4] Здесь можно зайти слишком далеко. Мы как-то познакомились с одним разработчиком, переписавшим весь переданный ему исходный код только потому, что у него были свои соглашения по именованию.

26. Допустим, требуется запрограммировать автоматический номеронабиратель для голосовых предупреждений и вести базу данных контактной информации. По спецификации Международного союза электросвязи (ITU) телефонные номера не могут быть длиннее 15 цифр, так что вы сохраняете номер телефона в числовом поле, гарантированно хранящем как минимум 15 цифр. Данное устройство было тщательно испытано на всей территории Северной Америки, и все было нормально... но неожиданно из других частей света устремился поток жалоб. Почему так произошло?

27. Допустим, вы написали приложение, которое масштабируется для ресторана круизного судна на 5000 посадочных мест. Вы получаете жалобы на то, что преобразования неточны. Вы проверяете прикладной код, в котором применяется формула преобразования 16 чашек в галлон. Правильно это или нет?

Тема 39 Быстродействие алгоритмов

В разделе "Тема 15. Оценивание" главы 2 "Прагматичный подход" речь шла о том, как оценить, сколько времени отнимет прогулка по городу или завершение проекта. Но имеется и другая разновидность оценок, которой программисты-прагматики пользуются ежедневно. Речь идет об оценке таких используемых в алгоритмах ресурсов, как, например, время, процессор, оперативная память и т.д.

Такого рода оценка нередко имеет решающее значение. Так, если имеются два способа сделать что-нибудь, то какой из них лучше выбрать? Известно, за какое время программа обрабатывает 1000 записей, но если ее нужно масштабировать до 1 000 000 записей? Какие ее части при этом необходимо оптимизировать?

Оказывается, что на все эти вопросы можно ответить, призвав на помощь здравый смысл, произведя некоторый анализ и воспользовавшись способом записи аппроксимаций, именуемых *асимптотическим обозначением*, или *обозначением "О большое"*.

Что подразумевается под оценкой алгоритмов

В большинстве нетривиальных алгоритмов обрабатываются переменные входные данные некоторого рода; например, сортируются n символьных строк, преобразуется матрица $m \times n$ или сообщение расшифровывается с помощью n-разрядного ключа. Как правило, объем входных данных оказывает влияние на работу алгоритма: чем больше этот объем, тем дольше обрабатываются данные и больше оперативной памяти для этого требуется.

Если бы такое соотношение всегда было линейным (т.е. чтобы время увеличивалось прямо пропорционально величине n), то этот раздел не стоило бы

даже писать. Но большинство важных алгоритмов не линейны. Хорошая новость — среди них имеется немало сублинейных алгоритмов. Например, алгоритму бинарного поиска не требуется просматривать все элементы для обнаружения нужного. Плохая новость в том, что другие алгоритмы значительно уступают в быстродействии линейным алгоритмам, поскольку время их выполнения и требования к оперативной памяти возрастают быстрее, чем n. Такой алгоритм, которому для обработки десяти элементов требуется минута, может потребовать целой жизни для обработки ста элементов.

Всякий раз, когда мы пишем какой-нибудь код, содержащий циклы или рекурсивные вызовы функций, мы неосознанно проверяем требования ко времени выполнения и оперативной памяти. Такой процесс редко носит формальный характер и скорее служит для быстрого подтверждения правильности того, что мы делаем в определенных обстоятельствах. Но иногда нам все же приходится выполнять более подробный анализ. Именно тогда и приходит на помощь асимптотическое обозначение.

Асимптотическое обозначение

Асимптотическое обозначение, записываемое как $O()$, является математическим средством обращения с аппроксимациями. Так, если сортировка n записей в конкретной подпрограмме отнимает время $O(n^2)$, такое обозначение означает, что в худшем случае время сортировки записей будет изменять как квадрат n. Стоит удвоить количество записей, как время их сортировки увеличится приблизительно в четыре раза. Обозначение O, по существу, означает "около..." или "порядка...".

Асимптотическое обозначение $O()$ устанавливает верхний предел измеряемой величины (времени, объема оперативной памяти и т.д.). Так, если говорят, что для выполнения функции требуется время $O(n^2)$, то известно, что верхний предел времени ее выполнения не будет расти быстрее, чем n^2. Иногда приходится иметь дело с довольно сложными функциями $O()$, но поскольку по мере увеличения n преобладает член высшего порядка, то обычно удаляются все члены низкого порядка, а также все постоянные множители, как показано ниже.

$O\left(\frac{n^2}{2}+3n\right)$ представляет собой то же, что и $O\left(\frac{n^2}{2}\right)$, и то же, что и $O(n^2)$.

Таково свойство асимптотического обозначения $O()$ — один алгоритм $O(n^2)$ может действовать в 1000 раз быстрее, чем другой алгоритм $O(n^2)$, но это никак нельзя узнать из самого асимптотического обозначения. Конкретные числовые значения времени, объема оперативной памяти или иных величин из асимптотического обозначения получить нельзя. Оно просто показывает, каким образом эти величины будут изменяться по мере изменения входных данных.

На рис. 7.1 показано несколько типичных асимптотических обозначений $O()$ наряду с графиком для сравнения времени выполнения алгоритмов каждой категории. Очевидно, что алгоритм выходит из повиновения, как только преодолевается предел $O(n^2)$.

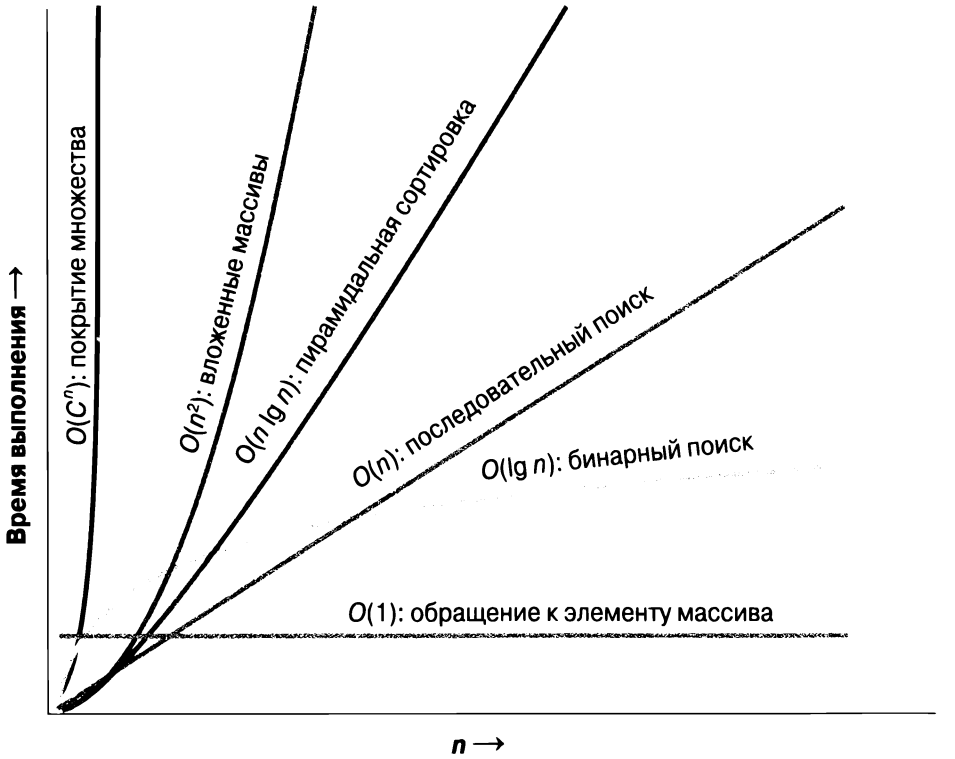


Рис. 7.1. Время выполнения различных алгоритмов

Допустим, имеется подпрограмма, которой требуется одна секунда для обработки 100 записей. Сколько времени отнимет обработка 1000 записей? Если имеется алгоритм $O(1)$, то для этой цели потребуется одна секунда. Если же это алгоритм $O(\lg n)$, то ждать, вероятно, придется около трех секунд. Алгоритм $O(n)$ покажет линейное увеличение времени обработки до десяти секунд, тогда как алгоритм $O(n \lg n)$ — до 33 секунд. Если же неудачно выбрать алгоритм $O(n^2)$, то окончания обработки придется ждать уже 100 секунд. А если воспользоваться алгоритмом $O(2^n)$, то можно смело сделать перерыв на чашку чая или кофе, поскольку обработка должна завершиться приблизительно через 10^{263} лет, и мы узнаем, каким окажется конец существования вселенной.

Асимптотическое обозначение $O()$ применяется не только ко времени. С его помощью можно представить любые ресурсы, используемые в алгоритме. Например, зачастую оказывается полезно смоделировать потребление оперативной памяти (см., например, упражнения в конце этого раздела).

Ниже перечислены типичные алгоритмы и их асимптотическое обозначение.

$O(1)$	Постоянная зависимость (доступ к элементу массива, простые операторы)
$O(\lg n)$	Логарифмическая зависимость (бинарный поиск). Основание логарифма значения не имеет, поэтому данная запись равнозначна записи $O(\log_2 n)$ или $O(\ln n)$
$O(n)$	Линейная зависимость (последовательный поиск)
$O(n \lg n)$	Линарифмическая зависимость — хуже, чем линейная, но не намного (среднее время выполнения быстрой или пирамидальной сортировки)
$O(n^2)$	Квадратичная зависимость (сортировка методом выборки или вставки)
$O(n^3)$	Кубическая зависимость (перемножение двух матриц $n \times n$)
$O(C^n)$	Экспоненциальная зависимость (задача коммивояжера, разбиение множества)

Разумное оценивание алгоритмов

Многие простые алгоритмы можно оценить, руководствуясь здравым смыслом, как поясняется ниже.

- **Простые циклы.** Если простой цикл выполняется от 1 до n, то ему, вероятнее всего, соответствует алгоритм $O(n)$, где время увеличивается линейно по мере увеличения n. Характерными тому примерами служат исчерпывающий поиск, нахождение максимального значения в массиве и вычисление контрольных сумм.
- **Вложенные циклы.** Если один цикл вложен в другой, то ему соответствует алгоритм $O(m \times n)$, где m и n — два предела выполнения циклов. Это обычно происходит в таких простых алгоритмах сортировки, как, например, пузырьковым методом, где во внешнем цикле каждый элемент массива просматривается по очереди, а во внутреннем цикле обнаруживается место, где следует разместить данный элемент в отсортированном результате. Такие алгоритмы сортировки обычно являются алгоритмами $O(n^2)$.
- **Бинарный поиск делением пополам.** Если алгоритм на каждом шаге делит множество элементов пополам, то он, вероятнее всего, имеет логарифмическое время выполнения $O(\lg n)$. Ту же сложность имеет бинарный поиск в отсортированном списке, обход бинарного дерева и обнаружение первого установленного бита в машинном слове.

- **Декомпозиция.** Алгоритмы, сначала разделяющие входные данные на две независимые половины, обрабатывающие их по отдельности, а затем объединяющие результат, обычно являются алгоритмами $O(n\lg n)$. Классическим примером служит быстрая сортировка, разделяющая данные на две половины и рекурсивно сортирующая каждую из них. И хотя формально это алгоритм $O(n^2)$, поскольку в наихудшем случае, когда на вход поступают отсортированные входные данные, его работа сильно замедляется, среднее время быстрой сортировки все же составляет $O(n\lg n)$.
- **Перестановка.** Всякий раз, когда в алгоритмах начинается анализ перестановок элементов, время их выполнения может выйти из-под контроля. Объясняется это тем, что в перестановках задействованы факториалы (например, $5!=5\times4\times3\times2\times1=120$ перестановок цифр от 1 до 5). Для перестановки шести элементов по сравнению с пятью потребуется в шесть раз больше времени, а для перестановки семи элементов — в 42 раза. Характерными примерами применения алгоритмов перестановки служат такие признанные трудно разрешимыми задачи, как задача коммивояжера, оптимальная упаковка товаров в контейнер, разбиение множества чисел таким образом, чтобы получить одинаковую сумму в каждом подмножестве, и т.д. Нередко для сокращения времени выполнения подобных алгоритмов в конкретной предметной области применяется эвристический анализ.

Быстродействие алгоритма на практике

Вы вряд ли будете тратить много времени на то, чтобы писать подпрограммы сортировки. Доступные в библиотеках подпрограммы, вероятно, превзойдут все, что вы могли бы написать без существенных усилий. Тем не менее время от времени приходится встречаться с описанными выше типами алгоритмов. Всякий раз, когда вам приходится писать простой цикл, знайте, что вы имеете дело с алгоритмом $O(n)$. Если же этот цикл содержит внутренний цикл, значит, перед вами алгоритм $O(m\times n)$. И тогда вы должны задаться вопросом: насколько крупными могут оказаться массивы сортируемых значений? Если они ограничены, то можно оценить, сколько времени потребуется для выполнения сортирующего кода. Но если размеры зависят от внешних факторов (например, количества записей для пакетной обработки ночью или количества имен в списке людей), то вам, возможно, придется остановиться и рассмотреть влияние, которое большие входные данные могут оказать на время выполнения программы или объем потребляемой ею оперативной памяти.

Совет 63 Оценивайте порядок производительности своих алгоритмов

Имеется ряд подходов, которые можно предпринять для разрешения потенциальных затруднений. Так, если у вас имеется алгоритм $O(n^2)$, попытайтесь воспользоваться подходом "разделяй и властвуй", чтобы найти алгоритм $O(n\lg n)$.

Если вы не знаете, как долго будет выполняться ваш код или сколько оперативной памяти ему потребуется, попробуйте сначала выполнить его для небольших количеств входных данных (того, что, вероятнее всего, оказывает влияние на время выполнения), а затем нанесите полученные результаты на график. По виду кривой этого графика вы сможете сразу же судить о зависимости времени выполнения от размера входных данных. Для того чтобы увидеть, изгибается ли кривая графика вверх, выпрямляется в линию или сглаживается по мере увеличения объема входных данных, обычно достаточно получить три-четыре точки.

Обратите также внимание на то, что вы делаете в самом коде. Простой цикл алгоритма $O(n^2)$ может проявить себя лучше, чем сложный цикл алгоритма $O(n\lg n)$ при малых значениях n, особенно если в цикле алгоритма $O(n\lg n)$ имеется дорогостоящий внутренний цикл.

Не забудьте и о том, что имеются и практические соображения. Так, при небольших массивах входных данных время выполнения может возрастать линейно. Но стоит подать на вход миллионы записей, как время выполнения кода резко возрастет, а производительность системы — упадет. Если вы проверяете подпрограмму сортировки со случайными входными данными, вас может удивить ее поведение, когда она столкнется с упорядоченными входными данными. Постарайтесь учесть как теоретические, так и практические соображения. В конечном итоге имеет значение лишь проверка быстродействия вашего кода в реальной производственной среде, с реальными входными данными. Это приводит к следующему совету:

Совет 64 Проверяйте свои оценки

Если трудно получить точные временны*е* характеристики, воспользуйтесь *профайлерами кода*, чтобы подсчитать, сколько раз в вашем алгоритме выполняются разные действия, а затем нанесите полученные данные на график зависимости от размера входных данных.

Лучшее не всегда лучшее

Прагматиком необходимо быть и при выборе подходящего алгоритма, поскольку самый быстродействующий алгоритм далеко не всегда оказывается наилучшим для решения конкретной задачи. Так, если имеется небольшой массив входных данных, простая их сортировка методом вставки проявит себя лучше, чем быстрая сортировка, не говоря уже том, что для написания и отладки

реализующего ее кода потребуется меньше времени. Необходимо также проявлять осмотрительность, если выбираемый алгоритм требует больших затрат на настройку. При небольших массивах входных данных такая настройка может нивелировать все преимущества во времени выполнения и сделать алгоритм непригодным с практической точки зрения.

Кроме того, отнеситесь осторожно к *преждевременной оптимизации*. Всегда рекомендуется убедиться, что выбранный алгоритм является узким местом производительности, прежде чем тратить драгоценное время на его усовершенствование.

Другие разделы, связанные с данной темой

- **Тема 15**. Оценивание, **глава 2** "Прагматичный подход".

Задачи

- Каждый разработчик должен знать, каким образом алгоритмы устроены и как они анализируются. На эту тему Роберт Седжвик (Robert Sedgewick) написал целый ряд книг, в том числе *Algorithms* [SW11] и *An Introduction to the Analysis of Algorithms* [SF13]. Рекомендуем вам пополнить этими книгами свою библиотеку и взять себе за правило читать их регулярно.
- Тем, кто интересуется большими подробностями, чем в книгах Роберта Седжвика, рекомендуются перечисленные ниже книги из каноничного издания *Art of Computer Programming*[5], где анализируется обширный ряд алгоритмов.
  - *The Art of Computer Programming, Volume 1: Fundamental Algorithms* [Knu98]
  - *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* [Knu98a]
  - *The Art of Computer Programming, Volume 3: Sorting and Searching* [Knu98b]
  - *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1* [Knu11].
- Приведенное ниже упражнение 28 посвящено сортировке массивов длинных целых чисел. Какое влияние на их сортировку окажет повышение сложности ключей и дополнительных издержек на их сравнение? Оказывает ли структура ключей воздействие на эффективность алгоритма сортировки или же самая быстрая сортировка таковой всегда и остается?

[5] Имеется русский перевод этих книг под названием *Искусство программирования*, выпущенный ИД "Вильямс". — *Примеч. пер.*

Упражнения

28. Мы запрограммировали ряд простых алгоритмов сортировки на языке Rust[6]. Выполните их на разных доступных вам машинах. Будут ли полученные вами данные соответствовать предполагаемым кривым зависимостей? Какой вывод можно сделать об относительном быстродействии алгоритмов на отдельных машинах? Каковы последствия оптимизации различных параметров настройки компилятора?

29. Ранее в разделе "Разумное оценивание алгоритмов" было сказано, что бинарный поиск делением пополам относится к категории алгоритмов $O(\lg n)$. Можете ли вы это доказать?

30. Ранее в разделе "Асимптотическое обозначение" было сказано, что асимптотические обозначения $O(\ln n)$ и $O(\log_2 n)$ означают одну и ту же сложность алгоритма. Можете ли вы пояснить причину?

Тема 40 Рефакторинг

"Перемена и упадок во всем, что я вижу..."

Х.Ф. Лит[7]*, **Abide With Me** (Пребудь со мной)*

По мере развития программы возникает потребность переосмыслить прежние решения и переделать отдельные части исходного кода. Это совершенно естественный процесс. Ведь код должен развиваться, он не является статическим.

К сожалению, самой распространенной метафорой для разработки программного обеспечения является построение здания. В своем классическом труде *Object-Oriented Software Construction* [Mey97] Бертран Мейер (Bertrand Meyer) употребляет термин "построение программного обеспечения", а скромным авторам данной книги приходилось даже редактировать колонку "Software Construction" (Построение программного обеспечения) в журнале *IEEE Software* в начале 2000-х годов[8].

Строительство в качестве руководящей метафоры подразумевает выполнение следующих действий.

1. Архитектор чертит чертежи.
2. Подрядчики углубляют фундамент, возводят стены, проводят канализацию и прочие коммуникации и выполняют отделочные работы.

---

[6] См. по адресу `https://media-origin.pragprog.com/titles/tpp20/code/algorithm_speed/sort/src/main.rs`.

[7] H. F. Lyte — шотландский англиканский пастор, теолог, поэт XIX века, а также автор церковных гимнов. — *Примеч. пер.*

[8] И мы, конечно, выражали свою озабоченность по поводу названия этой колонки.

3. Жильцы заселяют дом и счастливо живут в нем, периодически вызывая для устранения неполадок эксплуатирующий здание персонал — электриков, сантехников и др.

Безусловно, программное обеспечение таким способом не разрабатывается. Его разработку можно сравнить не со стройкой, а скорее с *выращиванием*, поскольку оно более органичное, а не бетонное. Растения высаживаются в саду по заранее составленному плану и в соответствии с местными условиями. Одни растения успешно вырастают, а другие в конечном итоге идут на компост. Растения можно пересаживать, чтобы выгодно воспользоваться взаимным влиянием света и тени, ветра и дождя. Слишком заросшие растения обрезаются, а те, что не ласкают взгляд, переносятся в эстетически более привлекательное место. Кроме того, выпалываются сорняки и удобряются те растения, которые требуют дополнительного ухода. Здоровое состояние сада требует постоянного контроля и ухода по мере надобности за почвой, растениями и их размещением.

Для деловых людей более удобной метафорой является построение здания, поскольку оно более научно, чем садоводство, повторяемо, ему присуща жесткая иерархия отчетности перед руководством и т.д. Но ведь мы строим не небоскребы и не ограничены в своих действиях пределами физики и реального мира.

Метафора садоводства намного ближе к реальностям разработки программного обеспечения. Так, одна подпрограмма может слишком разрастись, а попытка реализовать слишком крупное функциональное средство может потребовать его разделения. Все, что не работает нормально, необходимо искоренить, выполоть или обрезать.

Переписывание, переработка и переделка архитектуры кода сообща называется *реструктуризацией*. *Рефакторинг* представляет собой практическое подмножество такой деятельности.

В своей книге *Refactoring: Improving the Design of Existing Code*[9] [Fow19] Мартин Фаулер (Martin Fowler) определяет рефакторинг следующим образом:

> "дисциплинированная методика перестройки структуры существующего тела кода, изменение его внутренней структуры без изменения его внешнего поведения".

Самые важные части этого определения таковы.

1. Деятельность дисциплинирована, а не хаотична.
2. Внешнее поведение не меняется, а следовательно, при этом не вводятся новые функциональные средства.

---

[9] Имеется русский перевод: Мартин Фаулер, Кент Бек, Джон Брант, Уильям Опдайк, Дон Робертс. *Рефакторинг: улучшение проекта существующего кода*. — СПб.: "Диалектика", 2017.

Рефакторинг не предусмотрен как особый церемониальный вид деятельности, выполняемый лишь изредка, подобно запашке всего сада для пересадки растений. Напротив, рефакторинг представляет собой повседневную деятельность, выполняемую мелкими шажками с малым риском что-то испортить, подобно прополке или поливанию. Вместо глобального переписывания кода применяется точно нацеленный, выверенный подход, помогающий упростить внесение изменений в код.

Чтобы гарантировать неизменяемость внешнего поведения, требуется качественное, автоматизированное модульное тестирование, проверяющее правильность поведения кода.

Когда следует выполнять рефакторинг

Рефакторинг выполняется, когда вы что-то изучили, когда понимаете что-то лучше, чем в прошлом году, вчера или еще десять минут назад.

Вполне возможно, что вы наткнулись на камень преткновения, поскольку код уже не вполне отвечает исходным требованиям; а может быть, заметили, что два компонента следует непременно объединить, или обратили внимание на еще что-нибудь такое, что поражает вас, как "неверное". В таком случае *не колеблясь, измените код*. Сделать это лучше всего сразу же, не откладывая на будущее. Вот только некоторые из причин для рефакторинга кода.

- **Дублирование.** Обнаружено нарушение принципа DRY.
- **Неортогональный дизайн.** Обнаружено нечто такое, что можно сделать более ортогональным.
- **Устаревшие знания.** Изменились обстоятельства, сменились требования, ваше знание решаемой задачи расширилось. В таком случае код необходимо сделать актуальным, приведя его в соответствие с произошедшими изменениями.
- **Применение.** По мере того как система эксплуатируется реальными людьми в реальных условиях, приходит понимание того, что некоторые функциональные средства теперь больше важны, чем предполагалось ранее, тогда как "обязательные" прежде средства таковыми, вероятно, уже не являются.
- **Производительность.** Возникла потребность перенести функциональные возможности из одного участка системы в другой, чтобы повысить общую производительность.
- **Прохождение тестов.** Да, именно так. Ведь ранее говорилось, что рефакторинг должен быть мелкомасштабным видом деятельности, и поэтому он должен поддерживаться грамотно написанными тестами. Так, когда вы добавляете некоторый код, который проходит еще один дополнительный тест, у вас появляется отличная возможность углубиться в код и привести в порядок то, что вы только что написали.

Рефакторинг — перемещение функциональных возможностей и обновление прежних решений — на самом деле оказывается *управлением болью*. Будем откровенны, внесение изменений в исходный код может оказаться довольно болезненным. Раньше код работал нормально, так что его хочется оставить в покое. Многие разработчики неохотно снова возвращаются к фрагменту кода лишь потому, что он не полностью правильный.

Сложности реального мира

Итак, вы обращаетесь к коллегам по работе или клиенту и заявляете: "Этот код вполне работоспособен, но мне потребуется неделя, чтобы выполнить его рефакторинг". Из этических соображений мы не приводим их (непечатный) ответ.

Нехватка времени нередко служит оправданием отказа от рефакторинга. Но это оправдание не выдерживает никакой критики. Если не выполнить рефакторинг кода сейчас, то для устранения неполадок позже потребуется затратить намного больше времени. И чем дальше, тем с большим количеством зависимостей придется считаться. Будет ли у вас столько времени вообще? Вряд ли.

Этот принцип, возможно, придется объяснить другим, используя медицинскую аналогию, рассматривая код, требующий реализации, как "нарост". Чтобы удалить его, потребуется хирургическое вмешательство. Это можно сделать сразу, удалив нарост, пока он еще невелик. Можно подождать до тех пор, пока нарост не станет больше и не распространится дальше, но его удаление впоследствии может обойтись дороже и быть опаснее. Стоит подождать еще немного, и можно вообще потерять пациента.

Совет 65 Выполняйте рефакторинг кода как можно раньше и чаще

Столь же опасно со временем может разрастить и ущерб коду (см. раздел "Тема 3. Программная энтропия" главы 1 "Философия прагматизма"). Как и большинство других методик, рефакторинг проще выполнить, пока трудности невелики, как просто разновидность деятельности во время программирования. Для рефакторинга фрагмента кода не требуется неделя, в течение которой можно было бы полностью переписать код. Если уж необходим именно такой перерыв в основной работе — вы можете не иметь возможности приступить к исправлениям немедленно. В таком случае следует уведомить пользователей, что на некоторый конкретный срок запланирована переделка кода, а также пояснить, как это может повлиять на их работу.

Порядок выполнения рефакторинга

Первыми рефакторингами кода начали заниматься программисты на языке Smalltalk. Когда мы писали первое издание этой книги, данная методика начала приобретать все больше приверженцев, возможно, благодаря посвященной этой теме книге *Refactoring: Improving the Design of Existing Code* [Fow19] (у которой недавно вышло второе издание[10]).

Рефакторинг, по существу, является его перепроектированием. Все, что проектируете вы и ваши коллеги, может быть перепроектировано в свете новых фактов, более глубокого понимания, изменения требований и т.д. Но если вы станете с неистовым рвением бросаться кромсать огромные куски кода, то можете оказаться в еще худшем положении, чем перед началом данной операции.

Очевидно, что рефакторинг — это вид деятельности, который следует выполнять медленно, осторожно и внимательно. Мартин Фаулер предлагает следующие простые рекомендации относительно того, как реорганизовывать код, чтобы не нанести вреда больше, чем принести пользы[11].

1. Не пытайтесь выполнять рефакторинг, попутно добавляя функциональные возможности.

2. Прежде чем приступать к рефакторингу, обязательно напишите качественные тесты. Выполняйте тестирование как можно чаще. Это позволит вам быстро выяснить, не нарушили ли что-нибудь внесенные вами изменения.

3. Продвигайтесь короткими, осторожными шагами: перенесите поле из одного класса в другой, разделите метод на части, переименуйте переменную. Рефакторинг зачастую подразумевает внесение множества мелких локализованных изменений, приводящих в итоге к крупномасштабным переменам. Если сохранять темп продвижения мелкими шагами и выполнять тесты после каждого сделанного шага, то можно избежать продолжительной отладки[12].

---

[10] Имеется русский перевод: Мартин Фаулер. *Рефакторинг кода на JavaScript: улучшение проекта существующего кода, 2-е изд.* — СПб.: "Диалектика", 2019.

[11] Первоначально они появились в книге *UML Distilled: A Brief Guide to the Standard Object Modeling Language* [Fow00].

[12] Это, в общем, превосходный совет (см. также раздел "Тема 27. Не опережайте свет фар вашего автомобиля" главы 4 "Прагматичная паранойя").

Автоматический рефакторинг

В первом издании этой книги отмечалось, что данная методика еще не вышла за пределы среды Smalltalk, хотя такое положение, вероятнее всего, изменится. Так оно и произошло, и теперь автоматический рефакторинг доступен во многих интегрированных средах разработки для большинства ведущих языков программирования.

Такие интегрированные среды разработки способны автоматически переименовывать переменные и методы, разделять длинную подпрограмму на более мелкие составляющие, распространять необходимые изменения по коду, помогать переносить код с помощью перетаскивания мышью и выполнять прочие полезные действия для рефакторинга кода.

О тестировании на данном уровне речь пойдет подробнее в разделе "Тема 41. Тестировать, чтобы кодировать" далее в этой главе, а о крупномасштабном тестировании — в разделе "Строгое и непрерывное тестирование" главы 9 "Прагматичные проекты". Но рекомендация Мартина Фаулера относительно проведения качественных регрессионных тестов имеет решающее значение для надежного рефакторинга кода.

Если приходится выходить за пределы рефакторинга кода и в конечном счете изменять внешнее поведение или интерфейсы, то такие действия могут привести к преднамеренному нарушению сборки, когда прежним клиентам данного кода не удастся его скомпилировать. Это позволит выяснить, что именно требуется обновить. И если в следующий раз вы обнаружите фрагмент кода, работающий не совсем так, как требуется, — исправьте его. Управляйте болью: ведь если она беспокоит вас теперь, то еще больше будет беспокоить в дальнейшем, так что лучше вылечить ее сразу и окончательно. Вспомните уроки из раздела "Тема 3. Программная энтропия" главы 1 "Философия прагматизма": нельзя жить с разбитыми окнами.

Другие разделы, связанные с данной темой

- **Тема 3**. Программная энтропия, **глава 1** "Философия прагматизма".
- **Тема 9**. DRY — пороки дублирования, **глава 2** "Прагматичный подход".
- **Тема 12**. Трассирующие пули, **глава 2** "Прагматичный подход".
- **Тема 27**. Не опережайте свет фар вашего автомобиля, **глава 4** "Прагматичная паранойя".
- **Тема 44**. Именование.
- **Тема 48**. Сущность гибкости, **глава 8** "До начала проекта".

Тема 41 Тестировать, чтобы кодировать

Первое издание этой книги было написано в более простые времена, когда большинство разработчиков не писали тесты. Он считали, что делать это не стоит, поскольку все равно в 2000 году наступит конец света. В первом издании книги был раздел, посвященный построению кода, который легко тестировать. Это был хитрый способ убедить разработчиков все же писать тесты.

Теперь же настали более просвещенные времена. Если еще и остались разработчики, которые до сих пор не пишут тесты, то они, по крайней мере, должны знать, что обязаны это делать.

Тем не менее, когда мы спрашиваем разработчиков, зачем они пишут тесты, они смотрят на нас так, как будто мы спросили, пользуются ли они до сих пор перфокартами, и обычно отвечают: "Чтобы убедиться в работоспособности кода", хотя и не говорят вслух, что задающий такие вопросы — "чайник" в программировании. Но мы все же считаем такой ответ неверным.

Так что же мы считаем самым важным в тестировании и как оно должно проводиться? Отвечая на эти вопросы, начнем со следующего смелого заявления:

> **Совет 66** Тестирование предназначено не для выявления программных ошибок

Мы считаем, что основные выгоды из тестирования извлекаются в том случае, когда тесты обдумываются и пишутся, а не тогда, когда они выполняются.

Обдумывание тестов

Представьте, что в понедельник утром вы приступаете к работе над каким-то новым кодом. Вам нужно написать код, запрашивающий в базе данных список людей, просматривающих больше 10 видеофильмов в неделю на вашем веб-сайте, где размещаются "самые забавные в мире видеофильмы о мытье посуды".

С этой целью вы запускаете свой текстовый редактор и приступаете к написанию следующей функции, выполняющей запрос:

```
def return_avid_viewers do
  # ... м-да! ...
end
```

Постойте! Откуда вы знаете, что поступаете правильно?

Ответ в том, что ни вы и никто другой не можете этого знать. Но это может стать более вероятным, если думать о тестах. И вот как это делается.

Представьте сначала, что завершили написание упомянутой выше функции и теперь вам нужно ее проверить. Как это сделать? Ведь для этого потребуются

какие-то тестовые данные, а следовательно, придется обратиться к управляемой вами базе данных. В этом вам могут, конечно, помочь некоторые каркасы, проводящие тесты с помощью некоторой тестовой базы данных. Но в данном случае вы должны передать своей функции отдельный экземпляр базы данных, а не нечто глобальное, чтобы во время тестирования можно было внести в нее какие-то изменения.

```
def return_avid_users(db) do
```

Затем необходимо подумать о том, как заполнить базу тестовыми данными. Ведь исходно требуется получить список людей, просматривающих больше 10 видеофильмов в неделю. Следовательно, в схеме базы данных необходимо найти те поля, которые могут помочь в реализации такого требования. В таблице базы данных действительно имеются поля `opened_video` и `completed_video`, которые могут помочь в получении искомого списка. Чтобы сформировать тестовые данные, нужно точно знать, каким полем можно воспользоваться. Но в исходном требовании об этом ничего конкретного не сказано, а бизнес-контакты отсутствуют. Поэтому придется пойти на обман и, чтобы проверить функцию (а возможно, даже изменить ее в дальнейшем), просто передать ей имя поля.

```
def return_avid_users(db, qualifying_field_name) do
```

Итак, мы с вами стали думать о тестах. Не написав ни строчки кода, мы уже сделали два открытия и воспользовались ими с целью изменить API рассматриваемой здесь функции.

Кодирование на основе тестов

В предыдущем примере, думая о тестировании, мы сумели уменьшить связывание в своем коде, передавая разрабатываемой функции подключение к конкретной базе данных вместо того, чтобы воспользоваться ее глобальной версией, а также повысили степень гибкости, сделав тестируемое поле параметром. Размышляя о написании теста для рассматриваемой здесь функции, мы сумели взглянуть на нее снаружи, поставив себя на место клиента, а не автора написанного кода.

Совет 67 Тест — первый пользователь вашего кода

На наш взгляд, самое главное преимущество, которое дает тестирование, заключается в том, что оно дает жизненно важную ответную реакцию, которая, по существу, управляет программированием. Функция или метод, тесно связанные с другим кодом, с трудом поддаются тестированию, поскольку при этом, прежде чем выполнять данную функцию или метод, приходится подготавливать

всю необходимую среду. Таким образом, делая свой код легко тестируемым, вы уменьшаете его связывание с другим кодом.

Прежде чем протестировать какой-нибудь код, его необходимо понимать. На первый взгляд это кажется нелепым, но на самом деле всем нам приходилось иметь дело с фрагментами кода, основанными на не полном понимании того, что мы должны делать. Мы часто уверяем себя, что по ходу дела нам удастся прояснить ситуацию, а в дальнейшем добавить код для поддержки граничных условий и обработки ошибок. В итоге код получается в пять раз длиннее, чем следует, поскольку он наполнен условной логикой и частными случаями. Но достаточно пролить тестовый свет на такой код, как дело проясняется. Если вы подумаете о тестировании граничных условий и их пригодности еще *до* того, как приступить к программированию, то сможете обнаружить в логике шаблоны, упрощающие создаваемую функцию. А если вы подумаете об условиях возникновения ошибок, которые нужно проверять, то соответственно структурируете вашу функцию.

Разработка на основе тестирования

В программировании существует направление, которое задается следующим вопросом: если все преимущества обдумывания тестов имеют такое первостепенное значение, то почему бы не писать в первую очередь именно тесты? Приверженцы этого направления практикуют методику, называемую *разработкой на основе тестирования* (test-driven development — TDD). Иногда ее еще называют *разработкой с предварительным тестированием*[13].

Ниже приведен основной цикл разработки на основе тестирования.

1. Наметить небольшую часть функциональных возможностей, которые требуется добавить.
2. Написать тест, который будет успешно пройден, как только будет реализована данная часть функциональных возможностей.
3. Выполнить все тесты. Убедиться, что не проходит лишь только что написанный тест.
4. Написать минимальный фрагмент кода, требующийся для прохождения данного теста, и убедиться, что теперь он благополучно проходит.
5. Реорганизовать код, выяснив, можно ли каким-то образом усовершенствовать то, что было только что написано (тест или проверяемая функция). По завершении убедиться, что тесты по-прежнему успешно проходят.

[13] Некоторые утверждают, что разработка на основе тестирования и разработка с предварительным тестированием являются разными методиками, говоря, что у них разные цели. Но исторически сложилось так, что разработка с предварительным тестированием, происходящая от экстремального программирования, оказалась идентична тому, что теперь принято обозначать сокращением TDD.

Смысл в том, что данный цикл должен быть кратким и выполняться в считанные минуты. А это означает, что тесты приходится писать постоянно и программировать так, чтобы они в конечном итоге были успешно пройдены. На наш взгляд, методика TDD выгодна в основном тем разработчикам, которые начинают работу над своим проектом с тестирования. Если вы будете придерживаться последовательности операций по приведенной выше методике, то можно гарантировать, что у вас всегда будут тесты для проверки вашего кода. И это означает, что вы будете всегда думать о тестах.

Нам, однако, приходилось встречать и таких людей, которые стали буквально рабами разработки на основе тестирования. Такое раболепие перед данной методикой проявляется по-разному.

- Они тратят непомерно много времени, чтобы обеспечить полное покрытие написанного кода тестами.
- У них накапливается много избыточных тестов. Например, прежде чем написать класс в первый раз, многие приверженцы разработки на основе тестирования, пишут тест, который просто ссылается на имя этого класса. Убедившись, что тест сбоит, они пишут определение пустого класса, после чего данный тест оказывается успешно пройденным. Но ведь этот тест совершенно ничего не проверяет! В очередном написанном тесте также делается ссылка на класс, а следовательно, предыдущий тест просто не нужен. Если в дальнейшем имя класса изменится, то в тесты придется внести больше изменений. А ведь это самый простой пример.
- Их проектирование обычно начинается снизу и продвигается вверх. (См. далее врезку "Проектирование снизу вверх и сверху вниз. Как это следует делать".)

Как бы там ни было, непременно применяйте в своей практике разработку на основе тестирования. Но при этом не забывайте периодически останавливаться, чтобы оценить общую картину. Ведь очень легко, соблазнившись сообщением "тест прошел" с зеленым индикатором, написать немало кода, который никак не приближает вас к решению поставленной задачи.

Применяя разработку на основе тестирования нужно знать, куда идти

В старой шутке спрашивается: "Как съесть слона?", на что дается лаконичный ответ: "По частям". Эта идея нередко превозносится как преимущество разработки на основе тестирования. Если нельзя решить задачу целиком, ее можно решить по частям, предпринимая мелкие шаги, тест за тестом. Но такой подход может вводить в заблуждение, поощряя сосредоточиваться на простых задачах, бесконечно оттачивая их решение и в то же время пренебрегая главной причиной написания кода.

Любопытный случай, характеризующий такой подход, произошел в 2006 году, когда Рон Джеффрис, видный активист движения за гибкость в разработке программного обеспечения, опубликовал целый ряд блог-постов, где он описал свой метод программирования решателя головоломок судоку на основе тестирования[14]. Пять блог-постов спустя он уточнил представление исходной доски для судоку, неоднократно выполняя реорганизацию кода до тех пор, пока его не удовлетворила объектная модель. Но затем он забросил этот проект. Любопытно прочитать его блог-посты по порядку, следя за тем, как этот умный человек отвлекался на незначительные мелочи, будучи зачарованным блеском успешных тестов.

Проектирование снизу вверх и сверху вниз. Как это следует делать

Еще в те времена, когда вычислительная техника была молода и беззаботна, существовали два направления в проектировании: сверху вниз и снизу вверх. Приверженцы проектирования сверху вниз говорят, что начинать следует с общей задачи и пытаться решить ее, разбивая на мелкие части, а те — на еще более мелкие части и так далее до тех пор, пока не будут достигнуты настолько мелкие части, что их можно выразить непосредственно в коде.

Приверженцы проектирования снизу вверх предпочитают строить код, как дом. Они начинают с создания такого уровня кода, который дает им некоторые абстракции, приближающие их к той задаче, которую они пытаются решить. Затем они вводят еще один уровень с более общими абстракциями. И так продолжают до тех пор, пока не достигнут конечного уровня с абстракцией, решающей поставленную задачу.

Ни одна из этих методик проектирования на самом деле не годится, поскольку в обеих игнорируется одна из самых важных особенностей разработки программного обеспечения: когда мы начинаем разработку, мы не знаем, что делаем. Сторонники проектирования сверху вниз считают, что они способны изначально выразить все требование в целом, хотя это сделать нельзя. А сторонники проектирования снизу вверх считают, что они способны составить перечень абстракций, который позволит им постепенно прийти к одному решению на верхнем уровне. Но как им выбрать функциональные возможности отдельных уровней, если они не знают, куда направляются?

Мы твердо убеждены, что строить программное обеспечением можно только инкрементно. Создавайте функциональные возможности небольшими частями, изучая по ходу дела решаемую задачу. Применяйте приобретенные знания по мере реализации кода, на каждой стадии привлекайте заказчиков, позволяя им направлять данный процесс.

[14] См. по адресу `https://ronjeffries.com/categories/sudoku`. Выражаем искреннюю благодарность Рону за то, что позволил нам привести здесь его историю.

Совет 68 Стройте комплексно, а не сверху вниз или снизу вверх

В противоположность этому Питер Норвиг описывает альтернативный подход совсем иного характера[15]. Вместо того чтобы руководствоваться тестами, он начинает с уяснения, каким образом такого рода задачи решались традиционно с помощью распространения ограничений, а затем сосредоточивается на своем алгоритме. В частности, он реализует доску в десятке строк кода, которые вытекают непосредственно из обсуждения обозначений, принятых в судоку.

Тесты определенно могут помочь в разработке. Но если не иметь в виду конечной цели, то в результате можно просто бесконечно ходить по кругу.

Возврат к коду

Компонентная разработка уже давно стала возвышенной целью проектирования программного обеспечения[16]. Ее идея состоит в том, что обобщенные программные компоненты должны быть доступны и должны объединяться так же просто, как и большинство интегральных схем (ИС). Но такая методика пригодна лишь в том случае, если заранее известно, что применяемые компоненты надежны и соответствуют принятым нормам, таким как общее напряжение, стандарты соединений, временны*е* характеристики и прочие параметры в производстве ИС.

ИС проектируются с учетом их испытаний, причем не только на заводе-изготовителе или на месте их установки на печатные платы, но и в месте их эксплуатации. В более сложных ИС и системах могут быть предусмотрены средства встроенного самоконтроля (Built-In Self Test — BIST), выполняющие внутреннюю диагностику на каком-нибудь элементарном уровне, или же механизм тестового доступа (Test Access Mechanism — TAM), предоставляющий средства тестирования, которые позволяют подавать на ИС заданные воздействия из внешней среды и собирать их ответные реакции.

То же самое можно сделать и в программном обеспечении. Как и нашим коллегам, разработчикам электронного оборудования, нам нужно с самого начала встраивать тестируемость в разрабатываемое программное обеспечение и тщательно проверять каждый из его компонентов, прежде чем соединять их вместе.

---

[15] См. по адресу `http://norvig.com/sudoku.html`.

[16] Такая методика была опробована еще в 1986 году, когда Кокс (Cox) и Новобильски (Novobilski) изобрели термин "программная интегральная схема", введя его в своей книге *Object-Oriented Programming: An Evolutionary Approach* [CN91] по языку Objective-C.

Модульное тестирование

Испытание оборудования на уровне ИС приблизительно равнозначно *модульному тестированию* программного обеспечения, когда каждый модуль тестируется по отдельности для проверки его поведения. Тщательно протестировав модуль в контролируемых (и даже искусственных) условиях, можно составить более ясное представление о том, как он отреагирует на окружающий мир.

Модульный тест программного обеспечения представляет собой код, испытывающий модуль. Как правило, модульный тест сначала настраивает определенного рода искусственную среду, а затем вызывает подпрограммы тестируемого модуля. Далее он проверяет возвращаемые результаты, сравнивая их с известными значениями или результатами предыдущих прогонов того же самого теста (регрессионное тестирование).

В дальнейшем, когда модули собираются как "программные ИС" в готовую систему и есть уверенность, что отдельные ее части работают должным образом, можно воспользоваться теми же средствами модульного тестирования для испытания всей системы в целом. Подробнее о такой крупномасштабной проверке системы речь пойдет в разделе "Строгое и непрерывное тестирование" главы 9 "Прагматичные проекты".

Но прежде чем зайти в тестировании так далеко, необходимо решить, что именно следует тестировать на модульном уровне. В исторической перспективе программисты раньше вводили несколько произвольных битов данных в код, анализировали операторы вывода и называли это тестированием. Но ведь для тестирования кода можно сделать намного больше.

Тестирование соответствия контракту

Мы предпочитаем рассматривать модульное тестирование как *тестирование соответствия контракту* (см. раздел "Тема 23. Проектирование по контракту" главы 4 "Прагматичная паранойя"). При этом требуется написать такой тест, который гарантирует, что проверяемый модуль выполняет свой контракт. Такой тест позволит выяснить следующее: соответствует ли написанный код контракту и отвечает ли сам контракт своему назначению. Требуется также проверить, предоставляет ли тестируемый модуль обещанные им функциональные возможности в обширном ряде контрольных примеров и граничных условий.

Что же все это означает на практике? Начнем ответ на этот вопрос с простого числового примера: функции извлечения квадратного корня. Ее документированный контракт довольно прост:

```
предусловия:
   argument >= 0;

постусловия:
   ((result * result) - argument).abs <= epsilon*argument;
```

Это нам подсказывает, что в тесте необходимо сделать следующее.

- Передать отрицательный аргумент и убедиться, что он отвергается.
- Передать нулевой аргумент и убедиться, что он принимается (это граничное значение).
- Передать значения в пределах от нуля до максимального выражаемого аргумента и убедиться, что разность результата извлечения квадратного корня и исходного аргумента меньше некоторой мелкой дробной части аргумента.

Вооружившись этим контрактом и считая, что рассматриваемая здесь функция извлечения квадратного корня самостоятельно проверяет свои пред- и постусловия, можно написать тестовый сценарий, чтобы запустить его на выполнение и испытать данную функцию, как показано ниже.

```
assertWithinEpsilon(my_sqrt(0), 0)
assertWithinEpsilon(my_sqrt(2.0), 1.4142135624)
assertWithinEpsilon(my_sqrt(64.0), 8.0)
assertWithinEpsilon(my_sqrt(1.0e7), 3162.2776602)
assertRaisesException fn => my_sqrt(-4.0) end
```

Это довольно простой пример, но на практике любой нетривиальный модуль, вероятно, будет зависеть от целого ряда других модулей. Так как же протестировать их вместе?

Допустим, имеется модуль A, в котором применяются модули `DataFeed` и `LinearRegression`. В таком случае мы должны выполнить тестирование в следующем порядке.

1. Контракт модуля `DataFeed`, полностью.
2. Контракт модуля `LinearRegression`, полностью.
3. Контракт модуля A, который полагается на другие контракты, хотя и не раскрывает их непосредственно.

Для такого рода тестирования требуется, чтобы сначала были проверены модули, подчиненные тестируемому модулю. Как только эти подчиненные модули будут проверены, можно тестировать и основной модуль.

Если тесты модулей `DataFeed` и `LinearRegression` пройдут, а тест модуля A не пройдет, можно с уверенностью сказать, что причина сбоя кроется в модуле A или же в *применении* одного из подчиненных ему модулей. Такая методика тестирования позволяет существенно сэкономить на отладке, поскольку дает возможность быстро сосредоточиться на вероятном источнике сбоя в модуле A и не тратить зря время на перепроверку подчиненных ему модулей.

К чему все эти хлопоты? Прежде всего, требуется избежать создания "бомбы замедленного действия", которая может оказаться заложенной в исходном коде и незамеченной, взорвавшись в самый неподходящий момент позже в проекте.

Делая акцент на тестировании соответствия контракту, мы можем попытаться избежать стольких бед по ходу дальнейшего выполнения проекта, сколько удастся.

Совет 69 Проектируйте с учетом тестирования

Специальное тестирование

Тестирование называется *специальным* (ad hoc) при ручном дополнении в конкретном месте исходного кода. Это может быть простой вызов `console.log()` или фрагмент кода, введенный в диалоговом режиме работы отладчика, интегрированной среде разработки или цикле "чтение–вычисление–вывод" (REPL).

В конце сеанса отладки такой специальный тест требуется формализовать. Если в коде один раз произошла неприятность — их, вероятнее всего, будет больше. Не отбрасывайте написанный вами специальный тест, а добавьте его в арсенал своих средств модульного тестирования.

Создание тестового окна

Даже самые лучшие наборы тестов вряд ли сумеют обнаружить все программные ошибки. Выявить их могут некоторые особые условия эксплуатационной среды.

Это означает, что как только любой компонент программного обеспечения будет развернут в реальных условиях его эксплуатации, его обязательно следует проверить с помощью потока реальных данных. В отличие от печатной платы или ИС, в программном обеспечении нет контрольных выводов. Тем не менее *можно* предоставить различные представления внутреннего состояния модуля, не прибегая к отладчику, что может быть неудобно или вообще невозможно в условиях промышленной эксплуатации.

Одним из таких механизмов являются журнальные файлы, содержащие сообщения трассировки. Эти сообщения должны быть представлены в точном, последовательном формате. Возможно, их придется проанализировать автоматически, чтобы выявить время обработки или логические пути, которыми пошла тестируемая программа. Неудачно или несогласованно отформатированная диагностическая информация содержит столько лишнего, что ее трудно читать и бесполезно анализировать.

Еще одним механизмом проникновения в выполняющийся код является последовательность "горячих" клавиш или "магический" URL. Когда нажимается конкретная комбинация клавиш или осуществляется доступ к веб-ресурсу по указанному URL, появляется окно диагностического контроля с сообщениями о состоянии и т.д. И хотя это совсем не то, что обычно демонстрируется конечным пользователям, подобная информация может весьма пригодиться службе

технической поддержки. В более общем смысле можно было бы воспользоваться *переключателем функциональных средств*, чтобы активизировать дополнительную диагностику для отдельного пользователя или категории пользователей.

Культура тестирования

Все программы, которые вы пишете, должны быть *непременно* протестированы, если не вами и вашей командой, то хотя бы конечными пользователями, поэтому вы должны тщательно планировать стадию тестирования. Достаточно проявить немного предусмотрительности, чтобы свести к минимуму затраты на сопровождение и вызовы службы технической поддержки.

На самом деле у вас имеются лишь следующие возможности.

- Тестировать предварительно.
- Тестировать по ходу программирования.
- Вообще не тестировать.

Предварительное тестирование, включая и разработку на основе тестирования, вероятно, будет наилучшим вариантом выбора в большинстве случаев, поскольку оно гарантирует, что тестирование непременно произойдет. Но иногда оно оказывается неудобным или бесполезным, и тогда лучше выбрать тестирование по ходу программирования. В этом случае можно написать фрагмент кода, поработать с ним, написать для него тесты, а затем перейти к следующему фрагменту кода. Самый худший вариант выбора нередко называют "протестируем потом". Не будем себя обманывать, поскольку это, по существу, означает "не тестировать вовсе".

Культура тестирования означает прохождение всех тестов в любое время. Если не обращать внимания на те тесты, которые "никогда не проходят", то легче пренебречь *всеми* тестами вообще. И тогда возникнет порочный круг (см. раздел "Тема 3. Программная энтропия" главы 1 "Философия прагматизма").

Уделяйте тестовому коду такое же внимание, как и любому рабочему коду. Поддерживайте его развязанным, ясным и надежным. Не полагайтесь на такие ненадежные вещи, как, например, абсолютное положение виджетов в системе с GUI, точность отметок времени в журнале регистрации событий на сервере или подробности описания ошибок в сообщениях (см. выше раздел "Тема 38. Программирование по совпадению"). Тестирование такого рода вещей приведет к тому, что тесты окажутся хрупкими.

Совет 70 Тестируйте свои программы сами, иначе их будут тестировать пользователи

Признание

Я (Дэйв) однажды заявил, что больше не собираюсь писать тесты. И сделал я это отчасти для того, чтобы поколебать веру тех, кто превратили тестирование в религию, а отчасти потому, что это было в какой-то степени правдой.

Я программирую уже 45 лет и больше 30 лет из них пишу тесты. Думать о тестах во время программирования уже давно вошло в мою привычку. Мне так удобно. Но мой внутренний голос подсказывает, что когда я начинаю чувствовать себя удобно, я должен заняться чем-то другим.

В данном случае я решил на пару месяцев перестать писать тесты, чтобы посмотреть, как это повлияет на мой код. К моему искреннему удивлению, ничего особенного не произошло, поэтому я решил уделить время выяснению причин.

На *мой* взгляд, все дело в том, что наибольшая польза от тестирования проистекает от обдумывания тестов и их воздействия на код. И если практиковать это долго, то можно научиться думать о тестах, вообще не составляя их. Мой код по-прежнему оставался тестируемым, он просто не тестировался.

Но при таком подходе во внимание не принимается то обстоятельство, что тесты служат также средством общения с другими разработчиками. Поэтому теперь я *пишу* тесты для кода, общего с другими разработчиками или опирающегося на особенности внешних зависимостей.

Энди не советовал мне включать эту врезку в книгу. Он предостерегал, что мое признание может искусить неопытных разработчиков не тестировать свой код. Поэтому я предлагаю следующее компромиссное решение: должны ли вы писать тесты? Да, должны. Но после такой практики в течение 30 лет можете немного поэкспериментировать, чтобы посмотреть, что вам выгоднее: писать тесты или не писать.

Не сомневайтесь, что тестирование является неотъемлемой частью программирования. Эту стадию процесса разработки нельзя поручать другим подразделениям. Тестирование, проектирование, написание кода — все это входит в само программирование.

Другие разделы, связанные с данной темой

- **Тема 27**. Не опережайте свет фар вашего автомобиля, **глава 4** "Прагматичная паранойя".
- **Тема 51**. Начальный набор инструментальных средств программиста-прагматика, **глава 9** "Прагматичные проекты".

Тема 42 Тестирование на основе свойств

"Доверяй, но проверяй".

Русская пословица

Мы рекомендуем писать модульные тесты для создаваемых функций. Для этого следует обдумать типичные причины, которые могут вызвать осложнения, опираясь на знание того, что именно тестируется.

Но здесь возникает небольшое, но потенциально важно затруднение. Если вы пишете код, а вместе с ним и тесты, то могут ли и там, и там быть выражены неверные предположения? Код проходит тесты, поскольку он делает именно то, что и должен делать, — исходя из того, как вы это понимаете.

Из этого затруднения можно, в частности, выйти, поручив писать тесты и проверяемый код разным людям, но нам такое решение не по душе. Как пояснялось ранее в разделе "Тема 41. Тестировать, чтобы кодировать", одно из главных преимуществ обдумывания тестов заключается в том, что оно дает нужные сведения о написанном коде. Эти сведения теряются, если работа по тестированию и программированию разделяется. Вместо этого мы отдаем предпочтение автоматизации тестирования на компьютере, который действует, в отличие от людей, непредубежденно.

Контракты, инварианты и свойства

В разделе "Тема 23. Проектирование по контракту" главы 4 говорилось о том, что у написанного кода имеются *контракты*, которым он должен соответствовать. Это, по существу, условия, которым должен удовлетворять код, когда ему подаются на обработку входные данные, а также определенные гарантии, которые он дает относительно получаемых в нем результатов или выходных данных. У написанного кода имеются также *инварианты*, т.е. представления о какой-то части состояния, которые остаются истинными, когда оно проходит через функцию. Так, если сортируется список, то в результате данной операции в нем должно оказаться такое же количество элементов, как и первоначально, т.е. длина списка инвариантна.

Как только контракты и инварианты будут составлены, а мы склонны называть их вместе *свойствами*, ими можно воспользоваться для автоматизации тестирования. И в конечном счете такая методика называется *тестированием на основе свойств*.

> **Совет 71** Пользуйтесь тестами на основе свойств для проверки правильности ваших предположений

В качестве искусственного примера можно написать ряд тестов для отсортированного списка. Мы уже установили одно свойство: длина отсортированного списка остается такой же, как и у первоначального. Можно также утверждать, что ни один элемент результирующего списка не может быть больше следующего после него элемента.

Это можно теперь выразить непосредственно в коде. В большинстве языков программирования имеется особого рода библиотека или каркас для поддержки тестирования на основе свойств. И хотя данный пример реализован на языке Python с помощью инструментального средства Hypothesis и каркаса pytest, основные принципы остаются теми же и довольно универсальными.

Ниже приведен весь исходный код тестов на основе свойств.

proptest/sort.py

```
from hypothesis import given
import hypothesis.strategies as some

@given(some.lists(some.integers()))
def test_list_size_is_invariant_across_sorting(a_list):
    original_length = len(a_list)
    a_list.sort()
    assert len(a_list) == original_length

@given(some.lists(some.text()))
def test_sorted_result_is_ordered(a_list):
    a_list.sort()
    for i in range(len(a_list) - 1):
        assert a_list[i] <= a_list[i + 1]
```

А вот что произойдет, если выполнить данный код:

```
$ pytest sort.py
======================== test session starts ========================
...
plugins: hypothesis-4.14.0

sort.py ..                                                    [100%]

===================== 2 passed in 0.95 seconds ======================
```

Ничего драматического здесь вроде бы не наблюдается. Но подспудно инструментальное средство Hypothesis выполнило эти тесты сотню раз, передавая каждый раз другой список на сортировку. Длина этих списков будет разной, как, впрочем, и их содержимое. Это все равно, что состряпать 200 отдельных тестов с 200 произвольными списками.

Генерация тестовых данных

Как и большинство библиотек для тестирования на основе свойств, Hypothesis дает описание на мини-языке тех данных, которые следует сформировать. Такой язык основывается на вызовах функций из модуля `hypothesis.strategies`,

у которого имеется псевдоним some, употребляемый только ради повышения удобочитаемости.

Так, если написать следующую строку кода:

```
@given(some.integers())
```

тестовая функция будет выполнена многократно и всякий раз ей будет передано иное целочисленное значение. Если же вместо этого написать приведенную ниже строку кода, то в конечном счете будет получен ряд четных чисел в пределах от 10 до 20.

```
@given(some.integers(min_value=5, max_value=10).map(lambda x: x * 2))
```

Кроме того, обе упомянутые выше строки можно составить таким образом, чтобы получить в итоге списки натуральных чисел длиной не меньше 100 элементов, как показано ниже.

```
@given(some.lists(some.integers(min_value=1), max_size=100))
```

Данный пример не предназначен в качестве упражнения в применении какой-нибудь конкретной библиотеки или каркаса. Поэтому опустим здесь многие несущественные подробности и вместо этого рассмотрим реальный пример.

Выявление неудачных допущений

Итак, предположим, что разрабатывается простая система обработки заказов и контроля запасов, поскольку всегда есть место для их пополнения. В этой системе моделируются уровни запасов с помощью объекта Warehouse, представляющего склад. К такому складу можно обратиться с запросом, чтобы выяснить наличие запасов какого-нибудь товара, изъять товары из запасов и получить текущие уровни запасов.

Ниже показано, как это реализуется непосредственно в коде.

proptest/stock.py

```
class Warehouse:
    def __init__(self, stock):
        self.stock = stock

    def in_stock(self, item_name):
        return (item_name in self.stock)
                and (self.stock[item_name] > 0)

    def take_from_stock(self, item_name, quantity):
        if quantity <= self.stock[item_name]:
            self.stock[item_name] -= quantity
        else:
            raise Exception("Oversold {}".format(item_name))

    def stock_count(self, item_name):
        return self.stock[item_name]
```

А вот как выглядит элементарный модульный тест, который написан для проверки данного кода и который данный код успешно проходит:

proptest/stock.py

```
def test_warehouse():
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    assert wh.in_stock("shoes")
    assert wh.in_stock("hats")
    assert not wh.in_stock("umbrellas")

    wh.take_from_stock("shoes", 2)
    assert wh.in_stock("shoes")

    wh.take_from_stock("hats", 2)
    assert not wh.in_stock("hats")
```

Теперь перейдем к функции, написанной для обработки запросов на заказ товаров со склада. Эта функция возвращает кортеж, где первым элементом является признак "ok" (товар имеется) или "not available" (товар отсутствует), а вторым — количество запрашиваемого товара. Для проверки данной функции написаны также тесты, которые проходят. Исходный код самой функции и ее тестов приведен ниже.

proptest/stock.py

```
def order(warehouse, item, quantity):
    if warehouse.in_stock(item):
        warehouse.take_from_stock(item, quantity)
        return ( "ok", item, quantity )
    else:
        return ( "not available", item, quantity )
```

proptest/stock.py

```
def test_order_in_stock():
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    status, item, quantity = order(wh, "hats", 1)
    assert status   == "ok"
    assert item     == "hats"
    assert quantity == 1
    assert wh.stock_count("hats") == 1

def test_order_not_in_stock():
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    status, item, quantity = order(wh, "umbrellas", 1)
    assert status   == "not available"
    assert item     == "umbrellas"
    assert quantity == 1
    assert wh.stock_count("umbrellas") == 0
```

```
def test_order_unknown_item():
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    status, item, quantity = order(wh, "bagel", 1)
    assert status   == "not available"
    assert item     == "bagel"
    assert quantity == 1
```

На первый взгляд все выглядит вроде бы прекрасно. Но прежде чем выпустить рассматриваемый здесь код, дополним его некоторыми тестами на основании свойств.

Нам, в частности, известно, что запасы на складе не могут появляться и исчезать в течение транзакции. Это означает, что если взять некоторые товары со склада, количество взятых товаров плюс оставшиеся на складе должно быть таким же, как и первоначальное их количество. Поэтому приведенный ниже тест выполняется с произвольно выбранными значениями от `"hats"` (шапки) до `"shoes"` (туфли) параметра `item` (товар), а также значениями от 1 до 4 параметра `quantity` (количество).

proptest/stock.py

```
@given(item = some.sampled_from(["shoes", "hats"]),
       quantity = some.integers(min_value=1, max_value=4))
def test_stock_level_plus_quantity_equals_original_stock_level(
              item, quantity):
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    initial_stock_level = wh.stock_count(item)
    (status, item, quantity) = order(wh, item, quantity)
    if status == "ok":
        assert wh.stock_count(item) + quantity == initial_stock_level
```

Если выполнить данный тест, то в итоге будет получен приведенный ниже результат. При попытке изъять три шапки со склада, где хранятся лишь две шапки, данный тест завершится неудачно в функции `warehouse.take_from_stock()`.

```
$ pytest stock.py
. . .
stock.py:72:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
stock.py:76:
     in test_stock_level_plus_quantity_equals_original_stock_level
           (status, item, quantity) = order(wh, item, quantity)
stock.py:40: in order
     warehouse.take_from_stock(item, quantity)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
self = <stock.Warehouse object at 0x10cf97cf8>, item_name = 'hats'
quantity = 3

    def take_from_stock(self, item_name, quantity):
        if quantity <= self.stock[item_name]:
            self.stock[item_name] -= quantity
```

```
        else:
>         raise Exception("Oversold {}".format(item_name))
E         Exception: Oversold hats

stock.py:16: Exception
---------------------------- Hypothesis ----------------------------
Falsifying example:
    test_stock_level_plus_quantity_equals_original_stock_level(
        item='hats', quantity=3)
```

В ходе рассматриваемого здесь тестирования на основе свойств было обнаружено ошибочное предположение: в функции `in_stock()` проверяется лишь наличие на складе хотя бы одной единицы запрашиваемого товара. А ведь нужно убедиться, что количества запрашиваемого товара на складе достаточно, чтобы выполнить заказ, как показано ниже.

proptest/stock1.py

```
    def in_stock(self, item_name, quantity):
➤       return (item_name in self.stock)
                and (self.stock[item_name] >= quantity)
```

Кроме того, необходимо внести изменения в функцию `order()`, как приведено ниже. И тогда тест на основе свойств будет успешно пройден.

proptest/stock1.py

```
    def order(warehouse, item, quantity):
➤       if warehouse.in_stock(item, quantity):
            warehouse.take_from_stock(item, quantity)
            return ( "ok", item, quantity )
        else:
            return ( "not available", item, quantity )
```

Тесты на основе свойств способны удивлять

В предыдущем примере тест на основе свойств служил для того, чтобы проверить правильность коррекции уровней запасов. И хотя этот тест обнаружил программную ошибку, он все же не имел никакого отношения к коррекции уровней запасов. Вместо этого он обнаружил программную ошибку в функции `in_stock()`.

В этом состоит сила и слабость тестирования на основе свойств. Его сила состоит в том, что сначала вы устанавливаете ряд правил для формирования входных данных, задаете некоторые утверждения для проверки достоверности выходных данных, а затем просто даете ему выполняться. Вы никогда точно не знаете, чем все закончится. Тест может пройти. Утверждение может не выполняться. Код вообще может не суметь обработать предоставленные ему входные данные.

Слабость же заключается в том, что выяснение причины сбоя может оказаться нелегким делом.

Если тест на основе свойств не проходит, мы рекомендуем сначала выяснить, какие именно параметры были переданы тестовой функции, а затем воспользоваться их значениями для составления отдельного обычного модульного теста. Такой тест автоматически выполняет две функции. Во-первых, он позволяет сосредоточиться на обнаруженной неполадке, не делая дополнительные вызовы в проверяемом коде с помощью каркаса для тестирования на основе свойств. И во-вторых, такой модульный тест действует как *регрессионный*. Тесты на основе свойств формируют произвольные значения, передаваемые данному тесту, и поэтому нет никакой гарантии, что одни и те же значения будут использованы при выполнении тестов в следующий раз. Наличие же модульного теста, заставляющего использовать одни и те же значения, гарантирует, что данная программная ошибка не будет пропущена тестом.

Тесты на основе свойств помогают проектировать

Обсуждая особенности модульного тестирования, мы отмечали как одно из главных его преимуществ возможность посмотреть на написанный код и его API с точки зрения модульного теста как первого клиента. Это же можно сказать и о тестах на основе свойств, хотя и несколько иначе. Они заставляют рассматривать написанный код с точки зрения инвариантов и контрактов, т.е. думать о том, что не должно изменяться и что должно оставаться истинным. Такое дополнительное осмысление оказывает волшебное воздействие на код, устраняя крайние случаи и выделяя функции, оставляющие данные в несогласованном состоянии.

Мы полагаем, что тестирование на основе свойств дополняет модульное тестирование. Эти методики тестирования решают разные задачи, и каждая из них приносит свои выгоды. Если вы пока еще не пользуетесь ими, опробуйте их на практике.

Другие разделы, связанные с данной темой

- **Тема 23**. Проектирование по контракту, **глава 4** "Прагматичная паранойя".
- **Тема 25**. Утвердительное программирование, **глава 4** "Прагматичная паранойя".
- **Тема 45**. Западня требований, **глава 8** "До начала проекта".

Упражнения

31. Обратитесь снова к рассмотренному ранее примеру со складом. Имеются ли какие-нибудь другие свойства, которые вы могли бы протестировать?

32. Допустим, ваша компания поставляет машинное оборудование. Каждая машина упакована в ящик, имеющий прямоугольную форму и разные размеры. Ваша задача — написать некоторый код, чтобы упаковать как можно больше ящиков в одном слое в кузове грузовика. Написанный вами код должен выводить список всех ящиков. Из этого списка можно определить место для каждого ящика в грузовике наряду с шириной и высотой этого ящика. Какие свойства вывода можно было бы протестировать?

Задачи

- Подумайте о коде, над которым вы работаете в настоящий момент. Каковы его свойства — контракты и инварианты? Можете ли вы воспользоваться каркасом для тестирования на основе свойств, чтобы проверить эти свойства автоматически?

Тема 43 Будьте осторожны

"Хороший сосед начинается с высокого забора".

Роберт Фрост[17], ***Ремонт стены*** *(Mending a Wall)*

При обсуждении связывания кода в первом издании этой книги мы сделали смелое и наивное заявление, что разработчикам не следует быть чрезмерно подозрительными, как шпионы и диссиденты, но мы оказались не правы. На самом деле разработчикам необходимо не терять бдительность каждый день.

Когда мы работали над вторым изданием этой книги, ежедневные новости были полны историями об опустошительных взломах информационных систем, подвергшихся атакам злоумышленников и кибермошенничества. При этом крадутся сотни миллионов записей за один раз, наносятся миллиардные убытки, а на восстановление систем после атак требуются не меньшие затраты, и эти цифры быстро растут каждый год. В подавляющем большинстве случаев это происходит не потому, что атакующие злоумышленники оказались особенно смышлеными или даже компетентными, а потому, что разработчики проявили беспечность.

Другие 90%

Во время программирования вы неоднократно проходите путь от "это работает!" до "почему это не работает?", а иногда и до "каким образом это могло произойти?..."[18]. Добравшись до желанной вершины после стольких подъемов и

[17] Robert Frost — один из самых известных американских поэтов, четырехкратный лауреат Пулитцеровской премии. — *Примеч. пер.*

[18] См. раздел "Тема 20. Отладка" главы 2 "Прагматичный подход".

падений, можно, вздохнув с облегчением, сказать: "Уф, наконец-то все работает!" и объявить, что код готов. Разумеется, он еще не готов полностью. Работа выполнена на 90%, но теперь предстоят еще *другие* 90%.

Далее необходимо проанализировать код на возможные сбои и ввести их условия в набор тестов. При этом следует рассмотреть такие ситуации, как передача неверных параметров, утечка или недоступность ресурсов и т.п.

В старые добрые времена такого оценивания внутренних ошибок могло быть достаточно, но сейчас это лишь начало. Ведь помимо ошибок из-за внутренних причин, необходимо выяснить, каким образом внешнее действующее лицо могло бы намеренно вывести систему из строя. На это вы можете возразить: "Да никого данный код вообще не интересует, он не так важен, и вряд ли кому-нибудь вообще известен этот сервер..." Но окружающий мир велик и по большей части взаимосвязан, а значит, в нем всегда найдутся изнывающие от скуки школьники, поощряемые государством террористы, банды преступников, корпоративные шпионы или даже мстительные бывшие сотрудники, нацеленные взломать вашу систему. Время выживания устаревшей, давно не обновлявшейся системы в открытой сети измеряется минутами, если не меньше. Безвестность никак не защищает.

Основные принципы защиты

Программистам-прагматикам присуща изрядная доля здоровой подозрительности. Им хорошо известно, что любая система несовершенна и подвержена отказам, и поэтому атакующие извне злоумышленные готовы проникнуть в любую брешь, оставленную в защите системы, чтобы вывести ее из строя. Конкретные среды разработки и развертывания также испытывают потребность в защите, и в связи с этим ниже приведен ряд основных принципов, которые следует всегда иметь в виду.

1. Минимизация площади поверхности атак.
2. Принцип наименьших привилегий.
3. Безопасные значения настроек по умолчанию.
4. Шифрование секретных данных.
5. Организация регулярного обновления системы безопасности.

Рассмотрим каждый из этих принципов защиты по отдельности.

Минимизация площади поверхности атак

Площадь поверхности атак на систему — это суммарное количество всех точек доступа, где атакующий злоумышленник может ввести или извлечь данные, а также вызвать выполнение службы. Ниже приведены характерные примеры направлений атак.

- **Сложность кода.** Чрезмерная сложность кода расширяет поверхность атак, давая больше возможностей для проявления непредвиденных побочных эффектов. Сложный код способен сделать такую поверхность более проницаемой для атак и открытой для заражения вирусами. И в этом случае чем проще и компактнее код, тем лучше. Кроме того, чем меньше кода, тем меньше программных ошибок и возможностей проникновения сквозь бреши в защите. Наконец, более простой, краткий и менее сложный код легче анализировать и находить в нем потенциальные слабости.

- **Входные данные.** Никогда не доверяйте данным из внешнего источника и всегда подвергайте их саночистке, прежде чем передавать их на хранение в базу данных, воспроизводить для просмотра или обрабатывать[19]. Некоторые языки программирования могут в этом помочь. Например, в языке Ruby переменные, хранящие вводимые извне данные, считаются *загрязненными*, что ограничивает круг операций над ними. Например, в приведенном ниже фрагменте кода явно используется команда `wc`, выводящая количество символов в файле, имя которого предоставляется во время выполнения.

safety/taint.rb

```
puts "Enter a file name to count: "[20]
name = gets
system("wc -c #{name}")
```

- Бесчестный пользователь мог нанести таким кодом следующий ущерб:

```
Enter a file name to count:
test.dat; rm -rf /
```

- Но, если установить первый уровень безопасности (SAFE), внешние данные станут загрязненными, а это означает, что их нельзя использовать в опасном контексте, как показано ниже.

safety/taint.rb

```
➤   $SAFE = 1

    puts "Enter a file name to count: "
    name = gets
    system("wc -c #{name}")

    ~~~ session $ ruby taint.rb Enter a file name to count:
```

[19] Помните нашего хорошего знакомого малыша Бобби Тэйблса (Bobby Tables — персонажа веб-комикса, доступного по адресу `https://xkcd.com/327`)? Пока вы вспоминаете, зайдите на сайт по адресу `https://bobby-tables.com`, где перечислены способы саночистки данных, передаваемых по запросам базы данных.

[20] `Введите имя файла для подсчета`

```
test.dat; rm -rf /
code/safety/taint.rb:5:in system':
Insecure operation - system (SecurityError) from
code/safety/taint.rb:5:in main' ~~~[21]
```

- **Неаутентифицированные службы.** По своей природе любой пользователь где угодно в мире может вызывать неаутентифицированные службы, и поэтому запрет или ограничение доступа к таким службам сразу же создает возможность, по крайней мере, для атаки типа *отказа в обслуживании*. Появившееся недавно довольно большое число брешей в общедоступных данных было обусловлено тем, что разработчики ненамеренно размещали данные в неаутентифицированных, открыто доступных для чтения информационных хранилищах, организованных в "облаке".
- **Выходные данные.** Существует (возможно, апокрифическая) история о системе, которая прилежно выдавала сообщение об ошибке `"Password is used by another user"` (Этот пароль используется другим пользователем). Не разглашайте секретные данные. Убедитесь, что сообщаемые вами данные пригодны для авторизации данного пользователя. Обрезайте или скрывайте потенциально рискованные сведения, например, номер социального страхования или другие идентификационные номера.
- **Отладочная информация.** Вряд ли вас обрадует появление на экране местного банкомата, киоска саморегистрации в аэропорту или на сбойной веб-странице результатов трассировки стека. Сведения, предназначенные для упрощения отладки, могут также упростить несанкционированное проникновение в систему. Поэтому убедитесь, что любое "тестовое окно" (см. выше раздел "Построение тестового окна") и сообщение об исключении во время выполнения защищено от любопытных глаз соглядатаев[22].

Совет 72 Не усложняйте код и минимизируйте поверхности атак

Принцип наименьших привилегий

Еще один важный принцип состоит в использовании *наименьших* привилегий в *кратчайший* период времени. Это означает, что пользователю не предоставляется наивысший уровень привилегий, например `root` (корневой) или `Administrator` (административный). Если же требуется столь высокий уро-

[21] Небезопасная операция в системе (`SecurityError`) из файла `code/safety/taint.rb:5:in main' ~~~`

[22] Данная методика оказалась успешной на уровне микросхем процессоров, где хорошо известные взломы эксплуатировали отладку и средства администрирования. Как только машина взломана, она оказывается полностью доступной злоумышленнику.

вень привилегий, он предоставляется для выполнения минимального объема работы, а затем быстро освобождается с целью сократить риск. Появление такого принципа относится к началу 1970-х годов.

> *"Всякая программа и каждый привилегированный пользователь системы должны работать, пользуясь наименьшими привилегиями, требующимися для выполнения конкретного задания".*
>
> *— Джером Зальтцер (Jerome Saltzer), журнал* ***Communications of the ACM****, 1974 г.*

Возьмем, например, программу `login` в производных от Unix системах. Она выполняется с корневыми привилегиями. Но, как только эта программа завершает аутентификацию правильно установленного пользователя, она сразу сбрасывает привилегии высокого уровня до пользовательских привилегий.

Данный принцип распространяется не только на уровни привилегий в операционной системе. Так, если в приложении реализуются разные уровни доступа (например, "администратор" и "пользователь"), следует предусмотреть более мелкие градации уровней доступа, где уязвимые ресурсы разделены на разные категории, а отдельные пользователи наделяются привилегиями только для некоторых из этих категорий.

Такая методика придерживается того же принципа минимизации площади поверхности атак, когда пределы направлений атак сокращаются как по времени, так и по уровню привилегий. И в данном случае, действительно, соблюдается правило "лучше меньше, да лучше".

Безопасные значения настроек по умолчанию

Значения настроек по умолчанию приложения или веб-сайта для пользователей должны быть *самыми* безопасными. И хотя такие значения могут быть не особенно удобны или пригодны для пользователей, тем не менее каждому пользователю должна быть предоставлена возможность самостоятельно найти компромисс между безопасностью и удобством.

Например, вводимый пароль по умолчанию может скрываться, когда вместо вводимых с клавиатуры символов на экране отображаются звездочки. Такой режим благоразумно установить для ввода пароля в общественных людных местах или для демонстрации приложений в большой аудитории. Но тем категориям пользователей, которым требуются специальные возможности, вероятно, придется указывать вводимый пароль прописью.

Шифрование секретных данных

Удостоверяющие личность сведения, информация о финансовом состоянии, пароли и прочие учетные данные нельзя хранить в формате простого текста,

будь то в базе данных или в каком-нибудь другом внешнем файле. Чтобы такие данные не были раскрыты, их следует зашифровать, обеспечив тем самым дополнительный уровень их защиты.

В разделе "Тема 19. Контроль версий" главы 3 "Основные инструментальные средства" настоятельно рекомендовалось ставить все или *почти* все, что требуется для проекта, под систему управления версиями. Из этого правила имеется одно важное исключение: не регистрировать секретные данные, ключи API, ключи сетевого протокола SSH, зашифрованные пароли и прочие учетные данные вместе с исходным кодом, когда он возвращается после правки в систему контроля версий. Управление ключами и секретными сведениями должно осуществляться отдельно через файлы конфигурации или переменные окружения как отдельная стадия процесса сборки и развертывания.

Организация регулярного обновления системы безопасности

Обновление вычислительных систем может доставить немало хлопот. Так, вставка "заплаты" для обновления системы безопасности может нарушить в качестве побочного эффекта нормальную работу какой-нибудь части приложения. В таком случае можно было бы принять решение отложить обновление до более удобного момента. Но такое решение никуда не годится, поскольку приложение остается уязвимым к известной эксплуатирующей попытке взлома до тех пор, пока не будут обновлены его системы безопасности.

Совет 73 Устанавливайте обновления системы безопасности незамедлительно

Такой совет касается любого подключенного к сети устройства, включая мобильные телефоны, автомобили, бытовую электронику, персональные компьютеры, компьютеры для разработки и сборки программ, рабочие серверы и "облачные" образы — одним словом, все. И если вы думаете, что это не так уж и важно, запомните, что самые крупные взломы данных в истории происходили потому, что информационные системы своевременно не обновлялись. Не доводите того, чтобы нечто подобное случилось и с вами.

Здравый смысл и криптография

Очень важно иметь в виду, что здравый смысл может вас подвести, когда речь идет о криптографии. Первое и самое главное правило в отношении криптографии гласит: *никогда не реализуйте ее самостоятельно*[23].

[23] Если, конечно, вы не обладаете ученой степенью в области криптографии. Но даже в этом случае вы не должны этого делать без проверок со стороны коллег по работе, обширных эксплуатационных испытаний с поощрениями за обнаружение уязвимых мест в защите проверяемой системы и выделения средств на долгосрочное ее сопровождение.

Антишаблоны для паролей

Одно из самых главных затруднений, связанных с обеспечением безопасности, состоит в том, что качественная защита нередко противоречит здравому смыслу или обычной практике. У вас может, например, возникнуть мысль, что требования к строгости паролей повысят безопасность приложения или веб-сайта. Но вы окажетесь не правы.

Правила соблюдения строгости паролей на самом деле *снижают* уровень безопасности. Ниже перечислены вкратце неудачные идеи и некоторые рекомендации по обеспечению безопасности от Национального института стандартов и технологий США (NIST)[24].

- Не ограничивайте длину паролей меньше чем 64 символами. NIST рекомендует в качестве предельной длины пароля 256 символов.
- Не обрезайте выбранные пользователями пароли.
- Не ограничивайте применение специальных символов, таких как `[]();&%$#` или `/`. Если специальные символы в пароле способны скомпрометировать вашу систему — у вас большие проблемы. NIST требует принимать все печатные ASCII-символы, пробел и символы Unicode.
- Не предоставляйте подсказки для паролей неаутентифицированным пользователям или приглашения для ввода конкретных типов информации (наподобие "Как звали ваше первое домашнее животное?")
- Не отменяйте функцию **`paste`** в своем браузере. Нарушение функций браузера и диспетчеров паролей не делает вашу систему более безопасной и фактически побуждает пользователей составлять более простые и краткие пароли, которые намного проще разгадать.
- Не устанавливайте дополнительные правила составления паролей. Например, не запрещайте использовать в паролях какие-нибудь конкретные сочетания прописных и строчных букв, цифр, специальных знаков или повторяющихся символов.
- Не заставляйте волевым порядком пользователей изменять свои пароли по истечении некоторого периода времени. Делайте это лишь при веских основаниях (например, в случае взлома).

Поощряйте употребление длинных, произвольных паролей с высокой степенью энтропии. Наложение искусственных ограничений сдерживает энтропию и порождает скверные привычки в отношении паролей, оставляя учетные записи пользователей уязвимыми к овладению чужими.

[24] См. специальное издание *NIST Special Publication 800-63B: Digital Identity Guidelines Authentication and Lifecycle Management*, свободно доступное по адресу `https://doi.org/10.6028/NIST.SP.800-63b`.

Общие нормы практики неверны даже для таких простых вещей, как пароли (см. врезку "Антишаблоны для паролей"). Обратившись к криптографии, имейте в виду, что даже мельчайшая и едва заметная ошибка способна все нарушить: ваш изощренный доморощенный алгоритм шифрования может оказаться разгаданным опытным взломщиком в считанные минуты. Поэтому ни в коем случае не занимайтесь шифрованием самостоятельно.

Как упоминалось ранее, полагаться следует лишь на то, что действительно надежно: хорошо проверенные, тщательно изученные, регулярно сопровождаемые, часто обновляемые библиотеки и каркасы — желательно с открытым исходным кодом. Если речь не идет о простых задачах шифрования, тщательно проанализируйте другие связанные с безопасностью функциональные возможности своего веб-сайта или приложения (например, аутентификацию пользователей).

Чтобы самостоятельно реализовать функции регистрации с помощью пароля или биометрической аутентификации, вам придется как следует разобраться в принципе действия и назначении хеш-кодов и "соли" для шифрования; понять, каким образом взломщики пользуются такими средствами, как радужные таблицы; выяснить причины, по которым нельзя пользоваться алгоритмами шифрования MD5 и SHA1; а также разрешить немало других криптографических вопросов. И даже если вы сделаете все это правильно, то в конечном счете именно вам придется отвечать за безопасное хранение данных, учитывая появление новых законодательных норм и правовых обязательств.

С другой стороны, вы можете выбрать прагматичный подход и возложить бремя ответственности на кого-нибудь, обратившись, например, к стороннему поставщику услуг аутентификации. Это может быть служба, функционирующая в вашей организации, или же сторонняя служба, присутствующая в "облаке". Услуги аутентификации нередко предоставляются поставщиками услуг электронной почты, телефонной связи или социальных сетей и могут (или не могут) быть пригодны для вашего приложения. Но в любом случае поставщики услуг аутентификации постоянно занимаются подержанием своих систем в безопасном состоянии и лучше вас разбираются в этих вопросах. Итак, будьте осторожны!

Другие связанные с данной темой разделы

- **Тема 23**. Проектирование по контракту, **глава 4** "Прагматичная паранойя".
- **Тема 24**. Мертвые программы не лгут, **глава 4** "Прагматичная паранойя".
- **Тема 25**. Утвердительное программирование, **глава 4** "Прагматичная паранойя".
- **Тема 38**. Программирование по совпадению.
- **Тема 45**. Западня требований, **глава 8** "До начала проекта".

Тема 44 Именование

"Называть вещи своими именами — начало мудрости".

Конфуций

Что значит имя? В процессе программирования оно значит "все, что угодно"! Разработчики придумывают имена приложениям, подсистемам, модулям, функциям, переменным, ведь они постоянно создают что-нибудь новое и каким-то образом его называют. Следовательно, имена крайне важны, поскольку они в немалой степени выявляют намерения и убеждения разработчиков.

Мы считаем, что разрабатываемые компоненты должны именоваться в соответствии с той ролью, которую они играют в прикладном коде. Это означает, что всякий раз, когда вы создаете какой-нибудь компонент, вы должны остановиться на время и подумать над следующим вопросом: "Для чего я это создаю?"

И это своевременный и важный вопрос, поскольку он заставляет вас отвлечься от решения непосредственной задачи и бросить взгляд на общую картину. Рассматривая роль переменной или функции, вы размышляете, для чего она предназначена, что она может сделать и как с ней взаимодействовать непосредственно в коде. И зачастую вы осознаете, что предполагаемые вами дальнейшие действия не имеют никакого смысла просто потому, что вам не удалось придумать подходящее имя.

Мысль о том, что имена имеют глубокий смысл, имеет свое научное обоснование. Оказывается, мозг человека может читать и понимать слова очень быстро, причем быстрее, чем выполнять многие другие действия. Это означает, что слова приобретают определенный приоритет, когда мы пытаемся что-то осмыслить. И это можно продемонстрировать на примере эффекта Струпа[25].

Взгляните на приведенную ниже панель, где показаны названия цветов и оттенков, выделенных черным или белым цветом либо оттенком серого, хотя названия и цвета совсем не обязательно совпадают. И вот вам задача: произнести вслух название каждого цвета так, как оно написано[26].

[25] См. *Studies of Interference in Serial Verbal Reactions* [Str35].

[26] Имеются два варианта данной панели. В одном из них употребляются разные цвета, а в другом — оттенки серого. Здесь приведен черно-белый вариант данной панели, но если вам потребуется цветной ее вариант или же если вы плохо различаете оттенки серого, перейдите на веб-страницу, доступную по адресу `https://pragprog.com/the-pragmatic-programmer/stroop-effect`.

Повторите описанные выше действия, но вместо этого произнесите вслух *название цвета*, которым написано каждое слово. Сделать это труднее, не так ли? Бегло прочитать названия цветов проще, чем попытаться распознать сами цвета. Наш мозг интерпретирует написанные слова как нечто важное, поэтому нам следует сделать так, чтобы употребляемые нами имена отвечали данному наблюдению.

Рассмотрим ряд примеров.

- Допустим, на веб-сайте, продающем ювелирные изделия, изготовленные из бывших в употреблении графических плат, мы аутентифицируем людей, посещающих этот сайт. Это делается следующим образом:

  ```
  let user = authenticate(credentials)
  ```

  Переменная `user`, хранящая результаты аутентификации, называется `user`, поскольку она всегда обозначает *пользователя*. Но почему переменная называется именно так? Ведь с тем же успехом ее можно было бы назвать `customer` или `buyer`? Присвоив ей имя `user`, мы тем самым постоянно напоминаем себе по ходу программирования, что именно данное лицо пытается сделать и что это означает для нас самих.

- Допустим, имеется следующий метод экземпляра, в котором рассчитывается скидка на заказанные товары:

  ```
  public void deductPercent(double amount)
  // ...
  ```

  Здесь употребляются два имени. Во-первых, имя метода `deductPercent` обозначает, *что* именно он делает, а не *почему* он это делает. И во-вторых, имя параметра `amount`, по меньшей мере, вводит в заблуждение. Что именно оно обозначает: абсолютную величину или долю в процентах?

  Возможно, было бы лучше написать следующую строку кода:

  ```
  public void applyDiscount(Percentage discount)
  // ...
  ```

  Ведь имя данного метода теперь ясно обозначает его назначение. Кроме того, мы изменили тип параметра с `double` на заранее определяемый ссылочный тип `Percentage`. Нам ничего неизвестно о намерениях пользователя данного метода, и поэтому, оперируя долями в процентах, мы можем

только гадать, в каких именно пределах будет задано значение параметра `discount`: от `0` до `100` или же от `0.0` до `1.0`. Поэтому указанный тип документирует здесь предполагаемое значение параметра данного метода.

- Допустим, имеется модуль, оперирующий числами Фибоначчи. В одной из его функций вычисляется *n*-е число Фибоначчи. При этом возникает вопрос: как назвать такую функцию? Большинство разработчиков скажут, что этой функции следует присвоить имя `fib`. Такое имя, на первый взгляд, вполне обосновано, но не следует забывать, что эта функция будет вызываться в контексте данного модуля, например, так: `Fib.fib(n)`. А может быть, этой функции следует присвоить имя `of` или `nth` и вызывать ее так, как показано ниже.

```
Fib.of(0)   # => 0
Fib.nth(20) # => 4181
```

Присваивая имена компонентам, вы постоянно ищете способы прояснить, что именно вы имеете в виду, и такое прояснение приведет вас к лучшему пониманию своего кода *по ходу* его написания. Тем не менее далеко не все имена способны претендовать на соискание литературной премии.

Исключение, подтверждающее правило

Несмотря на стремление сделать исходный код более ясным, необходимо также принимать во внимание и фирменную символику, а это уже совсем другой вопрос.

По установившейся традиции проектам и их командам принято присваивать малопонятные, но "умные" названия, например: "Покемон", "Замечательные супергерои", "Крутые мачо", имена персонажей из серии фильмов "Властелин колец" в буквальном смысле слова.

Уважение к культуре

В программировании есть только две трудные вещи:
инвалидация кеша и именование переменных.

В большей части литературы по основам программирования читателей предостерегают никогда не присваивать переменным однобуквенные имена вроде `i`, `j` или `k`.[27] Мы считаем, что они не совсем правы.

[27] Знаете *ли* вы, *почему* переменная цикла обычно обозначается как `i`? Дело в том, что более 60 лет назад в первоначальной версии языка FORTRAN (на синтаксис которого оказала большое влияние алгебра) переменные, имена которых начинались на буквы от `I` до `N`, были целочисленными.

На самом деле это зависит от культуры программирования на конкретном языке или его выполняющей среды. Так, в языке C имена `i`, `j` или `k` традиционно употребляются в качестве переменных для увеличения шага цикла, а именем `s` обозначаются символьные строки и т.д. Если вы программируете именно в такой среде, то, именуя элементы кода, следуйте принятым нормам. Иначе вы, по существу, нарушаете эти нормы, а следовательно, поступаете неверно. Уважая сложившиеся традиции, вы вряд ли напишете нечто, подобное приведенному ниже фрагменту кода на языке Clojure, где переменной `i` присваивается символьная строка.

```
(let [i "Hello World"]
   (println i))
```

В одних сообществах программистов отдается предпочтение смешанному написанию имен вроде `camelCase`, где употребляются прописные буквы в средине имени (так называемый "горбатый регистр"), тогда как в других сообществах — "змеиный регистр" вроде `snake_case`, где слова, составляющие имя, разделяются знаком подчеркивания. В самих языках программирования, безусловно, приняты обе разновидности написания имен. Но это не позволяет уладить все вопросы, поскольку приходится уважать локальную культуру программирования. Так, в некоторых языках в именах компонентов допускается использовать символы Юникода. Но, прежде чем употреблять в исходном коде такие "заумные" имена, как ɹəsn или εξέρχεται, необходимо подумать, как это будет воспринято в сообществе программистов.

Согласованность

Ральф Уолдо Эмерсон однажды написал: "Безрассудная согласованность — это пугало для маленьких умов", но ему не посчастливилось работать в команде программистов. В каждом проекте вырабатывается свой словарь терминов — жаргонных словечек, имеющих особое значение для членов команды. Так, термин *order* означает "заказ" для команды, разрабатывающей интернет-магазин, тогда как для команды, создающей приложение, графически отображающее происхождение религиозных групп, — "орден". В связи с этим очень важно, чтобы все члены команды знали эти термины и пользовались ими согласованно.

С одной стороны, можно поощрять активное общение в команде разработчиков. Так, если программы пишутся парами, и эти пары часто меняются, то терминология усваивается быстро. С другой стороны, можно составить словарь проекта, перечислив в нем термины, имеющие особое значение для команды. Это неформальный документ, который может поддерживаться в форме викистраниц или карточек для записей, вывешиваемых где-нибудь на стене.

Со временем терминология проекта станет жить своей жизнью. И как только словарь терминов станет привычным для всех, им можно пользоваться как скорописью для точного и краткого обозначения сложных понятий. (На этом, собственно, основывается *язык шаблонов*.)

Переименовывать еще труднее

В программировании есть только две трудные вещи:
инвалидация кеша, именование переменных
и ошибки выхода за границы массивов

Сколько ни прилагай усилий, все равно все течет и изменяется. В частности, код реорганизуется, акценты применения смещаются, а смысловое значение немного меняется. И если не проявить бдительность в отношении обновления имен по ходу программирования, то можно очень быстро дойти до *запутанных* имен, вызывающих еще большие проблемы, чем имена бессмысленные. Приходилось ли вам когда-нибудь слышать такое объяснение несогласованностей в исходном коде: "Подпрограмма `getData` на самом деле записывает данные в архивный файл"?

Как пояснялось в разделе "Тема 3. Программная энтропия" главы 1 "Философия прагматизма", обнаружив ошибку, ее необходимо безотлагательно устранить. Так, если вы обнаружите имя, больше не выражающее ваше намерение или вводящее в заблуждение, его нужно исправить. Имея в своем распоряжении полный набор регрессионных тестов, выявите все экземпляры этого слова, которые вы могли пропустить.

Совет 74 Именуйте правильно;
а по мере надобности переименовывайте

Если по какой-то причине вы не можете изменить неверное в настоящий момент имя, значит, у вас возникло более серьезное затруднение: нарушен принцип легкости изменения (см. раздел "Тема 8. Сущность качественного проектирования" главы 2 "Прагматичный подход"). Сначала разрешите это затруднение, а затем измените указавшее на него имя. Упрощайте переименование и делайте это почаще. В противном случае вам придется объяснить новым членам своей команды, что подпрограмма `getData` на самом деле записывает данные в файл, причем делать это с невозмутимым видом.

Другие разделы, связанные с данной темой

- **Тема 3**. Программная энтропия, **глава 1** "Философия прагматизма".
- **Тема 40**. Рефакторинг.
- **Тема 45**. Западня требований, **глава 8** "До начала проекта".

Задачи

- Если вы обнаружите функцию или метод со слишком общим именем, попробуйте изменить это имя, чтобы точнее выразить назначение данной функции или метода. И тогда вам будет легче реорганизовать свой код.
- В одном из рассмотренных выше примеров предлагалось употребить более конкретные имена вроде `buyer` или `customer` вместо более традиционного и обобщенного имени `user`. Какими еще именами вы обыкновенно пользуетесь как наиболее подходящими в данном случае?
- Согласуются ли имена в вашей системе с пользовательскими терминами из предметной области? Если не согласуются, то почему? Вызвано ли это когнитивным диссонансом вроде эффекта Струпа для вашей команды?
- Трудно ли изменять имена в вашей системе? Что можно сделать, чтобы устранить нарушенное в итоге отображение конкретного окна?

Глава 8

До начала проекта

В самом начале проекта вам и вашей команде требуется изучить исходные требования. Просто приказание, что нужно делать, или выслушать объяснения пользователей явно недостаточно, поэтому прочитайте раздел "Западня требований", чтобы научиться избегать общих ловушек.

Обыкновенной мудрости и управлению ограничениями посвящен раздел "Решение неразрешимым головоломок". Трудности появятся в любом случае, удовлетворяете ли вы требования, выполняете ли анализ, программируете или тестируете. Зачастую они бывают еще более серьезными, чем казались вначале.

Если по ходу проекта перед нами встают неразрешимые трудности, мы любим прибегать к своему секретному оружию: работать вместе. Под работой вместе мы подразумеваем не коллективный труд над массивным документом с требованиями, интенсивный обмен рассылаемыми по электронной почте сообщениями или проведение бесконечных совещаний, а совместное разрешение затруднений во время программирования. В разделе "Совместная работа" мы покажем, как приступить к такой работе, кто и что для этого требуется.

Несмотря на то что в *Манифесте гибкой разработки*, в частности, заявлено: "Люди и взаимодействие важнее процессов и инструментов", буквально все "гибкие" проекты начинались с иронического обсуждения процессов и инструментальных средств, которые предстоит применять. Но как бы тщательно ни был рассмотрен этот вопрос и какие бы нормы передовой практики ни применялись, ни одна методика не может заменить способность *мыслить*. Следовательно, требуется не конкретный процесс или инструмент, а осмысление самой сущности гибкости, как поясняется в соответствующем разделе.

Разобрав все самые важные вопросы *до* начала проекта, можно лучше подготовиться к работе над ним, чтобы избежать "аналитического паралича", когда много мыслей, да мало толку, и фактически начать, а затем успешно завершить проект.

Тема 45 Западня требований

"Совершенство достигается не тогда, когда нечего добавить, а когда нечего отнять..."

*Антуан де Сент-Экзюпери, **Планета людей**, 1939 г.*

Во многих книгах и учебных пособиях *сбор требований* объясняется как ранняя стадия проекта. Слово "сбор" здесь, по-видимому, означает племя счастливых аналитиков, собирающих повсюду крупицы мудрости под звуки "Пасторальной симфонии" Людвига ван Бетховена, исполняемой на заднем плане. Кроме того, слово "сбор" подразумевает, что требования уже существуют, остается лишь найти и разместить их в своей корзине и беззаботно идти своим путем.

В действительности все обстоит совсем иначе. Ведь требования редко лежат на поверхности. Как правило, они зарыты глубоко под слоями допущений, недоразумений и убеждений. Хуже того, требования зачастую вообще не существуют.

Совет 75 Никто точно не знает, чего он хочет

Миф о требованиях

На заре истории разработки программного обеспечения ЭВМ были более ценными (с точки зрения амортизированной стоимости часа работы), чем люди, которые на них работали. Поэтому программисты экономили средства, пытаясь сделать свои программы работоспособными сразу. В ходе этого процесса они старались точно указать, что должна делать ЭВМ. И с этой целью составлялась спецификация требований, которая превращалась сначала в проектную документацию, а затем в блок-схемы и псевдокод и, наконец, в исходный код. Но, прежде чем ввести исходный код в ЭВМ, программисты тщательно проверяли его за столом.

Такой подход к разработке программного обеспечения обходится очень дорого. И столь большие затраты послужили причиной того, что программисты старались автоматизировать что-нибудь лишь тогда, когда они точно знали, чего хотят добиться. А поскольку вычислительные возможности первых ЭВМ были очень скромными, круг решаемых задач был весьма ограниченным. В частности, *можно* было полностью понять задачу еще до того, как приступить к ее решению.

Но в реальном мире все обстоит совсем иначе. В нем царит беспорядок, конфликты и много неизвестного. В таком мире точные спецификации чего-нибудь редки, если они вообще возможны.

И здесь вступают в действие программисты. Их задача — помочь людям понять, чего они хотят на самом деле. И это, вероятно, самое ценное преимущество программистов, так что повторим еще раз:

Совет 76 Программисты помогают людям понять, чего они не хотят на самом деле

Программирование как терапия

Назовем клиентами тех людей, которые просят нас написать программу.

Типичный клиент приходит к нам со своей нуждой, которая может быть как стратегической, так и тактической потребностью отреагировать на возникшее затруднение. Это может быть потребность изменить существующую систему или создать новую. Иногда она выражается в деловых терминах, а порой — в технических.

Ошибка, которую часто совершают разработчики, состоит в том, что они воспринимают сформулированную их клиентами потребность как задачу и реализуют ее решение. На самом деле это лишь приглашение к исследованию, хотя клиент может этого и не осознавать.

Рассмотрим простой пример. Допустим, что вы работаете в издательстве, выпускающем книги как в печатном, так и в электронном виде. Вам поставили новое требование, сформулированное следующим образом:

> *Доставка должна быть бесплатной по всем заказам стоимостью свыше 50 долларов.*

Призадумайтесь на секунду и поставьте себя на место заказчика. Какие мысли придут вам сразу на ум? Скорее всего, у вас возникнут следующие вопросы:

- Включает ли сумма заказа 50 долларов налог?
- Включает ли сумма заказа 50 долларов текущие транспортные расходы?
- Распространяется ли сумма заказа 50 долларов только на печатные издания или же может охватывать и электронные книги?
- Какой вид доставки имеется в виду? Срочная? Наземным транспортом?
- А как насчет международных заказов?
- И как часто предельная сумма заказа 50 долларов будет изменяться в дальнейшем?

Когда вам дают простое задание, рассматривайте крайние случаи, докучая тем, кто вам его дал, подробными вопросами, чтобы прояснить дело. Скорее всего, клиент уже думал о некоторых из этих крайних случаев и лишь подразумевал, что они будут учтены в реализации. Задавая наводящие вопросы, можно выведать все эти сведения.

Но другие вопросы могут, вероятнее всего, касаться того, о чем клиент раньше даже не задумывался. Именно здесь и начинается самое интересное, а гра-

мотный разработчик учится быть дипломатом, как демонстрируется в приведенном ниже диалоге с клиентом.

Вы: *У нас возникли вопросы по поводу итоговой суммы заказа 50 долларов. Входят ли в эту сумму обычно взимаемые транспортные расходы?*

Клиент: *Конечно. Это все, что заказчики должны заплатить.*

Вы: *Заказчики должны ясно и просто усмотреть в этом привлекательность такого предложения. Но мне представляется, что некоторые менее добросовестные заказчики попытаются обойти эти правила нашей системы заказов.*

Клиент: *Каким образом?*

Вы: *Допустим, они покупают сначала книгу за 25 долларов, а затем выбирают ночную доставку как самый дорогой вариант. В таком случае доставка книги будет, вероятнее всего, стоить около 30 долларов, а общая сумма заказа составит 55 долларов. Если мы делаем доставку бесплатной, то за ночную доставку книги стоимостью 25 долларов они заплатили бы всего 25 долларов.*

(В этот момент опытный разработчик делает многозначительную паузу. Он изложил факты, а решение принимать клиенту.)

Клиент: *Ой, да это совсем не то, что я предполагал! Ведь мы понесем убытки на таких заказах. Есть ли какие-то пути выхода из этого положения?*

Именно здесь и начинается исследование исходных требований. Ваша роль — интерпретировать то, что говорит клиент, и в ответ пояснить ему возможные последствия. Это одновременно интеллектуальный и творческий процесс, поскольку вы обдумываете требования и в то же время способствуете принятию такого решения, которое, скорее всего, окажется лучше, чем то, что могло бы быть принято вами или клиентом по отдельности.

Требования — это процесс

В приведенном выше примере разработчик принял требования и в ответ объяснил их последствия клиенту, что и послужило началом для их исследования. В ходе этого исследования вы, вероятно, придумаете еще множество вопросов, реагируя на предлагаемые клиентом варианты решений. Это реальность любого сбора требований.

Совет 77 Требования изучаются с помощью обратной связи

Ваша задача — помочь клиенту лучше понять последствия сформулированных им требований. Для этого вы образуете цепь обратной связи, пользуясь которой клиент может обдумывать и уточнять свои требования.

В приведенном выше примере выразить обратную связь словами было нетрудно, но так бывает далеко не всегда. Иногда вы просто недостаточно знаете особенности предметной области для этого.

В подобных случаях программисты-прагматики полагаются на методику ответной реакции вроде наводящего вопроса: "Вы имели в виду именно это?" Они создают макеты и прототипы и дают клиентам возможность поэкспериментировать с ними. В идеальном случае макеты и прототипы получаются достаточно гибкими, чтобы их можно было изменять прямо во время бесед с клиентами и реагировать на их замечания вроде "это не то, что я имел в виду" репликами типа "а вот так больше похоже на то, что вы хотели?".

Иногда такие макеты и прототипы могут быть отвергнуты в течение часа или около того. Очевидно, что они являются лишь поделками, подсказывающими общий замысел.

Но на практике *вся* работа, которую разработчикам приходится выполнять, на самом деле является некоторой формой макетирования. И даже в конце проекта мы все еще интерпретируем таким образом пожелания своих клиентов. Фактически к этому времени количество клиентов только возрастает, включая сотрудников отделов контроля качества, эксплуатации, сбыта, а возможно, и тестовых групп потенциальных пользователей.

Таким образом, программист-прагматик рассматривает *весь* проект как большое упражнение в сборе требований. Именно поэтому мы предпочитаем краткие, но частые беседы с клиентами, чтобы получать их непосредственную реакцию. Это дает нам возможность быть в курсе дела и быть уверенными, что в том случае, если мы пойдем по неверному пути, потраченное на него время сведется к минимуму.

Поставьте себя на место клиента

Чтобы лучше понять ход мыслей клиента, достаточно воспользоваться простым, но нечасто применяемым методом: поставить себя на место клиента. Так, если вы разрабатываете систему для службы технической поддержки, проведите пару дней рядом с опытным сотрудником этой службы, следя за тем, как он отвечает на звонки. А если вы автоматизируете систему ручного контроля запасов, поработайте с неделю на складе[1].

[1] Целая неделя — не слишком ли это долго? Нет, не слишком, особенно в том случае, если требуется проанализировать производственные процессы, где руководители и исполнители работают в разных условиях. Руководители поделятся с вами одним (зачастую самым общим) представлением о производственных процессах, а когда вы опуститесь на землю, то столкнетесь с совершенно иной реальностью, на усвоение которой также потребуется время.

Помимо ясного понимания, каким образом проектируемая система будет применяться *на практике*, вы будете приятно удивлены тем, как ваша просьба понаблюдать за производственными процессами непосредственно на рабочих местах поможет установить доверие и создать прочный фундамент для общения с вашими клиентами. Только не мешайте людям работать!

Совет 78 Работайте с пользователем, чтобы мыслить как пользователь

Во время сбора ответной реакции стоит начать выстраивать тесные отношения со своей клиентурой, изучая ожидания и надежды клиентов в отношении проектируемой вами системы. Подробнее об этом см. в разделе "Тема 52. Доставляйте своим пользователям удовольствие" главы 9 "Прагматичные проекты".

Требования и правила

Представьте, что в беседе с сотрудниками отдела кадров клиент заявит: "Записи о работниках могут просматривать только их непосредственные начальники и сотрудники отдела кадров". Следует ли считать такое заявление истинным требованием? Возможно, в настоящий момент так и стоит поступать, но оно включает в себя определенный набор бизнес-правил.

Отличия бизнес-правил от требований едва заметны, но тем не менее они могут иметь глубокие последствия для разработчиков. Так, если требование сформулировано следующим образом: "Только непосредственные начальники и сотрудники отдела кадров могут просматривать записи о работниках", то разработчик может в конечном счете запрограммировать и проверять это требование явным образом всякий раз, когда приложение получает доступ к данным. Но если требование сформулировано таким образом: "Записи о работниках могут быть доступны только уполномоченным пользователям", то разработчик, вероятнее всего, спроектирует и реализует некоторого рода систему управления доступом. И если бизнес-правила изменятся (а это неизбежно), то обновить придется только метаданные такой системы. Фактически сбор требований естественным образом приводит к системе, хорошо приспособленной к поддержке метаданных.

В действительности общее правило таково:

Совет 79 Бизнес-правила — это метаданные

Реализуйте общий случай, приняв во внимание сведения о бизнес-правилах в качестве примера того, что должно поддерживаться в системе.

Требования и реальность

В статье из журнала *Wired*, вышедшей в январе 1999 года[2], продюсер и музыкант Брайан Эно описал невероятную на то время технологию окончательного микширования звука на одном пульте. Однако эта технология только мешала творческому процессу вместо того, чтобы дать музыкантам возможность создавать более качественную музыку, а звукорежиссерам — записывать ее быстрее и дешевле.

Чтобы стали понятнее причины, выясним, каким образом работают звукорежиссеры. Они выполняют сведение звука, руководствуясь своей интуицией. С годами у них вырабатывается внутренняя цепь обратной связи между их ушами и пальцами, перемещающими ползунки регуляторов громкости, вращающими рукоятки балансировки звуковых каналов и т.д. Тем не менее все эти профессиональные способности звукорежиссеров в упомянутом выше микшерском пульте учтены не были. Вместо этого его пользователи вынуждены были вводить данные с клавиатуры или щелкать кнопками мыши. Пульт обеспечивал все необходимые для звукозаписи функции, но они были реализованы незнакомым и чуждым звукорежиссерам образом. Зачастую очень нужные звукорежиссерам функции этого микшерского пульта были скрыты за непонятными названиями или же становились доступными через неинтуитивные комбинации основных функциональных средств.

Данный пример наглядно иллюстрирует наше убеждение, что удобные инструменты должны быть приспособлены к рукам мастера, который ими пользуется. Обычно это положение учитывается при удачном сборе требований. И поэтому ранняя ответная реакция, достигаемая с помощью прототипов или методом трассирующих пуль, приводит к реакции клиентов "Да, система делает то, *что* мне требуется, но не так, *как* я хочу".

Документирование требований

Мы считаем самой лучшей и, возможно, единственной документацией требований рабочий код. Но это совсем не означает, что можно обойтись без документирования своего понимания того, что требуется клиенту. Это просто означает, что такие документы не выпускаются, т.е. не даются клиентам на подпись. Вместо этого они просто служат вехами, помогающими направить процесс реализации требований в нужное русло.

Требования документируются не для клиентов

В прошлом мы оба работали над проектами, в ходе которых составлялись невероятно подробные требования. Такие основательные документы существенно расширяли исходное двухминутное пояснение клиентом его требований к проекту до образцового произведения толщиной с целый дюйм, испещренного

[2] См. по адресу `https://www.wired.com/1999/01/eno/`.

диаграммами и таблицами. В подобных документах требования уточнялись настолько, что их реализация почти не допускала никакой неоднозначности. Если бы можно было применить к такому документу современные довольно эффективные инструментальные средства, его можно было бы сразу превратить в готовую программу.

Составление таких документов было ошибкой по двум причинам. Во-первых, как обсуждалось ранее, клиенту на самом деле неизвестно, что ему, собственно, требуется. Следовательно, превращая то, что заявлено клиентом, в едва ли не нормативно-правовой документ, разработчики пытаются построить невероятно сложный замок на зыбучих песках.

На это вы могли бы возразить: "Но ведь когда мы приносим документ клиенту на подпись, то получаем его ответную реакцию". И это приводит нас ко второй причине, по которой такое документирование требований ошибочно: клиент вообще не читает спецификации требований к программному обеспечению.

Клиенты прибегают к услугам программистов потому, что программисты вникают в мельчайшие подробности, тогда как клиента интересует лишь общее решение не совсем ясно поставленной им задачи. Документация на требования составляется для разработчиков и содержит подробные сведения, которые иногда непонятны, а зачастую неинтересны самому клиенту.

Выпустите документацию требований объемом 200 страниц, и клиент, вероятнее всего, решит: если она столь объемистая, значит, достаточно важная. Он может позволить себе прочитать два первых ее абзаца (именно поэтому они и называются *резюме для руководства*) и бегло просмотреть все остальное, изредка останавливаясь на аккуратно построенной диаграмме.

Документация не подавляет клиента. Но предоставить ему крупную техническую документацию — это все равно, что дать разработчику средней квалификации экземпляр "Илиады" Гомера и попросить его написать на ее основе видеоигру.

Требования документируются для планирования

Итак, мы не верим в монолитную и настолько тяжелую, что ею можно оглушить и быка, документацию требований к проекту. Но мы знаем, что требования должны быть написаны просто потому, что разработчикам нужно знать, что им делать в команде.

Какую же форму должна принимать такая документация? Мы предпочитаем нечто, способное уместиться на реальной (или виртуальной) карточке для записей. Столь краткие записи обычно называются *пользовательскими историями*, описывающими, что именно должна делать небольшая часть приложения с точки зрения его пользователя. Если требования написаны именно таким образом, их можно разместить на доске и переставлять их, чтобы показать состояние и приоритетность их выполнения.

Вы можете, конечно, возразить, что на одной карточке для записей нельзя уместить все сведения, необходимые для реализации компонента приложения, и будете совершенно правы. Но это лишь частичный довод. Стремясь к краткости формулировки требований, вы побуждаете разработчиков задавать уточняющие вопросы. И в то же время усиливаете процесс ответной реакции клиентов и программистов на взаимные действия до и во время создания фрагмента кода.

Излишне подробная спецификация

Еще одна немалая опасность документирования требований состоит в том, что они могут оказаться слишком подробными. Грамотно составленные требования абстрактны. Простейшая формулировка, точно отражающая потребности в конкретной предметной области, подходит в качестве требования лучше всего. Это совсем не означает, что требования могут быть неопределенными. В их формулировке необходимо зафиксировать исходные семантические инварианты и задокументировать конкретные или текущие нормы практики выполнения работ в виде набора правил. Требования не имеют никакого отношения к архитектуре, конструкции или пользовательскому интерфейсу. Требования отражают *потребность*.

"Еще одну мятную вафельную пластинку..."[3]

Многие неудачи в проектах объясняются расширением их границ, иначе именуемым раздуванием либо расползанием функциональных возможностей или требований. Это похоже на синдром сваренной лягушки (см. раздел "Тема 4. Суп из камней и вареные лягушки" главы 1 "Философия прагматизма"). Что же можно сделать, чтобы избежать расползания требований?

Ответ снова кроется в обратной связи. Так, если вы работаете с клиентом в режиме последовательных итераций с обратной связью, то клиент будет на собственном опыте испытывать воздействие "еще одного функционального средства". Увидев, как очередная карточка с пользовательской историей переместилась вверх по доске, он поможет выбрать следующую карточку, чтобы перейти к следующей итерации и освободить место на доске. И в обоих случаях срабатывает обратная связь.

Ведение словаря терминов проекта

Как только вы начнете обсуждать требования, пользователи и знатоки предметной области тотчас воспользуются определенными терминами, имеющими для них конкретный смысл. Они, например, могут проводить различие между

[3] Кульминационный момент интермедии *Мистер Креозот* в исполнении английской группы комиков "Монти Пайтон"; см. по адресу `https://www.youtube.com/watch?v=GxRnenQYG7I`.

понятиями "клиент" и "заказчик", поэтому в системе нецелесообразно пользоваться тем или иным из этих понятий случайно.

Во избежание подобных недоразумений рекомендуется создать и вести *словарь терминов проекта* как единое место, где определяются все конкретные термины, употребляемые в проекте. Все участники проекта, от конечных пользователей до персонала технической поддержки, должны пользоваться этим словарем ради согласованности своих действий. При этом подразумевается, что словарь терминов должен быть широко доступным, что служит веским доводом в пользу онлайн-документации.

Совет 80 Пользуйтесь словарем терминов проекта

В проекте нельзя достичь успеха, если пользователи и разработчики называют один и тот же предмет по-разному (или, что еще хуже, называют разные предметы одинаково).

Другие разделы, связанные с данной темой

- **Тема 5**. Подходящее программное обеспечение, **глава 1** "Философия прагматизма".
- **Тема 7**. Общайтесь!, **глава 1** "Философия прагматизма".
- **Тема 11**. Обратимость, **глава 2** "Прагматичный подход".
- **Тема 13**. Прототипы и памятные записки, **глава 2** "Прагматичный подход".
- **Тема 23**. Проектирование по контракту, **глава 4** "Прагматичная паранойя".
- **Тема 43**. Будьте осторожны, **глава 7** "По ходу кодирования".
- **Тема 44**. Именование, **глава 7** "По ходу кодирования".
- **Тема 46**. Решение неразрешимых головоломок, **глава 8** "До начала проекта".
- **Тема 52**. Доставляйте своим пользователям удовольствие, **глава 9** "Прагматичные проекты".

Упражнения

33. Какие из перечисленных ниже требований, вероятнее всего, окажутся истинными? Переформулируйте, если это возможно, не истинные требования, чтобы они стали более полезными.

1) Время реакции должно быть меньше ~500 мс.
2) У модальных окон должен быть серый фон.
3) Приложение должно быть организовано в виде целого ряда внешних процессов и внутреннего сервера.

4) Если пользователь введет нечисловые символы в числовое поле, начнется мигание фона этого поля, а введенные данные будут отвергнуты.
5) Код и данные для данного встроенного приложения должны вписываться в пределы 32 Мбайт.

Задачи

- Можете ли вы сами воспользоваться программой, которую пишете? Можно ли иметь ясное представление о требованиях *без* возможности самому пользоваться написанной программой?
- Выберите задачу, не связанную с вычислительной техникой и требующую безотлагательного решения в настоящий момент. Составьте требования для решения этой задачи.

Тема 46 Решение неразрешимых головоломок

Гордий, царь Фригии, однажды завязал такой узел, который никому не удавалось развязать. Считалось, что тот, кто разгадает тайну гордиева узла, будет править Азией. И вот явился Александр Македонский и разрубил гордиев узел мечом на куски. Это была всего лишь иная интерпретация требований. И в конечном счете он действительно правил большей частью Азии.

Время от времени вы обнаруживаете, что оказались посредине проекта, и вам приходится решать по-настоящему трудную головоломку — техническую задачу, с которой вы не можете справиться, или фрагмент кода, написать который оказалось труднее, чем вы думали. Разрешить возникшее затруднение вроде бы невозможно, но так ли это трудно сделать, как кажется на первый взгляд?

Рассмотрим настоящие головоломки, состоящие из фигурных кусочков дерева, ковкого металла или пластмассы, которые обычно дарят детям на Рождество или которые можно найти среди распродаваемых домашних вещей. Чтобы решить такую головоломку, достаточно снять кольцо, разместить Т-образные кусочки в коробке или сделать что-нибудь другое.

Итак, вы тянете за кольцо или пытаетесь разместить Т-образные кусочки в коробке и тотчас убеждаетесь, что очевидные решения здесь не годятся. Решить головоломку таким способом нельзя. Но очевидность этого вывода не останавливает людей в их попытках делать то же самое снова и снова, надеясь, что таким способом удастся разгадать головоломку.

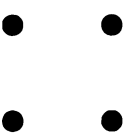
Решение, конечно, лежит где-то в другом месте. Секрет решения любой головоломки состоит в том, чтобы выявить настоящие (а не мнимые) ограничения и найти решение с их учетом. При этом одни ограничения *абсолютны*, а другие — лишь *предубеждения*. Абсолютные ограничения должны *непременно* учитываться, какими бы неугодными или нелепыми они ни были. Но, с другой стороны, некоторые очевидные ограничения могут вообще оказаться не настоящими (что и доказал на деле Александр Македонский). Такими же коварными могут оказаться и многие задачи в разработке программного обеспечения.

Степени свободы

Модное выражение "творческое, нешаблонное мышление" побуждает нас распознавать ограничения, которые могут оказаться не таковыми, и пренебречь ими. Но эта фраза не совсем точна. Так, если шаблон — это граница ограничений и условий, то нужно *найти* такой шаблон, который может оказаться значительно *бо*льшим, чем кажется на первый взгляд.

Самое главное в разгадывании головоломок — уметь распознавать накладываемые ограничения и имеющиеся степени свободы, поскольку именно в них находится решение. Некоторые головоломки столь эффективны именно потому, что, разгадывая их, можно слишком легко отклонить потенциальные решения.

Например, сможете ли вы соединить все точки на приведенном ниже рисунке таким образом, чтобы вернуться в исходную точку, проведя всего лишь три прямые линии, не поднимая ручку от бумаги и не возвращаясь по своим следам назад (см. *Math Puzzles & Games* [Hol92])?



Необходимо преодолеть любые предубеждения и проверить, являются ли они настоящими, незыблемыми ограничениями. И дело не в том, мыслите ли вы шаблонно или нешаблонно, а в том, как *найти* шаблон, выявив тем самым настоящие ограничения.

Совет 81 Вместо того чтобы мыслить нешаблонно, *находите* шаблон

Столкнувшись с трудноразрешимой задачей, перечислите *все* возможные пути, которые опробовали прежде. Не отвергайте ни одного из этих путей, каким бы бесполезным или нелепым он ни казался. Затем пройдитесь по списку всех путей и поясните, почему нельзя выбрать тот или иной путь. Насколько вы в этом уверены и сможете ли вы это *доказать*?

Обратимся к троянскому коню — эпическому примеру решения трудноразрешимой задачи. Как войску проникнуть незаметно в укрепленный стенами город? Можно побиться об заклад, что решение пробиться через ворота было сразу отвергнуто как самоубийственное.

Разделите свои ограничения по категориям и приоритетам. Когда столяры приступают к работе, они отрезают сначала самый длинный кусок, затем из оставшегося дерева — куски покороче. Аналогично требуется выявить сначала самые жесткие ограничения и подогнать под них все остальные. Между прочим, решение упомянутой выше головоломки с четырьмя точками, соединяемыми тремя прямыми линиями, приведено в конце приложения с решениями некоторых задач.

Идите своим путем!

Иногда приходится работать над задачей, которая кажется намного труднее, чем думалось изначально. Возможно, при этом возникает ощущение, что был выбран неверный путь и должно быть какое-то более простое решение! А может быть, трудноразрешимая задача возникла на завершающей стадии работы над проектом, и вы уже отчаялись довести свою систему до работоспособного состояния.

Это идеальный момент отвлечься немного на что-нибудь другое, например, пойти на прогулку с собачкой или просто поспать.

Ваш мозг осознает стоящую перед вами задачу, но на самом деле он немного глуповат (только без обид). Поэтому самое время задействовать ваш настоящий ум — эту поразительную ассоциативную нейронную сеть, скрывающуюся в подсознании. И вы будете удивлены, как часто в вашем уме возникает ответ, когда вы намеренно отвлекаетесь от мучающей вас задачи.

На первый взгляд, все это кажется слишком таинственным, но на самом деле это совсем не так. В журнале *Psychology Today*[4] по этому поводу сообщается следующее:

> *Откровенно говоря, люди, отвлекавшиеся от решения сложной задачи, добивались больших успехов, чем люди, которые прилагали для этого сознательные усилия.*

Если вы по-прежнему стремитесь отложить на время решение задачи, то следующий наилучший шаг, вероятно, состоит в том, чтобы найти кого-нибудь и пояснить ему свои трудности. И зачастую отвлечение просто на беседу с кем-нибудь приводит к прозрению.

---

[4] См. по адресу `https://www.psychologytoday.com/us/blog/your-brain-work/201209/stop-trying-solve-problems`.

Беседуя с кем-нибудь, дайте ему возможность задать вопросы, аналогичные перечисленным ниже.

- Почему вы решаете эту задачу?
- Какие преимущества дает ее решение?
- Связаны ли возникшие у вас трудности с крайними случаями? Можно ли их исключить?
- Есть ли более простая родственная задача, которую вы могли бы решить?

Это еще один характерный пример употребления на практике метода резинового утенка.

Судьба благоволит подготовленному уму

Известный французский ученый-микробиолог Луи Пастер как-то сказал:

"Что касается наблюдения,
то судьба благоволит подготовленному уму".

Это высказывание справедливо и для решения задач. Чтобы дойти до моментов прозрения, необходимо бессознательно накопить немало сырого материала; предыдущий опыт может способствовать выработке правильного решения.

Чтобы дать своему уму необходимую пищу для размышлений, лучше всего, когда вы выполняете свою повседневную работу, проанализировать отзывы о том, что работает и что не работает. Для этого удобно вести технический дневник, как пояснялось в разделе "Тема 22. Технические дневники" главы 3 "Основные инструментальные средства". И никогда не забывайте совет, начертанный на обложке книги *The Hitchhiker's Guide to the Galaxy*[5]: НЕ ПАНИКУЙ!

Другие разделы, связанные с данной темой

- **Тема 5**. Подходящее программное обеспечение, **глава 1** "Философия прагматизма".
- **Тема 37**. Прислушивайтесь к своим инстинктам, **глава 7** "По ходу кодирования".
- **Тема 45**. Западня требований.
- На эту тему Энди написал целую книгу под названием *Pragmatic Thinking and Learning: Refactor Your Wetware* [Hun08].

[5] Научно-фантастический роман английского писателя Дугласа Адамса (Douglas Adams; в русском переводе — *Автостопом по галактике*), по мотивам которого была создана одноименная компьютерная игра и целый телесериал. — *Примеч. пер.*

Задачи

- Тщательно проанализируйте любую трудную задачу, которую вам приходится решать в настоящее время. Удастся ли вас разрубить этот гордиев узел? Вынуждены ли вы поступать именно таким образом и следует ли это вообще делать?
- Представляли ли вы себе все множество ограничений, когда отдавали на подпись свой текущий проект? Применимы ли они до сих пор и верна ли по-прежнему их интерпретация?

Тема 47 Совместная работа

"Мне никогда не встречался человек, желающий прочитать 17 тысяч страниц документации, и если бы такой человек нашелся, я бы его убил, чтобы удалить его из генофонда".

Джозеф Костелло, президент компании Cadence Design Systems

Это был один из тех "невозможных" проектов, о которых вам, вероятно, приходилось слышать как о воодушевляющих и одновременно ужасающих. Существование морально устаревшей системы приходило к концу, а оборудование было физически изношено, и поэтому совершенно новая система была призвана заменить старую в *точном* соответствии с (зачастую недокументированным) режимом ее работы. Через эту систему проходили многие сотни миллионов долларов чужих людей, а ее новая версия должна была быть создана от начальной стадии проектирования и до конечной стадии развертывания в считанные месяцы.

Именно тогда и познакомились Энди и Дэйв — авторы этой книги как участники столь невозможного проекта с нелепыми предельными сроками выполнения. Шумному успеху этого проекта способствовало лишь одно обстоятельство: специалист, который должен был управлять данной системой годами, сидел в своем кабинете прямо напротив нашего похожего на чулан для метел помещения, где мы занимались разработкой. Так что он был постоянно доступен для вопросов, уточнений, решений и демонстраций промежуточных результатов нашей работы.

На страницах данной книги мы уже не раз рекомендовали работать в тесном контакте с пользователями, считая их частью команды разработчиков. В своем первом совместном проекте мы практиковали то, что теперь можно было бы назвать *парным* или *групповым программированием*, когда один программист набирает исходный код, тогда как другой программист или несколько членов команды комментируют, анализируют и решают задачи вместе с ним. Это весь-

ма эффективный способ совместной работы, заменяющий бесконечные совещания, памятные записки и раздутую до бюрократических размеров документацию, которая может похвалиться лишь своим весом, а не полезностью. Под совместной работой мы подразумеваем следующее: не просто задавать вопросы, вести дискуссии и делать замечания, но делать все это *непосредственно по ходу* программирования.

Закон Конвея

В 1967 году Мелвин Конвей выдвинул в своей статьей *How do Committees Invent?* [Con68] следующую идею, которая затем стала называться законом Конвея:

> *"Организации, проектирующие системы, ограничиваются проектными решениями, копирующими структуру связей в этих организациях".*

Таким образом, социальные структуры и каналы связи команды разработчиков с их организацией отражаются в проектируемом приложении, веб-сайте или ином программном продукте. Различные исследования эмпирически подтвердили эту идею. И мы можем засвидетельствовать, что неоднократно наблюдали действие закона Конвея на практике. Например, в одних командах, члены которых вообще не общаются друг с другом, проектируются разобщенные системы, а командах, разделенных надвое, получаются проекты типа "клиент–сервер".

Исследования эмпирически подтвердили и обратный принцип: структуру своей команды можно сознательно организовать по аналогии с требующейся структурой исходного кода. Например, в географически распределенных командах проявляется тенденция к разработке программного обеспечения, носящего более модульный и распределенный характер.

Но самое главное — команды разработчиков, включающие в себя пользователей, будут производить программное обеспечение, ясно отражающее их непосредственное участие в разработке. И это будет отражаться даже на тех командах, которые не особенно заботятся о привлечении пользователей к разработке программного обеспечения.

Парное программирование

Парное программирование является одной из тех норм практики экстремального программирования, которые нашли широкое распространение за его пределами. При парном программировании один разработчик вводит исходный код на клавиатуре, а другой анализирует введенный код. Они совместно работают над решением одной задачи и могут по мере надобности периодически меняться своими обязанностями.

Парное программирование дает немало преимуществ. Разные люди привносят в него разную подготовку и опыт, различные методы и подходы к решению задач с разной степенью сосредоточенности внимания на любой конкретной задаче. В частности, разработчик, вводящий исходный код с клавиатуры, должен быть сосредоточен на рядовых подробностях синтаксиса и стиля программирования, тогда как другой разработчик волен рассматривать общие вопросы в пределах решаемой задачи. И хотя такое различие может показаться незначительным, не следует забывать, что умственные способности каждого человека в той или иной мере ограничены. Так, возня с ключевыми словами и операторами, неохотно принимаемыми компилятором, отнимает немало вычислительной мощности мозга. Работа же в паре с другим разработчиком позволяет в немалой степени задействовать его умственные способности в процессе решения задачи. Как говорится, один ум хорошо, а два — лучше.

Присущее данной методике давление со стороны коллеги в паре помогает преодолевать моменты слабости и бороться с дурными привычками присваивать переменным имена вроде `foo`. Когда за вашими действиями активно наблюдают, вы в меньшей степени склонны прибегать к потенциально постыдным ухищрениям ради достижения цели кратчайшим путем. И это также приводит в итоге к повышению качества программного обеспечения.

Групповое программирование

Но если два ума лучше, чем один, то не привлечь ли с десяток разных людей к решению одной задачи, поручив одному из них набирать исходный код на клавиатуре? *Групповое программирование*, несмотря на свое название, не подразумевает толпы людей с факелами и кольями. Оно просто расширяет парное программирование, чтобы в совместной работе принимали участие не только два разработчика. Поборники данной методики сообщают о замечательных результатах группового решения трудных задач. В такие группы можно без особого труда ввести людей, которые не являются членами команды разработчиков, в том числе пользователей, спонсоров проекта и тестировщиков.

На самом деле в нашем первом "невозможном" совместном проекте, о котором речь шла ранее в этом разделе, мы считали привычным явлением, когда один из нас набирал исходный код, а другой обсуждал текущий вопрос со специалистом в предметной области данного проекта. По существу, мы работали группой из трех человек. Групповое программирование можно рассматривать как *тесное сотрудничество при активном программировании.*

Что следует делать?

Если в настоящий момент вы программируете индивидуально, опробуйте методику парного программирования. Выделите на это как минимум две недели, по несколько часов в день, поскольку эта методика может показаться вам

поначалу непривычной. Для выработки новых идей в режиме мозгового штурма или решения щекотливых вопросов можно опробовать методику группового программирования. Если вы уже практикуете парное или групповое программирование, то привлекаете ли вы к совместной работе только разработчиков или же допускаете участие в ней пользователей, спонсоров, тестировщиков и прочих лиц?

Как и при всяком другом сотрудничестве, необходимо руководствоваться как техническими, так и человеческими особенностями. В приведенных ниже некоторых рекомендациях поясняется, с чего следует начать.

- Пишите код, не выпячивая свое "я". Тут дело не в том, кто самый лучший, поскольку у каждого из нас свои достоинства и недостатки.
- Начинайте с малого. Привлеките к совместной работе 4–5 человек или же начните лишь с нескольких пар, работающих вместе в течение коротких периодов времени.
- Критикуйте написанный код, а не его автора. Фраза "рассмотрим этот блок кода" звучит намного лучше, чем фраза "ты ошибся".
- Внимательно слушайте, пытаясь понять чужую точку зрения. Отличие от вашей точки зрения еще не означает ее неправоту.
- Почаще проводите ретроспективы, чтобы постараться работать лучше в дальнейшем.

Программирование в одном и том же помещении или в удаленном режиме индивидуально, в парах или группах — все это эффективные методы совместной работы для решения насущных задач. Если вам и вашей команде приходилось когда-нибудь применять какой-нибудь из этих методов на практике, попробуйте поэкспериментировать с другими методами. Но не подходите прямолинейно к внедрению выбранного метода. Ведь для каждого стиля разработки имеются определенные правила, предположения и руководящие принципы. Например, при групповом программировании рекомендуется менять набирающего исходный код разработчика через каждые 5–10 минут.

Почитайте литературу, а также изучите прочие материалы на данную тему, чтобы уяснить преимущества и недостатки каждой упоминавшейся в этом разделе методики программирования. Возможно, стоит начать с простого упражнения в программировании, а не просто сразу опробовать выбранную методику на самом трудном рабочем коде. Но, как бы вы ни поступили, примите к сведению еще один завершающий совет.

Совет 82 Не вникайте в исходный код в одиночку

Тема 48 Сущность гибкости

"Вы все еще пользуетесь этим словом, но я не думаю, что оно означает именно то, что вы подразумеваете".

Иниго Монтойя[6]

Прилагательное "гибкий" обозначает, как именно сделать что-нибудь. Так, можно быть гибким разработчиком или членом команды, в которой приняты нормы практики гибкой разработки, или же команды, гибко реагирующей на перемены и провалы. Гибкость — это ваш стиль, а не вы сами.

Совет 83 Гибкость — это не существительное, а порядок выполнения действий

На момент написания данной книги прошло почти двадцать лет после принятия Манифеста гибкой разработки программного обеспечения[7], и с тех пор многие разработчики успешно применяют его ценные положения на практике. Мы наблюдаем немало чудесных команд, находящих способы принять ценности гибкой разработки и руководствоваться ими в том, что они делают и как они изменяют то, что делают.

Но мы также наблюдаем и обратную сторону гибкости: явное стремление команд и компаний к готовым решениям, где гибкость упакована в привлекательную обертку. И многие консультанты и компании будут только рады продать то, что хотят их клиенты. Мы видим, что в компаниях принимается все больше уровней управления, формальной отчетности, узкой специализации разработчиков и должностей с забавными названиями, которые просто означают чьи-то обязанности следить за работой других с папкой-планшетом и хронометром[8]. И мы считаем, что многие просто утратили ясное представление об истинном назначении гибкости, поэтому нам хотелось бы вернуть их к основным ценностям гибкой разработки.

Напомним основные положения *Манифеста гибкой разработки программного обеспечения*, где перечисленные приведенные ниже ценности.

> *"Мы раскрываем лучшие способы разработки программного обеспечения, выполняя ее сами и помогая делать это другим. В своей работе мы пришли к признанию следующих ценностей.*

---

[6] Inigo Montoya — вымышленный персонаж романа *The Princess Bride* (Принцесса-невеста) современного американского писателя Уильяма Голдмана.

[7] См. по адресу `https://agilemanifesto.org`.

[8] Подробнее о том, насколько неудачным может оказаться такой подход к разработке программного обеспечения, см. *The Tyranny of Metrics* [Mul18].

- Отдельные личности и взаимодействия превыше процессов и инструментальных средств.
- Работоспособное программное обеспечение превыше всеобъемлющей документации.
- Сотрудничество с заказчиками превыше согласования контрактов.
- Реагирование на перемены превыше следования плану.

Это означает, что, несмотря на ценность элементов, указанных справа в каждом из перечисленных выше положений, мы больше ценим элементы, указанные слева".

Всякий, предлагающий нечто, явно повышающее важность элементов, указанных справа в каждом из положений Манифеста гибкой разработки программного обеспечения, по сравнению с элементами, указанными слева, на самом деле не разделяет ценности наших и авторов этого документа. А всякий, предлагающий готовое решение, просто не читал вводную часть этого документа. К заявленным в нем ценностям побуждает и извещает о них действие, раскрывающее лучшие способы производства программного обеспечения. Это не статический документ, а предложения относительно порождающего процесса.

Гибкий процесс вообще невозможен

В действительности всякий раз, когда кто-нибудь говорит "сделайте это — и это и будет гибко", он не прав по определению. Гибкость, как в физическом мире, так и в разработке программного обеспечения, — это своевременное реагирование на неизвестность и перемены, возникающие тотчас, как только вы приступаете к делу. Газель не бежит по прямой линии, а гимнаст каждую секунду вносит сотни корректив в свои движения, реагируя на изменения в окружающей его среде, чтобы допустить минимум ошибок в расположении своих конечностей.

Это же относится к командам и индивидуальным разработчикам. Для разработки программного обеспечения не существует единого плана. Об этом упорно твердит упомянутый выше манифест и перечисленные в нем ценности, и все они имеют отношение к сбору ответной реакции и реагированию на нее. Эти ценности сообщают не о том, что нужно делать, а о том, к чему следует стремиться, решившись что-то делать.

Такие решения всегда принимаются в определенном контексте, поскольку они зависят от вас самих, характера вашей команды, разрабатываемого приложения, применяемых инструментальных средств, вашей организации, вашего заказчика, внешнего мира и невероятно большого числа других фактов, одни из которых более важны, чем другие. Никакой фиксированный, статический план действий не позволит преодолеть подобную неопределенность.

Что же тогда делать?

Никто не скажет вам, *что* нужно делать. Но мы думаем, что можем подсказать вам общее направление, в котором следует двигаться к желанной цели. Все сводится к тому, как справиться с неопределенностью. В упомянутом выше манифесте предлагается собирать ответную реакцию и действовать соответственно. Итак, рецепт гибкой работы состоит в следующем.

---

1. Выясните, где находитесь.
2. Сделайте наименьший значимый шаг в нужную сторону.
3. Оцените, где вы оказались, и исправьте все, что нарушили.

Повторяйте приведенные выше действия до тех пор, пока не завершите намеченную работу. Пользуйтесь данным рецептом рекурсивно на каждом уровне всей своей деятельности. Иногда даже самое тривиальное на первый взгляд решение становится важным, когда вы собираете ответную реакцию, как показано в приведенном ниже примере.

"Теперь мне нужен код, чтобы получить владельца счета.

```
let user = accountOwner(accountID);
```

М-да... имя `user` *кажется бесполезным. Изменю-ка я его* `owner`.

```
let owner = accountOwner(accountID);
```

Но теперь это кажется несколько избыточным. Что я на самом деле пытаюсь здесь сделать? В пользовательской истории говорится, что этому лицу следует отправить сообщение по электронной почте, поэтому мне нужно найти адрес его электронной почты. Может быть, сам владелец счета мне вообще не нужен?"

```
let email = emailOfAccountOwner(accountID);
```

Применив эту цепь обратной связи на довольно низком уровне именования переменной, мы фактически улучшили архитектуру всей системы, сократив связывание данного фрагмента кода с фрагментом, который оперирует счетами.

Цепь обратной связи применима и на наивысшем уровне проекта. Одна из наших наиболее успешных работ была сделана, когда мы приступили к проработке требований клиента, предприняли один шаг и осознали, что все, что мы предполагали делать, — не нужно, и что самое лучшее решение вообще не требовало программного обеспечения.

Применение цепи обратной связи можно расширить за пределы одного проекта. Команды должны применять ее для анализа своих процессов и того, насколько хорошо они работают. Те команды, которые не экспериментируют постоянно со своими процессами, нельзя считать гибкими.

И это движет проект

В разделе "Тема 8. Сущность качественного проектирования" главы 2 "Прагматичный подход" утверждалось, что мерой качества проектирования может служить простота изменения его результата: грамотное проектирование дает нечто, что изменить легче, чем при неграмотном проектировании. И в этом обсуждении гибкости поясняется, *почему* это именно так.

Сначала вы вносите изменения, а затем обнаруживаете, что они вам не нравятся. Как пояснялось в п. 3 приведенного выше рецепта, вам придется исправить все, что вы нарушили. Чтобы цепь обратной связи действовала эффективно, такое исправление должно быть как можно более безболезненным. В противном случае возникнет искушение оставить все как есть, без исправления. Подробнее о последствиях такого решения речь шла в разделе "Тема 3. Программная энтропия" главы 1 "Философия прагматизма". Для того чтобы вся рассматриваемая здесь методика гибкой разработки оказалась пригодной, следует практиковать качественное проектирование, поскольку именно оно способствует простоте изменений. Если внести изменения нетрудно, то можно, не колеблясь, вносить соответствующие коррективы на каждом уровне. Это и есть гибкость.

Другие разделы, связанные с данной темой

- **Тема 27**. Не опережайте свет фар вашего автомобиля, **глава 4** "Прагматичная паранойя".
- **Тема 40**. Рефакторинг, **глава 7** "По ходу кодирования".
- **Тема 50**. Кокосами не обойтись, в **главе 9** "Прагматичные проекты".

Задачи

Простая цепь обратной связи подходит не только для разработки программного обеспечения. Обдумайте другие решения, которые вам приходилось принимать в последнее время. Можно ли было бы их улучшить, размышляя над тем, как бы удалось их отменить, если бы дело приняло нежелательный оборот? Можете ли вы придумать способы улучшить то, что делаете, собирая ответную реакцию и соответственно реагируя на нее?

Глава 9

Прагматичные проекты

По ходу работы над проектом разработчикам приходится отступать от вопросов, связанных с индивидуальными философскими воззрениями и принципами программирования, переходя к обсуждению более общих вопросов масштаба целого проекта. Мы не собираемся здесь вдаваться в особенности управления проектами, но все же рассмотрим ряд очень важных областей, способствующих удачному или неудачному завершению любого проекта.

Как только над проектом станет работать не один человек, а команда, придется установить ряд основных правил и поручить выполнение соответствующих частей проекта конкретным лицам. В разделе "Прагматичные команды" будет показано, как это делается с учетом прагматичной философии.

Назначение конкретной методики разработки программного обеспечения — помочь людям работать вместе. Делаете ли вы и ваша команда то, что вам подходит, или же только разменяетесь на тривиальные поверхностные артефакты, не извлекая настоящих выгод для себя и своей команды? Ответы на этот и другие насущные вопросы разработки программного обеспечения даются в разделе "Кокосами не обойтись", где раскрываются также секреты успешной работы над проектами.

И, разумеется, ничто из этого вообще не важно, если вы не в состоянии выпускать программное обеспечение согласованно и надежно. А этому способствует волшебная троица: контроль версий и тестирование исходного кода, а также автоматизация процесса разработки, как поясняется в разделе "Начальный набор инструментальных средств программиста-прагматика".

В конечном счете успешность завершения проекта оценивается с точки зрения его спонсора. И здесь важно понимать, что значение имеет прежде всего восприятие успеха. Поэтому в разделе "Доставляйте своим пользователям удовольствие" показывается, как удовлетворить каждого спонсора проекта.

Последний совет, приведенный в данной книге, является прямым следствием всех остальных советов. В разделе "Гордость и предубеждение" поясняется, как сдавать проект заказчику на подпись с гордостью за проделанную работу.

Тема 49 Прагматичные команды

"В группе L Штоффель надзирает за шестью первоклассными программистами. Трудности руководства ими можно грубо сравнить с выпасом выводка котов".

***Washington Post Magazine** от 9 июня 1985 г.*

Даже в 1985 году шутка по поводу выпаса выводка котов считалась устаревшей. А ко времени выхода в свет первого издания данной книги под конец столетия она вообще стала анахронизмом, хотя до сих пор остается расхожей, поскольку в ней есть доля правды. Программисты действительно похожи чем-то на котов: они образованы, своевольны, самоуверенны, независимы и зачастую прославлены в сети.

В этой книге до сих пор рассматривались прагматичные методики, помогающие отдельным программистам стать лучше. Могут ли эти методики пригодиться командам, пусть даже состоящим из своевольных и независимых людей? Однозначно могут! Преимущества быть прагматичным программистом многократно усиливаются, когда программист работает в прагматичной команде.

На наш взгляд, команда — это небольшая, самостоятельная и в основном устойчивая организационная единица. Пятьдесят людей — это не команда, а толпа[1]. Команды, члены которых постоянно привлекаются к выполнению чужих заданий и не знают друг друга, вообще не считаются командами. Они похожи на незнакомых людей, временно столпившихся под козырьком автобусной остановки в дождливый день.

Прагматичная команда невелика и состоит не более, чем из 10–12 человек, которые редко приходят и уходят. Все хорошо знакомы и полагаются друг на друга.

Совет 84 Поддерживайте небольшие, устойчивые команды

В этом разделе будет вкратце показано, как применять прагматичные методики к командам в целом. Приведенные здесь заметки служат лишь отправной точкой. Как только образуется группа прагматичных разработчиков, работающих в благоприятных условиях, они быстро выработают и уточнят пригодную для них динамику работы в команде. Итак, изложим материал предыдущих глав иначе с точки зрения команд.

---

[1] По мере увеличения численного состава команд связи между программистами растут со скоростью $O(n^2)$, где n — число членов команды. В крупных командах связь начинает ослабевать и становится неэффективной.

Никаких разбитых окон

Качество — это дело всей команды. Даже самому прилежному разработчику, работающему в беспечной команде, будет трудно поддерживать свой энтузиазм, необходимый для решения пустячных задач. Трудности еще больше усугубляются, если команда активно отговаривает такого разработчика не тратить время на такие пустяки.

Команды в целом не должны мириться с разбитыми окнами, т.е. с теми мелкими несовершенствами, которые некому устранить. Команда *обязана* взять на себя ответственность за качество разрабатываемого ею программного продукта, поддерживая тех разработчиков, которые ясно понимают принцип "*никаких разбитых окон*", описанный в разделе "Тема 3. Программная энтропия" главы 1 "Философия прагматизма", а также помогая уяснить данный принцип тем, кто еще не открыл его для себя.

В некоторых командных методологиях предусматривается "ответственный за качество", которому команда поручает отвечать за качество выпускаемого программного продукта. Но это нелепо, поскольку качество может быть достигнуто лишь в результате индивидуального вклада *всех* членов команды. Качество закладывается как основание, а не прикрепляется болтами.

Сваренные лягушки

Помните, как в разделе "Тема 4. Суп из камней и вареные лягушки" главы 1 "Философия прагматизма" упоминалось о вымышленной лягушке в кастрюле с водой? Она не замечает постепенное изменение своего окружения и в конечном итоге оказывается сваренной. То же самое может произойти и с отдельными людьми, потерявшими бдительность. Ведь не так-то просто следить за всем окружением в самый разгар работы над проектом.

Команды могут очень легко свариться целиком, как лягушки. Ее члены могут посчитать, что возникший вопрос должен решить кто-то другой, а отвечать за внесение запрашиваемого пользователем изменения должен руководитель команды. И даже команды с наилучшими намерениями могут предавать забвению существенные изменения в своих проектах.

С подобными явлениями следует бороться, поощряя всех членов команды активно и постоянно контролировать изменения в окружающей их среде. Необходимо быть начеку, следя за расширением рамок проекта, сокращением сроков выполнения работ, внедрением дополнительных функциональных возможностей, появлением новых сред — другими словами, за всем, что первоначально не было ясно и понятно. Кроме того, необходимо следить за количественными показателями новых требований[2]. Команда не должна отвергать изменения с по-

[2] Для этой цели диаграмма *выгорания* подходит лучше, чем диаграмма *сгорания*. Пользуясь диаграммой выгорания, можно ясно видеть, как дополнительные функциональные средства меняют цели и намерения по ходу проекта.

рога, а должна осознавать, что они все равно происходят. В противном случае она рискует свариться в кипящей воде, как лягушка.

Планирование пополнения багажа знаний

В разделе "Тема 6. Ваш багаж знаний" главы 1 "Философия прагматизма" были рассмотрены способы своевременного пополнения багажа индивидуальных знаний. Команды, стремящиеся к успеху, должны анализировать и пополнять багаж своих знаний и умений.

Если ваша команда серьезно относится к усовершенствованиям и нововведениям, их следует планировать. Попытка сделать что-нибудь, когда настанет удобный момент, означает, что это не произойдет *никогда*. С каким бы видом задела, списка задач или порядка работ ни приходилось иметь дело, его не следует откладывать на будущую стадию разработки. Ведь команда работает не только над новыми функциональными средствами. Ниже перечислены некоторые из возможных путей пополнения командой своего багажа знаний.

- **Сопровождение прежних систем.** Несмотря на то что всем нравится работать в новой с иголочки системе, вполне возможно, что придется поработать и над сопровождением старой системы. Нам встречались команды, пытавшиеся отложить такую работу в долгий ящик. Если команда обязана выполнять такие задачи, она должна выполнять их точно и в срок.
- **Обсуждение и уточнение процессов.** Непрерывное совершенствование может происходить только в том случае, если есть время оглянуться и выяснить, что работает и что не работает, а затем внести необходимые изменения (см. раздел "Тема 48. Сущность гибкости" главы 8 "До начала проекта"). Слишком много команд оказываются настолько занятыми вычерпыванием воды из тонущей лодки, что не имеют времени на устранение течи. Планируйте работы по выявления и устранению неполадок.
- **Экспериментирование с новыми технологиями.** Не внедряйте новые технологии, каркасы или библиотеки только потому, что это делают все, или же на основании увиденного на конференции либо прочитанного в Интернете. Технологии, претендующие на внедрение, следует проверять с помощью прототипов. Поставьте соответствующие задачи в график выполнения работ, опробуйте новые технологии и проанализируйте полученные результаты.
- **Обучение и совершенствование навыков.** Личное обучение и совершенствование навыков служит отличной отправной точкой, хотя многие навыки оказываются более эффективными, если применяются на уровне команды. Планируйте подобные мероприятия, будь то в виде неформальных встреч или более формальных учебных занятий.

Совет 85 Планируйте и воплощайте в жизнь пополнение багажа знаний

Внешнее общение команды

Очевидно, что разработчики могут общаться друг с другом в своей команде. Свои рекомендации по поводу того, как облегчить общение, мы дали в разделе "Тема 7. Общайтесь!" главы 1 "Философия прагматизма". Но при этом очень легко забыть, что команда сама присутствует в конкретной организации. Поэтому, как организационная единица, команда должна свободно общаться с остальным миром.

Для посторонних наихудшими являются замкнутые и скрытные проектные команды. Они проводят совещания без всякого порядка, и на них никто не хочет говорить. Их обмен сообщениями по электронной почте и проектные документы беспорядочны: у них нет двух одинаковых документов и единой употребляемой терминологии.

Хорошие проектные команды заметно отличаются своей индивидуальностью. Другие люди стремятся встретиться с такими командами на совещаниях, поскольку они знают, что придут на хорошо подготовленное мероприятие, где все участники чувствуют себя уютно. Такие команды составляют ясную, точную и согласованную документацию и говорят одним голосом[3]. У них даже может быть чувство юмора.

Имеется простой маркетинговый прием, помогающий командам общаться как единое целое. Он состоит в том, чтобы сформировать фирменный стиль. Начиная проект, придумайте ему подходящее название — в идеальном случае взяв его с потолка. (В прошлом мы называли свои проекты, например, попугаями-убийцами, охотящимися на овец, разными оптическими иллюзиями, песчанками, персонажами мультфильмов и мифическими городами.) Уделите полчаса, чтобы придумать забавный логотип, а затем воспользуйтесь им в своем проекте. На первый взгляд, такая рекомендация кажется нелепой, но на самом деле она дает вашей команде возможность создать свою индивидуальность, а окружающему миру — запомнить нечто, связанное с вашей работой.

Не повторяйтесь

В разделе "Тема 9. DRY — пороки дублирования" главы 2 "Прагматичный подход" речь шла о том, как трудно исключить дублирование работы отдельными членами команды. Такое дублирование приводит к напрасным затратам тру-

[3] Команда говорит одним голосом с внешним миром; внутри команды настоятельно рекомендуется проводить живые, активные дебаты. Ведь грамотные разработчики обычно относятся весьма ревностно к своей работе.

да и способно превратить сопровождение программного обеспечения в сущий кошмар. В тех командах, где это происходит, зачастую проектируются разрозненные системы, у которых мало общего, но много дублирующихся функциональных возможностей.

Хорошо налаженное общение в команде служит главным средством, помогающим избежать подобных осложнений. Под фразой "хорошо налаженное" здесь понимается *мгновенное* и *свободное* общение без всяких трений. Это означает возможность задать членам команды вопрос и почти мгновенно получить ответ. Так, если команда совместно размещается в одном месте, для этого достаточно высунуть голову из своей кабинки или в коридор. В командах, работающих в удаленном режиме, возможно, придется полагаться на систему обмена сообщениями или другие электронные средства.

Если вам приходится целую неделю ждать совещания в команде, чтобы задать вопрос или поделиться положением своих дел, это может вызвать немало трений[4]. Свободное общение без всяких трений означает простую и нецеремонную атмосферу в команде, позволяющую задавать вопросы, делиться своими достижениями и затруднениями, открытиями и знаниями, а также оставаться в курсе того, что делают ваши коллеги. Поддерживайте свою осведомленность обо всем, что происходит в команде, чтобы соблюдать принцип DRY.

Трассирующие пули в команде

Проектной команде приходится выполнять много разных заданий в различных областях проекта, касающихся самых разных технологий. Для этого необходимо уяснить исходные требования, спроектировать архитектуру, запрограммировать и протестировать клиентскую и серверную части разрабатываемой системы. Существует типичное недопонимание, состоящее во мнении, что эти виды деятельности и задания можно выполнять по отдельности и обособленно, — хотя в действительности делать этого нельзя.

Некоторые методологии поддерживают самые разные роли и должности в команде или создание совершенно отдельных специализированных команд. Но недостаток такого подхода заключается в том, он вносит *шлюзы* и *сдачи*. И тогда вместо плавного перехода от разработки к развертыванию образуются искусственные шлюзы, где работа команды останавливается. А сдачи работ вынуждены ждать своего приема, утверждения, бумажной волокиты. Бережливые люди называют это *расточительством* и поэтому стремятся активно исключить его.

Все эти разные роли и виды деятельности фактически являются разными представлениями одной и той же задачи, и поэтому их искусственное разделение может вызвать немало хлопот. Например, те программисты, которые отделены двумя или тремя уровнями от фактических пользователей написанного

[4] Энди встречались команды, проводившие свои "ежедневные" летучки по пятницам.

ими кода, вряд ли понимают контекст, в котором используются результаты их трудов. В итоге они не могут принимать обоснованные решения.

Мы рекомендуем пользоваться методом *трассирующих пуль* (см. раздел "Тема 12. Трассирующие пули" главы 2 "Прагматичный подход"), разрабатывая отдельные, пусть небольшие и первоначально ограниченные функциональные средства, которые проходят сквозь всю систему. И для этого потребуются все ваше умение и опыт, чтобы применить данный метод в команде, где всем ее членам (проектировщику GUI и взаимодействия с пользователем, администратору базы данных, контролеру качества и прочим) удобно и привычно работать вместе. Применяя метод трассирующих пуль, можно довольно быстро реализовать очень мелкие фрагменты функциональных средств, чтобы немедленно получить ответную реакцию, позволяющую судить, насколько хорошо команда взаимодействует и доставляет результаты своих трудов. Благодаря этому создается среда, в которой можно быстро и просто вносить изменения и настраивать команду и процесс разработки.

Совет 86 Организовывайте полнофункциональные команды

Создавайте команды таким образом, чтобы разрабатывать код комплексно, постепенно и итерационно.

Автоматизация

Чтобы обеспечить согласованность и точность, лучше всего автоматизировать все, что делает команда. Зачем, например, разбираться со стандартами форматирования кода, если текстовый редактор или интегрированная среда разработки может сделать это автоматически? Зачем проводить тестирование вручную, если это можно делать автоматически в процессе непрерывной сборки? Наконец, зачем выполнять развертывание вручную, если этот процесс можно автоматизировать, чтобы надежно повторять его всякий раз, когда в этом возникнет потребность?

Автоматизация является существенной составляющей работы каждой проектной команды. Поэтому следует позаботиться о том, чтобы у команды было достаточно навыков *комплектации инструментальных средств* для построения и развертывания инструментальных средств, автоматизирующих разработку проектов и развертывания в условиях эксплуатации.

Знайте, когда остановиться

Помните, что команды состоят из индивидуальностей. Дайте каждому члену возможность проявить себя, а также обеспечьте их поддержу и возможность вклада в проект. И тогда вам останется лишь бороться с искушением нанести

еще немного краски, как художнику, упоминавшемуся в разделе "Тема 5. Подходящее программное обеспечение" главы 1 "Философия прагматизма".

Другие разделы, связанные с данной темой

- **Тема 2**. Кот съел мой исходный код, **глава 1** "Философия прагматизма".
- **Тема 7**. Общайтесь!, **глава 1** "Философия прагматизма".
- **Тема 12**. Трассирующие пули, **глава 2** "Прагматичный подход".
- **Тема 19**. Контроль версий, **глава 3** "Основные инструментальные средства".
- **Тема 50**. Кокосами не обойтись.
- **Тема 51**. Начальный набор инструментальных средств программиста-прагматика.

Задачи

- Поищите успешные команды вне области разработки программного обеспечения. В чем причина их успешности? Пользуетесь ли вы какими-нибудь процессами, обсуждавшимися в этом разделе?
- Приступая к проекту в следующий раз, попытайтесь убедить его участников сформировать свой особый фирменный стиль. Дайте своей организации время привыкнуть к этой идее, а затем проведите быструю проверку, чтобы выяснить, какие отличия внес новый фирменный стиль как в самой команде, так и за ее пределами.
- Вам, вероятно, приходилось решать задачи вроде следующей: "Если четырем рабочим требуется шесть часов, чтобы вырыть канаву, то сколько времени для этого потребуется восьми рабочим?" Но какие реальные факты могут повлиять на решение подобной задачи, если речь идет не о рабочих, а о программистах? И сколько имеется вариантов фактически сократить время выполнения подобной работы?
- Прочитайте книгу Фредерика Брукса *The Mythical Man Month* (Мифический человеко-месяц) [Bro96]. В качестве дополнительного задания приобретите два экземпляра этой книги, чтобы читать ее в два раза быстрее.

Тема 50 Кокосами не обойтись

Туземные обитатели одного острова никогда не видели прежде самолет и не встречались с прилетевшими на нем чужеземцами. В обмен на право пользоваться их землей чужеземцы предоставили островитянам возможность наблюдать за этими механическими птицами, которые целый день то прилетали, то улетали со "взлетной полосы", принося на их родной остров невероятные материальные ценности, иногда перепадающие туземцам. Чужеземцы упоминали о

чем-то вроде войны и сражения. Однажды все это кончилось тем, что чужеземцы покинули остров, прихватив с собой свои чудесные богатства.

Отчаянно пытаясь вернуть подарок судьбы, островитяне восстановили копию аэродрома с диспетчерской вышкой и прочим оборудованием, используя местные материалы: лианы, скорлупы кокосового ореха, пальмовые ветки и т.п. Но по какой-то непонятной им причине самолеты не прилетали, несмотря на то, что островитяне все расставили по своим местам. А дело в том, что они сымитировали форму, а не содержание. Антропологи называют это *карго-культом*.

Разработчики программного обеспечения слишком часто оказываются в роли этих незадачливых островитян. Ведь очень легко попасться в западню карго-культа, вложив средства и труд в построение чисто внешней имитации, надеясь вызвать скрывающиеся под ней волшебные силы, способные привести ее в действие. Но, как и в первобытных карго-культах Меланезии[5], бутафорский аэродром, построенный из скорлупы кокосовых орехов, не может заменить собой настоящий аэропорт.

Нам, например, приходилось лично встречаться с командами, заявлявшими, будто бы они пользуются методикой Scrum в своей работе. Но при более внимательном рассмотрении оказалось, что они проводили стоя ежедневные летучки лишь один раз в неделю, а четырехнедельные повторяющиеся циклы нередко превращались у них в шести- и даже восьминедельные циклы. Они считали это нормальным, поскольку пользовались весьма распространенным инструментальным средством "гибкого" планирования работ. На самом же деле они вкладывали средства и труд лишь в искусственные артефакты, суеверно твердя как заклинание такие термины, как "летучка" или "итерация". Не удивительно, что им не удавалось вызвать к жизни по-настоящему волшебные силы.

Все дело в контексте

Приходилось ли вам или вашей команде попадать в подобную западню? Спросите себя: почему вы вообще пользуетесь данной конкретной методикой разработки, каркасом, библиотекой или методикой тестирования? Действительно ли все это подходит для выполнения текущего задания и вам лично? А может быть, все это было принято лишь потому, что применялось в каком-нибудь недавнем проекте, история успеха которого была почерпнута из Интернета?

Ныне наблюдается тенденция внедрять у себя правила и процессы таких успешных компаний, как Spotify, Netflix, Stripe, GitLab и пр. У каждой из них свой взгляд на разработку программного обеспечения и руководство ею. Но давайте рассмотрим контекст: разве вы работаете на тот же рынок, с теми же ограничениями и возможностями, имеете аналогичный опыт и масштабы организации, у вас аналогичные подходы к руководству и такая же культура, пользовательская база и требования?

[5] См. по адресу `https://ru.wikipedia.org/wiki/Карго-культ`.

Не попадайтесь на эту удочку! Определенных артефактов, внешних структур, правил, процессов и методов явно недостаточно.

Совет 87 Делайте то, что пригодно, а не то, что модно

А как узнать, что именно пригодно? Для этого достаточно воспользоваться одним из самых основных прагматичных приемов, а именно:

Пробуйте!

Опробуйте саму идею в небольшой команде или ряде команд. Оставьте то, что годится, и откажитесь от всего остального как расточительства или накладных расходов. Никто не сможет принизить вашу организацию только потому, что она действует иначе, чем компании Spotify или Netflix, и не следует принятым в них процессам, которые способствовали их успешному росту. Ведь через несколько лет эти компании, созрев, упрочившись и продолжая расти дальше, станут в очередной раз делать нечто иное. Именно в этом и кроется секрет их успеха.

Один и тот же подход годится не всем

Назначение методологии разработки программного обеспечения — помочь разработчикам работать вместе. Как пояснялось в разделе "Тема 48. Сущность гибкости" главы 8 "До начала проекта", не существует единого плана, которого можно придерживаться, разрабатывая программное обеспечение, и особенно того плана, который был составлен кем-то чужим в другой организации.

Многие обучающие программы на самом деле оказываются еще хуже, поскольку они основываются на способностях учащихся запоминать и соблюдать правила. Но вам требуется совсем не это, а способность видеть дальше существующих правил и выгодно пользоваться имеющимися возможностями. Это совсем иной образ мышления, чем мнение вроде "так принято в методике Scrum, Lean, Kanban, экстремального программирования, гибкой разработки..." и т.д.

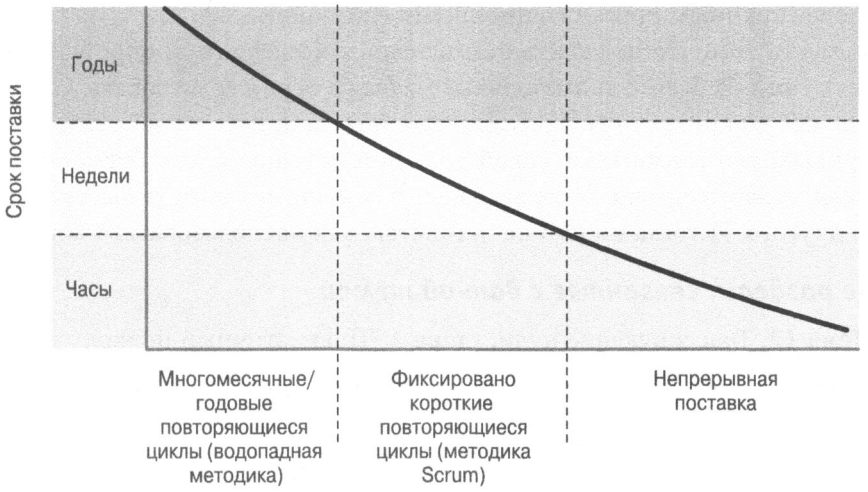
Вместо этого следует взять все самое лучшее из любой методологии и приспособить к применению в своей практике. Не существует такого единого подхода, который годился бы всем. Современные методы разработки программного обеспечения далеки от совершенства, и поэтому вам придется рассмотреть не только один широко распространенный метод. Например, в методике Scrum определяется ряд норм практики управления проектами, но сама эта методика не может служить достаточным руководством к действию команд на техническом уровне или же на уровне портфеля заказов либо организации управления для высшего руководства. Так с чего же начать?

Будьте как они!

Нам часто приходится слышать, как начальство, руководящее разработкой программного обеспечения, говорит своим подчиненным: "Мы должны действовать как Netflix" (или одна из ведущих компаний в данной отрасли). Разумеется, вы *могли бы* это сделать, но прежде вам пришлось бы установить несколько сотен серверов и привлечь десятки миллионов пользователей...

Главная цель

Безусловно, главная цель состоит не в том, чтобы действовать по методике Scrum, Lean, "гибко" или по иной выбранной вами методологии разработки программного обеспечения, а также иметь все необходимые возможности, чтобы выпускать работоспособное программное обеспечение, дающее пользователям новые возможности *уже теперь*, а не через недели, месяцы или годы. Многим командам и организациям непрерывная поставка программного обеспечения кажется слишком возвышенной, недостижимой целью, особенно если они обременены процессом, ограничивающим поставку месяцами или даже неделями, как показано ниже. Но какова бы ни была конечная цель, самое главное — постоянно двигаться в правильном направлении.



Если программное обеспечение выпускается годами, можно попытаться сократить данный цикл до месяцев, а затем до недель. Если же применяется методика четырехнедельных спринтов, их можно сократить до двух недель, а затем до одной недели и даже ежедневных выпусков. Наконец, поставки программного обеспечения можно делать по требованию. Следует, однако, иметь в виду,

что возможность выпускать программное обеспечение по требованию совсем не означает, что это необходимо делать ежеминутно каждый день. В этом режиме программное обеспечение поставляется именно тогда, когда оно требуется пользователям и когда это экономически целесообразно.

Совет 88 Выпускайте программное обеспечение, когда оно требуется пользователям

Чтобы перейти к такому стилю непрерывной разработки программного обеспечения, необходимо создать довольно устойчивую инфраструктуру, как обсуждается далее в разделе "Тема 51. Начальный набор инструментальных средств программиста-прагматика". При этом разработка ведется в главном стволе системы контроля версий исходного кода, а не в ее ветвях, а также применяются такие методики, как *переключатели функциональности*, дающие возможность выборочно включать средства тестирования для пользователей.

Как только инфраструктура будет приведена в должный порядок, необходимо решить, каким образом организовать работу. Начинающим командам, возможно, придется выбрать сначала методику Scrum для управления проектом, а затем технические нормы практики экстремального программирования. А более опытные и дисциплинированные команды могут остановить свой выбор на методиках Kanban и Lean, пригодных для управления как отдельными командами, так и более крупными организационными единицами.

Но вместо того чтобы верить нам на слово, исследуйте и опробуйте все эти методики сами. И будьте внимательны, чтобы не переусердствовать. Слепо полагаясь на любую конкретную методологию, можно просто не заметить альтернатив. Привыкнув пользоваться какой-то одной методикой, вам будет очень трудно заметить любые другие. Утвердившись в ней, вы не сможете быстро внедрить ничего другого. И тогда ничего не останется, как воспользоваться кокосами.

Другие разделы, связанные с данной темой

- **Тема 12**. Трассирующие пули, **глава 2** "Прагматичный подход".
- **Тема 27**. Не опережайте свет фар вашего автомобиля, **глава 4** "Прагматичная паранойя".
- **Тема 48**. Сущность гибкости, **глава 8** "До начала проекта".
- **Тема 49**. Прагматичные команды.
- **Тема 51**. Начальный набор инструментальных средств программиста-прагматика.

Тема 51 Начальный набор инструментальных средств программиста-прагматика

"Цивилизация развивается,
расширяя количество важных операций,
которые можно выполнять, не задумываясь".

Альфред Норт Уайтхед[6]

В те времена, когда автомобили считались редкостью, инструкции по запуску двигателя автомобиля модели Ford T занимали больше двух страниц. Для запуска двигателя современных автомашин достаточно нажать одну кнопку, и вся процедура запуска будет выполнена автоматически и безопасно. Водитель, следующий инструкции, мог залить двигатель топливом, тогда как автоматическое пусковое устройство этого просто не допустит.

Несмотря на то что разработка программного обеспечения все еще находится на стадии, сравнимой с производством автомобилей модели Ford T, мы не можем себе позволить обращаться к двухстраничным инструкциям всякий раз, когда требуется выполнить какую-нибудь типичную операцию. Такая операция, будь то процедура сборки и выпуска, тестирование, составление проектной документации или любая другая задача, повторяющаяся по ходу проекта, должна быть автоматизирована на всякой машине, где ее можно повторить.

Кроме того, требуется обеспечить согласованность и повторяемость операций в проекте. Ведь ручные процедуры оставляются на волю случая. Их повторяемость не гарантируется, особенно если порядок выполнения процедуры может по-разному интерпретироваться разными людьми.

После выхода в свет первого издания данной книги у нас возникла потребность написать больше книг, помогающих командам разрабатывать качественное программное обеспечение. И мы решили начать все с самого начала, т.е. с самых основных и важных элементов, которые требуются *каждой* команде независимо от применяемой методологии, языка программирования или комплекса технологий. И тогда у нас зародилась идея *начального набора инструментальных средств программиста-прагматика*, охватывающая следующие три крайне важные и взаимосвязанные темы.

- Контроль версий.
- Регрессионное тестирование.
- Полная автоматизация.

[6] Alfred North Whitehead — британский математик, логик, философ XX века. — *Примеч. пер.*

На эти три столпа опирается всякий проект. Ниже поясняется, каким образом это происходит.

Ведение проекта путем контроля версий

Как упоминалось в разделе "Тема 19. Контроль версий" главы 3 "Основные инструментальные средства", все, что требуется для построения проекта, должно постоянно находиться под системой контроля версий исходного кода. И этот принцип становится еще более важным в контексте самого проекта.

Прежде всего, это позволяет избавиться от сборочных машин. Вместо одной видавшей виды машины в углу помещения, к которой все боятся прикоснуться, как к священной корове[7], сборочные машины или их кластеры могут создаваться по требованию как точечные экземпляры в "облаке". Конфигурирование развертывания также должно находиться под контролем версий, чтобы все стадии от выпуска до ввода в эксплуатацию выполнялись автоматически. И при этом очень важно подчеркнуть, что на уровне проекта контроль версий приводит в действие процесс сборки и выпуска.

Совет 89 Пользуйтесь контролем версий, чтобы проводить сборки, тесты и выпуски

Это означает, что сборка, тестирование и развертывание запускается посредством операций фиксации или продвижения в системе контроля версий, а также размещения в контейнере, находящемся в "облаке". Все стадии от выпуска до размещения или ввода в эксплуатацию обозначаются отдельными маркерами в системе контроля версий. И тогда выпуски становятся намного менее церемониальными частями повседневной деятельности, способствуя непрерывной поставке, не привязанной ни к одной из вычислительных машин: как сборочных, так и разработчиков.

Строгое и непрерывное тестирование

Многие разработчики тестируют мягко, подсознательно зная, где будет нарушен исходный код, и при этом избегая слабых мест в нем. Но программисты-прагматики поступают иначе. Ими движет стремление обнаружить программные ошибки *сейчас*, чтобы не быть пристыженными другими, обнаружившими их ошибки впоследствии.

Обнаружение ошибок похоже на рыбалку с сетью. Мы используем тонкие, небольшие сети (юнит-тесты), чтобы поймать пескарей, и большие, грубые сети (тесты интеграции) для ловли акул-убийц. Иногда рыбе удается уйти, поэтому мы чиним и латаем любые отверстия, которые находим, в надежде поймать все больше и больше скользких дефектов которые плавают в нашем бассейне-проекте.

[7] Мы убеждались в этом воочию больше раз, чем вы могли бы подумать.

Совет 90 Тестируйте как можно раньше, чаще и автоматически

Тестирование исходного кода необходимо начинать как можно раньше. Ведь программные ошибки имеют скверную привычку очень быстро вырастать из мелкой рыбешки в гигантских акул-людоедов, выловить которые будет намного труднее. Поэтому программисты-прагматики пишут модульные тесты, причем в большом количестве.

В действительности качественно разрабатываемый проект отличается тем, что в нем *больше* тестового, чем рабочего кода. Время и труд, затраченные на создание тестового кода, того стоят. В долгосрочной перспективе это будет обходиться намного дешевле, а у вас появится возможность производить программные продукты практически без изъянов. Кроме того, зная, что тест прошел, вы обретаете большую степень уверенности в готовности проверенного фрагмента кода к эксплуатации.

Совет 91 Программирование нельзя считать завершенным до тех пор, пока не пройдут все тесты

При автоматической сборке выполняются все имеющиеся тесты. При этом очень важно стремиться тестировать "по-настоящему". Это означает, что среда тестирования должна как можно более точно соответствовать среде эксплуатации, поскольку любые расхождения служат местом для размножения программных ошибок.

Сборка может охватывать несколько важных видов тестирования программного обеспечения: модульное и интеграционное, проверку на достоверность и верификацию, а также тестирование производительности. Этот перечень видов тестирования ни в коем случае не следует считать исчерпывающим, поскольку в некоторых специализированных проектах могут потребоваться и другие разновидности тестирования. Тем не менее данный перечень видов тестирования может послужить неплохой отправной точкой.

Модульное тестирование

Модульный тест содержит код, проверяющий отдельный модуль, как пояснялось в разделе "Тема 41. Тестировать, чтобы кодировать" главы 7 "По ходу кодирования". Модульное тестирование служит основанием для всех остальных форм тестирования, рассматриваемых в этом разделе. Если отдельные части программы не работают по отдельности, они не будут работать и вместе. Поэтому все используемые модули должны пройти все свои модульные тесты, прежде чем вы сможете продолжить работу над проектом.

Как только все относящиеся к проекту модули пройдут модульные тесты, можно перейти к следующей стадии проекта. При этом необходимо проверить, каким образом все модули используются и взаимодействуют в системе.

Интеграционное тестирование

Результаты *интеграционного тестирования* показывают, насколько в рамках текущего проекта хорошо работают и взаимодействуют основные подсистемы. Так, если контракты правильно расставлены по местам и тщательно проверены, любые трудности интеграции могут быть легко обнаружены. В противном случае интеграция становится благодатной почвой для размножения программных ошибок. В действительности интеграция зачастую оказывается единственным и самым крупным источником программных ошибок в системе. Интеграционное тестирование на самом деле служит лишь расширением описанного выше модульного тестирования, поскольку в данном случае только проверяется, каким образом во всех подсистемах в целом учитываются их контракты.

Проверка на достоверность и верификация

Как только у вас появится выполняемый пользовательский интерфейс или его прототип, вам придется найти ответ на следующие очень важные вопросы: пользователи сообщили вам свои пожелания, но действительно ли это именно то, что им нужно? Соответствуют ли их пожелания функциональным требованиям к системе? И это также следует проверить. От бездефектной системы, дающей отрицательные ответы на эти вопросы, вряд ли будет много проку. Следует здраво оценивать схемы доступа конечных пользователей, а также их отличия от тестовых данных разработчиков (вспомните, например, историю с мазками кистью, поведанную в разделе "С чего начинать отладку" главы 3 "Основные инструментальные средства").

Тестирование производительности

Тестирование производительности и тестирование под предельной нагрузкой может также оказаться очень важной стадией работы над проектом. Спросите себя: соответствует ли программное обеспечение требованиям производительности в реальных условиях с предполагаемым числом пользователей, соединений или транзакций в секунду и допускает ли оно масштабирование? Для тестирования некоторых приложений может потребоваться специализированное испытательное оборудование или программное обеспечение, позволяющее правдоподобно сымитировать нагрузку.

Тестирование тестов

Если нельзя написать идеальное программное обеспечение, то нельзя также написать и идеальные тесты, поэтому необходимо проверять сами тесты.

В этой связи наборы тестов следует рассматривать как тщательно продуманную систему безопасности, призванную вовремя предупреждать о появлении программной ошибки. Насколько лучше протестировать систему безопасности, чем пытаться взломать ее? Как только вы напишете тест для выявления конкретной программной ошибки, вызовите ее намеренно и убедитесь, что тест предупредит о ней. Этим гарантируется, что тест отловит программную ошибку, если она действительно произойдет.

Совет 92 Пользуйтесь саботажем для проверки своих тестов

Если вы по-настоящему серьезно относитесь к тестированию, сделайте отдельное ответвление от дерева исходного кода, намеренно внедрите программные ошибки и убедитесь, что тесты отловят их. На более высоком уровне можете воспользоваться чем-то вроде инструментального средства Chaos Monkey от компании Netflix[8], чтобы нарушить функционирование служб и проверить устойчивость работы своего приложения. Когда вы пишете тесты, примите меры, чтобы они вовремя предупреждали об обнаруженных ошибках.

Тщательное тестирование

Убедившись в правильности своих тестов и обнаружив ошибки, сделанные вами в программе, как вы узнаете, что протестировали свою кодовую базу в достаточной степени тщательно? Если кратко — то никак, и вам вряд ли это вообще удастся. С этой целью можно было бы воспользоваться инструментальными средствами для *анализа покрытия тестами*, которые наблюдают за прикладным кодом во время его тестирования и отслеживают выполнявшиеся и не выполнявшиеся строки кода. Такие инструментальные средства помогают в общем понять, насколько всеобъемлюще проводимое тестирование, хотя ожидать полного покрытия проверяемого кода тестами при этом не стоит[9].

Даже если вам и удастся пройти каждую строку кода, вы все равно не получите цельную картину. Ведь реально важно число состояний, в котором может находиться программа, но состояния не равнозначны строкам кода. Допустим, имеется функция, принимающая два параметра, целочисленные значения которых могут находиться в пределах от 0 до 999:

```
int test(int a, int b) {
    return a / (a + b);
}
```

[8] См. по адресу `https://netflix.github.io/chaosmonkey`.

[9] Любопытное исследование корреляции между покрытием тестами и дефектами в проверяемом коде см. в *Mythical Unit Test Coverage* [ADSS18].

Теоретически эта функция, состоящая из трех строк кода, имеет 1 000 000 логических состояний, 999 999 из которых окажутся правильными и лишь одно — неправильным, когда `a + b = 0`. Простого знания, что данная строка кода была выполнена, еще недостаточно, чтобы сделать вывод о выявлении всех возможных состояний программы. К сожалению, в общем случае это довольно трудная задача, иллюстрируемая утверждением — "Солнце превратится в холодную твердую глыбу раньше, чем вы решите эту задачу".

Совет 93 Проверяйте покрытие тестами состояний, а не исходного кода

Тестирование на основе свойств

Чтобы проверить, каким образом в прикладном коде обрабатываются неожиданные состояния, лучше всего заставить компьютер сгенерировать эти состояния. А для того чтобы сформировать тестовые данные по контрактам и инвариантам проверяемого кода, можно прибегнуть к тестированию *на основе свойств*. Более подробно этот вопрос рассматривается в разделе "Тема 42. Тестирование на основе свойств" главы 7 "По ходу кодирования".

Затягивание сетки

Наконец, нам хотелось бы поделиться с читателем самым важным принципом тестирования. Это вполне очевидный принцип, и он поясняется буквально в каждой книге на данную тему. Но по какой-то не вполне понятной причине в большинстве проектов он по-прежнему не применяется.

Так, если программная ошибка проникает через сеть существующих тестов, для ее перехвата в следующий раз потребуется новый тест.

Совет 94 Обнаруживайте ошибки единожды

Как только тестировщик обнаружит программную ошибку, это должен быть *последний* раз, когда он обнаружил данную ошибку. Автоматизированные тесты должны быть модифицированы с целью проверять данную ошибку впредь, без всяких исключений и независимо от того, насколько она тривиальна и сколько раз разработчик будет заверять, что она больше не повторится.

Все это необходимо сделать, потому что программная ошибка может возникнуть снова. И если у вас, как всегда, нет времени на выявление ошибок, которые могли бы вместо вас обнаружить автоматизированные тесты, вам все равно придется потратить время на написание нового кода, причем с новыми программными ошибками.

Полная автоматизация

Как упоминалось в начале этого раздела, современная разработка опирается на сценарные, автоматические процедуры. Пользуетесь ли вы чем-то вроде сценариев командного процессора, или просто оболочки с командами `rsync` и `ssh`, или же такими полнофункциональными решениями, как Ansible, Puppet, Chef или Salt, ни в коем случае не полагайтесь на ручное вмешательство в процесс тестирования своего кода.

Однажды мы побывали у своего клиента, где все разработчики пользовались одной и той же интегрированной средой разработки. Их системный администратор дал каждому разработчику ряд инструкций по установке дополнительных пакетов в этой интегрированной среде разработки. Эти инструкции занимали много страниц и в основном сводились к тому, чтобы щелкнуть кнопкой мыши там, прокрутить содержимое экрана сям, перетащить то, дважды щелкнуть кнопкой мыши на этом и повторить все снова.

Не удивительно, что компьютер каждого разработчика загружался немного иначе, чем другие. И когда разные разработчики выполняли один и тот же прикладной код, в поведении приложения наблюдались незначительные отличия. Программные ошибки могли возникнуть на одном компьютере, отсутствуя при этом на остальных, а отслеживание отличий в версиях любого компонента обычно преподносило сюрпризы.

Совет 95 Не пользуйтесь ручными процедурами

Дело в том, что люди просто не в состоянии повторять процедуры с такой же точностью, как и компьютеры, да этого и нельзя ожидать от них. Сценарий оболочки или программа будет всякий раз выполнять одни и те же инструкции в неизменном порядке, самостоятельно осуществляя контроль версий. Это дает также возможность проверять изменения, происходящие в процедурах сборки и выпуска, с течением временем ("но ведь *раньше* же работало...").

Все зависит от автоматизации. Нельзя построить проект на анонимном сервере в "облаке", если не автоматизировать этот процесс полностью. И развернуть программный продукт нельзя автоматически, если этот процесс включает в себя стадии, выполняемые вручную. Стоит вам внедрить выполняемые вручную стадии хотя бы частично, вы тотчас разобьете очень большое окно[10]. Опираясь на три столпа контроля версий, строгого тестирования и полной автоматизации, ваш проект получит прочное основание, позволяющее вам сосредоточиться на самом трудном: доставить удовольствие своим пользователям.

[10] Никогда не забывайте о *программной энтропии*. Помните о ней всегда.

Другие разделы, связанные с данной темой

- **Тема 11**. Обратимость, **глава 2** "Прагматичный подход".
- **Тема 12**. Трассирующие пули, **глава 2** "Прагматичный подход".
- **Тема 17**. Игры в скорлупки, **глава 3** "Основные инструментальные средства".
- **Тема 19**. Контроль версий, **глава 3** "Основные инструментальные средства".
- **Тема 41**. Тестировать, чтобы кодировать, **глава 7** "По ходу кодирования".
- **Тема 49**. Прагматичные команды.
- **Тема 50**. Кокосами не обойтись.

Задачи

- Если сборки, выполняемые вами по ночам или непрерывно, автоматизированы, то почему не автоматизировано развертывание на рабочем сервере? В чем особенность этого сервера?
- Можете ли вы протестировать весь свой проект автоматически? Многие команды вынуждены признать, что они этого не могут. Почему? Может быть, им трудно получить приемлемые результаты? Не мешает ли им это убедить спонсоров проекта, что он "готов"?
- Если так трудно протестировать логику приложения независимо от GUI, то что это говорит о самом GUI и связывании?

Тема 52 Доставляйте удовольствие своим пользователям

"Когда вы обвораживаете людей, ваша цель — не выманить у них деньги или заставить их сделать то, что вам требуется, а доставить им огромное удовольствие".

Гай Кавасаки[11]

Наша цель как разработчиков — *доставлять удовольствие* своим пользователям. Именно для этого мы создаем программные продукты, а не для того, чтобы добыть личные данные пользователей, привлечь их внимание или опорожнить

[11] Guy Kawasaki — один из самых известных сотрудников компании Apple Computer, внесший в маркетинг компьютера Macintosh образца 1984 г. концепцию "евангелизма", ныне повсеместно применяемую в вычислительной технике. — *Примеч. пер.*

их кошельки. Но если оставить в стороне злонамеренные цели, то даже своевременной поставки работоспособного программного обеспечения окажется явно недостаточно. Одно лишь это не доставит удовольствие пользователям.

Ваших пользователей не особенно вдохновляет написанный вами код. Напротив, им нужно решать экономические задачи в контексте их собственных целей и бюджета. И они верят, что, сотрудничая с вашей командой, сумеют это сделать. Они связывают свои ожидания непосредственно с программным обеспечением. Они даже не полностью отдают себе отчет в спецификации, которую передают вашей команде на рассмотрение, поскольку эта спецификация будет неполной до тех пор, пока ваша команда не переработает ее несколько раз подряд.

Как же тогда выявить ожидания пользователей? Для этого следует задать простой вопрос:

> *Как знать, будет ли все успешно работать через месяц, год или иной срок после завершения проекта?*

Ответ на этот вопрос может вас удивить. Проект, призванный усовершенствовать рекомендации по применению программного продукта, на самом деле можно оценивать с точки зрения сохранения клиентской базы, а проект, призванный соединить две базы данных, — с точки зрения качества данных, а возможно, и экономии затрат. Но на самом деле значение имеют лишь ожидания коммерческой ценности, но не сам программный проект. Ведь программное обеспечение служит лишь средством для достижения этих целей.

И как только удастся прояснить исходные ожидания коммерческой ценности, стоящей за проектом, можно приступить к обдумыванию способов получения такой ценности, чтобы удовлетворить эти ожидания, выполнив перечисленные ниже действия.

- Примите меры, чтобы все члены вашей команды ясно понимали эти ожидания.
- Принимая решения, подумайте, какой путь быстрее всего удовлетворит эти ожидания.
- Критически проанализируйте требования пользователей в свете их ожиданий. Во многих проектах нам не раз приходилось обнаруживать, что сформулированное "требование" на самом деле оказывалось всего лишь предположением о возможностях конкретной технологии. Это был на самом деле любительский план реализации, облаченный в ризы документально оформленных требований. Не бойтесь делать предположения, изменяющие требования, если можете продемонстрировать, что они способны приблизить проект к конечной цели.
- Продолжайте обдумывать эти ожидания, продвигаясь дальше в работе над проектом.

Как показывает наш опыт, по мере того, как мы лучше узнаем предметную область, нам удается точнее делать свои предположения относительно возможных путей решения исходных задач в этой области. Мы твердо уверены, что разработчики, которым открыты разные особенности функционирования конкретной организации, нередко способны найти способы связать вместе различные части ее деятельности, не всегда очевидные ее отдельным подразделениям.

Совет 96 Доставляйте пользователям не просто код, а удовольствие

Если вы хотите доставить удовольствие своему клиенту, установите с ним такие отношения, которые позволят вам активно помогать ему решать стоящие перед ним задачи. Даже если ваша должность называется "разработчик программного обеспечения" или "инженер-программист", на самом деле она должна называться "решатель задач". Ведь программисты-прагматики, по существу, занимаются именно тем, что постоянно решают поставленные перед ними задачи.

Другие разделы, связанные с данной темой

- **Тема 12.** Трассирующие пули, **глава 2** "Прагматичный подход".
- **Тема 13.** Прототипы и памятные записки, **глава 2** "Прагматичный подход".
- **Тема 45.** Западня требований, **глава 8** "До начала проекта".

Тема 53 Гордость и предубеждение

"Вы довольно долго радовали нас".

Джейн Остин, ***Гордость и предубеждение***

Программисты-прагматики не уклоняются от ответственности. Напротив, они только рады принять вызов и сделать свой опыт и знания общим достоянием. Если программисты-прагматики отвечают за проектное решение или фрагмент кода, они выполняют свою работу так, чтобы ею можно было гордиться.

Совет 97 Подписывайте свою работу

В прежние времена ремесленники с гордостью подписывали свои работы, и вам следует поступать точно так же. Но проектные команды по-прежнему состоят из разных людей, и поэтому соблюдение данного правила может доставить немало хлопот. В некоторых проектах идея *владения кодом* может за-

труднить сотрудничество. Люди могут ревностно охранять свои границы или неохотно работать над элементами общего пользования. В итоге проект может стать похожим на целый ряд обособленных мелких княжеств. А вы можете быть предубеждены относительно своего кода и настроены против своих коллег.

Но ведь требуется совсем другое. С одной стороны, вам не следует ревностно защищать свой код от вторгающихся в него чужаков. А с другой стороны, вы должны уважительно относиться к чужому коду. Соблюдение золотого правила "Поступайте с людьми так, как вы хотели бы, чтобы они поступали с вами" и основание для взаимного уважения среди разработчиков имеют решающее значение для того, чтобы следовать приведенному выше совету на практике.

Анонимность, особенно в крупных проектах, может послужить благоприятной почвой для развития в команде таких неприятных качеств, как небрежность, оплошности, леность, а в конечном счете — некачественный код. И тогда можно легко себе внушить, что вы просто мелкая сошка, теша себя слабым оправданием, что от вас мало что зависит, вместо того, чтобы писать качественный код.

И хотя у написанного кода должен быть владелец, он не должен принадлежать кому-то одному. В действительности в своем скромном введении в экстремальное программирование[12] Кент Бек рекомендует общее владение кодом. Хотя для защиты от опасностей, кроющихся в анонимности, потребуются дополнительные нормы практики (например, парное программирование).

Нам хотелось бы видеть гордость владения кодом, выражаемую словами: "Я это написал и отвечаю за свою работу". Ваша подпись должна стать признанным знаком качества, а люди должны видеть ваше имя на фрагменте кода, ожидая, что он написан грамотно, тщательно проверен, подробно задокументирован и будет работать надежно. Это должна быть профессиональная работа, написанная профессионалом — программистом-прагматиком.

Благодарим вас за внимание.

[12] См. по адресу `http://www.extremeprogramming.org`.

Приложение A

Послесловие

"В долгосрочной перспективе мы формируем свою жизнь сами. И этот процесс никогда не кончается вплоть до нашей смерти. В конечном счете мы несем ответственность за свой выбор.

Элеонора Рузвельт[1]

Когда наша профессиональная деятельность послужила основанием для написания первого издания данной книги двадцать лет назад, мы участвовали в эволюции вычислительной техники от частного любопытства к современному императиву всякой деятельности. Спустя двадцать лет, программное обеспечение выросло за пределы ЭВМ для решения коммерческих задач и по-настоящему овладело всем миром. Но что это действительно значит для нас?

В своей книге *The Mythical Man-Month: Essays on Software Engineering* [Bro96] Фред Брукс пишет: "Труд программиста, как и поэта, лишь отчасти не является плодом чистого разума. Он строит свои замки в воздухе и из воздуха, создавая их усилием своего воображения". Начиная с чистого листа, мы можем создать все, что может нам подсказать наше воображение. И все, что мы создаем, способно изменить мир.

Все технологические новшества (от социальной сети Twitter, позволяющей людям планировать революции, процессора в бортовом компьютере, помогающем вести автомобиль по скользкой дороге, и вплоть до смартфона) избавляют нас от необходимости запоминать мелкие повседневные подробности. Нас всюду окружают программы, и нас нигде и никогда не покидает воображение.

Мы, разработчики, находимся в весьма привилегированном положении, поскольку действительно строим будущее. Это чрезвычайно большая власть, которая требует необычайной ответственности. Как часто мы перестаем думать об этом? И как часто обсуждаем в своем узком или в более широком кругу, что это на самом деле означает?

---

[1] Eleanor Roosevelt — американская общественная деятельница, супруга президента США Франклина Делано Рузвельта. — *Примеч. пер.*

Во встроенных устройствах применяется на порядок больше компьютеров, чем на переносных и настольных компьютерах или в центрах обработки данных. Встроенные компьютеры нередко управляют жизненно важными системами: от электростанций до автомашин и медицинских приборов. И даже простая система центрального отопления или бытовая техника может оказаться смертельно опасной, если она спроектирована или реализована неграмотно. Когда вы разрабатываете программы для таких систем и устройств, вы берете на себя непомерную ответственность.

Многие невстроенные системы могут также быть как большим благом, так и нанести немалый вред. Так, социальные сети могут способствовать мирной революции или дать повод для отвратительной ненависти. Большие данные могут как упростить совершение покупок, так и разрушить всякие остатки конфиденциальности, которые, как вам кажется, у вас еще имеются. Банковские системы способны принимать решения о займах, коренным образом меняющие жизнь людей. И практически любая система может быть использована для соглядатайства за своими пользователями.

Мы уже заметили признаки возможного утопического будущего, а также видели примеры неумышленных последствий, ведущих к кошмарным антиутопиям. Отличия этих двух последствий могут оказаться менее заметными, чем вы думаете. И это все в ваших руках.

Нравственный ориентир

Ценой, которую приходится платить за столь неожиданно полученную власть, является бдительность. Ведь наши действия оказывают непосредственное воздействие на людей. Никто больше не программирует в качестве любителя на компьютере с 8-разрядным процессором в гараже, не выполняет в пакетном режиме технологический процесс на изолированной от внешнего мира большой ЭВМ в центре обработки данных или даже на настольном компьютере. Наше программное обеспечение вплетено в саму ткань повседневной современной жизни.

Мы просто обязаны задавать себе два следующих вопроса о каждом фрагменте кода, который выпускаем.

1. Защитил ли я пользователя?
2. Воспользовался бы я этим кодом сам?

Во-первых, следует себя спросить: "Сделал ли я все от меня зависящее, чтобы защитить пользователей данного кода от возможного нанесения им вреда? Принял ли я меры к постоянной вставке "заплат" в простую систему слежения за ребенком? Обеспечил ли я ручное управление системой центрального отопления на случай выхода из строя автоматического термостата? Храню ли я только нужные мне данные и шифрую ли любые личные данные?"

Никто не идеален, и все время от времени упускают что-нибудь из виду. Но если вы не можете честно сказать, что предусмотрели все возможные последствия и приняли необходимые меры, чтобы защитить от них пользователей, то вы понесете определенную ответственность, если дело приобретет скверный оборот.

Совет 98 Прежде всего, не нанесите вред

Во-вторых, упомянутое выше золотое правило можно было бы переосмыслить, задав следующие вопросы: "Был бы я рад воспользоваться этим программным обеспечением? Хотел бы я поделиться своими подробностями или сделать мои перемещения достоянием розничных торговых точек? Был бы я рад воспользоваться данным автономным транспортным средством и насколько это было бы мне удобно?"

Некоторые изобретательные идеи начинают обходить границы этического поведения, и если вы вовлечены в такой проект, то несете такую же ответственность, как и его спонсоры. Независимо от того, сколько степеней разделения вы могли бы обосновать, остается в силе следующее правило:

Совет 99 Не потакайте всякой шушере

Представляйте будущее таким, каким вы хотите его видеть

Это ваше дело, ваше воображение, ваши надежды и ваши заботы, дающие пищу для размышлений чистого разума на последующие двадцать и больше лет. Вы строите будущее для себя и своих потомков, и ваш долг — сделать такое будущее, какое все мы хотели бы унаследовать. Признайте это, когда пытаетесь делать что-нибудь против такого идеала и имеете смелость противиться ему. Представьте будущее таким, каким бы мы могли его иметь, и найдите в себе смелость создать его. Стройте воздушные замки каждый день. Ведь у всех у нас замечательная жизнь.

Совет 100 Это ваша жизнь.
Делитесь ею, празднуйте и стройте ее.
И ЖЕЛАЕМ ПОЛУЧИТЬ ОТ ЭТОГО УДОВОЛЬСТВИЕ!

Приложение Б

Библиография

[ADSS18] Vard Antinyan, Jesper Derehag, Anna Sandberg, and Miroslaw Staron. *Mythical Unit Test Coverage. IEEE Software*. 35:73-79, 2018

[And10] Jackie Andrade. *What does doodling do? Applied Cognitive Psychology*. 24(1):100-106, 2010, January.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, Raleigh, NC, 2007.

[BR89] Albert J. Bernstein and Sydney Craft Rozen. *Dinosaur Brains: Dealing with All Those Impossible People at Work*. John Wiley & Sons, New York, NY, 1989.

[Bro96] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, Anniversary, 1996.

[CN91] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, MA, Second, 1991.

[Con68] Melvin E. Conway. How do Committees Invent? *Datamation*. 14(5):28-31, 1968, April.

[de 98] Gavin de Becker. *The Gift of Fear: And Other Survival Signals That Protect Us from Violence*. Dell Publishing, New York City, 1998.

[DL13] Tom DeMacro and Tim Lister. *Peopleware: Productive Projects and Teams*. Addison-Wesley, Boston, MA, Third, 2013.

[Fow00] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, MA, Second, 2000.

[Fow04] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, MA, Third, 2004.

[Fow19] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, Second, 2019.
Русский перевод: Мартин Фаулер. *Рефакторинг кода на JavaScript: улучшение проекта существующего кода. 2-е изд.* — СПб.:"Диалектика", 2019.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Hol92] Michael Holt. *Math Puzzles & Games*. Dorset House, New York, NY, 1992.

[Hun08] Andy Hunt. *Pragmatic Thinking and Learning: Refactor Your Wetware*. The Pragmatic Bookshelf, Raleigh, NC, 2008.

[Joi94] T.E. Joiner. Contagious depression: Existence, specificity to depressed symptoms, and the role of reassurance seeking. *Journal of Personality and Social Psychology*. 67(2):287–296, 1994, August.

[Knu11] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley, Boston, MA, 2011.

Русский аперевод: Дональд Эрвин Кнут. *Искусство программирования, том 4A. Комбинаторные алгоритмы, часть 1*. — М.:Издательский дом "Вильямс", 2013.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA, Third, 1998.

Русский аперевод: Кнут Дональд Эрвин. *Искусство программирования, том 1. Основные алгоритмы, 3-е изд.* — М.:Издательский дом "Вильямс", 2000.

[Knu98a] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, Third, 1998.

Русский аперевод: Кнут Дональд Эрвин. *Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.* — М.:Издательский дом "Вильямс", 2000.

[Knu98b] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, MA, Second, 1998.

Русский перевод: Кнут Дональд Эрвин. *Искусство программирования, том 3. Сортировка и поиск, 2-е изд.* — М.:Издательский дом "Вильямс", 2000.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Reading, MA, 1999. Русский перевод: Брайан У. Керниган, Роб Пайк. *Практика программирования* — М.:Издательский дом "Вильямс", 2005.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, Second, 1997.

[Mul18] Jerry Z. Muller. *The Tyranny of Metrics*. Princeton University Press, Princeton NJ, 2018.

[SF13] Robert Sedgewick and Phillipe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Boston, MA, Second, 2013.

[Str35] James Ridley Stroop. *Studies of Interference in Serial Verbal Reactions*. *Journal of Experimental Psychology*. 18:643–662, 1935.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, Boston, MA, Fourth, 2011. Русский перевод: Роберт Седжвик, Кевин Уэйн. *Алгоритмы на Java, 4-е изд.* — М.:Издательский дом "Вильямс", 2012.

[Tal10] Nassim Nicholas Taleb. *The Black Swan: Second Edition: The Impact of the Highly Improbable*. Random House, New York, NY, Second, 2010.

[WH82] James Q. Wilson and George Helling. *The police and neighborhood safety*. *The Atlantic Monthly*. 249[3]:29–38, 1982, March.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, Englewood Cliffs, NJ, 1979.

[You95] Edward Yourdon. *When good-enough software is best*. *IEEE Software*. 1995, May.

Приложение B

Возможные ответы на упражнения

"Я бы предпочел иметь вопросы, на которые нельзя найти ответы, чем ответы, на которые нельзя поставить вопросы".

Ричард Фейнман[1]

Ответ на упражнение 1

Мы думаем, что класс `Split2` более ортогональный. Он сосредоточен на собственной задаче (разбиении строк), пренебрегая такими подробностями, как место поступления строк. Благодаря этому код становится не только легче разрабатывать, но и повышается его гибкость. Класс `Split2` можно разбить на строки, читаемые из файла, сформированного другой подпрограммой, или же передаваемые через среду.

Ответ на упражнение 2

Итак, начнем со следующего утверждения: качественный, ортогональный код можно написать практически на любом языке. В то же время у каждого языка имеются свои приманки, т.е. средства, способные привести к увеличению степени связывания и уменьшению ортогональности.

В языках ООП такие средства, как множественное наследование, исключения, перегрузка операций и переопределение родительских методов (через наследование), предоставляют обширные возможности увеличить степень связывания не вполне очевидными путями. Определенное связывание возникает и потому, что код связывается с данными в самом классе. И как правило, это хорошо (когда связывание — благо, мы называем его сцеплением). Но если не сделать классы в достаточной степени сосредоточенными на решении конкретных задач, что это может привести к появлению довольно скверных интерфейсов.

[1] Ричард Фейнман — американский физик-теоретик XX века. — *Примеч. пер.*

В языках функционального программирования поощряется написание большого количества небольших развязанных функций, а также их объединение в самых разных сочетаниях для решения конкретной задачи. В теории это кажется замечательным, да и на практике все зачастую оказывается именно так. Но при этом может все же возникнуть некоторая форма связывания. Подобные функции, как правило, преобразуют данные, а это означает, что результат выполнения одной функции может стать входными данными другой. И если не проявить внимательность, то изменение формата данных в одной функции может привести к сбою где-нибудь дальше в потоке преобразования этих данных. Хорошие системы типов в языках функционального программирования отчасти помогают устранять подобные затруднения.

Ответ на упражнение 3

Спасение — в простых технологиях! Сделайте несколько эскизов автомобиля, телефона или дома с маркерами на белой доске. Это совсем не обязательно должны быть произведения искусства — можно обойтись и контурными схематическими рисунками. Затем разместите памятные записки, описывающие содержимое целевых страниц, на участках, активизируемых щелчком мышью. По ходу совещания можете уточнить рисунки и расположение памятных записок.

Ответ на упражнение 4

Этот язык требуется сделать расширяемым, поэтому он должен приводиться в действие с помощью таблицы синтаксического анализатора, каждая ячейка которой содержит букву выполняемой команды, признак, указывающий, требуется ли аргумент, а также имя подпрограммы, вызываемой для выполнения данной конкретной команды, как показано ниже.

lang/turtle.c

```
typedef struct {
  char cmd;   /* буква выполняемой команды */
  int hasArg; /* признак, указывающий, требуется ли аргумент */
  void (*func)(int, int); /* вызываемая подпрограмма */
} Command;

static Command cmds[] = {
  { 'P', ARG, doSelectPen },
  { 'U', NO_ARG, doPenUp },
  { 'D', NO_ARG, doPenDown },
  { 'N', ARG, doPenDir },
  { 'E', ARG, doPenDir },
  { 'S', ARG, doPenDir },
  { 'W', ARG, doPenDir }
};
```

Основная программа довольно проста. Она читает строку, находит команду, получает аргумент, если таковой требуется, а затем вызывает функцию обработки, как показано ниже.

lang/turtle.c

```
while (fgets(buff, sizeof(buff), stdin)) {
  Command *cmd = findCommand(*buff);
  if (cmd) {
    int arg = 0;
    if (cmd->hasArg && !getArg(buff+1, &arg)) {
      fprintf(stderr, "'%c' needs an argument\n", *buff);
      continue;
    }
    cmd->func(*buff, arg);
  }
}
```

Приведенная ниже функция, находящая команду, выполняет линейный поиск в таблице, возвращая совпавшую ячейку или пустое значение NULL.

lang/turtle.c

```
Command *findCommand(int cmd) {
  int i;
  for (i = 0; i < ARRAY_SIZE(cmds); i++) {
    if (cmds[i].cmd == cmd)
      return cmds + i;
  }
  fprintf(stderr, "Unknown command '%c'\n", cmd);
   return 0;
}
```

Наконец, чтение числового аргумента реализуется довольно просто с помощью функции `sscanf()`:

lang/turtle.c

```
int getArg(const char *buff, int *result) {
  return sscanf(buff, "%d", result) == 1;
}
```

Ответ на упражнение 5

Эта задача была фактически решена в предыдущем упражнении, где написан интерпретатор внешнего языка. В примерах исходного кода к данной книге этот внутренний интерпретатор реализуется с помощью функций типа do*Xxx*.

Ответ на упражнение 6

Используя запись BNF, спецификацию формата времени можно было бы определить следующим образом:

```
time    ::= hour ampm | hour : minute ampm | hour : minute
ampm    ::= am | pm
hour    ::= digit | digit digit
minute  ::= digit digit
digit   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Чтобы еще лучше определить формат *часов* и *минут*, следует учесть, что часы могут изменяться в пределах от 00 до 23, а минуты — в пределах от 00 до 59, как показано ниже.

```
hour    ::= h-tens digit | digit
minute  ::= m-tens digit
h-tens  ::= 0 | 1
m-tens  ::= 0 | 1 | 2 | 3 | 4 | 5
digit   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Ответ на упражнение 7

Ниже приведен синтаксический анализатор, написанный на языке JavaScript с помощью библиотеки Pegjs.

lang/peg_parser/time_parser.pegjs

```
time
  = h:hour offset:ampm { return h + offset }
  / h:hour ":" m:minute offset:ampm { return h + m + offset }
  / h:hour ":" m:minute { return h + m }

ampm
  = "am" { return 0 }
  / "pm" { return 12*60 }

hour
  = h:two_hour_digits { return h*60 }
  / h:digit { return h*60 }

minute
  = d1:[0-5] d2:[0-9] { return parseInt(d1+d2, 10); }

digit
  = digit:[0-9] { return parseInt(digit, 10); }

two_hour_digits
  = d1:[01] d2:[0-9 ] { return parseInt(d1+d2, 10); }
  / d1:[2] d2:[0-3] { return parseInt(d1+d2, 10); }
```

А так выглядит тест, демонстрирующий применение этого синтаксического анализатора:

lang/peg_parser/test_time_parser.js

```
let test = require('tape');
let time_parser = require('./time_parser.js');

// time     ::= hour ampm |
//              hour : minute ampm |
//              hour : minute
//
// ampm     ::= am | pm
//
// hour     ::= digit | digit digit
//
// minute   ::= digit digit
//
// digit    ::= 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

const h  = (val) => val*60;
const m  = (val) => val;
const am = (val) => val;
const pm = (val) => val + h(12);

let tests = {
  "1am": h(1),
  "1pm": pm(h(1)),
   "2:30": h(2) + m(30),
  "14:30": pm(h(2)) + m(30),
  "2:30pm": pm(h(2)) + m(30),
}

test('time parsing', function (t) {
    for (const string in tests) {
      let result = time_parser.parse(string)
      t.equal(result, tests[string], string);
    }
    t.end()
});
```

Ответ на упражнение 8

Ниже приведено одно из возможных решений на языке Ruby.

lang/re_parser/time_parser.rb

```
TIME_RE = %r{
(?<digit>[0-9]){0}
(?<h_ten>[0-1]){0}
(?<m_ten>[0-6]){0}
(?<ampm> am | pm){0}
(?<hour> (\g<h_ten> \g<digit>) | \g<digit>){0}
(?<minute> \g<m_ten> \g<digit>){0}

\A(
     ( \g<hour> \g<ampm> )
```

```
    | ( \g<hour> : \g<minute> \g<ampm> )
    | ( \g<hour> : \g<minute> )
)\Z

}x

def parse_time(string)
  result = TIME_RE.match(string)
  if result
    result[:hour].to_i * 60 +
    (result[:minute] || "0").to_i +
    (result[:ampm] == "pm" ? 12*60 : 0)
  end
end
```

В приведенном выше коде применяется специальный прием сначала для определения именованных шаблонов в начале регулярного выражения, а затем для обращения к ним как к подшаблонам при фактическом сопоставлении.

Ответ на упражнение 9

Наш ответ должен быть основан на нескольких предположениях.

- Информация, которую требуется передать, хранится в запоминающем устройстве.
- Нам известна скорость ходьбы человека.
- Нам известно расстояние между компьютерами.
- Мы не учитываем время, требующееся для передачи информации как в запоминающее устройство, так из него.
- Издержки на хранение данных приблизительно равны издержкам на отправку данных по линии связи.

Ответ на упражнение 10

С учетом допущений, сделанных в предыдущем ответе, на накопителе на магнитной ленте емкостью 1 Тбайт хранится 8×2^{40}, или 2^{43} бит, поэтому данные равнозначного объема могут быть переданы по линии связи с пропускной способностью 1 Гбит/с приблизительно за 9000 секунд, или около 2,5 часа. Если человек ходит с постоянной скоростью 5,6 км/час, то компьютеры должны отстоять один от другого на расстоянии почти 14,5 км, чтобы линия связи превзошла человека по скорости передачи данных, а иначе он превзойдет ее.

Ответ на упражнение 14

Приведем ниже сигнатуры функций на языке Java с пред- и постусловиями в комментариях.

Сначала покажем инвариант для класса:

```
/**
  * @invariant getSpeed() > 0
  *      implies isFull()                    // не выполнять, если пусто!
  *
  * @invariant getSpeed() >= 0 &&
  *       getSpeed() < 10                    // проверить пределы
  */
```

Затем пред- и постусловия:

```
/**
  * @pre Math.abs(getSpeed() - x) <= 1   // изменить только на единицу
  * @pre x >= 0 && x < 10                // проверить пределы
  * @post getSpeed() == x              // учесть запрашиваемую скорость
  */
public void setSpeed(final int x)
/**
  * @pre !isFull()                       // не заполнять дважды
  * @post isFull()                       // удостовериться в выполнении
  */

void fill()
/**
  * @pre isFull()                        // не заполнять дважды
  * @post !isFull()                      // удостовериться в выполнении
  */
void empty()
```

Ответ на упражнение 15

В этом ряду 21 число. Если же вы скажете, что их 20, то совершите ошибку, подсчитывая промежутки между числами, а не сами числа.

Ответ на упражнение 16

- В сентябре 1752 года было только 19 дней[2]. Это было сделано специально, чтобы синхронизировать календари как часть Григорианской реформы.
- Каталог мог быть удален в другом процессе, у вас могло не быть разрешения на чтение его содержимого, диск мог быть не смонтирован — в общем, картина должна быть вам ясна.
- Мы специально не указали типы переменных `a` и `b`. Перегрузка операций `+`, `=` или `!=` могла бы иметь неожиданные последствия. Кроме того, обозначения `a` и `b` могут быть псевдонимами одной и той же переменной, поэтому при втором присваивании будет перезаписано значение, сохраняемое при первом присваивании. А если программа выполняется в параллельном режиме и написана неудачно, то значение переменной `a` может быть обновлено ко времени выполнения операции сложения.

[2] Речь идет о переходе на Григорианский календарь Великобритании. В разных странах этот переход (с появлением укороченных месяцев) происходил в разные годы. — *Примеч. ред.*

- В неевклидовой геометрии сумма углов треугольника не будет равна 180°. Представьте треугольник, начерченный на поверхности сферы.
- Високосные минуты могут насчитывать 61 или 62 секунды.
- В зависимости от конкретного языка числовое переполнение может оставить результат операции `a+1` отрицательным.

Ответ на упражнение 17

В большинстве реализаций C и C++ никак не проверяется, ссылается ли фактически указатель на действительную область памяти. Типичная ошибка состоит в том, что блок памяти освобождается вместе со ссылкой на него далее в программе. Но к тому времени область памяти, на которую делается ссылка по указателю, может быть вполне выделена уже для других целей. Устанавливая пустое значение `NULL` в указателе, программисты надеются предотвратить эти ошибки.

Ответ на упражнение 18

Устанавливая значение ссылки `NULL`, вы тем самым сокращаете на единицу количество указателей на ссылаемый объект. И как только это количество достигнет нуля, объект может стать пригодным для сборки мусора. Установка значения ссылок равным `NULL` может иметь существенное значение для долго выполняющихся программ, где программисты должны гарантировать, что объем используемой памяти со временем не увеличится.

Ответ на упражнение 19

Простая реализация данного компонента может выглядеть следующим образом:

event/strings_ex_1.rb

```
class FSM
  def initialize(transitions, initial_state)
    @transitions = transitions
    @state = initial_state
  end
  def accept(event)
     @state, action = TRANSITIONS[@state][event]
                   || TRANSITIONS[@state][:default]
  end
end
```

Ответ на упражнение 20

- ...получены три события о *выходе из строя* сетевого интерфейса в течение пяти минут.

Такое поведение можно было бы реализовать с помощью конечного автомата, но сделать это было бы труднее, чем кажется на первый взгляд. Ведь если получить события в 1-ю, 4-ю, 7-ю и 8-ю минуты, то придется выдать предупреждение на четвертом событии. Это означает, что конечному автомату придется сбросить себя в исходное состояние.

По этой причине избранной технологией, по-видимому, могут оказаться потоки событий. У этих потоков имеется реактивная функция `buffer()`, принимающая параметры `size` и `offset` и возвращающая каждую группу из трех входящих событий. И тогда можно проанализировать отметки времени наступления первого и последнего события в группе, чтобы выяснить, следует ли выдавать предупреждение.

- ...после захода солнца обнаруживается движение сначала внизу, а затем вверху лестницы, включить лестничное освещение.

  Этот режим, вероятно, можно было бы реализовать, используя определенное сочетание шаблона "Издатель–подписчик" и конечных автоматов. Так, используя шаблон "Издатель–подписчик", можно было бы сначала распространить события среди любого количества конечных автоматов, а затем дать конечным автоматам возможность определить, что именно следует делать.

- ...уведомить различные системы учета о выполнении заказа.

  Этот режим лучше всего было бы реализовать, используя шаблон "Издатель–подписчик". И хотя для этой цели можно было бы воспользоваться потоками событий, но тогда и уведомляемые системы учета пришлось бы проектировать на основе потоков.

- ...отправить соответствующие запросы трем серверным службам, ожидая от них ответов.

  Это похоже на рассмотренный пример, в котором потоки используются для выборки пользовательских данных.

Ответ на упражнение 21

1. Налоги на поставку и с продаж вводятся в заказ:

   ```
   основной заказ → окончательный заказ
   ```

   В обычном коде, вероятнее всего, должна быть определена одна функция для расчета затрат на поставку, а другая — для расчета налога. Но если принять во внимание выполняемые здесь преобразования, то заказ просто с перечисленными в нем товарами можно преобразовать в новый вид заказа на поставку.

2. Приложение загружает конфигурацию из именованного файла:

   ```
   имя файла → структура конфигурации
   ```

3. Пользователь входит в веб-приложение, регистрируясь в нем:

```
учетные данные пользователя → сеанс работы
```

Ответ на упражнение 22

Исходное высокоуровневое преобразование

```
field contents as string
     → [validate & convert]
        → {:ok, value} | {:error, reason}
```

можно разделить следующим образом:

```
field contents as string
     → [convert string to integer]
        → [check value >= 18]
        → [check value <= 150]
        → {:ok, value} | {:error, reason}
```

При этом предполагается наличие конвейера для обработки ошибок.

Ответ на упражнение 23

Ответим сначала на вторую часть данного упражнения: мы предпочитаем первый фрагмент кода.

Во втором фрагменте кода на каждом шаге возвращается объект, реализующий следующий вызов метода. В частности, объект, возвращаемый методом `content_of()`, должен реализовать метод `find_matching_lines()` и т.д.

Это означает, что объект, возвращаемый методом `content_of()`, привязан к нашему коду. Представьте, что требование изменилось, и тогда нам придется пренебречь строками, начинающимися со знака #. В стиле преобразований это нетрудно выразить следующим образом:

```
const content     = File.read(file_name);
const no_comments = remove_comments(content)
const lines       = find_matching_lines(no_comments, pattern)
const result      = truncate_lines(lines)
```

Этот код будет работать, даже если поменять порядок вызова методов `remove_comments()` и `find_matching_lines()`, хотя сделать это в стиле составления методов в цепочку будет труднее. А где должен находиться метод `remove_comments()`: в объекте, возвращаемом методом `content_of()`, или же в объекте, возвращаемом методом `find_matching_lines()`? И нарушится ли работа какого-нибудь другого кода, если изменить такой объект? Вследствие именно такого связывания стиль составления методов иногда еще называют *крушением поездов*.

Ответ на упражнение 24

- **Обработка изображений.** Для простого планирования рабочей нагрузки на параллельные процессы может лучше всего подойти общая рабочая очередь. А если требуется учитывать ответную реакцию, т.е. воздействие результатов обработки одного участка изображения на другие его участки, как, например, в приложениях машинного зрения или сложных трехмерных преобразованиях, искажающих изображение, то можно рассмотреть возможность применения системы типа классной доски.
- **Групповое ведение календаря.** Система типа классной доски может быть в данном случае вполне пригодным вариантом. Она позволит оповещать на классной доске о планируемых совещаниях и сделать ее доступной для всех заинтересованных лиц. Отдельные подразделения могут функционировать автономно, но при этом важна их реакция на принимаемые решения, а участники проекта могут приходить и уходить.

  Можно также рассмотреть возможность разделения такого рода системы типа классной доски на отдельные категории в зависимости от того, кто осуществляет в ней поиск. Так, младший персонал может интересовать только то, что происходит непосредственно в их учреждении, отдел кадров — англоязычные подразделения по всему миру, а высшее руководство — все в целом.

  Допускается также определенная гибкость в отношении форматов данных, позволяющая не обращать внимание на непонятные форматы или языки. Понимание разных форматов данных требуется только в тех подразделениях, которые принимают участие в совместных совещаниях, хотя их участникам совсем не обязательно знать все возможные форматы. Благодаря этому связывание сокращается до нужной степени, не налагая никаких искусственных ограничений.
- **Инструментальное средство текущего контроля сети.** Это очень похоже на прикладную программу, принимающую и обрабатывающую заявки на залог или заем (см. раздел "Классная доска в действии" главы 6 "Параллельность"). В данном случае от пользователей поступают сообщения о неполадках в сети и автоматически составляются статистические отчеты — и все это размещается на классной доске. Одушевленный или неодушевленный (программный) посредник может анализировать содержимое классной доски, чтобы диагностировать сетевые сбои. Так, две ошибки в линии связи могут быть вызваны всего лишь воздействием космических лучей, но 20 тысяч ошибок явно означают неисправность сетевого оборудования. Подобно тому, как сыщики раскрывают тайну убийства совместными усилиями, работу нескольких подразделений можно организовать таким образом, чтобы они анализировали и предлагали варианты устранения неполадок в сети.

Ответ на упражнение 25

Когда речь идет о списке, состоящем из пар "ключ–значение", обычно считается, что ключи в нем уникальны. И библиотеки хеширования обычно соблюдают это правило, обеспечивая надлежащее поведение самого хеш-кода или выдавая сообщения об ошибках дублирования ключей. Но на массив такие ограничения, как правило, не накладываются, поэтому в нем могут благополучно храниться дублирующиеся ключи, если только в прикладном коде против этого не приняты специальные меры. Так, в данном случае одерживает верх первая же запись, найденная по совпадению с заданным ключом `DepositAccount`, тогда как все остальные записи игнорируются. При этом порядок следования записей не гарантируется, поэтому в одних случаях такая структура данных оказывается пригодной, а в других — непригодной. А что касается компьютеров, не задействованных на стадии разработки и эксплуатации, то речь может идти только о простом совпадении.

Ответ на упражнение 26

Тот факт, что исключительно числовое поле вполне пригодно для хранения телефонных номеров в США, Канаде и на Карибских островах, является простым совпадением. По спецификации ITU формат международного телефонного звонка начинается со знака +. В некоторых местных телефонных сетях употребляется также знак *, и зачастую номер телефона может содержать начальные нули. Поэтому номера телефонов ни в коем случае нельзя хранить в числовом поле.

Ответ на упражнение 27

Все зависит от того, где именно вы находитесь. Так, в США единицы измерения объема основываются на галлоне, обозначающем объем цилиндра высотой 6 дюймов и диаметром 7 дюймов, округленный до ближайшего кубического дюйма. А в Канаде "одна чашка" в рецепте может обозначать одну из следующих мер объема:

- 1/5 британской кварты, или 227 мл
- 1/4 американской кварты, или 236 мл
- 26 метрических столовых ложек, или 240 мл
- 1/4 литра, или 250 мл

Если только речь не идет о рисоварке, где "одна чашка" обозначает объем 180 мл. Эта мера объема происходит от японской единицы измерения *коку*, устанавливающей объем сухого риса, требующегося одному человеку на один год, т.е. около 180 л. Чашки для рисоварки имеют объем 1 *го*, т.е. 1/1000 коку, а следовательно, они обозначают количество риса, которое человек обычно съедает за один присест[3].

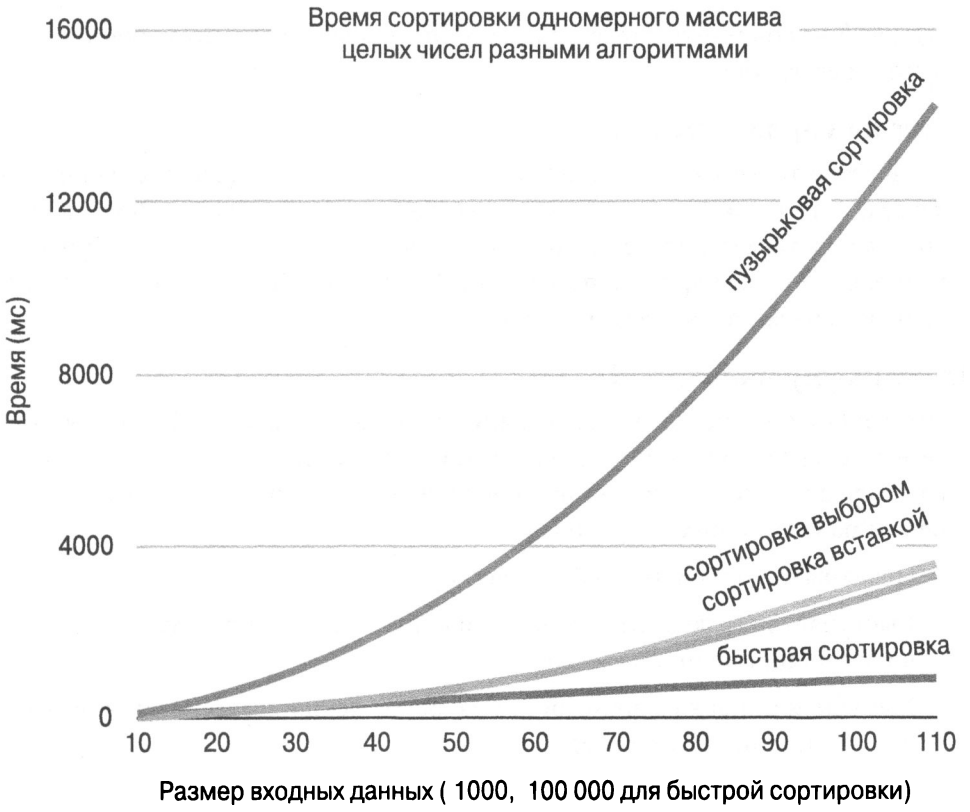
[3] Благодарим за разъяснение всех этих мер риса Эви Брайянта (Avi Bryant; `@avibryant`).

Ответ на упражнение 28

Очевидно, что на это упражнение нельзя дать какой-то однозначный ответ. Но в то же время можно дать пару указаний, в каком направлении следует двигаться в поисках ответа.

Если вы обнаружите, что полученные вами результаты не укладываются на плавную кривую, в таком случае, возможно, придется проверить, не используются ли вычислительные мощности процессора вашего компьютера для какого-нибудь другого вида деятельности. Вряд ли вы получите хорошие цифры, если фоновые процессы периодически отнимают процессорное время у ваших программ. Возможно, стоит также проверить, как используется оперативная память. Стоит, например, приложению начать пользоваться областью подкачки, как производительность резко упадет.

Ниже приведен график результатов выполнения кода на одном из наших компьютеров.



Ответ на упражнение 29

Это можно сделать двумя способами. Один из них состоит в том, чтобы обдумать задачу в уме. Так, если в массиве содержится лишь один элемент, то для его перебора цикл не требуется. На каждом дополнительном шаге цикла размер массива, в котором осуществляется поиск, удваивается. Следовательно, общая формула для определения размера массива такова: $n = 2^m$, где m — количество шагов цикла. Если же применить логарифм по основанию 2 к каждой стороне формулы, по получим $\lg n = \lg 2^m$, и, по определению логарифма, $\lg n = m$.

Ответ на упражнение 30

Чтобы ответить на данное упражнение, придется вспомнить школьный курс математики и, в частности, следующую формулу преобразования логарифма по основанию a в логарифм по основанию b:

$$\log_b x = \frac{\log_a x}{\log_a b}$$

где $\log_a b$ — константа, которой можно пренебречь в асимптотическом выражении результата.

Ответ на упражнение 31

В качестве тестируемого свойства можно проверить успешное выполнение заказа, если на складе имеются достаточные запасы товаров. Заказы можно сначала составить для произвольных количеств товаров, а затем проверить, возвращается ли кортеж с признаком `"OK"`, если на складе имеются достаточные запасы заказываемых товаров.

Ответ на упражнение 32

Это удобный случай для тестирования на основе свойств. Модульные тесты можно сосредоточить на отдельных случаях, когда результат получен другими средствами, а в тестах на основе свойств уделить основное внимание решению следующих вопросов.

- Перекрываются ли два любых ящика?
- Выступает ли какая-нибудь часть любого ящика за пределы кузова грузовика по ширине или по высоте?
- Меньше или равна 1 плотность упаковки, т.е. площадь, занимаемая ящиками и деленная на площадь кузова грузовика?
- Если это часть требования, то превышает ли плотность упаковки минимально допустимую плотность?

Ответ на упражнение 33

1. Данное требование сформулировано как подлинное. В частности, среда приложения может накладывать на него свои ограничения.
2. Само по себе данное требование не сформулировано как таковое. Но для того чтобы выяснить, что *действительно* требуется, придется задать волшебный вопрос: "Зачем?"

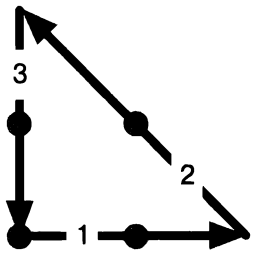
   Возможно, это корпоративная норма, и тогда конкретное требование должно быть сформулировано, например, следующим образом: "Все элементы пользовательского интерфейса должны соответствовать принятым нормам MegaCorp User Interface Standards V12.76".

   А возможно, данный цвет просто понравился команде, оформляющей пользовательский интерфейс. В таком случае следует подумать, каким образом эта команда меняет свое мнение, и сформулировать данное требование, например, таким образом: "Цвет фона всех модальных окон должен быть настраиваемым, а в первоначальной поставке — серым". А еще лучше сформулировать более обширное требование: "Все визуальные элементы приложения (цвета, шрифты и языки) должны быть настраиваемыми".

   Наконец, данное требование может просто означать, что пользователю требуется различать модальные и немодальные окна. В таком случае данное требование может потребовать дополнительного обсуждения.
3. Это требование сформулировано не как таковое, а как архитектурное решение. Сталкиваясь с чем-то подобным, необходимо уточнить мнение пользователя по данному вопросу. В чем здесь дело: в масштабировании, производительности, стоимости или безопасности? Ответы на эти вопросы помогут принять более обоснованное проектное решение.
4. Исходное требование, вероятно, ближе к следующему: "Система должна не давать пользователю ввести недостоверные данные в отдельных полях и предупреждать его при всякой попытке это сделать".
5. Это требование вряд ли можно считать сформулированным как таковое, исходя из некоторых аппаратных ограничений.

   И, наконец, ниже приведено решение задачи проведения трех прямых линий через четыре точки, не отрывая ручку от листа бумаги и возвращаясь в исходную точку.

Здесь цифрами показано, как соединить четыре точки тремя линиями, не отрывая ручку от листа бумаги и возвращаясь в исходную точку



Предметный указатель

M

macOS, 189

U

Unix, 111; 135; 187

А

Аварийное завершение, 151
Автоматизация, 325
Автономный агент, 148
Актор, 227
  модель, 214
  порядок выполнения, 228
  простой, 228
  реализация, 232
Алгоритм, 255
  оценка производительности, 255
  производительность, 252; 256
Архитектура
  гибкая, 81
  прототипирование, 91
Асимптотическое обозначение, 252; 253

Б

Багаж знаний, 47
База данных, 21
Безопасность, 284
  входные данные, 285
  настройки по умолчанию, 287
  обновление, 288
  пароль, 289
  площадь поверхности атак, 284
  принцип наименьших привилегий, 287
  принципы, 284
  шифрование данных, 288

Г

Гейзенбаг, 154
Гибкость, 315

Д

Делегирование, 204
Диаграмма действий, 216
Документация, 54; 76
  назначение, 54
  составление, способы, 54
Дублирование
  в документации, 64
  в исходном коде, 62
  данных, 65
  знаний, 69
  коллективное, 68
  представительное, 67
  у разработчиков, 68

З

Закон
  Деметры, 172
  Конвея, 312
Знания, 43
  багаж, 43
    пополнение, 47
    создание, 44

И

Именование, 291
  запутывающее, 295
  пояснение намерений, 293
  согласованность, 294
  традиции, 294
Инвариант, 144; 276
  семантический, 146
Инстинкт, 241
Инструментальные средства, 107
Интерфейс, 203

К

Кайзен, 27
Карго-культ, 249; 327
Классная доска, 214; 233
  преимущества, 234
Команда, 26; 320
  необходимость доверия, 32
  прагматичная, 320
Командная оболочка, 113
  автозавершение, 115
  настройка, 114
  псевдонимы, 115
Компромисс, 41
Конвейер, 190
Конечный автомат, 177
Контекст, 48
  случайности, 249

Контракт, 140; 276
  динамический, 148
  инвариант, 141
  постусловие, 141
  предусловие, 141
Контроль версий, 119; 332
  ветви, 121
Конфигурирование, 208
  как служба, 209
  рекомендации, 211
  статическое, 208
Криптография, 288
Критическое мышление, 48

М

Миксин, 205
Мьютекс, 226

Н

Наследование, 199; 200
  альтернативы, 202
  иерархия, 202
  и связывание, 200
  множественное, 202

О

Обратимость, 80
Общение, 50
  в Интернете, 56
  в команде, 324
  диалог, 53
  документация как форма общения, 54
  на родном языке, 50
  обратная связь, 51
  особенности и формы, 50
  свободное, 324
  стиль, 52
  удобный момент, 52
Объект, 21
Ортогональность, 69
  и тестирование, 76
  поддержание, 75
  преимущества, 71
  при документировании, 76
  при проектировании, 73
Осведомленность об окружающей обстановке, 39
Ответственность
  взятие на себя, 32
  определение, 32
Отладка, 125
  бинарный поиск ошибки, 128; 129
  вопросы психологии, 125
  воспроизведение ошибок, 127
  и тестирование, 127
  контрольный список, 134
  метод резинового утенка, 131
  мысленная установка на выполнение, 126
  наблюдения, 126
  проверка предположений, 133
  протоколирование и трассировка, 130
  стратегии, 127
  утечка ресурсов, 131
Оценка, 100; 101
  графика работ, 104
  проектов PERT, 104
  точность, 100
  шкала временных оценок, 101

П

Параллелизм, 213; 216; 217
Параллельность, 213; 216
  пассивная, 235
  средства поддержки, 226
Перемены, 30
Повторное использование, 69
Полиморфизм, 200
Поток
  данных, 183; 194
  событий, 185
Прагматичность
  знание меры, 42
  основы мышления, 30
Преждевременная оптимизация, 258
Принцип
  единообразия доступа, 67
  легкости изменения, 58
  наименьших привилегий, 286
  не навреди, 345
  не повторяться, 16; 60; 61; 62
  ортогональности, 69; 77
  поедания слона кусочками, 105; 164; 268
  указывай, а не спрашивай, 171
Программирование
  гибкость, 315
  групповое, 311; 313
  именование, 291

метод трассирующих пуль, 84
в команде, 325
недостатки, 87
преимущества, 86
налог на наследование, 167
на основе тестирования, 267
обдуманное, 250
парное, 311; 312
по совпадению, 145; 239; 246
преобразовательное, 186
прототипирование, 88; 89
профилирование, 257
развязывание, 167; 169
реактивное, 183
решение сложных задач, 310
связывание, 167
утвердительное, 152
Программист
особенности профессии, 23
прагматик
автоматизация, 325
анонимность, 341
взлом сознания, 243
взятие на себя ответственности, 32
знания, 43
индивидуальные навыки, 26
инстинкт, 243
критическое осмысление, 240
манипулирование текстом, 134
начальный набор инструментов, 331
обратная связь, 301
обучение, 47
общение, 50
общие характеристики, 24
озарение, 243
определение, 23
ответственность, 340
отладка, 125
отличительные особенности, 29
оценка, 100
подозрительность, 284
пополнение знаний, 47
синдром самозванца, 242
становление, 25
удовольствие клиента, 340
чужой код, 244
этика, 344
свобода выбора, 31
Программное обеспечение
деградация, 34
факторы, 34
достаточно хорошее, 40
качество, 41
раздутость функциональности, 43
требования, 298
Проектирование, 58; 72
нисходящее, 189
ортогональных систем, 73
по контракту, 141
реализация, 145
сверху вниз, 269
снизу вверх, 269
Проектный шаблон
декоратор, 74
издатель-подписчик, 182
недостатки, 182
обозреватель, 181
недостатки, 182
синглтон, 75
стратегия, 75
Протокол, 203
Прототип, 91; 244
Прототипирование, 88; 89; 90
архитектуры, 91
и метод трассирующих пуль, 88
предостережения, 92
Профайлер, 257
Процесс, 227

P

Разработка
гибкая, 315; 316
на основе тестирования (TDD), 144; 267
Ресурс
балансировка, 158
взаимоблокировка, 159
вложенное выделение, 159
выделение и освобождение, 156
инкапсуляция, 159
исчерпание, 43
Рефакторинг, 62; 76; 260
автоматический, 264
причины, 261
рекомендации, 263
Риск, 45

С

Связывание, 168
  временное, 214
  глобальные данные, 173
  конвейеры, 173
  признаки, 169
  транзитивность, 169
  цепочки вызовов, 170
  через наследование, 200
Семафор, 221
  блокировка и освобождение, 222
  реализация, 221
Сигнатура, 189
Событие, 176; 183
Совет, 25; 30; 33; 35; 38; 39; 41; 45; 48; 50; 53; 54; 58; 61; 69; 71; 81; 82; 84; 90; 94; 105; 109; 114; 116; 121; 125; 126; 128; 132; 133; 135; 139; 143; 151; 152; 156; 159; 164; 166; 169; 171; 172; 174; 175; 100; 188; 193; 202; 204; 205; 207; 208; 215; 219; 225; 228; 236; 243; 249; 256; 257; 262; 265; 266; 270; 273; 274; 276; 286; 288; 295; 298; 299; 300; 302; 306; 308; 314; 315; 320; 323; 325; 328; 330; 332; 333; 335; 336; 337; 340; 345
Состояние, 144
  общее, 214; 220

Т

Текст, 134
  простой, 109
  редактор, 116
    навыки, 116
  удобочитаемость, 109; 111
  формат, 110
  хранение информации, 109
  языки, 135
Теория разбитых окон, 35
  и программирование, 35
Тестирование, 76; 127; 265; 266; 333
  автоматизация, 337
  анализ покрытия, 335
  интеграционное, 334
  и связывание, 266
  культура, 274
  модульное, 144; 271; 333
  на основе свойств, 276; 281; 336
  под предельной нагрузкой, 334
  производительности, 334
  регрессионное, 282
  соответствия контракту, 271
  специальное, 273
  тестов, 335
Технический дневник, 137
Трассирующие пули, 83

У

Утверждение, 145; 152
  assert, 152
  и побочные эффекты, 153
  отключение, 154
  применение, 153

Э

Энтропия, 34

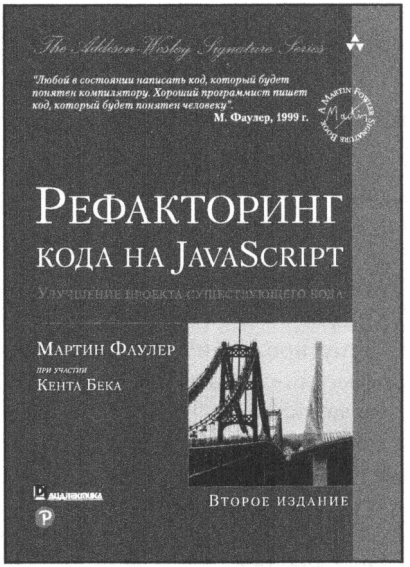
Я

Язык
  Cucumber, 94
  JSON, 98
  UML, 215
  YAML, 96
  манипулирования текстом, 135
  предметно-ориентированный, 94; 97
    внешний, 97
    внутренний, 97
  программирования
    C++, 78; 146; 160; 200; 202
    Clojure, 97; 142; 191
    Eiffel, 144; 145
    Elixir, 97; 143; 194
    Elm, 191
    Erlang, 232
    F#, 191; 194
    Haskell, 191; 194
    Java, 200
    JavaScript, 191
    Python, 135
    Ruby, 135
    Rust, 160
    Scala, 194
    Simula 67, 199
    Smalltalk, 200
    Swift, 191
    назначение и особенности, 93
    поддержка параллельности, 226

РЕФАКТОРИНГ КОДА НА JAVASCRIPT

ВТОРОЕ ИЗДАНИЕ

Мартин Фаулер



www.williamspublishing.com

Рефакторинг уже давно и прочно занимает свое достойное место среди технологий программирования, и не в последнюю очередь благодаря Мартину Фаулеру — автору одной из тех книг, которые написаны "на все времена".

Сам принцип рефакторинга прост: это последовательность небольших шагов, таких как перемещение поля из одного класса в другой, вынесение фрагмента кода из метода и его превращение в самостоятельный метод или даже перемещение кода по иерархии классов. Кумулятивный эффект таких малых изменений состоит в существенном улучшении архитектуры существующего кода. Новое издание классической книги достойно того, чтобы занять свое место на книжной полке каждого серьезного программиста — вне зависимости от используемого языка программирования.

ISBN 978-5-907144-59-0 **в продаже**

"Это одна из самых важных книг в моей жизни."
ОБИ ФЕРНАНДЕС, автор книги *The Rails Way*

"В этом издании вы найдете немало практических советов как технического, так и профессионального характера, которые еще многие годы сослужат вам верную службу в ваших проектах."
АНДРЕА ГУЛЕ, генеральный директор компании Corgibytes; учредитель компании LegacyCode.Rocks

"Молния иногда поражает дважды, чему эта книга служит явным доказательством."
ВМ (ВИКИ) БРАССЕР, директор программы Open Source Strategy в компании Juniper Networks

Настоящее издание относится к числу тех редких образцов технической литературы, которые стоит читать, перечитывать и снова читать в течение многих лет. Из него читатель, будь он начинающим или опытным разработчиком программного обеспечения, сможет всегда почерпнуть свежие идеи.
Дэвид Томас и Эндрю Хант написали первое издание этой замечательной книги в 1999 году, чтобы помочь своим клиентам в создании более качественного программного обеспечения и помочь открыть для себя удовольствие от программирования. Уроки, извлеченные из этой книги, помогли целому поколению программистов усвоить саму суть разработки программного обеспечения, независимо от конкретного языка, библиотеки или методики. Предложенный авторами книги прагматичный философский подход к разработке программного обеспечения нашел широкое распространение, породив сотни других книг и статей, а также послужил началом для тысяч успешных карьер и историй профессионального роста.
Теперь, двадцать лет спустя, в новом издании авторы по-новому взглянули на то, что такое современный программист. В этом издании затрагиваются самые разные темы: от личной ответственности разработчика до развития его карьеры, архитектурные приемы, обеспечивающие гибкость исходного кода и возможность легкого его изменения. Прочитав эту книгу, вы узнаете как:

- бороться с деградацией программного обеспечения
- постоянно учиться
- избегать ловушек, кроющихся в дублировании знаний
- писать гибкий, динамический и адаптируемый код
- овладевать основными инструментальными средствами
- избегать программирования по совпадению
- изучать подлинные требования
- защищаться от уязвимостей в системе безопасности
- решать задачи, лежащие в основе параллельного программирования
- организовывать команды программистов-прагматиков
- брать на себя ответственность за свою работу и карьеру
- строго и эффективно тестировать
- реализовывать начальный набор инструментальных средств программиста-прагматика
- доставлять удовольствие своим пользователям

Эта книга написана в виде последовательного ряда автономных тем-разделов, сдобрена немалой долей классических и свежих анекдотов, тщательно продуманными примерами и интересными аналогиями. В ней показаны наилучшие подходы к разработке программного обеспечения и основные ловушки на этом пути. Начинающие или опытные программисты, как и руководители программных проектов, смогут извлечь немало уроков из этой книги в своей повседневной деятельности, быстро добившись улучшений в производительности труда, пунктуальности и удовлетворенности своей работой. Книга поможет читателю выработать и развить навыки и отношения, образующие прочный фундамент его успешной карьеры в долгосрочной перспективе. В конечном счете он станет программистом-прагматиком.

Категория: программирование
Предмет рассмотрения: методики программирования
Уровень: промежуточный/опытный

Cover illustration: Mihalec/Shutterstock, Stockish/Shutterstock

ISBN 978-5-907203-32-7

Диалектика
www.dialektika.com

Pearson
Addison-Wesley