Дастин Босуэлл Тревор Фаучер

Читаемый код

или Программирование как искусство

O'REILLY®

ПИТЕР®

Dustin Boswell, Trevor Foucher

The Art of Readable Code

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Дастин Босуэлл, Тревор Фаучер

Читаемый код

или Программирование как искусство

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2012

ББК 32.973.2-018
УДК 004.42
Б85

Босуэлл Д., Фаучер Т.
Б85 Читаемый код, или Программирование как искусство. — СПб.: Питер, 2012. — 208 с.: ил.

ISBN 978-5-459-01188-3

Любому программисту доводилось видеть код, который настолько неаккуратен и так пестрит ошибками, что от его чтения начинает болеть голова. За пять лет авторы этой книги проанализировали сотни примеров «плохого» кода (в основном — собственного), пытаясь определить, чем плох тот или иной код и как его можно улучшить. К какому выводу они пришли? Необходимо писать такой код, который читатель сможет понять максимально быстро, даже если этот читатель — сам создатель этого кода.

В данной книге рассматриваются базовые принципы и практические методы, которые можно применять при написании кода. В каждой главе на примере несложных образцов кода, написанного на разных языках программирования, изучается отдельный аспект создания кода и демонстрируется, как сделать код простым для понимания.

ББК 32.973.2-018
УДК 004.42

ISBN 978-0596802295 англ.
ISBN 978-5-459-01188-3

Краткое содержание

Часть I. Поверхностные улучшения

Часть II. Упрощение цикла и логики

Часть III. Реорганизация кода

Часть IV. Избранные темы

Оглавление

Часть I. Поверхностные улучшения

Часть II. Упрощение цикла и логики

Часть III. Реорганизация кода

Предисловие

Мы работали в успешных компаниях, занимающихся разработкой программного обеспечения, с выдающимися инженерами, но код, с которым мы сталкивались, был далек от совершенства. Встречался нам и ужасный код (возможно, вы представляете, что мы имеем в виду под словом «ужасным»).

Чтение красиво написанного кода вдохновляет. Хороший код позволяет быстро разобраться в том, что он делает. Его приятно использовать, он мотивирует вас на оптимизацию собственного кода.

Цель этой книги — помочь вам улучшить ваш код. Под «кодом» мы подразумеваем строки программы, которые вы видите в редакторе. Мы не говорим об архитектуре или о выборе шаблонов проектирования. Безусловно, это очень важные вопросы. Но, исходя из нашего опыта, в повседневной жизни программисты занимаются более «приземленными» вещами: выбирают имена переменных, создают циклы, а также решают задачи на уровне функций. Большую часть вышеперечисленной работы составляет чтение и редактирование уже существующего кода. Надеемся, что данная книга поможет вам в повседневном программировании и вы порекомендуете ее всей своей команде.

О чем эта книга

В этой книге рассказано, как написать приятный, читаемый код. Основная идея издания — код должен быть простым для понимания. Наша цель — сократить время, которое потребуется постороннему человеку для того, чтобы понять ваш код.

В книге эта идея подробно объясняется и иллюстрируется множеством примеров из различных языков программирования, в частности C++, Python, JavaScript и Java. Мы намеренно старались не использовать специфических особенностей данных языков, поэтому, даже если вы не знакомы со всеми этими языками, вам все равно будет легко воспринимать материал. (По нашему опыту, основные концепции читаемости обычно не зависят от языка.)

Каждая глава рассматривает отдельный аспект написания кода, а также способы его улучшения. Книга разделена на четыре части.

- *Часть I. Поверхностные улучшения.* Включает в себя выбор имен, комментирование и оформление кода — все это довольно просто применить для каждой строки вашей кодовой базы.
- *Часть II. Упрощение цикла и логики.* Описывает способы совершенствования циклов, логики и переменных программы с целью их упрощения.
- *Часть III. Реорганизация кода.* Содержит рекомендации общего характера по организации больших блоков кода, а также решение задач на уровне функций.
- *Часть IV. Избранные темы.* Рассказывает о применении принципа «простоты» для тестирования, а также для большого фрагмента кода, содержащего структуры данных.

Как читать эту книгу

Мы хотели написать эту книгу максимально простым и понятным языком. Надеемся, что большинство читателей прочтут ее достаточно быстро.

Главы книги размещены по сложности: простые темы располагаются в начале, а более сложные — в конце. Однако каждая глава вполне самодостаточна и может быть прочитана в отдельности. Поэтому при желании вы вполне можете пропустить несколько глав.

Использование примеров кода

Эта книга призвана помочь вам при выполнении работы. Вы можете применять любой код из данного издания в ваших программах и документации. Вам не нужно связываться с нами и просить разрешение на использование кода, если только вы не собираетесь скопировать его значительный фрагмент. Например, написание программы, которая содержит фрагменты кода из этой книги, не требует разрешения. Однако, если вы запишете на диск примеры из книг издательства O'Reilly и захотите раздавать или продавать такие диски — на это нужно получить разрешение. Ответ на вопрос, построенный на цитате или фрагменте кода из этой книги, не требует разрешения. Для использования в документации значительного объема кода, взятого из примеров этой книги, необходимо разрешение.

Если вам кажется, что вы использовали большое количество нашего кода без разрешения, напишите нам по адресу permissions@oreilly.com.

Как с нами связаться

Все вопросы и комментарии отправляйте издателю:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

У нас есть веб-страница, посвященная этой книге, где мы перечисляем ошибки, приводим примеры и дополнительную информацию. Вы можете посетить ее, перейдя по адресу http://shop.oreilly.com/product/9780596802301.do.

Для получения более подробной информации о наших книгах, курсах, конференциях и новостях посетите веб-сайт http://www.oreilly.com.

Мы в сети Facebook: http://facebook.com/oreilly.

Мы в Twitter: http://twitter.com/oreillymedia.

Наш канал на YouTube: http://www.youtube.com/oreillymedia.

Благодарности

Мы хотим поблагодарить своих коллег, потративших время на то, чтобы прочесть всю эту книгу. В их числе Алан Дэвидсон, Джош Эрлих, Роб Конигсберг, Арчи Рассел, Гейб В. и Асаф Земах. Любая ошибка в данном издании на их совести (шутка).

Мы благодарны нашим многочисленным рецензентам, среди которых Майкл Хангер, Джордж Хайнеман и Чак Хадсон.

Кроме того, мы получили множество идей и отзывов от следующих людей: Джона Блэкберна, Тима Дасилва, Денниса Гилса, Стива Гердинга, Криса Харриса, Джоша Хьюмена, Джоэль Ингрэм, Эрика Мавринака, Грега Миллера, Анатолия Пена и Ника Уайта. Благодарим множество онлайновых комментаторов, которые рецензировали нашу рукопись в системе OFPS издательства O'Reilly.

Спасибо команде издательства O'Reilly за бесконечное терпение и поддержку, особенно Мэри Трезелер (редактор), Терезе Элси (выпускающий редактор), Нэнси Котари (ответственный секретарь), Робу Романо (иллюстратор), Джессике Гозман и Эбби Фокс. Мы хотим также поблагодарить нашего карикатуриста Дэйва Оллреда, который воплотил в жизнь придуманные нами смешные картинки.

Наконец, мы хотим сказать спасибо Мелиссе и Сюзанне за то, что они вдохновляли нас и терпели бесконечные разговоры о программировании.

Об авторах

Дастин Босуэлл был воспитан в цирковой среде, но очень рано понял, что ему гораздо лучше удается работа на компьютере, нежели акробатика. Дастин получил степень бакалавра в Калифорнийском технологическом университете, где и увлекся информатикой. Затем поступил в калифорнийский университет Сан-Диего, где получил степень магистра. На протяжении пяти лет он работал в компании Google. Участвовал в различных проектах, в том числе занимался разработкой поискового робота. Дастин создал множество веб-сайтов. Он любит работу с «большими объемами данных», а также машинное обучение. Сейчас Дастин увлекается стартапами, возникающими в Интернете. В свободное время любит гулять по горам Санта-Моники, а также осваивает новую для себя роль отца.

Тревор Фаучер работал над крупными проектами в компаниях Microsoft и Google более десяти лет. В данный момент он является инженером поисковой инфраструктуры компании Google. В свободное время он посещает игровые конвенции, читает научную фантастику, а также работает на должности главного операционного директора в компании-стартапе своей жены. Тревор получил степень бакалавра в области электротехники и информатики в калифорнийском университете Беркли.

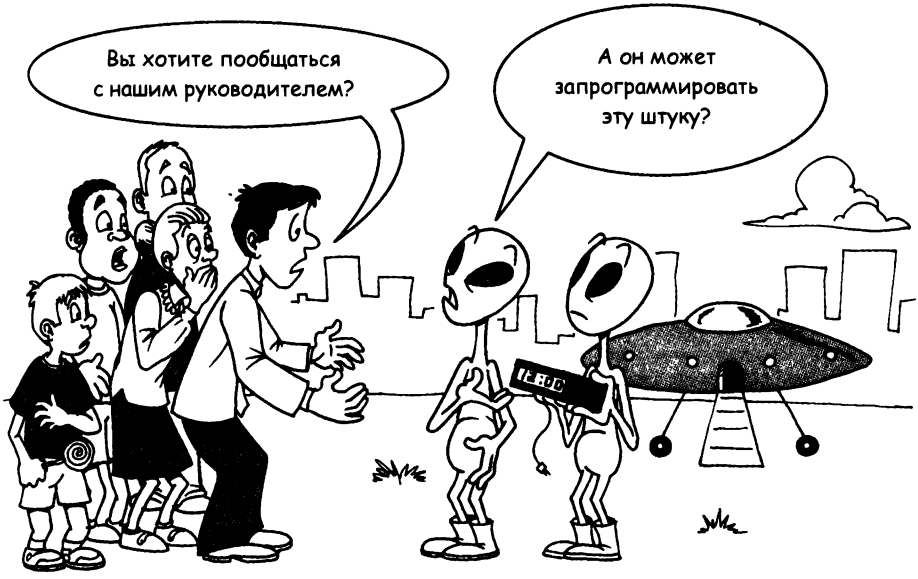
От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты halickaya@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

1 Код должен быть простым для понимания



За последние пять лет у нас скопились сотни образцов плохого кода (в основном наши собственные). Мы проанализировали, почему он является плохим и какие принципы/техники использовались для того, чтобы сделать его лучше. И мы заметили, что все эти принципы вытекают из одного основополагающего.

ОСНОВНАЯ ИДЕЯ

Код должен быть простым для понимания.

Мы считаем, что это наиболее важный принцип, которым следует руководствоваться при написании кода. В книге мы покажем, как применять этот принцип к различным аспектам повседневного программирования. Начнем с того, что подробно разберем этот принцип и объясним, почему он так важен.

Что делает код «лучше»?

Большинство программистов (включая авторов этой книги) принимают решения, связанные с программированием, на основании собственных ощущений и интуиции. Все мы знаем, что следующий код:

```
for (Node* node = list->head; node != NULL; node = node->next)
     Print(node->data);
```

гораздо лучше, чем такой:

```
Node* node = list->head;
if (node == NULL) return;

while (node->next != NULL)  {
    Print(node->data);
    node = node->next;
}
If (node != NULL) Print (node->data)
```

(даже несмотря на то, что оба примера кода работают совершенно одинаково).

Однако в большинстве случаев выбор сделать труднее. Например, лучше ли этот фрагмент кода:

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1
                                                    << -exponent);
```

чем следующий:

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

Первый вариант более компактен, но второй менее удручает. Какой критерий более важен? В общем, как вы определяете, каким способом написать код?

Фундаментальная теорема читаемости

После изучения множества подобных примеров кода мы пришли к выводу, что есть один наиболее важный параметр читаемости. Он настолько важен, что мы назвали его *фундаментальной теоремой читаемости*.

ОСНОВНАЯ ИДЕЯ

Код должен быть написан так, чтобы можно было максимально быстро понять, как он работает.

Что мы имеем в виду? Мы советуем предложить коллегам прочесть ваш код и измерить время, которое потребуется им для того, чтобы понять его. «Время-для-понимания» — это теоретический параметр, значение которого вам нужно уменьшить.

Обратите внимание, что, говоря «понимание», мы устанавливаем планку очень высоко. *Полное понимание* кода означает возможность вносить в него изменения, находить ошибки, а также понимать, как именно он взаимодействует с остальными фрагментами кода.

Сейчас вы можете задуматься, *а зачем кому-то еще понимать мой код? Ведь его использую только я один!* Даже если в проекте заняты только вы, стоит пытаться достичь этой цели. Этим *кем-нибудь еще* можете быть вы сами шесть месяцев спустя. Код может показаться вам незнакомым. Вы не можете предусмотреть все: кто-то может присоединиться к проекту или ваш код может использоваться повторно, уже в другом проекте.

Меньше — значит лучше?

Вообще-то чем меньше кода используется для решения проблемы, тем лучше (см. гл. 13). Скорее всего, потребуется меньше времени для того, чтобы разобраться в 2000-строчном классе, чем в 5000-строчном.

Но меньшее количество строк не всегда делает код лучше. Как правило, чтобы понять следующее однострочное выражение:

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

требуется больше времени, чем на понимание того же выражения, записанного в две строки:

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

Подобным образом комментарий позволит вам понять код намного быстрее, даже если этот комментарий состоит из строк кода:

```
// Версия кода "hash = (65599 * hash) + c", которая выполняется быстрее.
hash = (hash << 6) + (hash << 16 ) - hash + c;
```

Таким образом, меньшее количество строк кода — это хорошая цель, однако гораздо важнее стремиться к сокращению времени-для-понимания.

Противоречит ли время-для-понимания другим целям?

Вы, должно быть, сейчас задумались об остальных требованиях, таких как эффективность кода, продуманная архитектура, простота тестирования и т. д. Не противоречат ли они стремлению сделать код простым для понимания?

Мы считаем, что эти цели не мешают друг другу. Даже высокооптимизированный код всегда можно сделать еще более удобным для чтения. В результате приведения кода в читаемый вид он становится также более структурированным и простым для тестирования.

В этом издании рассматривается следующий вопрос: как повысить читаемость кода при различных обстоятельствах. Но запомните главное: фундаментальная теорема читаемости более приоритетна, чем любое другое правило или принцип этой книги. Некоторым программистам приходится исправлять код, над которым не был проведен рефакторинг. Всегда необходимо еще раз просматривать готовый код и задавать себе вопрос: *легко ли его понять*? Если ответ утвердительный, то можно переходить к другому фрагменту кода.

Самое сложное

Да, непросто постоянно думать о том, сможет ли воображаемый посторонний человек легко понять ваш код. Вам понадобится подумать теми «серыми клеточками», которые, возможно, «отдыхают» во время написания кода.

Но, если вы поставите перед собой такую цель (как это сделали мы), гарантируем, что вы станете хорошим программистом, будете допускать меньше ошибок, станете гордиться своей работой и создавать код, который с удовольствием будут использовать окружающие. Итак, приступим!

ЧАСТЬ I

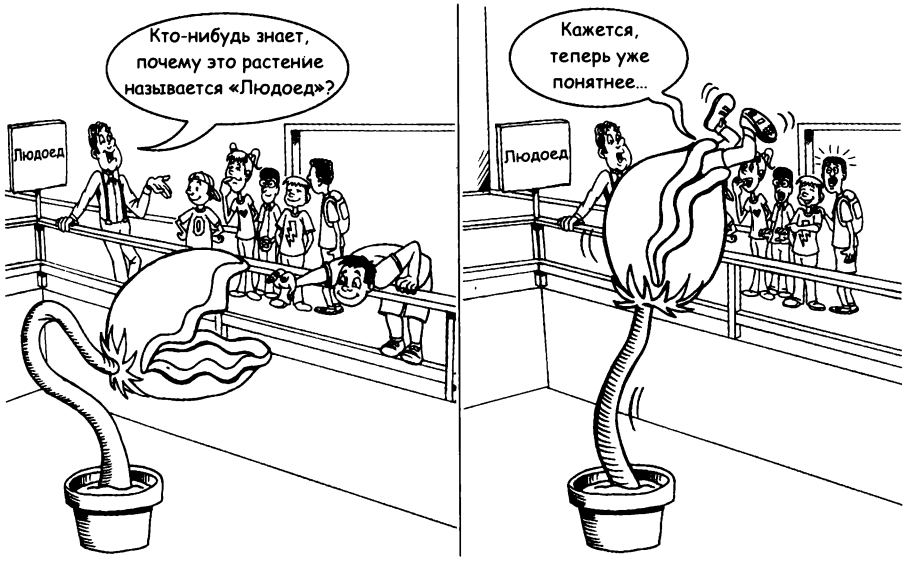
Поверхностные улучшения

Начнем нашу экскурсию в мир красивого кода с рассмотрения так называемых поверхностных улучшений: выбора хороших имен, написания полезных комментариев, а также качественного форматирования кода. Эти изменения довольно легко применить на практике. Их можно делать постепенно, не тратя времени в дальнейшем. После внесения этих изменений отпадает также необходимость рефакторинга или изменения способа работы программы.

Данные темы очень важны, поскольку **они касаются каждой строчки кода вашей программы**. Хотя каждое отдельное изменение может показаться совсем несущественным, в совокупности они могут значительно улучшить всю программу. Код будет *намного* проще читать, если в нем будут отличные имена, хорошо написанные комментарии, а также продуманное форматирование.

Конечно, под этим поверхностным уровнем скрывается гораздо больший объем работ по приведению кода в читаемый вид (и мы рассмотрим соответствующие примеры в книге далее). Но материал этой части имеет довольно широкое применение и сравнительно прост, поэтому его стоит изучить в самом начале.

2 Помещаем в имена полезную информацию



Когда вы придумываете имя для переменной, функции или класса, вы руководствуетесь примерно одинаковыми для всех программистов принципами. Мы предлагаем вам относиться к имени как к небольшому комментарию. Вы можете сообщить довольно много информации, выбрав хорошее имя.

ОСНОВНАЯ ИДЕЯ

Помещайте в имена полезную информацию.

Большое количество имен, встречающихся в программе, ни о чем не говорят (например, `tmp`). Даже слова, применение которых обычно оправданно (такие как `size` или `get`), не содержат много информации. Из этой главы вы узнаете, как подобрать более подходящие имена.

Данная глава состоит из шести отдельных тем:

- выбор конкретных слов;
- избегание общих имен (или умение использовать их к месту);
- использование конкретных имен вместо абстрактных;
- добавление дополнительной информации с использованием суффикса или префикса;
- определение длины имени;
- форматирование имени для того, чтобы сообщить дополнительную информацию.

Выбираем конкретные слова

Одним из важных этапов добавления информации в имена является выбор конкретных слов, а также избегание пустых и неинформативных.

Например, слово `get` в следующем примере совершенно неконкретное:

```
def GetPage(url):
```

Слово `get` (получать) не скажет вам многого. Откуда получает информацию этот метод: из локальной кэш-памяти, из базы данных или из Интернета? Если верным является последний вариант, то было бы правильнее использовать имя `FetchPage()` или `DownloadPage()`.

Рассмотрим также пример класса `BinaryTree`:

```
class BinaryTree {
    int Size();

};
```

Что возвращает метод `Size()`? Высоту бинарного дерева, количество узлов или объем используемой памяти?

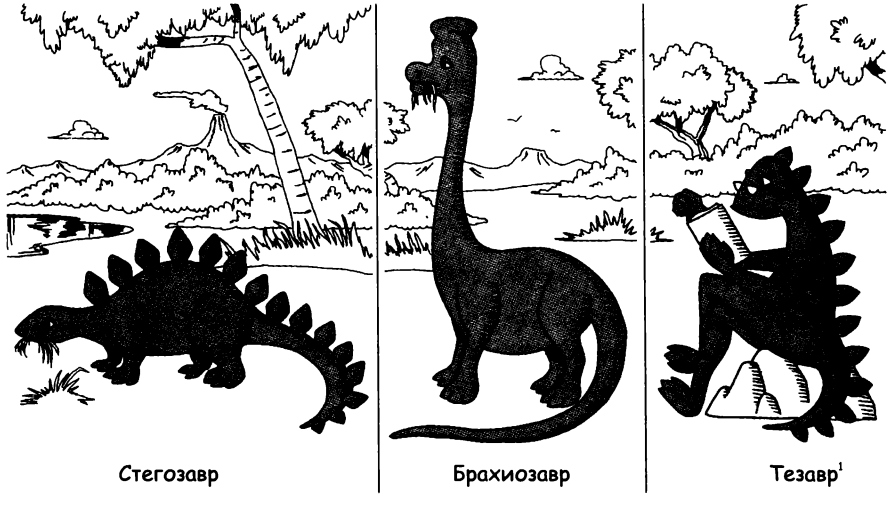
Проблема в том, что имя метода `Size()` не сообщает нам достаточной информации. Более конкретными именами в данном случае были бы `Height()`, `NumNodes()` или `MemoryBytes()`.

В качестве еще одного примера рассмотрим вариант реализации класса Thread:

```
class Thread {
    void Stop();

};
```

Имя Stop() вполне подходит, но, в зависимости от того, что именно делает этот метод, можно выбрать и более конкретное имя. Например, вместо него можно использовать имя Kill() (убить), если результат действия этой операции нельзя откатить. Или можно применить имя Pause() (приостановить), если у вас есть возможность продолжить его работу (Resume()).

Находим более «яркие» слова. Не бойтесь воспользоваться тезаурусом или попросить друга помочь вам придумать более подходящее имя. Английский язык богат различными синонимами, вы можете выбрать имя из огромного множества слов.



В табл. 2.1 рассмотрим «яркие» версии слов, которые вы повседневно используете. Они могут пригодиться в определенных ситуациях.

Таблица 2.1

Слово	Альтернативы
send (посылать)	deliver (доставлять), dispatch (отсылать), announce (извещать), distribute (распространять), route (направлять)
find (искать)	search (искать), extract (извлекать), locate (обнаруживать), recover (восстанавливать)
start (начинать)	launch (запускать), create (создавать), begin (начинать), open (открывать)
make (создавать)	create (создавать), set up (устанавливать), build (строить), generate (генерировать), compose (составлять), add (добавлять), new (создавать)

[1] Здесь имеет место игра слов, обыгрываются названия видов динозавров и слова «тезаурус» (вид словарей). — *Примеч. пер.*

Однако не слишком фантазируйте. В языке PHP есть функция для работы со строками, которая называется `explode()` (взорвать). Это довольно яркое имя, которое рисует прекрасную картину того, как объект разлетается на кусочки. Но чем эта функция отличается от `split()` (разделить)? (Эти функции используются для разных целей, но как определить это по названию?)

ОСНОВНАЯ ИДЕЯ

Лучше быть ясным и четким, чем крутым и оригинальным.

Избегаем общих имен, например таких, как tmp и retval

Имена вроде `tmp`, `retval` и `foo` можно сравнить с отговорками вроде «никак не могу придумать имя». Вместо использования подобного бесполезного имени **придумайте такое имя, которое описывает назначение или содержание объекта**.

В качестве примера приведем функцию на языке JavaScript, которая использует параметр `retval`:

```
var euclidian_norm = function (v) {
    var retval = 0.0;
    for (var i = 0; i < v.lenght; i += 1)
        retval += v[i] * v[i];
    return Math.sqrt(retval);
};
```

Очень хочется воспользоваться именем `retval`, если вы не можете придумать более подходящее имя для возвращаемого значения. Но `retval` не содержит никакой другой информации, кроме как о том, что это возвращаемое значение (а это в любом случае очевидно).

Более подходящее имя описывает назначение переменной или значение, которое она содержит. В данном случае в переменную записывается сумма квадратов значений массива `v`. Поэтому лучше назвать ее `sum_squares` (сумма квадратов). Такое имя говорит само за себя и может помочь определить ошибку.

Например, представьте, что внутри цикла вы случайно написали:

```
retval += v[i];
```

Ошибка будет гораздо более очевидной, если переменная будет называться `sum_squares`:

```
sum_squares += v[i]; // Почему мы суммируем значения,
                     // а не их квадраты? Ошибка!
```

СОВЕТ

Имя переменной retval недостаточно информативное. Используйте вместо него имя, которое описывает значение переменной.

Однако иногда общие имена довольно информативны. Рассмотрим, в каких ситуациях целесообразно их использовать.

tmp

Представим классический вариант обмена двух переменных:

```
if (right < left) {
    tmp = right;
    right = left;
    left = tmp;
}
```

В подобных случаях имя tmp отлично подходит. Единственная цель, с которой оно создавалось, — временное хранилище значения. Продолжительность ее жизни равна всего нескольким строкам. Имя tmp имеет особое значение для читателя — у переменной с таким именем нет другого предназначения, кроме как функции временного хранилища. Она не передается в другие функции, не меняет свое значение и не используется несколько раз.

А в следующем случае имя tmp использовано только из-за того, что программист поленился:

```
String tmp = user.name();
tmp += " " + user.phone_number();
tmp += "" + user.email();
...
template.set("user_info", tmp);
```

Хотя жизненный цикл этой переменной очень краток, она важна не только потому, что применяется как временное хранилище. Вместо имени tmp лучше использовать более содержательное название вроде user_info.

В следующем примере мы могли использовать имя tmp, но применили его как *часть* имени:

```
tmp_file = tempfile.NamedTemporaryFile()
...
SaveData(tmp_file, ...)
```

Обратите внимание на то, что мы назвали переменную tmp_file, а не просто tmp, поскольку она является объектом файла. Представьте, что мы назвали ее tmp:

```
SaveData(tmp, ...)
```

Если взглянуть только на эту строку кода, не совсем понятно, чем является tmp: файлом, именем файла или данными, которые мы записываем в файл.

СОВЕТ

Имя tmp должно использоваться только в тех случаях, когда вы объявляете переменную с небольшой продолжительностью жизни и эта информация о ней является самой важной.

Итераторы циклов

Имена вроде `i`, `j`, `iter` и `it` чаще всего используются в качестве индексов и итераторов цикла. Даже несмотря на то, что их имена абстрактны, чаще всего они рассматриваются именно как итераторы. (Фактически использование этих имен для какой-нибудь другой цели может кого-нибудь запутать — не делайте так!)

Однако иногда можно выбрать более полезные, чем `i`, `j` и `k`, имена для итераторов. Например, с помощью следующих циклов определяется, какие клубы посещают пользователи:

```
for (int i = 0; i < clubs.size(); i++)
    for (int j = 0; j < clubs[i].members.size(); j++)
        for (int k = 0; k < users.size(); k++)
            if (clubs[i].members[k] == users[j])
                cout << "user[" << j << "] is in club[" << i << "]" << endl;
```

В конструкции `if` массивы `members[]` и `users[]` используют неправильные индексы. Такие ошибки очень трудно распознать, поскольку в отдельности эта строка кода выглядит вполне правильной:

```
if (clubs[i].members[k] == users[j])
```

В данном случае может быть целесообразно применять более подробные имена. Вместо того чтобы называть индексы циклов (`i`, `j`, `k`), можно выбрать следующий вариант — `club_i`, `members_i`, `users_i` или более краткий — `ci`, `mi`, `ui`. Такой подход поможет вам обнаруживать некоторые ошибки:

```
if (clubs[ci].members[ui] == users[mi]) # Ошибка! Первые буквы имен
                                        # не совпадают.
```

Если же индексы использованы правильно, первые буквы их имен будут соответствовать первым буквам названий массивов:

```
if (clubs[ci].members[mi] == users[ui]) # Все в порядке.
                                        # Первые буквы совпадают
```

Вердикт по общим именам

Как вы могли заметить, применение общих имен иногда может быть полезным.

СОВЕТ

Если вы собираетесь использовать общее имя, например tmp, it или retval, у вас на это должна быть веская причина.

В большинстве случаев общие имена используются из-за обыкновенной лени. И это можно понять: когда ничего не приходит на ум, гораздо проще придумать какое-нибудь бессмысленное имя, как, например, `foo`, и писать программу дальше. Но если вы заведете привычку тратить несколько секунд на то, чтобы придумать хорошее имя, то вскоре обнаружите, что у вас это получается все лучше и лучше.

Используйте конкретные имена вместо абстрактных

Когда вы даете имя переменной, функции или какому-либо другому элементу программы, лучше описывать их конкретно, нежели абстрактно.

Например, представьте, что у вас есть метод, который называется `ServerCanStart` (сервер может быть запущен), проверяющий, может ли сервер прослушивать данный порт TCP/IP. Как видите, имя `ServerCanStart` (сервер может быть запущен) довольно абстрактно. Имя `CanListenOnPort` (может прослушивать порт) более конкретно. Оно описывает непосредственно то, что делает метод.

Следующие два примера иллюстрируют эту концепцию более подробно.

Пример: DISALLOW_EVIL_CONSTRUCTORS (отключение злобных конструкторов)

Этот пример взят из базы кода Google. В языке C++, если вы не определяете конструктор копирования или оператор присваивания для класса, вам предоставляется его версия, заданная по умолчанию. Хотя такие методы довольно удобны, их использование может привести к утечкам памяти или прочим проблемам, поскольку они выполняются «за кулисами», в тех местах, о которых вы даже не подозревали.

В результате компания Google ввела соглашение, отключающее эти «злобные» конструкторы с помощью использования макроса:

```
class ClassName {
  private:
    DISALLOW_EVIL_CONSTRUCTORS(ClassName);

  public:

};
```

Этот макрос определен как:

```
#define DISALLOW_EVIL_CONSTRUCTORS(ClassName) \
    ClassName(const ClassName&); \
    void operator=(const ClassName&);
```

Размещение данного макроса в секции класса `private:` приводит к тому, что два метода, описанных в нем, также становятся `private`, поэтому они не могут быть использованы даже случайно.

Однако применение имени `DISALLOW_EVIL_CONSTRUCTORS` — не самый лучший вариант. Использование слова `evil` вызвало множество споров. Самое важное — не совсем ясно, что именно отключает этот макрос. Он запрещает метод `operator=()`, который даже не является конструктором!

Это имя применялось много лет и в конечном итоге было заменено менее провокационным и более конкретным:

```
#define DISALLOW_COPY_AND_ASSIGN[1](ClassName) ...
```

Пример: --run_locally (запускать на локальной машине)

У одной из наших программ есть опциональный флаг командной строки, который называется `--run_locally` (запускать на локальной машине). Если он установлен, то программа будет выводить дополнительную информацию об отладке, но при этом замедлит работу. Данный флаг обычно используется при тестировании на локальном компьютере, например на ноутбуке. Но в тех случаях, когда эта программа запускается на удаленном сервере, довольно важна производительность, поэтому флаг не используется.

Понятно, почему флаг имеет такое имя (запускать на локальной машине). Однако при этом возникли некоторые проблемы.

- Новый член нашей команды не знал, что именно делает этот флаг. Он использовал бы его при запуске программы на локальной машине, но он не знал, для чего именно нужен этот флаг.
- Однажды нам пришлось вывести информацию об отладке в то время, когда программа была запущена удаленно. Передавать флаг `--run_locally` программе, которая запущена удаленно, было бы не только смешно, но и странно.
- Иногда мы запускали на локальной машине тест производительности, и нам не хотелось, чтобы запись процесса отладки замедляла систему, поэтому мы не использовали флаг `--run_locally`.

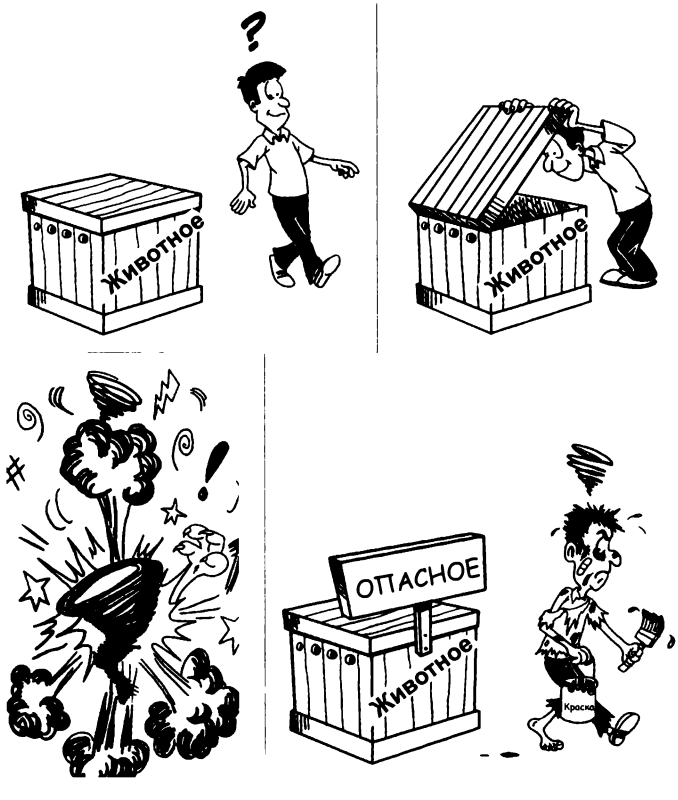
[1] Отключить копирование и присваивание. — *Примеч. пер.*

Проблема заключалась в том, что флаг `--run_locally` был назван так исходя из обстоятельств, в которых он использовался чаще всего. Имя флага вроде `--extra_logging` (записывать больше информации) было бы более понятным и однозначным.

Но что, если флаг `--run_locally` будет использоваться не только для того, чтобы записывать больше информации? Например, представьте, что вам необходимо установить и использовать специальную локальную базу данных. В этом случае имя `--run_locally` кажется более подходящим, поскольку оно охватывает оба предназначения.

Но его применение — не самый лучший вариант, поскольку это решение бесполезно и туманно, что не очень хорошо. Лучшим решением было бы создать второй флаг и назвать его `--use_local_database` (использовать локальную базу данных). Даже несмотря на то, что теперь вам придется оперировать двумя флагами, их имена гораздо более понятны. Они не пытаются объединить две пересекающиеся идеи в одну. Это также позволяет одновременно использовать один флаг и не применять другой.

Добавление дополнительной информации к имени



Как мы говорили выше, имя переменной можно использовать как маленький комментарий. Любая дополнительная информация, которую вы можете добавить в имя переменной, будет отображаться всякий раз, когда вы видите эту переменную.

Если есть какая-нибудь очень важная информация, которую должен знать читатель кода, стоит добавить к имени дополнительное слово. Например, представьте, что у вас есть переменная, содержащая строку в шестнадцатеричном формате:

```
string id; // например: "af84ef845cd8"
```

Возможно, следует использовать имя `hex_id`, если вы хотите, чтобы человек, читающий код, помнил формат номера.

Значения, содержащие единицы измерения

Если переменная хранит результаты измерений (такие как истекшее время или количество байтов), довольно удобно записывать единицы измерения в ее имя.

В качестве примера рассмотрим код на языке JavaScript, который измеряет время загрузки веб-страницы:

```
var start = (new Date()).getTime(); // верх страницы
...
var elapsed = (new Date()).getTime() - start; // низ страницы
document.writeln("Load time was: " = elapsed + " seconds");
```

В этом коде нет ни одной очевидной ошибки, но он не будет работать, поскольку функция `getTime()` возвращает значение в миллисекундах, а не в секундах.

Добавив к имени переменных постфикс `_ms`, мы сделаем код более понятным:

```
var start_ms = (new Date()).getTime(); // верх страницы
...
var elapsed_ms = (new Date()).getTime() - start_ms; // низ страницы
document.writeln("Load time was: " = elapsed_ms + " seconds");
```

При программировании вам могут встретиться и другие единицы измерения. В табл. 2.2 приведены параметры функций без единиц измерения в названиях, а также их более понятные версии.

Таблица 2.2

Параметр функции	Переименованный параметр с единицей измерения
Start(int **delay**)	delay -> **delay_secs**
CreateCache(int **size**)	size -> **size_mb**
ThrottleDownload(float **limit**)	limit -> **max_kbps**
Rotate(float **angle**)	angle -> **degrees_cw**

Кодирование прочих важных атрибутов

Вы можете добавлять в имена не только единицы измерения. Это нужно делать всякий раз, когда значение переменной может преподнести сюрприз и нарушить работу программы.

Например, существует множество хакерских программ, использующих уязвимости различных систем. Такие программы возникают из-за непонимания создателями этих систем того, что некоторые данные, получаемые программой, небезопасны. Чтобы избежать этого, можно использовать такие имена переменных, как `untrustedUrl` (небезопасный URL) или `unsafeMessageBody` (небезопасный текст сообщения). После вызова функций, которые очищают подобное содержимое, итоговые переменные можно назвать `trustedUrl` (проверенный URL) или `safeMessageBody` (безопасный текст сообщения).

В табл. 2.3 приведены подобные примеры случаев, когда возможно добавление важной информации к имени переменной.

Таблица 2.3

Ситуация	Имя переменной	Более подходящее имя
Пароль хранится в открытом виде и должен быть зашифрован	password (пароль)	**plaintext**_password (незашифрованный пароль)
Пользовательский комментарий, который должен быть экранирован, прежде чем он будет отображен	comment (комментарий)	**unescaped**_comment (неэкранированный комментарий)
Байты переменной html преобразованы в кодировку UTF-8	html	html_**utf8**
URL входящих данных был закодирован	data	data_**urlenc**

Не стоит использовать атрибуты вроде `unescaped_` или `_utf8` для *каждой* переменной в вашей программе. Они наиболее полезны в тех местах, где кто-либо может ошибиться, не разобравшись, что именно хранит переменная. И особенно если это может привести к появлению уязвимости в системе.

ЭТО ВЕНГЕРСКАЯ НОТАЦИЯ?

Венгерская нотация — это система образования имен, широко используемая компанией Microsoft. При такой нотации к имени каждой переменной в виде префикса добавляется ее тип. В табл. 2.4 показаны несколько примеров.

Таблица 2.4

Имя	Значение
pLast	Указатель (p) на последний элемент в какой-либо структуре данных
pszBuffer	Указатель (p) на буфер строк (s), который был заполнен нулями
Cch	Количество (c) символов (ch)
mpcopx	Ассоциативный контейнер (m), содержащий информацию об указателях на цвет (pco) и указателях на осевую длину (px)

Безусловно, это хороший пример добавления атрибутов к именам переменных. Но эта система является более формальной и прямолинейной, она сосредоточена на добавлении лишь определенного множества атрибутов.

Подход, описанный нами выше, более свободный. При нем мы определяем важнейшие атрибуты переменной, а затем добавляем их к ее имени, если это необходимо. Можно назвать это английской нотацией.

Насколько длинным должно быть имя?

Когда вы хотите придумать хорошее имя, вам, наверное, очевидно, что оно не должно быть слишком длинным. Никто не любит работать с идентификаторами вроде:

```
newNavigationControllerWrappingControllerForDataSourceOfClass
```

Более длинное имя сложнее запомнить, оно занимает больше места на экране (возможно, даже несколько строк).

С другой стороны, некоторые программисты могут воспринять этот совет слишком буквально и использовать односложные (или даже однобуквенные) имена. Как же сделать правильный выбор? Какое имя вы выберете для переменной: `d`, `days` или `days_since_last_update`?

Решение этой проблемы заключается в том, что перед выбором имени необходимо определить, как именно будет использоваться данная переменная. Далее мы приведем несколько указаний, которые вам пригодятся.

Более короткие имена подходят для небольших областей видимости

Когда вы отправляетесь в короткую поездку, то обычно пакуете меньше багажа, чем во время сборов в длинный отпуск. Так же и в программировании: идентификаторы с небольшой *областью видимости* (количество других строк кода, в которых может использоваться это имя) могут не содержать большого количества информации. В следующем примере для переменной использовалось короткое имя, поскольку всю информацию об этой переменной (ее тип, исходное значение и способ уничтожения) получить довольно легко:

```
if (debug) {
    map<string, int> m;
    LookUpNamesNumbers(&m);
    Print(m);
}
```

Даже несмотря на то, что имя `m` не несет никакой полезной информации, читатель без труда сможет почерпнуть из кода все необходимые сведения.

Однако представьте, что `m` — член какого-нибудь класса или глобальная переменная, и при этом вы видите следующий фрагмент кода:

```
LookUpNamesNumbers(&m);
Print(m);
```

Этот код гораздо менее удобочитаемый, не совсем ясны тип и предназначение переменной `m`.

Поэтому, если область видимости идентификатора довольно велика, его имя должно быть более информативным, чтобы упростить понимание кода.

Набрать длинное имя — больше не проблема

Есть множество причин не использовать длинные имена, но тот факт, что «их сложно набирать», такой причиной не является. В каждом текстовом редакторе для программистов есть встроенная функция автозавершения. К нашему удивлению, многие программисты об этом не знают. Если вы до сих пор ею не воспользовались, пожалуйста, отложите книгу и попробуйте сделать следующее.

1. Наберите несколько первых символов какого-либо имени.
2. Запустите функцию автозавершения (табл. 2.5).
3. Если на экране появилось не то слово, которое вам нужно, продолжайте запускать функцию автозавершения, пока не появится правильное имя.

Таблица 2.5

Редактор	Команда
Vi	Ctrl+P
Emacs	Meta+/ (нажмите Esc, затем /)
Eclipse	Alt+/
Intellij DEA	Alt+/
TextMate	Esc

Эта функция удивительно точна. Она работает во всех типах файлов, в любом языке, а также с любой меткой, даже когда вы набираете комментарий.

Акронимы и аббревиатуры

Некоторые программисты для сокращения имен используют акронимы и аббревиатуры, например называют класс `BEManager` вместо `BackEndManager`. Стоит ли подобное сокращение потенциальной путаницы в будущем?

По нашему опыту, использовать аббревиатуры, встречающиеся только в данном конкретном проекте, — порочная практика. Они могут показаться загадочными и устрашающими для людей, которые только недавно присоединились к проекту. По прошествии некоторого времени они могут показаться таковыми и самим авторам.

Ответьте при создании имени переменной на главный вопрос: **сможет ли новый член команды понять, что обозначает это имя**? Если да, то такое имя можно использовать.

Например, большинство программистов применяют следующие сокращения: `eval` вместо `evaluation`, `doc` вместо `document`, `str` вместо `string`. Новый член команды, взглянув на имя `FormatStr()`, скорее всего, поймет, что оно означает. Однако ему будет довольно трудно разобраться, что именно обозначает имя `BEManager`.

Убираем ненужные слова

Иногда некоторые слова, из которых состоит имя, можно убрать. При этом имя будет сообщать тот же объем информации. Например, чтобы не использовать имя `ConvertToString()` (преобразовать к строчному типу), можно ограничиться более простым вариантом — `ToString()` (к строчному типу). Это имя короче предыдущего, но сообщает ту же информацию, что и его длинный аналог. Так же, например, вместо имени `DoServeLoop()` (выполнить служебный цикл) можно использовать `ServeLoop()` (служебный цикл).

Использование форматирования имен для передачи их смысла

Использование подчеркиваний, тире и прописных букв также поможет вам передать с помощью имени больше информации. В качестве примера рассмотрим код на языке C++, который отформатирован согласно правилам, используемым компанией Google для проектов с открытым исходным кодом:

```
static const int kMaxOpenFiles = 100;

class LogReader {
  public:
    void OpenFile(string local_file);

   private:
     int offset_;
     DISALLOW_COPY_AND_ASSIGN(LogReader);
};
```

Применение различного форматирования для разных объектов (своего рода синтаксическое выделение) помогает читать код быстрее.

Большая часть форматирования в этом примере довольно распространена, например использование «верблюжьего регистра» (`CamelCase`) для имен классов

и нижнего подчеркивания для имен переменных. Но прочие примеры могут удивить вас.

Например, имена констант приведены в форме `kConstantName` (`k`, а затем имя константы) вместо использования привычного форматирования, когда имена написаны прописными буквами. Этот подход имеет свои преимущества — имена констант проще отличить от имен макросов, которые написаны прописными буквами исходя из соглашения.

Имена переменных, являющихся членами класса, выглядят так же, как и имена прочих переменных, но должны заканчиваться подчеркиванием. На первый взгляд это соглашение может показаться странным, но оно позволяет проще отличать переменные, являющиеся членами классов, от прочих, что довольно удобно. Представьте, что вы читаете длинный метод и видите следующую строку:

```
stats.clear();
```

Скорее всего, вы зададитесь вопросом, принадлежит ли переменная `stats` этому классу и меняет ли этот участок кода внутреннее состояние класса. Если бы код данного фрагмента был отформатирован с использованием вышеупомянутого соглашения, то вы смогли бы быстро разобраться в этом вопросе и понять, что переменная `stats` локальная. В противном случае она бы называлась `stats_`.

Другие соглашения, касающиеся форматирования

В зависимости от содержания проекта или языка программирования, на котором он написан, вы можете использовать другие соглашения, чтобы наполнить имена информацией.

Например, Дуглас Крокфорд, автор книги «JavaScript: положительные стороны» (JavaScript: The Good Parts, O'Reilly, 2008), советует делать прописной первую букву имени конструктора (функции, вызывающиеся оператором `new`), а первые буквы имен прочих функций оставить строчными:

```
var x = new DatePicker(); // DatePicker() – это функция-«конструктор»
var y = pageHeight();     // pageHeight – это обыкновенная функция
```

Еще один пример, написанный на языке JavaScript. Когда вы вызываете из библиотеки jQuery функцию, чье имя состоит из одного символа — `$`, вам будет довольно удобно добавлять к имени переменной, хранящей результат вызова, символ `$`:

```
var $all_images = $("img")  // $all_images – объект, связанный с jQuery
var height = 250;           // a height – не связан с jQuery
```

Если взглянуть на любой фрагмент кода, то становится ясно, что переменная `$all_object` хранит в себе результат работы функции jQuery.

Приведем еще один пример, касающийся HTML/CSS. Когда вы в HTML-теге указываете атрибуты `id` и `class`, вы можете использовать подчеркивания и тире. Одним из возможных соглашений может быть следующее: для разделения слов в атрибуте `id` можно применять подчеркивание, а в атрибуте `class` — дефис:

```
<div id="middle_column" class="main-content">
```

Решение о соглашениях принимаете вы сами и ваша команда. Однако при любой системе будьте последовательны на протяжении всего проекта.

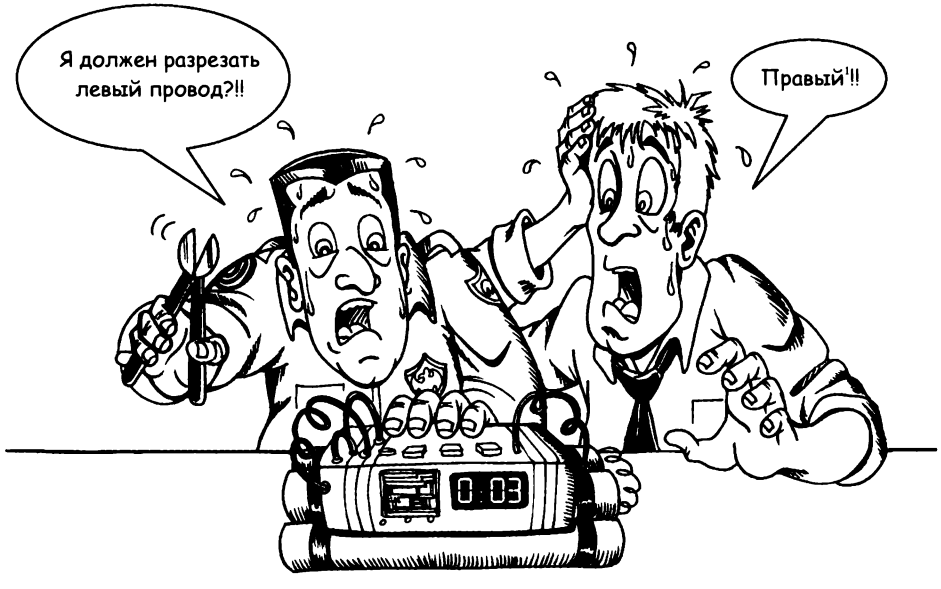
Итог

Основная идея этой главы — **помещайте информацию в имена**. Это следует делать для того, чтобы имена переменных были более информативными и человек, читающий вашу программу, смог быстрее разобраться в ней.

Мы определили несколько важных рекомендаций.

- **Используйте специфические слова:** например, вместо слова `Get` в зависимости от контекста можно применять слова `Fetch` или `Download`.
- **Избегайте использования общих имен** вроде `tmp` и `retval`, если только у вас нет для этого особой причины.
- **Применяйте конкретные имена,** которые описывают процесс работы программы более детально, — имя `ServerCanStart()` бесполезно по сравнению с `CanListenOnPort()`.
- **Добавляйте важные детали** к именам переменных, например постфикс `_ms` для переменных, значение которых измеряется в миллисекундах, или префикс `raw_` для переменных, содержащих необработанные данные.
- **Для больших областей видимости** используйте более длинные имена. Избегайте использования одно- или двухбуквенных имен для переменных, которые встречаются на протяжении всего кода. Более короткие имена предпочтительны для переменных, которые встречаются всего несколько раз.
- **Применяйте прописные буквы, подчеркивания и прочее форматирование,** чтобы сделать код более понятным. Например, вы можете добавить символ «_» к переменным, которые являются членами класса, чтобы отличать их от локальных переменных.

3 Имена, которые нельзя понять неправильно



[1] Здесь также имеет место игра слов: в английском языке слово right означает как «правый», так и «верно». – *Примеч. пер.*

В предыдущей главе мы рассмотрели, как делать имена информационно насыщенными. В этой главе мы обратим внимание на то, как гарантировать, что имена будут поняты правильно.

ОСНОВНАЯ ИДЕЯ

Внимательно изучайте имена, применяемые в программе, и задавайтесь вопросом: «Как еще могут интерпретировать это имя люди, которые будут читать код в дальнейшем?»

Постарайтесь быть креативным, активно отыскивая «неверные интерпретации». Этот шаг поможет вам обнаружить двусмысленные имена, которые вы сможете изменить на верные.

В каждом примере этой главы мы будем говорить о том, как именно можно интерпретировать каждое имя, которое мы видим. Затем будем подбирать более подходящий вариант.

Пример: Filter() (фильтрация)

Представьте, что вы пишете код, который манипулирует несколькими результатами, полученными из базы данных:

```
results = Database.all_objects.filter("year <= 2011")
```

Что будет содержать переменная `results` после выполнения этой строки?

- Объекты, чье значение атрибута `year` <= 2011?
- Объекты, чье значение атрибута `year` *не* <= 2011?

Проблема заключается в том, что слово `filter` может иметь два значения. Не совсем понятно, что именно требуется сделать – выбрать или удалить значения. Постарайтесь избегать слова `filter`, поскольку его можно понять неправильно.

Если вы хотите выбрать значения, то лучше использовать имя `select()`. Если же необходимо удалить значения, то подойдет, например, вариант `exclude()` (исключить).

Пример: Clip(text, lenght) (обрезать)

Представьте, что у вас есть функция, которая обрезает содержимое абзаца:

```
# обрезаем конец текста и добавляем в конец текста "..."
def Clip(text, lenght):
```

Теоретически функция `Clip()` может вести себя по-разному:

- удалять участок заданной длины из конца текста;
- уменьшить текст так, чтобы его длина была равна указанной.

Второй вариант более вероятен, но вы не можете быть в этом уверены. Вместо того чтобы озадачить человека, читающего ваш код, лучше переименовать функцию в `Truncate(text, length)` (укоротить).

Однако имя параметра `length` (длина) также может ввести читателя в заблуждение. Если бы этот параметр назывался `max_length` (максимальная длина), то разобраться в коде можно было бы быстрее.

В то же время и `max_lenght` можно интерпретировать по-разному. Оно может означать количество:

- байтов;
- символов;
- слов.

Как вы знаете из предыдущей главы, в таких случаях к имени необходимо добавить единицы измерения. В этом случае мы имеем в виду количество символов, поэтому вместо имени `max_lenght` следует использовать имя `max_chars`.

Применяйте префиксы min и max для (включающих) границ

Давайте предположим, что приложение-корзина в интернет-магазине должно иметь ограничение, запрещающее пользователям покупать более 10 товаров за один заказ:

```
CART_TOO_BIG_LIMIT = 10

if shopping_cart.num_items() >= CART_TOO_BIG_LIMIT:
    Error("Too many items in cart.")
```

В этом коде есть классическая ошибка «на единицу». Мы можем с легкостью исправить ее, заменив `>=` на `>`:

```
if shopping_cart.num_items() > CART_TOO_BIG_LIMIT:
```

(Или же мы можем изменить значение константы `CART_TOO_BIG_LIMIT`, сделав его равным 11.) Суть проблемы в том, что имя константы `CART_TOO_BIG_LIMIT` имеет два значения: оно может подразумевать как «меньше 10 товаров», так и «до 10 товаров».

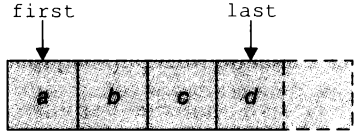
СОВЕТ

Имя границы будет выглядеть гораздо понятнее, если добавить префиксы max_ и min_ перед ограничиваемой величиной.

В нашем случае имя должно звучать как `MAX_ITEMS_IN_CART`. Код, использующий это имя, будет простым и понятным:

```
MAX_ITEMS_IN_CART = 10

if shopping_cart.num_items() >= MAX_ITEMS_IN_CART:
    Error("Too many items in cart.")
```

Используйте именах границ слова first и last



Вот еще один пример, в котором вы не можете сказать точно, какой элемент списка будет последним:

```
print integer_range(start=2, stop=4)
# Будет напечатано [2,3] или [2,3,4] (или что-то еще)?
```

Несмотря на то что `start` — приемлемое имя параметра, имя `stop` может быть интерпретировано по-разному.

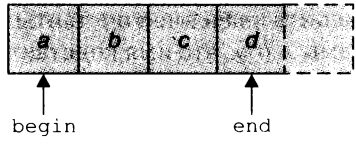
Для *включающих* границ, как в нашем случае (границы должны включать обе конечные точки), хорошо подойдут имена `first` и `last`. Например:

```
Set.PrintKeys(first="Bart", last="Maggie")
```

В отличие от `stop`, слово `last` явно указывает на то, что эти границы включающие.

Для подобных границ также могут подойти и имена `min` и `max`, при условии, что их употребление возможно в этом контексте.

Используйте имена begin и end для включающе-исключающих границ



На практике часто оказывается, что удобнее использовать включающе-исключающие границы. Например, если вы хотите вывести список всех событий, произошедших 16 октября, гораздо проще написать:

```
PrintEventsInRange("OCT 16 12:00am", "OCT 17 12:00am")
```

чем:

```
PrintEventsInRange("OCT 16 12:00am", "OCT 16 11:59:59.9999pm")
```

Так как же лучше назвать эту пару параметров? По соглашению включающе-исключающие границы называются `begin/end`.

Однако имейте в виду, что слово end может иметь еще одно значение. Например, в предложении I'm at the end of the book (Я нахожусь в конце книги) слово end обозначает включающую границу. К несчастью, в английском языке нет слова для обозначения предпоследнего значения.

Поскольку begin и end — устоявшаяся пара антонимов (по крайней мере, она используется в стандартных библиотеках C++, а также в большинстве мест, где необходимо перемещаться по массиву таким способом), она является более подходящим вариантом.

Называем булевы переменные

Подбирая имя для булевой переменной или функции, которая возвращает булево значение, убедитесь, что это имя ясно дает понять, что будут обозначать значения true и false.

Рассмотрим опасный пример:

```
bool read_password = true;
```

Основываясь только на этой строке кода, вы можете интерпретировать ее по-разному:

- нужно прочесть пароль;
- пароль уже был прочитан.

В этом случае лучше избегать употребления слова read и назвать переменную need_password (необходим пароль) или user_is_authenticated (пользователь идентифицирован).

В общем, добавление в имена переменных таких слов, как is, has, can или should (есть, имеется, может, должен), может упростить чтение кода с булевыми переменными.

Например, имя функции SpaceLeft() подразумевает, что возвращается некоторое число. Но если на самом деле она возвращает булево значение, то ее лучше назвать HasSpaceLeft().

Наконец, при выборе имени следует избегать *отрицающих* конструкций. Например, вместо:

```
bool disable_ssl = false;
```

лучше использовать следующую конструкцию, которую проще прочитать (она также более компактна):

```
bool use_ssl = true;
```

Оправдываем ожидания пользователей

Некоторые имена могут ввести в заблуждение только потому, что у пользователя уже есть представление о том, что значит это слово, несмотря на то, что вы имеете в виду что-то другое. В таких случаях лучше всего просто изменить имя на более понятное.

Пример: get*() (получать)

Многие программисты считают, что методы, имена которых начинаются со слова get, являются «средствами быстрого доступа», просто возвращающими внутренний член. Если метод с подобным именем будет исполнять другие функции, то программа может ввести в заблуждение многих пользователей.

Приведем пример на языке Java, который показывает, как не надо поступать:

```
public class StatisticsCollector {
    public void addSample(double x) {     }

    public double getMean() {
       // Проходим по всем значениям и возвращаем их сумму,
       // деленную на их количество.
    }

}
```

В данном случае назначение метода getMean() заключается в том, что он проходит по каждому полученному значению и рассчитывает на ходу среднее значение. Этот метод может выполняться очень долго, если программа оперирует огромным количеством данных. Но ничего не подозревающий программист может использовать этот метод, поскольку будет считать, что его применение не займет много времени.

Чтобы такого не произошло, данный метод следует назвать как-нибудь вроде computeMean(). Такое имя подразумевает, что эта операция будет выполняться довольно долго. (Возможно также изменить реализацию этого метода, сделав его более быстрым.)

Пример: list::size() (размер списка)

Приведем пример из стандартной библиотеки C++. В следующем фрагменте кода довольно сложно найти ошибку. Это привело к тому, что работа одного из наших серверов практически остановилась:

```
void ShrinkList(list<Node>& list, int max_size) {
    while (list.size() > max_size) {
        FreeNode(list.back());
        list.pop_back();
    }
}
```

«Ошибка» заключалась в следующем: автор не знал о том, что функция list.size() является операцией типа $O(n)$ — она подсчитывает элементы списка один за другим вместо того, чтобы возвращать предварительно вычисленное значение, что делает метод ShrinkList() операцией типа $O(n^2)$.

Технически код был совершенно правильным и прошел все наши тесты компонентов. Но как только функция ShrinkList() вызывалась для списка, который содержал миллион элементов, ей требовалось больше часа, чтобы закончить работу!

Возможно, сейчас вы думаете: «Вызвавший эту функцию сам виноват — следует более внимательно изучать документацию». Верно, но в данном случае тот факт, что функция `list.size()` не является мгновенной операцией, несколько удивляет. Все прочие контейнеры в C++ имеют мгновенный метод `size()`.

Если бы метод `size()` (размер) был назван `countSize()` (подсчитать размер) или `countElements()` (подсчитать количество элементов), подобная ошибка вряд ли бы повторилась. Возможно, создатели стандартной библиотеки C++ хотели назвать метод `size()`, чтобы контейнер походил на другие, такие как `vector` или `map`. Из-за этого несоответствия программисты ошибочно считают этот метод быстрым, таким, как он реализован в других контейнерах. К счастью, последние стандарты C++ описывают эту функцию как *O(1)*.

КТО ЖЕ МАСТЕР?

Некоторое время назад один из авторов этой книги устанавливал операционную систему OpenBSD. На этапе форматирования диска появилось меню с большим количеством пунктов, запрашивающее параметры диска. Одним из пунктов этого меню был переход в режим мастера (wizard). Автор был рад встретить этот удобный для пользователя пункт и выбрал его. К его ужасу, установка прервалась и появились низкоуровневые приглашения, ожидающие ручного ввода форматирующих команд. Причем способ выйти из него не был очевидным. Вероятно, подразумевалось, что «мастер» — это вы!

Пример: оценка нескольких вариантов названия

Когда вы выбираете имя, у вас может быть множество возможных вариантов. Обычно многие мысленно разбирают каждый вариант, перед тем как сделать окончательный выбор. Следующий пример иллюстрирует этот процесс.

Загруженные трафиком веб-сайты часто проводят эксперименты для того, чтобы узнать, пойдут ли на пользу бизнесу изменения на веб-сайте. Перед вами пример конфигурационного файла, который управляет некоторыми экспериментами:

```
experiment_id: 100
description: "increase font size to 14pt"
traffic_fraction: 5%
```

Каждый эксперимент определен примерно 15 парами «значение/атрибут». К несчастью, при определении очередного эксперимента, который очень похож на вышеописанный, вы скопировали и вставили большую часть его строк:

```
experiment_id: 100
description: "increase font size to 13pt"
[остальные строки, такие же, как в эксперименте с параметром experiment_id, равным 100]
```

Предположим, что мы хотим исправить эту ситуацию, разрешив одному эксперименту использовать свойства другого. (Этот паттерн называется «Наследование прототипа».) В итоге вы можете получить примерно следующий результат:

```
experiment_id: 101
the_other_experiment_id_I_want_to_reuse: 100
[здесь можно изменить необходимые свойства]
```

Вопрос заключается в следующем: как назвать параметр, который в данный момент имеет имя `the_other_experiment_id_I_want_to_reuse` (другой эксперимент, данные которого я хочу использовать)?

Рассмотрим четыре варианта.

1. `template` (шаблон).
2. `reuse` (повторное использование).
3. `copy` (копирование).
4. `inherit` (наследование).

Любое из этих имен имеет смысл для нас, поскольку именно мы добавляем новые функции в конфигурацию. Но давайте представим, как будут звучать эти имена для стороннего человека, который будет читать код, не зная об этой особенности. Проанализируем каждое имя с точки зрения того, могут ли пользователи неправильно его понять.

1. Предположим, что мы применяем имя `template`:

```
experiment_id: 101
template: 100
```

Имя `template` не очень подходит. Во-первых, не совсем ясно, что именно мы хотим этим сказать: «Я шаблон» или «Я использую шаблон»? Во-вторых, слово «шаблон» часто обозначает что-либо *абстрактное*. Нужно его заполнить данными, и тогда оно станет *конкретным*. Кто-то может подумать, что шаблонный эксперимент — это не настоящий эксперимент. В силу вышеназванных фактов можем заключить, что имя `template` в данном случае бесполезно.

2. Теперь проанализируем слово `reuse`:

```
experiment_id: 101
reuse: 100
```

Слово `reuse` подходит лучше, однако некоторые люди могут подумать, что мы повторяем эксперимент не более ста раз. Мы можем исправить это, изменив имя на `reuse_id`. Однако запутавшийся читатель может подумать, что строка `reuse_id: 100` означает «мой номер для повторного использования — 100».

3. Рассмотрим слово `copy`:

```
experiment_id: 101
copy: 100
...
```

`copy` — это подходящее слово. Но отдельная строка `copy: 100` может означать, что эксперимент необходимо скопировать сто раз или же что это сотая копия чего-либо. Чтобы прояснить тот факт, что слово `copy` относится к *другому* эксперименту, изменим имя на `copy_experiment`. Это имя пока является лучшим из предложенных.

4. Теперь разберем слово `inherit`:

```
experiment_id: 101
inherit: 100
```

Слово `inherit` знакомо многим программистам. Легко понять, что все дальнейшие модификации осуществляются уже после наследования. При наследовании классов вы получаете все методы и члены другого класса, а затем изменяете их или добавляете новые. Можно также привести пример из реальной жизни. Когда вы получаете в наследство собственность от родственников, вы можете продать ее или оставить вещи себе.

Но опять же для того, чтобы прояснить тот факт, что мы наследуем от другого эксперимента, мы должны изменить имя на `inherit_from` или даже на `inherit_from_experiment_id`.

В результате мы определили, что имена `copy_experiment` и `inherit_from_experiment_id` подходят в данной ситуации лучше всего, поскольку они довольно ясно могут описать происходящее и, скорее всего, будут поняты правильно.

Итог

Лучшие имена — это такие имена, которые не могут быть поняты неправильно. Человек, читающий ваш код, поймет именно то, что вы хотите сказать, и ничто другое. К несчастью, в английском языке множество слов, имеющих несколько значений (например, `filter`, `length` и `limit`), что особенно влияет на процесс программирования.

Перед тем как вы окончательно определитесь с именем, попробуйте представить, как еще его могут понять. Вероятность того, что ваши лучшие имена будут поняты неправильно, стремится к нулю.

Когда речь идет о верхней или нижней границах какого-либо значения, вам могут помочь префиксы `max_` и `min_`. Для включающих границ больше подходят имена `first` и `last`. Для включающе-исключающих границ лучше использовать имена `begin` и `end`, поскольку они образуют устойчивую пару антонимов.

Когда вы называете булеву переменную, используйте слова `is` и `has`, чтобы таким образом показать ее тип. Избегайте отрицающих конструкций (например, `disable_ssl`).

Вам следует также предусмотреть ожидания пользователей насчет некоторых слов. Например, многие могут полагать, что работа функций с именами `get()` и `size()` не займет много времени.

4 Эстетичность

Бывает непросто создать макет журнала: следует обдумать размер страниц, ширину колонок, порядок статей, а также содержимое обложки. Хороший журнал приятно как прочитать от корки до корки, так и просто пролистать.

Хороший исходный код также должен быть приятным на вид. В этой главе мы рассмотрим, как разумное использование пробелов, выравниваний и размещения может улучшить читаемость кода.

Мы будем использовать три следующих принципа:

- необходимо применять постоянный шаблон, к элементам которого читатель может легко привыкнуть;
- похожие строки кода следует оформлять одинаково;
- строки кода, решающие одну и ту же задачу, нужно объединять в блоки.

КРАСОТА ПРОТИВ ДИЗАЙНА

В этой главе мы сфокусируемся на простых эстетических улучшениях, которые вы можете внести в свой код. Эти изменения довольно легко произвести, и при этом они могут значительно повысить читаемость кода. Иногда серьезный рефакторинг кода (например, выделение новых функций или классов) может помочь еще больше. С нашей точки зрения, эстетика и качественный дизайн — это независимые критерии. Вы должны стремиться к тому, чтобы ваш код удовлетворял обоим этим понятиям.

Почему красота имеет значение?

Представьте, что вам нужно использовать следующий класс:

```
class StatsKeeper {
public:
```

```
// Класс для отслеживания последовательности чисел в формате double
   void Add(double d); // и методы, необходимые для получения
                       // статистической информации о них.
  private: int count;       /* подсчитываем их      количество
*/ public:
       double Average();

private:     double minimum;
list<double>
  past_items
     :double maximum;
};
```

Чтобы разобраться в нем, вам потребуется гораздо больше времени, чем на изучение более понятной версии:

```
// Класс для отслеживания последовательности чисел в формате double
// и методы, необходимые для получения статистической информации о них.
class StatsKeeper {
  public:
    void Add(double d);
    double Average();

  private:
    list<double> past_items;
    int count;  // подсчитываем их количество

    double minimum;
    double maximum;
};
```

Очевидно, что гораздо проще работать с кодом, который приятнее на вид. Ведь если вдуматься, то окажется, что во время программирования большую часть времени вы проводите, вглядываясь в код! Чем быстрее вы сможете просматривать код, тем проще будет другим людям использовать его.

Перераспределение разрывов строк сделает код более последовательным и компактным

Представьте, что вы пишете приложение на языке Java, которое исследует поведение программы при различных скоростях сетевого соединения. У вас есть класс `TcpConnectionSimulator`, конструктор которого принимает четыре параметра:

- скорость соединения (Кбит/с);
- среднее значение времени ожидания (мс);
- колебание времени ожидания (мс);
- потерянные пакеты (в процентах).

В коде необходимо создать три объекта класса TcpConnectionSimulator:

```
public class PerformanceTester {
    public static final TcpConnectionSimulator wifi = new TcpConnectionSimulator(
        500, /* Кбит/с */
        80, /* время ожидания в миллисекундах*/
        200, /* колебание */
        1 /* процент потерянных пакетов */);

    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(
            45000, /* Кбит/с */
            10, /* время ожидания в миллисекундах */
            0, /* колебание */
            0 /* процент потерянных пакетов */);

    public static final TcpConnectionSimulator cell = new TcpConnectionSimulator(
        100, /* Кбит/с */
        400, /* время ожидания в миллисекундах */
        250, /* колебание */
        5 /* процент потерянных пакетов */);
}
```

В этом фрагменте кода нужно применить множество разрывов строки для того, чтобы «вписать» его в заданные ограничения (80 символов, стандарт программирования в вашей компании именно таков). К несчастью, их появление привело к тому, что определение объекта t3_fiber стало отличаться от прочих объектов того же класса. Код стал выглядеть довольно странно, а также обращать внимание читателей на объект t3_fiber без причины. Такая ситуация не соответствует принципу «похожие строки кода следует оформлять одинаково».

Чтобы код выглядел более последовательно, добавим еще несколько разрывов строки (а также подровняем комментарии):

```
public class PerformanceTester {
    public static final TcpConnectionSimulator wifi =
        new TcpConnectionSimulator(
            500,    /* Кбит/с */
            80,     /* время ожидания в миллисекундах */
            200,    /* колебание */
            1       /* процент потерянных пакетов */);

    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(
            45000,  /* Кбит/с */
            10,     /* время ожидания в миллисекундах */
            0,      /* колебание */
            0       /* процент потерянных пакетов */);

    public static final TcpConnectionSimulator cell =
        new TcpConnectionSimulator(
```

```
            100,   /* Кбит/с */
            400,   /* время ожидания в миллисекундах */
            250,   /* колебание */
            5      /* процент потерянных пакетов */);
}
```

Этот код оформлен по замечательному последовательному шаблону, его довольно удобно просматривать. Но, к несчастью, он занимает слишком много места по вертикали. Кроме того, комментарии в нем повторяются трижды.

Перепишем его более компактно[1]:

```
public class PerformanceTester {
    // TcpConnectionSimulator
    // (скорость соединения, время ожидания, колебание, потерянные пакеты)
    // [Кбит/с]              [мс]            [мс]       [процент]

    public static final TcpConnectionSimulator wifi =
        new TcpConnectionSimulator
       (500,                 80,             200,       1);

    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator
       (45000,               10,             0,         0);

    public static final TcpConnectionSimulator cell =
        new TcpConnectionSimulator
       (100;                 400,            250,       5);
}
```

Мы переместили комментарии в верхнюю часть кода и записали все параметры в одну строку. Теперь, даже несмотря на то, что комментарии располагаются дальше от чисел, которые они поясняют, данные упорядочены более аккуратно.

Избавляемся от неоднородности с помощью методов

Предположим, что у вас есть база данных, содержащая информацию о персонале, которая предоставляет следующую функцию:

```
// Преобразуем сокращенное имя вроде "Doug Adams" в "Mr. Douglas Adams".
// Если это невозможно, заполняем поле значением 'error'
// и объяснением причины.
string ExpandFullName(DatabaseConnection dc, string partial_name, string* error);
```

[1] К сожалению, из-за того, что аналогичные выражения в русском языке длиннее, чем в английском, нам не удалось так прекрасно выровнять данный код, как он был представлен в оригинале. В англоязычном издании этот код гораздо компактнее и занимает меньше места по высоте. — *Примеч. ред.*

Эта функция проверяется рядом примеров:

```
DatabaseConnection database_connection;
string error;
assert(ExpandFullName(database_connection, "Doug Adams", &error)
    == "Mr. Douglas Adams");
assert(error == "");
assert(ExpandFullName(database_connection, " Jake  Brown ", &error)
    == "Mr. Jacob Brown III");
assert(error == "");
assert(ExpandFullName(database_connection, "No Such Guy", &error) == "");
assert(error == "no match found");
assert(ExpandFullName(database_connection, "John", &error) == "");
assert(error == "more than one result");
```

Этот код выглядит довольно небрежно. Некоторые его строки настолько длинны, что переходят на следующую строку. Его внешний вид никуда не годится, код не имеет последовательного шаблона.

В этом случае ситуацию может исправить перестановка разрывов строк. Однако большей проблемой является наличие множества повторяющихся строк вроде `assert(ExpansFullName(database_connection...` и `error`, которые только мешают. Чтобы по-настоящему улучшить этот код, нужно ввести вспомогательный метод. В итоге код будет выглядеть следующим образом:

```
CheckFullName("Doug Adams", "Mr. Douglas Adams", "");
CheckFullName(" Jake  Brown ", "Mr. Jake Brown III", "");
CheckFullName("No Such Guy", "", "no match found");
CheckFullName("John", "", "more than one result");
```

Теперь становится понятнее, что проводятся четыре теста, каждый со своими параметрами. Вся «грязная работа» теперь выполняется внутри метода `CheckFullName()`, который, несмотря на это, также выглядит не так уж и некрасиво:

```
void CheckFullName(string partial_name,
                   string expected_full_name,
                   string expected_error) {
    // объект database_connection теперь является членом класса
    string error;
    string full_name = ExpandFullName(database_connection,
                                      partial_name, &error);
    assert(error == expected_error);
    assert(full_name == expected_full_name);
}
```

Помимо того что код стал более аккуратен на вид, это изменение приносит и немало пользы:

- мы избавились от массы повторяющегося кода, сделав текст программы более компактным;
- важные элементы каждого теста (имена и строки с сообщениями об ошибках) теперь находятся на виду. До этого они перемежались с другими, такими как `database_connection` и `error`, из-за чего их было трудно обнаружить;
- добавить новые тесты теперь будет намного проще.

Мораль такова — приведение кода в «опрятный» вид часто сулит не только внешние улучшения, оно помогает качественнее его структурировать.

Выравнивание столбцов

Ровные границы и столбцы позволяют читателям быстрее просматривать текст программы.

Иногда вы можете выравнивать столбцы, чтобы сделать код более удобным для чтения. Например, в предыдущем разделе можно было применить пробелы и выстроить аргументы функции `CheckFullName()` следующим образом:

```
CheckFullName():
CheckFullName("Doug Adams"    , "Mr. Douglas Adams" , "");
CheckFullName(" Jake  Brown ", "Mr. Jake Brown III", "");
CheckFullName("No Such Guy"   , ""                  , "no match found");
CheckFullName("John"          , ""                  , "more than one result");
```

В этом фрагменте кода проще заметить второй и третий аргументы функции `CheckFullName()`.

Рассмотрим простой пример. Допустим, имеется большая группа определений переменных:

```
# Извлекаем параметры из POST и помещаем их в локальные переменные
details  = request.POST.get('details')
location = request.POST.get('location')
phone    = equest.POST.get('phone')
email    = request.POST.get('email')
url      = request.POST.get('url')
```

Как видите, в третьем определении есть опечатка (`equest` вместо `request`). Подобные ошибки гораздо легче заметить, когда код выровнен именно так.

В базе кода `wget` доступные опции командной строки (всего их более 100) записаны следующим образом:

```
commands[] = {
    ...
    { "timeout",          NULL,                 cmd_spec_timeout },
    { "timestamping",     &opt.timestamping,    cmd_boolean },
    { "tries",            &opt.ntry,            cmd_number_inf },
    { "useproxy",         &opt.use_proxy,       cmd_boolean },
    { "useragent",        NULL,                 cmd_spec_useragent },
    ...
};
```

Такой подход позволяет быстро просмотреть весь список, а также переходить от одного столбца к другому.

Обязательно ли выравнивать столбцы?

Ровные границы столбцов позволяют быстрее просматривать код. Это хороший пример принципа «похожие строки кода следует оформлять одинаково».

Однако некоторые программисты не любят пользоваться выравниванием. Одной причиной является необходимость проделывать массу работы для того, чтобы выравнивание выглядело как следует. Другая причина — при изменении одной строки может измениться сразу несколько других (в основном только лишь отступы).

Мы советуем вам попробовать применять выравнивание. По нашему опыту, оно не займет слишком много времени. А если все же займет, то вы можете просто отказаться от него.

Выберите определенный порядок и придерживайтесь его

Во многих случаях порядок кода не влияет на его правильность. Например, эти пять определений переменных могут быть записаны в любом порядке:

```
details  = request.POST.get('details')
location = request.POST.get('location')
phone    = request.POST.get('phone')
email    = request.POST.get('email')
url      = request.POST.get('url')
```

В подобных ситуациях удобно расставлять переменные не в случайном, а в определенном порядке. Есть несколько идей, как лучше это сделать:

- соотнесите порядок следования переменных и порядок следования полей <input> соответствующей HTML-формы;
- выстройте их от наиболее до наименее важного;
- расположите их в алфавитном порядке.

Независимо от выбора любой из этих идей, вы должны придерживаться выбранного порядка на протяжении всего кода. Не следует изменять порядок их следования в дальнейшем:

```
if details:  rec.details  = details
if phone:    rec.phone    = phone     # Эй, куда пропала переменная 'location'?
if email:    rec.email    = email
if url:      rec.url      = url
if location: rec.location = location  # Почему она опустилась сюда?
```

Объединяем объявления в блоки

Наш мозг привык использовать различные группы и классификации, поэтому вы можете помочь читателю быстро разобраться в вашем коде, если хорошо организуете его.

В качестве примера изучим класс фронтального сервера, написанный на языке C++. Рассмотрим описание всех его методов:

```
class FrontendServer {
  public:
```

```
    FrontendServer();
    void ViewProfile(HttpRequest* request);
    void OpenDatabase(string location, string user);
    void SaveProfile(HttpRequest* request);
    string ExtractQueryParam(HttpRequest* request, string param);
    void ReplyOK(HttpRequest* request, string html);
    void FindFriends(HttpRequest* request);
    void ReplyNotFound(HttpRequest* request, string error);
    void CloseDatabase(string location);
    ~FrontendServer();
};
```

Этот код выглядит не так плохо, но его компоновка не позволяет читателю быстро разобраться во всех методах. Вместо банального перечисления методов в одном гигантском блоке нужно разбить их на небольшие локальные группы следующим образом:

```
class FrontendServer {
 public:
    FrontendServer();
    ~FrontendServer();

    // обработчики
    void ViewProfile(HttpRequest* request);
    void SaveProfile(HttpRequest* request);
    void FindFriends(HttpRequest* request);

    // утилиты запроса/ответа
    string ExtractQueryParam(HttpRequest* request, string param);
    void ReplyOK(HttpRequest* request, string html);
    void ReplyNotFound(HttpRequest* request, string error);

    // вспомогательные функции для работы с базами данных
    void OpenDatabase(string location, string user);
    void CloseDatabase(string location);
};
```

Такой вариант кода гораздо легче усвоить. Его также довольно просто читать, даже несмотря на то, что количество строк увеличилось. Причина заключается в том, что вы быстро можете выделить четыре основных раздела, а затем при необходимости уточнить их детали.

Разбиваем код на абзацы

Обычный текст разбивают на абзацы по нескольким причинам:

- это хороший способ сгруппировать схожие идеи и отделить их от прочих;
- это позволяет наметить визуальные ориентиры, без которых довольно легко потеряться в тексте;
- это упрощает навигацию в тексте.

Код необходимо разбивать на абзацы по тем же причинам. Например, никому не нравится просматривать такие большие участки кода:

```
# Импортируем контакты из электронной почты пользователя, а затем
# проверяем их на совпадения с адресами уже существующих пользователей.
# Потом отображаем список тех пользователей, которые пока не находятся
# у него в списке друзей.
def suggest_new_friends(user, email_password):
    friends = user.friends()
    friend_emails = set(f.email for f in friends)
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email__in=non_friend_emails)
    display['user'] = user
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends
    return render("suggested_friends.html", display)
```

Возможно, это неочевидно, но данная функция выполняет несколько отдельных шагов. Было бы полезно объединить соответствующие строки кода в абзацы:

```
def suggest_new_friends(user, email_password):
    # Получаем адреса электронной почты пользователей,
    # которые находятся в списке друзей.
    friends = user.friends()
    friend_emails = set(f.email for f in friends)

    # Импортируем все адреса электронной почты с аккаунта пользователя.
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)

    # Находим соответствия среди пользователей,
    # которые пока не находятся в списке друзей.
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email__in=non_friend_emails)

    # Отображаем на странице список возможных друзей.
    display['user'] = user
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends

    return render("suggested_friends.html", display)
```

Обратите внимание, мы добавили к каждому абзацу комментарий, содержащий описание того, что будет происходить далее. Это также помогает читать код бегло (подробнее об этом читайте в гл. 5).

Как и для обычного текста, есть несколько способов разбить код на абзацы. Программисты могут предпочитать более длинные или более короткие абзацы.

Персональный стиль или единообразие?

Некоторые эстетические особенности зависят исключительно от персонального стиля программиста. Например, выбор способа оформления открывающих скобок при описании класса:

```
class Logger {

};
```

или

```
class Logger
{

};
```

Предпочтение одного из этих стилей другому мало повлияет на читаемость вашей кодовой базы. Однако смешение двух стилей повлияет на нее очень сильно.

Мы работали над многими проектами, в которых использовался «неправильный» стиль. Но следовали соглашениям, принятым в проекте, поскольку знали, что единообразие гораздо важнее.

ОСНОВНАЯ ИДЕЯ

Последовательный стиль гораздо важнее «правильного» стиля.

Итог

У каждого из нас свое понятие о красивом коде. Последовательное и осмысленное форматирование кода сделает его гораздо более читаемым.

В этой главе мы обсудили следующие приемы:

- если несколько блоков кода выполняют похожие операции, попробуйте оформить их одинаково;
- выравнивание фрагментов кода в столбцы упрощает его беглое чтение;
- если в одном месте кода упоминаются А, Б и В, не меняйте их расстановку (например, Б, В и А) в другом. Выберите осмысленный порядок и придерживайтесь его;
- используйте пустые строки для того, чтобы разбить большие блоки кода на логические абзацы.

5 Комментируем мудро

РУКОВОДСТВА ПОЛЬЗОВАТЕЛЯ

ТРЕБУЮТСЯ!!

НЕ ТРЕБУЮТСЯ

Цель этой главы — донести до читателя, что именно нужно комментировать. Вы можете считать, что смысл комментирования заключается в том, чтобы объяснить, что делает код. Однако это всего лишь верхушка айсберга.

ОСНОВНАЯ ИДЕЯ

Смысл комментирования заключается в том, чтобы помочь читателю получить столько же информации, сколько имеет разработчик.

Когда вы пишете код, у вас в голове содержится большое количество ценной информации. Когда ваш код читают другие люди, эта информация им недоступна. Все, что у них есть, — это ваша программа.

В этой главе мы покажем вам множество примеров того, когда именно следует записывать эту информацию, а также рассмотрим многие недостаточно известные аспекты комментирования, не останавливаясь на самых распространенных.

Мы разбили эту главу на разделы, в которых рассматриваем следующие действия:

- определяем, что *не* нужно комментировать;
- записываем свои мысли вместе с кодом;
- ставим себя на место читателя, чтобы представить, что именно ему необходимо знать.

Что НЕ нужно комментировать

Чтение комментариев занимает какое-то время, которое можно было бы потратить на чтение кода. Каждый комментарий занимает еще и место на экране. Поэтому комментарии должны иметь значение. Как же провести границу между бесполезным и хорошим комментарием?

Все комментарии, встречающиеся в следующем фрагменте кода, бесполезны:

```
// описание класса Account
class Account {
  public:
    // конструктор
    Account();

    // устанавливаем новое значение члена profit
    void SetProfit(double profit);

    // возвращаем значение члена profit этого объекта класса Account
    double GetProfit();
};
```

Эти комментарии бесполезны, поскольку не предоставляют никакой новой информации и не помогают лучше разобраться в коде.

ОСНОВНАЯ ИДЕЯ

Не комментируйте строки, предназначение которых можно легко и быстро понять и без комментариев.

Слово «быстро» здесь немаловажно. Рассмотрим пример комментария к коду, написанному на языке Python:

```
# удаляем все после второго символа '*'
name = '*'.join(line.split('*')[:2])
```

Технически этот комментарий также не предоставляет никакой новой информации. Если вы изучите код, то в конечном счете разберетесь, что именно он делает. Но большинство программистов предпочитают читать код с комментариями, это помогает им разбираться в нем быстрее, чем в коде без комментариев.

Не комментируйте только для того, чтобы комментировать

Некоторые профессора требуют, чтобы их студенты комментировали в своей домашней работе каждую функцию. В результате некоторые программисты чувствуют

себя виноватыми из-за того, что оставили какую-либо функцию без комментариев, и переписывают имя функции и ее аргументы в виде предложения:

```
// Находим узел в заданном поддереве, с заданным именем,
// опускаясь на заданную глубину.
Node* FindNodeInSubtree(Node* subtree, string name, int depth)[1];
```

Подобные комментарии попадают в категорию бесполезных — объявление функции и комментарий практически совпадают. Такой комментарий следует либо удалить, либо оптимизировать.

Если вы хотите оставить здесь комментарий, то следует обратить внимание на более важные детали:

```
// Находим узел с заданным именем 'name' или возвращаем NULL.
// Если глубина <= 0, исследуем только поддерево 'subtree'.
// Если глубина равна == N, исследуем только поддерево 'subtree'
// и N уровней под ним.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

Не комментируйте плохие имена — лучше исправьте их

Комментарий не должен описывать плохое имя. Например, рассмотрим совершенно безобидный комментарий функции, которая называется `CleanReply()` (очищаем ответ):

[1] Имя функции — «поиск узла в поддереве». Ее аргументы — поддерево, имя и глубина. — *Примеч. пер.*

```
// Применяем к параметру Reply ограничения, указанные в параметре Request,
// такие как количество возвращаемых элементов, количество байтов и т. д.
void CleanReply(Request request, Reply reply);
```

Большая часть комментария — это не более чем простое объяснение того, что означает слово `Clean`. Вместо этого фраза «применим ограничения» (enforce limits) должна быть помещена в имя функции:

```
// Убедимся, что параметр 'reply' соответствует ограничениям по количеству
// элементов, размеру и т. д., заложенным в параметр 'request'.
void EnforceLimitsFromRequest(Request request, Reply reply);
```

Теперь имя этой функции говорит само за себя. Хорошее имя лучше хорошего комментария потому, что оно видно в любом месте кода, где используется функция.

Вот еще один пример комментария к неудачно названной функции:

```
// Возвращает обработчик для этого ключа.
// Не изменяет содержимое самого реестра.
void DeleteRegistry(RegistryKey* key);
```

Имя `DeleteRegistry()` звучит весьма угрожающе (эта функция *удаляет* реестр?!). Комментарий: «Не изменяет содержимое самого реестра» – является попыткой исправить недоразумение.

Вместо этого мы можем использовать более подходящее имя, например `ReleaseRegistryHandle` (возвращает обработчик реестра):

```
void ReleaseRegistryHandle(RegistryKey* key);
```

Не советуем вам использовать вспомогательные комментарии (crutch comments), исправляющие недостоверности в именах функций. Программисты часто записывают это правило следующим образом:

хороший код > плохой код + хорошие комментарии.

Записываем ваши мысли

Теперь, когда вы знаете, что *не* нужно комментировать, обсудим, что именно *должно* быть прокомментировано (но часто остается без комментариев).

Большинство хороших комментариев получаются в результате того, что вы просто записываете свои мысли — важные замечания, которые возникли у вас в процессе написания кода.

Добавьте «комментарий режиссера»

К фильмам часто прилагаются записи с режиссерскими комментариями, в которых создатели фильма рассказывают истории и делятся своими мыслями, возникшими в процессе съемки, чтобы помочь вам понять, как именно создавался этот фильм. Подобным образом вы должны добавлять в ваш код комментарии, помогающие лучше понять его.

Рассмотрим следующий пример:

```
// Удивительно для этого типа данных, но бинарное дерево оказалось
// на 40% быстрее, чем хэш-таблица.
// На вычисление хэша потребовалось больше времени,
// чем для сравнений право/лево.
```

Этот комментарий сообщает читателю полезные сведения, а также информирует любых возможных оптимизаторов и сохраняет их время.

Еще один пример:

```
// Данная эвристика может не содержать нескольких слов.
// Это нормально, сложно предусмотреть все на 100 %.
```

Без этого комментария читатель может подумать, что нашел ошибку, и потерять время, пробуя исправить ее либо пытаясь обнаружить отсутствующие слова путем тестов.

Комментарий также может пояснить, почему приведенный код неидеален:

```
// Содержание этого класса становится все более путаным.
// Возможно, нам стоило создать подкласс 'ResourceNode' для обеспечения
// лучшей организации.
```

Этот комментарий подтверждает, что код неидеален, но также вдохновляет следующего разработчика (сообщая о том, с чего можно начать). Без данного комментария множество читателей не рискнули бы трогать этот беспорядочный код.

Комментируем недостатки кода

Код постоянно развивается, это связано с наличием в нем недостатков. Не стыдитесь документировать подобные изъяны. Например, вы можете записать информацию о потенциальных улучшениях:

```
// TODO: использовать более быстрый алгоритм.
```

Или же отметить, что код незавершен:

```
// TODO(dustin): обрабатывать другие форматы изображений помимо JPEG.
```

Ваша команда может пользоваться специальными соглашениями о том, в каких случаях использовать эти обозначения (табл. 5.1). Например, `TODO:` может быть зарезервировано для самых значительных идей. Если вы применяете это соглашение, то более мелкие изъяны можете обозначать как-нибудь вроде `todo:` (в нижнем регистре) или `maybe-later:` (сделать позже).

Таблица 5.1

Обозначение	Значение
TODO:	Задуманное, но нереализованное
FIXME:	Известно, что здесь есть проблема
HACK:	Неэлегантное решение проблемы
XXX:	Внимание! Серьезная проблема

Важно понимать, что вы можете свободно записывать собственные мысли о том, как, по вашему мнению, следует изменить код в будущем. Подобные комментарии позволяют читателям понять качество и состояние кода или даже указать им направление, действуя в котором они могут улучшить код.

Комментируйте ваши константы

Каждый раз, когда вы определяете константу, вы можете рассказать о том, что именно она делает и почему имеет специфическое значение. Например, в коде вы можете встретить следующую константу:

```
NUM_THREADS = 8
```

Возможно, эта строка не требует комментариев, но наверняка программист, создавший ее, знает о ней больше, чем вы:

```
NUM_THREADS = 8  # Значение является подходящим,
                 # пока оно  >= 2 * num_processors.
```

Теперь человек, читающий код, имеет некоторое представление о том, как можно изменить значение (например, значение 1 слишкое маленькое, а значение 50 — слишком большое).

Иногда нет необходимости в точном значении константы, однако комментарий все равно может быть полезным:

```
// Определим обоснованные границы — ни один человек
// не сможет столько прочитать.
const int MAX_RSS_SUBSCRIPTIONS = 1000;
```

Иногда константа содержит значение, которое было тщательно выверено, и, возможно, его не следует значительно изменять:

```
image_quality = 0.72; // Пользователи считают, что 0.72 — это лучшее
                      // соотношение размера и качества.
```

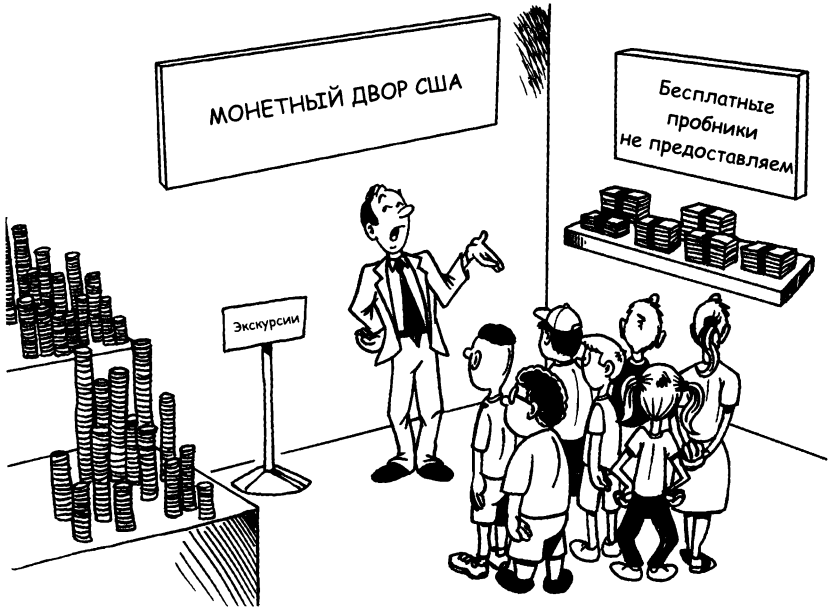
Во всех приведенных выше примерах можно вообще не добавлять комментарий, но они все довольно полезны.

Некоторые константы не нуждаются в комментариях, поскольку их имя и без того довольно ясное (например, SECONDS_PER_DAY (секунд в день)). Но, по нашему опыту, большинству констант добавление комментария идет на пользу. Всего лишь кратко записывайте все, о чем вы думали в тот момент, когда определяли значение константы.

Поставьте себя на место читателя

Главный прием, который мы используем на протяжении всей книги, — **представить, как выглядит ваш код для постороннего человека**, который непричастен к его разработке. Этот прием особенно полезен тогда, когда вы определяете, что именно необходимо комментировать.

Опережаем часто задаваемые вопросы



«Есть ли какие-то вопросы?..
...На которые не отвечает эта вывеска».

Человеку, читающему ваш код, могут встретиться фрагменты, которые заставят его задуматься: «Что здесь имеется в виду?» Ваша обязанность — прокомментировать эти фрагменты.

Например, рассмотрим определение функции `Clear()`:

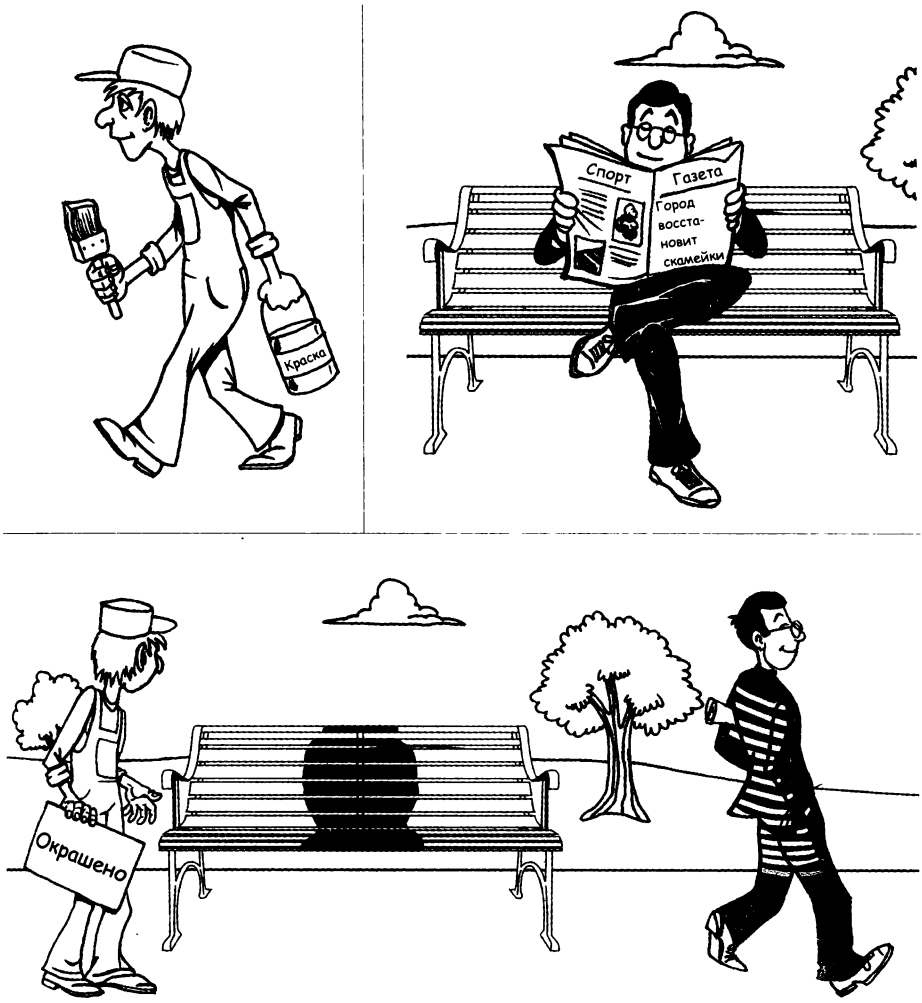
```
struct Recorder {
    vector<float> data;
    ...
    void Clear() {
        vector<float>().swap(data);  // Хм? Почему бы здесь не использовать
                                     // функцию data.clear()?
    }
};
```

Когда большинство программистов C++ видят этот код, они задаются вопросом, *почему бы здесь не использовать функцию data.clear() вместо того, чтобы меняться значениями с пустым вектором*? Однако далее оказывается, что это единственный способ передать память, принадлежащую контейнеру `vector`, распределителю памяти. Это особенность C++, которая недостаточно хорошо изучена.

Следует прокомментировать нижнюю строку:

```
// Принуждаем вектор передать свою память
// (для получения более подробной информации прочтите о трюке
// с использованием обмена для STL).
vector<float>().swap(data);
```

Отмечаем возможные ловушки



При документировании функции или класса полезно задаться вопросами: «Что удивительного может встретиться в этом коде? Как можно это неправильно использовать?» Проще говоря, вы хотите продумать наперед и предвосхитить возможные проблемы, с которыми люди могут столкнуться при применении вашего кода.

Например, представьте, что вы написали функцию, посылающую электронное письмо заданному пользователю:

```
void SendEmail(string to, string subject, string body);
```

Реализация этой функции включает в себя соединение с внешним сервисом электронной почты, а на это может уйти целая секунда или даже больше. Какой-нибудь

программист, пишущий веб-приложение, может и не подозревать об этом и ошибочно вызвать данную функцию при обработке HTTP-запроса (что заставит приложение зависнуть, если сервер отключен).

Чтобы предотвратить это, вы должны прокомментировать эту особенность реализации:

```
// Вызывает внешний сервис для доставки почты.
// (Выходит по тайм-ауту через одну минуту.)
void SendEmail(string to, string subject, string body);
```

Еще один пример: представьте, что у вас есть функция `FixBrokenHtml()`, которая пытается перезаписать нерабочий HTML-код, вставляя недостающие закрывающие теги:

```
Def FixBrokenHtml(html): ...
```

Функция работает замечательно практически всегда, за исключением тех случаев, когда теги расположены слишком глубоко в иерархии. В таких ситуациях функция может выполняться несколько *минут*.

Вместо того чтобы предоставить пользователю возможность обнаружить эту особенность самостоятельно, объявим об этом непосредственно перед функцией:

```
// Время работы равно O (количество тегов * средняя глубина тегов).
// Следите за тем, чтобы в функцию не передавались теги,
// спрятанные слишком глубоко.
Def FixBrokenHtml(html):
```

Комментарии «общей картины»

Одна из самых сложных задач для нового члена команды — это понимание общей картины: как взаимодействуют классы, как проходят данные через всю систему и где точки входа в программу. Создатель системы часто забывает о том, чтобы закомментировать эту информацию, поскольку он был слишком занят созданием самой системы.

Проведем мысленный эксперимент: **кто-то новый только что присоединился к вашей команде; этот человек сидит рядом с вами, и вам нужно познакомить его с базой кода**.

Во время «экскурсии» по базе кода вы должны показывать ему определенные файлы или классы и говорить что-нибудь вроде:

- «Этот код связывает логику приложения с базой данных. Ни один другой фрагмент кода не использует его напрямую»;
- «Этот класс выглядит довольно сложным, но на самом деле он является всего лишь умным кэшем. Он ничего не знает об остальной части системы».

Через минуту такого разговора ваш новый сотрудник будет знать гораздо больше, чем если бы он просто прочитал исходный код.

Эта информация должна присутствовать в высокоуровневых комментариях.

Приведем простой пример комментария на уровне файла:

```
// Этот файл содержит вспомогательные функции,
// предоставляющие более удобный интерфейс нашей файловой системе.
// Он обрабатывает права допуска к файлам и прочие мелкие детали.
```

Не следует писать здесь объемную официальную документацию. Несколько хорошо подобранных предложений — **это гораздо лучше, чем совсем ничего**.

Итоговые комментарии

Общую картину принято комментировать даже глубоко внутри функции. Рассмотрим пример комментария, подытоживающего низкоуровневый код, который расположен под ним:

```
# Найдем все предметы, которые приобрели покупатели.
for customer_id in all_customers:
    for sale in all_sales[customer_id].sales:
        if sale.recipient == customer_id:
```

Без этого комментария каждая строка данного кода может показаться несколько загадочной. (Читатель видит, что мы проходим по массиву всех покупателей... но для чего?)

Особенно полезны такие комментарии внутри больших функций, содержащих объемные фрагменты кода:

```
def GenerateUserReport():
    # Получаем ключ данного пользователя.


    # Считываем информацию о пользователе из базы данных.
```

```
# Записываем информацию в файл.


# Возвращаем ключ данного пользователя.
```

Эти комментарии также сообщают нам о том, что делает функция, поэтому читатель может понять основные шаги, не вдаваясь в подробности. (Если эти объемные участки кода легко разделить, то вы можете обособить их фрагменты в отдельные функции. Как мы упоминали ранее, хороший код без комментариев всегда лучше, чем плохой код с хорошими комментариями.)

ЧТО ИМЕННО СЛЕДУЕТ КОММЕНТИРОВАТЬ: ОБЪЕКТ, ПРИЧИНУ ИЛИ СПОСОБ?

Возможно, вы слышали совет вроде «необходимо комментировать причину, а не объект (или способ)»?

Мы советуем писать комментарии, помогающие читателю быстрее понять код. При необходимости вы можете прокомментировать объект, причину или способ (или все вместе).

Преодоление «творческого кризиса»

Множество программистов не любят писать комментарии, поскольку считают, что на написание хорошего комментария следует потратить много времени и сил. Когда у писателей наступает подобный творческий кризис, лучшее решение — просто начать писать. Поэтому, когда вы в следующий раз будете сомневаться, писать комментарий или нет, просто приступите к работе и просто напишите в комментарии все, о чем вы думаете, какими бы сырыми ни были ваши мысли.

Например, представьте, что вы работаете над функцией и думаете про себя: «Вот черт, эта штука становится слишком сложной, как только в списке появляются совпадающие значения». Просто запишите это:

```
// Вот черт, эта штука становится слишком сложной,
// как только в списке появляются совпадающие значения.
```

Это было несложно, правда? На самом деле это уже довольно неплохой комментарий — в любом случае гораздо лучше, чем ничего. Хотя язык немного простоват. Чтобы исправить это, заменим каждую фразу на более специализированную.

- «Вот черт» заменим на «На это следует обратить внимание».
- «Эта штука» — на «код, обрабатывающий эту входящую информацию».
- «Слишком сложной» — на «сложно реализовать».

У нас получился новый комментарий:

```
// На это следует обратить внимание: код, обрабатывающий
// эту входящую информацию, сложно реализовать,
// когда в списке появляются совпадающие значения.
```

Обратите внимание: мы разбили процесс написания комментариев на три простых шага.

1. Записали основную мысль комментария.
2. Прочитали комментарий и нашли места, которые следует улучшить.
3. Улучшили комментарий.

Чем больше вы комментируете, тем более качественные комментарии будут получаться у вас на этапе 1. В конечном итоге вам даже не понадобится вносить исправления. Заметьте, что, если вы будете комментировать часто и вовремя, вы сможете избежать необходимости создания нескольких комментариев в конце.

Итог

Цель комментирования — помочь читателю разобраться в том, что имел в виду автор кода. Вся эта глава помогла вам осознать то, что вы, как создатель кода, обладаете уникальной информацией, которую следует записать.

Вы узнали, что *не* нужно:

- комментировать информацию, которую можно получить непосредственно из кода;
- делать «вспомогательные комментарии», с помощью которых пытаются скрыть плохой код (например, плохое имя функции); вместо этого лучше исправить код.

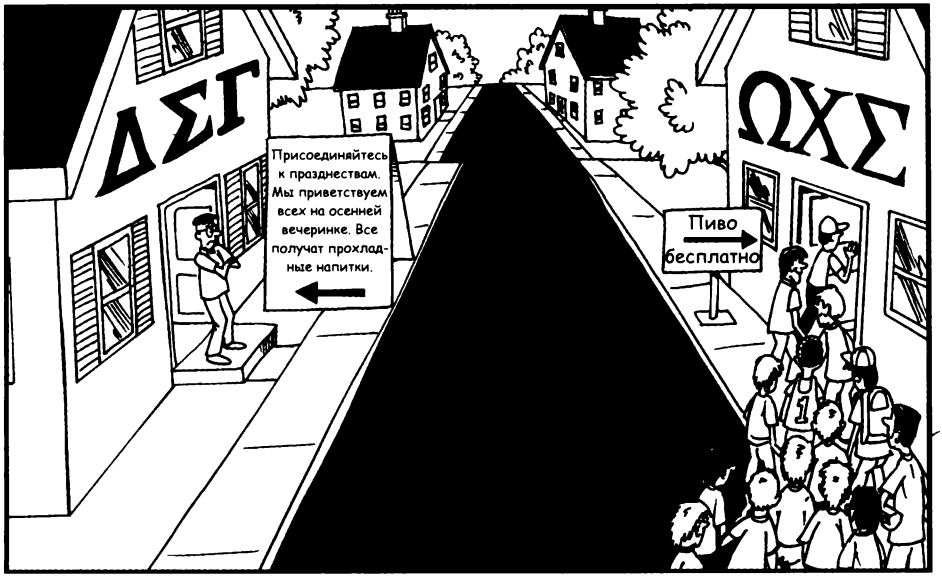
Мысли, которые следует закомментировать:

- объяснения того, почему код написан так, а не иначе («комментарии режиссера»);
- недостатки вашего кода (при этом можно также использовать обозначения, например `TODO:` или `XXX:`);
- назначение констант, а также причины выбора значения.

Поставьте себя на место читателя:

- определите, какие части вашего кода могут вызвать недоумение у читателей, и прокомментируйте их;
- документируйте любое неожиданное поведение, которое может смутить среднестатистического читателя;
- используйте комментарии, проясняющие общую картину на уровне класса/файла, а также объясните, как именно эти фрагменты собираются воедино;
- подводите итог блоков кода комментариями, чтобы читатель не запутался, разбираясь в деталях.

6 Комментарии должны быть четкими и компактными



В предыдущей главе мы разобрались, *что* именно нужно комментировать. Эта глава поможет понять, *как* делать комментарии четкими и компактными.

Комментарии, которые вы пишете, должны быть *четкими* — конкретными и настолько подробными, насколько это возможно. С другой стороны, комментарий, который занимает слишком много места на экране, приходится читать намного дольше. Поэтому комментарии также должны быть *компактными*.

ОСНОВНАЯ ИДЕЯ

Соотношение информативности комментария к его объему должно быть максимальным.

Далее в этой главе описаны способы достижения этой цели.

Старайтесь комментировать компактно

Рассмотрим в качестве примера комментарий к определению одного из типов языка C++:

```
// Типом int обладает параметр CategoryType.
// Первый параметр с типом float во внутренней паре называется 'score',
// второй - 'weight'.
typedef hash_map<int, pair<float, float> > ScoreMap;
```

Но зачем писать три строки комментариев, когда можно проиллюстрировать пример одной строкой:

```
// CategoryType -> (score,weight)
typedef hash_map<int, pair<float, float> > ScoreMap;
```

Некоторые комментарии вполне имеют право достигать трех строк, но приведенный выше явно к таким не относится.

Избегайте двусмысленных местоимений и указательных слов

Местоимения могут запутать кого угодно. Читателю требуется приложить дополнительные усилия, чтобы понять, какое именно слово заменяет местоимение. В некоторых случаях не совсем понятно, к чему именно относятся слова «он» или «это». Рассмотрим следующий пример:

```
// Добавим фрагмент данных в кэш, но сначала проверим,
// не слишком ли он велик.
```

В этом комментарии слово «он» может относиться либо к фрагменту данных, либо к кэшу. Конечно, можно разобраться в этом вопросе, прочитав остальной код. Но если приходится вчитываться в код, то зачем же тогда нужен комментарий?

В таких ситуациях наилучшим решением является «наполнение» местоимений. Например, в предыдущем примере можно заменить слово «он» на фразу «этот фрагмент»:

```
// Добавим фрагмент данных в кэш, но сначала проверим,
// не слишком ли велик этот фрагмент.
```

Это простейшее изменение, которое можно внести. Можно также изменить структуру предложения, чтобы окончательно исключить из него двусмысленность:

```
// Если фрагмент данных достаточно мал, вставляем его в кэш.
```

«Полируем» нечеткие предложения

Довольно часто повышение четкости комментария приводит к тому, что он становится и более компактным.

Рассмотрим следующий пример из поискового робота:

```
# В зависимости от того, находили ли мы этот URL ранее,
# задаем ему тот или иной приоритет.
```

Это предложение может показаться вполне нормальным, но сравните его со следующим:

```
# Устанавливаем более высокий приоритет тем URL,
# которые мы еще никогда не находили.
```

Это предложение более лаконично. В нем также поясняется, что *более высокий* приоритет получают новые URL, — предыдущий комментарий не содержит подобной информации.

Четко описываем поведение функции

Представьте, что вы написали функцию, которая считает количество строк в файле:

```
// Возвращает количество строк в заданном файле.
int CountLines(string filename) { ... }
```

Этот комментарий не совсем точный — есть множество различных интерпретаций понятия «строка». Следует рассмотреть множество спорных ситуаций, например следующие:

- `""` (пустой файл) — 0 или 1 строка?
- `"привет"` — 0 или 1 строка?
- `"привет\n"` — 1 или 2 строки?
- `"привет\n мир"` — 1 или 2 строки?
- `"привет\n\r жестокий\n мир\r"` — 2, 3 или 4 строки?

Простейшая реализация функции, определяющей количество строк, — подсчет количества символов `\n` (символов новой строки). (Подобным образом работает команда `wc`, применяемая в UNIX.) В таком случае более подходящим комментарием будет следующий:

```
// Подсчитываем количество символов ('\n') в файле.
int CountLines(string filename) { ... }
```

Этот комментарий не намного длиннее первоначальной версии, но содержит гораздо больше информации. Он сообщает читателю, что функция вернет 0, если в файле не будет символов новой строки. Он также несет информацию о том, что символы перевода каретки (\r) игнорируются.

Используйте примеры ввода/вывода, иллюстрирующие спорные ситуации

Когда дело доходит до комментариев, хорошо подобранный пример работы функции может стоить тысячи слов.

Например, рассмотрим функцию, удаляющую части строки:

```
// Удаляем суффиксы и префиксы, состоящие из 'chars',
// из входной строки 'src'.
String Strip(String src, String chars) {     }
```

Этот комментарий не совсем понятен, поскольку он не отвечает на следующие вопросы:

- chars представляет собой подстроку, которую нам необходимо удалить, или же неупорядоченный набор букв?
- Что произойдет, если в конце строки src встретится несколько наборов chars?

Хорошо подобранный пример позволяет ответить на эти вопросы:

```
// ...
// Пример: Strip("abba/a/ba", "ab") возвращает "/a/"
String Strip(String src, String chars) {     }
```

В этом примере предельно ясно демонстрируется вся функциональность метода Strip(). Обратите внимание на то, что более простой пример может не быть столь полезным, если он не отвечает на поставленные вопросы:

```
// Пример: Strip("ab", "a") возвращает "b"
```

Вот еще один пример функции, которая объясняется при помощи иллюстрации:

```
// Переставим элементы 'v' так, чтобы те из них,
// которые < pivot, располагались перед теми, которые >= pivot.
// Затем возвратим наибольшее значение 'i',
// для которого v[i] < pivot (или -1, если таких элементов нет).
int Partition(vector<int>* v, int pivot);
```

Этот комментарий очень четкий, однако не так просто представить работу этой функции на примере. Следующий комментарий лишен указанного недостатка:

```
// ...
// Пример: Partition([8 5 9 8 2], 8) вернет 1
// и расставит элементы следующим образом: [5 2 | 8 9 8].
int Partition(vector<int>* v, int pivot);
```

Стоит подчеркнуть несколько особенностей приведенного примера ввода/вывода:

- параметр `pivot` равен некоторым элементам вектора; это сделано, чтобы проиллюстрировать пограничный случай;
- мы добавили в вектор повторяющиеся значения (`8`), чтобы показать допустимость такого факта;
- результирующий вектор не отсортирован; в противном случае читатель мог бы получить неправильное представление о ситуации;
- поскольку функция в данном случае возвращает значение `1`, мы позаботились о том, чтобы такого значения не оказалось в векторе, поскольку это может запутать читателя.

Описывайте цели вашего кода

Как мы упомянули в предыдущей главе, идея комментирования часто заключается в том, чтобы сообщить читателю ваши мысли, которые возникли при написании кода. К несчастью, многие комментарии состоят лишь из словесного описания того, что делает код, и содержат не так много новой информации.

Приведем пример подобного комментария:

```
void DisplayProducts(list<Product> products) {
   products.sort(CompareProductByPrice);

   // Проходим от конца списка к началу.
   for (list<Product>::reverse_iterator it = products.rbegin();
        it != products.rend(); ++it)
      DisplayPrice(it->price);


}
```

Этот комментарий описывает лишь строку под ним. Для сравнения рассмотрим более информативный комментарий:

```
// Отображаем все цены, от наибольшей до наименьшей.
   for (list<Product>::reverse_iterator it = products.rbegin(); ... )
```

Этот комментарий объясняет принцип работы программы на более высоком уровне. Он гораздо точнее соответствует мыслям программиста, которые могли возникнуть у него, когда он писал код.

Интересно отметить, что в этой программе есть ошибка! Функция `CompareProductByPrice` (не показана) заранее сортирует товары с высокой стоимостью. Код выполняет задачу, совершенно противоположную той, которую хотел решить автор.

Есть достаточные основания считать, что второй комментарий лучше. Несмотря на ошибку, первый комментарий написан технически правильно (цикл проходит по списку в обратном порядке). Однако более вероятно, что второй комментарий

донесет до читателя информацию о том, что намерения автора (сначала показать самые дорогие товары) не соответствуют тому, что на самом деле делает код. Этот комментарий служит для *проверки на избыточность (redundancy check)*.

В конечном итоге наилучшей проверкой на избыточность является тестирование компонентов (см. гл. 14). Но в любом случае стоит оставлять подобные комментарии, чтобы объяснить читателю свои намерения.

Комментарии, содержащие названия параметров функций

Рассмотрим следующий вызов функции:

```
Connect(10, false);
```

Этот вызов функции довольно странный, поскольку непонятно, что именно означают передаваемые параметры.

В языках наподобие Python аргументы можно присваивать в алфавитном порядке:

```
def Connect(timeout, use_encryption): ...

# Вызываем функцию, используя названные параметры.
Connect(timeout = 10, use_encryption = False)
```

В языках наподобие C++ и Java этого сделать нельзя. Однако с тем же успехом можно использовать комментарии внутри строки:

```
void Connect(int timeout, bool use_encryption) { ... }

// Вызываем функцию с параметрами, названными в комментариях.
Connect(/* timeout_ms = */ 10, /* use_encryption = */ false);
```

Обратите внимание: мы назвали первый параметр `timeout_ms` вместо `timeout`. В идеале аргумент, передаваемый в функцию, также должен иметь имя `timeout_ms`, но здесь мы не можем внести такое изменение. Поэтому подобные комментарии — удобный способ «улучшить» название.

При работе с булевыми аргументами особенно важно ставить комментарий вида `/* имя = */` *перед* значением. Если разместить его *после* значения, это может запутать многих читателей:

```
// Не делайте так!
   Connect(     , false /* use_encryption */);

// Так тоже!
   Connect( ... , false /* = use_encryption */);
```

Из этих примеров непонятно, что именно означает `false`: «использовать шифрование» или «не использовать шифрование».

Большинство функций не нуждается в таких комментариях, но это довольно удобный (и компактный) способ объяснить замысловатые аргументы.

Употребляйте максимально содержательные слова

Если вы программируете уже несколько лет, то вы, возможно, замечали, что периодически возникают практически аналогичные общие проблемы и решения. Часто существуют специфические слова или фразы, которые были созданы только для того, чтобы описать эти паттерны/идиомы. При использовании таких слов комментарии получаются гораздо более компактными.

Например, представьте, что комментарий выглядит так:

```
// Этот класс содержит ряд членов, включающих ту же информацию,
// что и база данных, они сохраняются здесь для ускорения работы.
// Когда этот класс считывается позже, проверяется наличие этих членов.
// Если они существуют, то функция их возвращает. В противном случае
// информация считывается из базы данных и заносится в эти поля на будущее.
```

Вместо этого вы можете просто сказать:

```
// Этот класс ведет себя как кэширующий слой базы данных.
```

В качестве другого примера рассмотрим еще один комментарий:

```
// Удаляем лишние пробелы из адресов улиц, а также внедряем множество
// сокращений, например "Avenue" заменяем на "Ave". Таким образом,
// если существуют два разных адреса улиц, которые печатаются примерно
// одинаково, их сокращенные версии будут совпадать и мы сможем
// определить, что они одинаковы.
```

Данный комментарий может выглядеть и так:

```
// Приводим к канонической форме адреса улиц (удаляем лишние пробелы,
// "Avenue" -> "Ave." и т. д.)
```

Существует множество слов и фраз, содержащих большой объем информации, например «эвристический», «полный перебор», «упрощенное решение» и т. д. Если у вас есть комментарий, который кажется вам чересчур длинным, попробуйте описать его в контексте обычной для программиста ситуации.

Итог

В этой главе содержится информация о том, как писать информативные, но компактные комментарии. Можно выделить несколько советов:

- избегайте местоимений или указательных слов вроде «он» или «это», когда они могут обозначать несколько объектов;
- описывайте поведение функции четко и с практической точки зрения;
- иллюстрируйте комментарии хорошо подобранными примерами;
- описывайте свои намерения, а не очевидные детали;
- используйте внутристрочные комментарии (например, `Function(/* arg = */ ... )`), чтобы объяснить значение таинственных аргументов функции;
- делайте ваши комментарии более компактными, используя слова, содержащие большой объем информации.

ЧАСТЬ II

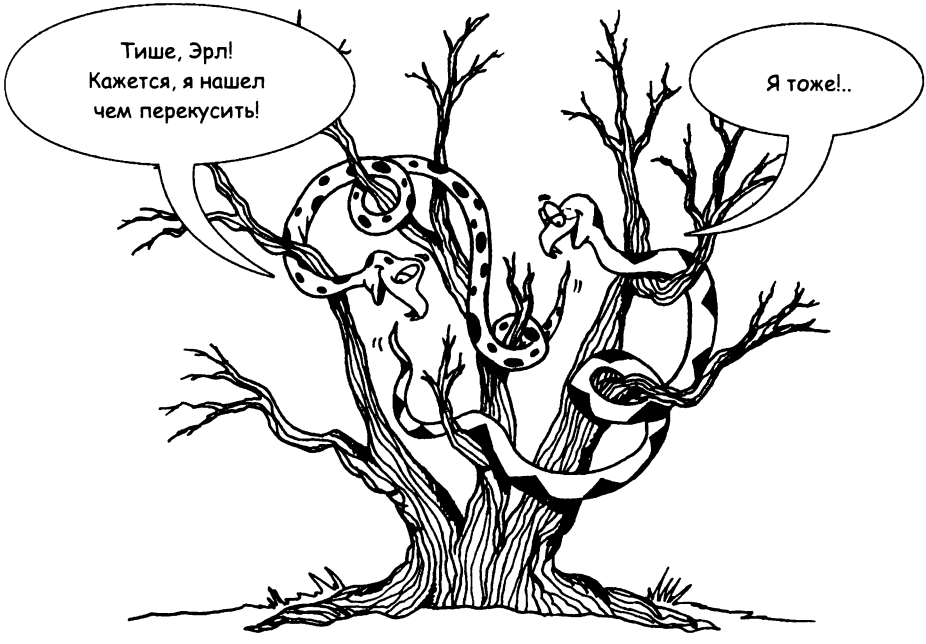
Упрощение цикла и логики

В первой части книги мы рассмотрели поверхностные улучшения — простые способы облегчить читаемость кода, которые несложно применить на практике, при этом ничем не рискуя и не затрачивая много усилий.

В этой части мы копнем глубже и обсудим циклы и логику вашей программы: порядок выполнения, логические выражения и переменные, благодаря которым код работает. Как и ранее, наша цель — упростить части кода, сделав их более понятными.

Чтобы достичь этой цели, мы попытаемся минимизировать «ментальный багаж» вашего кода. Каждый раз, когда вы видите запутанный цикл, огромное выражение или большое количество переменных, к вашим знаниям добавляется ментальный багаж. Это требует от вас тратить больше времени на раздумья, а также больше запоминать. Данное понятие противоположно «простому пониманию». Если код имеет большое количество ментального багажа, то с этим кодом не так приятно работать, его сложнее изменить и масса ошибок могут остаться незамеченными.

7 Как сделать поток команд управления удобочитаемым



Если в коде нет условных конструкций, циклов или других элементов, управляющих порядком выполнения программы, то читать его будет очень легко. А вот переходы и ветви сложны для понимания и легко могут сделать код непонятным. Из этой главы вы узнаете, как упростить для понимания порядок выполнения программы (данный порядок называется *потоком управления*, англ. control flow).

ОСНОВНАЯ ИДЕЯ

Старайтесь приводить все условные конструкции, циклы и другие изменения потока управления к максимально «естественному» виду — они должны быть написаны так, чтобы читатель не останавливался и не перечитывал код заново.

Порядок аргументов в условных конструкциях

Какой из этих двух фрагментов кода проще читать? Первый:

```
if (length >= 10)
```

или второй:

```
if (10 <= length)
```

Для большинства программистов первая строка будет гораздо более понятной. А как насчет следующих двух:

```
while (bytes_received < bytes_expected)
```

и

```
while (bytes_expected > bytes_received)
```

Опять же первая версия читается лучше. Но почему? По какому правилу это определяется? Как решить, какой фрагмент лучше: `a < b` или `b > a`?

Рекомендуем в таких случаях придерживаться следующего принципа.

Левостороннее выражение	Правостороннее выражение
Выражение, с которым вы сравниваете свое, меняется чаще	Выражение, с которым вы сравниваете свое, меняется реже

Этот принцип соответствует принципу построения предложений в нашей обычной речи. Фразы «Если вы зарабатываете в год как минимум $100 000» или «Если вам есть 18 лет» звучат довольно естественно. А выражение «Если 18 лет — это ваш возраст или меньше» звучит неестественно.

Это правило объясняет, почему строка `while (bytes_received < bytes_expected)` более удобочитаема. Мы проверяем значение переменной `bytes_received`, которое увеличивается в каждой итерации цикла. Значение переменной `bytes_expected`, с которым мы сравниваем предыдущую переменную, более «стабильно».

«НОТАЦИЯ МАСТЕРА ЙОДЫ» ПО-ПРЕЖНЕМУ ИСПОЛЬЗУЕТСЯ?

В некоторых языках (таких как C++ и C#, но не Java) компилятор позволяет выполнить операцию присваивания внутри условной конструкции:

```
if (obj = NULL)
```

Скорее всего, такая строка содержит ошибку и программист на самом деле имел в виду:

```
if (obj == NULL) ...
```

Чтобы избежать подобных ошибок, многие программисты **меняют порядок следования аргументов**:

```
if (NULL == obj) ...
```

В таком случае, если вы случайно напишете «=» вместо «==», выражение `if (NULL == obj)` даже не скомпилируется.

Тем не менее изменение порядка следования аргументов делает ваш код неестественным. (Как сказал бы Мастер Йода: «Понять ничего в этом не могу я»[1].) К счастью, современные компиляторы предупреждают программиста о том, что использована конструкция вида `if (obj = NULL)`, поэтому «нотация Мастера Йоды» становится достоянием прошлого.

Порядок блоков if/else

«Дети, сначала послушаем Бобби.
Он расскажет нам о своей ручной лягушке!»

Когда вы пишете условную конструкцию, обычно вполне допустимо поменять местами порядок блоков. Например, вместо этого кода:

[1] Мастер Йода — гуманоид из сериала «Звездные войны». Известен тем, что строит предложения очень витиевато, подобно тому, как мы наблюдаем в этой нотации. — *Примеч. пер.*

```
if (a == b) {
   // Первый блок ...
} else {
   // Второй блок ...
}
```

можно написать так:

```
if (a != b) {
   // Второй блок ...
} else {
   // Первый блок ...
}
```

Возможно, вы об этом никогда ранее не задумывались, но в некоторых случаях целесообразно предпочесть один порядок следования блоков другому и сначала выполнить инструкции:

- *положительного* блока вместо отрицательного, например `if (debug)` вместо `if (!debug)`;
- более *простого* блока, чтобы далее уже не думать о нем. При таком подходе возможно одновременное появление на экране как слова `if`, так и слова `else`, что выглядит довольно красиво;
- более *интересного* или заметного блока.

Иногда правила, отмеченные выше, противоречат друг другу и приходится делать выбор. Но в большинстве случаев есть вариант, явно выигрывающий у остальных.

Например, представьте, что у вас есть веб-сервер, который создает `response` (ответ), основываясь на том, имеет ли URL параметр запроса `expand_all`:

```
if (!url.HasQueryParameter("expand_all")) {
    response.Render(items);
    ...
} else {
    for (int i = 0; i < items.size(); i++) {
        items[i].Expand();
    }
    ...
}
```

Когда читатель смотрит на первую строку, ему сразу вспоминается ситуация, когда параметр `expand_all` присутствует в запросе. Очень похоже на случай, когда кто-то говорит: «Не думай о розовом слоне». И вы совершенно непроизвольно начинаете о нем думать: на частицу «не» зачастую не обращают внимания, поскольку замечают лишь необычного «розового слона».

В данном случае нашим «розовым слоном» является параметр `expand_all`. Поскольку ситуация, когда этот параметр присутствует в запросе, более интересна (она также является положительной), разберемся сначала с ней:

```
if (url.HasQueryParameter("expand_all")) {
    for (int i = 0; i < items.size(); i++) {
        items[i].Expand();
```

```
    }
    ...
} else {
    response.Render(items);

}
```

С другой стороны, можно привести пример, когда отрицательная ситуация более проста и более интересна/опасна, поэтому сначала следует обработать именно ее:

```
if not file:
    # Регистрация ошибки ...
else:
    # ...
```

Опять же следует делать выбор в зависимости от деталей — он может быть субъективным.

Резюмируя, мы советуем вам обращать внимание на эти факторы и следить за тем, чтобы блоки ваших условных конструкций `if/else` не следовали в странном порядке.

Условная конструкция ?: (также известная как тернарный оператор)

В языках, подобных C, можно использовать условную конструкцию вроде `cond ? a : b`, которая является гораздо более компактной версией стандартной конструкции `if (cond) { a } else { b }`.

Однако такая конструкция очень негативно влияет на читаемость кода. Сторонники такой записи считают, что гораздо удобнее написать что-либо в одну строку, чем использовать несколько. Противники полагают, что ее неудобно читать, а также сложно обработать отладчиком.

В следующем примере тернарный оператор улучшает читаемость и компактность кода:

```
time_str += (hour >= 12) ? "pm" : "am";
```

Без использования этого оператора потребуется написать:

```
if (hour >= 12) {
    time_str += "pm";
} else {
    time_str += "am";
}
```

Такая запись является немного неясной и избыточной. В этом случае условное выражение — именно то, что нужно.

Тем не менее подобные выражения быстро могут значительно усложнить чтение:

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

Этот случай — уже не только выбор между двумя простыми значениями. Обычно написание подобного кода мотивируется желанием «втиснуть все в одну строку».

ОСНОВНАЯ ИДЕЯ

Вместо того чтобы уменьшать количество строк, сократите время, требуемое для понимания кода.

Если акцентировать логику программы при помощи условных конструкций, можно сделать код более естественным:

```
if (exponent >= 0) {
   return mantissa * (1 << exponent);
} else {
   return mantissa / (1 << exponent);
}
```

СОВЕТ

По умолчанию используйте конструкцию if/else. Тернарный оператор ?: следует применять только в простейших случаях.

Избегайте циклов do/while

Во многих популярных языках программирования, например в Perl, есть цикл вида `do { выражение } while (условие)`. При этом выражение выполняется как минимум один раз. Приведем пример такого цикла:

```
// Ищем в списке заданное имя 'name', начиная с узла node.
// Не обрабатываем более 'max_length' узлов.
public Boolean ListHasNode(Node node, String name, int max_length) {
    do {
```

```
        if (node.name().equals(name))
            return true;
        node = node.next();
    } while (node != null && --max.length  > 0);

    return false;
}
```

В этом фрагменте кода есть довольно странный момент, связанный с циклом do/while, — выражение, заключенное в этом блоке кода, может быть выполнено или не выполнено в зависимости от условия, которое располагается *под* ним. Обычно логические выражения размещаются над управляемым ими кодом — таким способом оформляются конструкции с ключевыми словами if, while и for. Поскольку обычно вы читаете код сверху вниз, конструкция do/while выглядит слегка неестественно. Многие читатели будут вынуждены перечитывать код дважды.

Циклы с использованием while более просты для чтения, поскольку условие будет известно до того, как вы прочтете блок кода внутри. Но было бы довольно глупо дублировать код только для того, чтобы убрать конструкцию do/while:

```
// Имитируем конструкцию do/while – НЕ ДЕЛАЙТЕ ТАК!
body // код внутри цикла

while (condition) {
    body (again) // Дублируем код, написанный выше.
}
```

К счастью, мы обнаружили, что на практике циклы с конструкциями while и do/while пишутся одинаково и начинаются так:

```
public boolean ListHasNode(Node node, String name, int max_length) {
    while (node != null && max_length-- > 0) {
        if (node.name().equals(name)) return true;
        node = node.next();
    }
    return false;
}
```

Этот вариант выгодно отличается от предыдущего тем, что он будет работать в тех случаях, когда параметр max_length равен 0 или параметр node равен null.

Другая причина, по которой следует избегать конструкций вида do/while, заключается в том, что ключевое слово continue выглядит внутри его довольно неуместно. Например, что делает следующий фрагмент кода:

```
do {
    continue;
} while (false);
```

Этот цикл вечный или инструкции выполняются всего один раз? Большинству программистов придется остановиться и задуматься об этом. (Инструкция выполнится всего один раз.)

В общем, Бьерн Страуструп, создатель языка C++, в книге «Язык программирования C++» (The C++ Programming Language) пишет:

По моему опыту, выражение do — источник ошибок и замешательств... Я предпочитаю выражения, расположенные сверху, так мне удобнее их видеть. Следовательно, я стараюсь избегать конструкций с ключевым словом do.

Слишком быстрый возврат из функции

Некоторые программисты считают, что в функциях не следует использовать несколько утверждений с ключевым словом `return`, поскольку это не имеет смысла. Быстрый возврат из функций довольно удобен — и многие программисты довольно часто пользуются этим. Например:

```
public boolean Contains(String str, String substr) {
    if (str == null || substr == null) return false;
    if (substr.equals("")) return true;

}
```

Реализация этой функции без данных граничных операторов была бы очень неестественной.

Одна из причин использования единственной точки выхода — намерение гарантировать, что обязательно будет выполняться код очистки. Однако современные языки предлагают более тонкие способы достижения этой цели (табл. 7.1).

Таблица 7.1

Язык	Идиомы, используемые для выполнения кода очистки
C++	Деструкторы
Java, Python	Ключевые слова try finally
Python	with
C#	using

В чистом языке C нет механизма, вызывающего выполнение кода очистки. Поэтому, если вы пишете обширную функцию с большим количеством кода очистки, выйти заранее из этой функции будет довольно проблематично. В этом случае при проведении рефакторинга можно воспользоваться разными вариантами, в том числе и рационально применить конструкцию `goto cleanup;`.

Пресловутый goto

В языках, отличных от C, нет особой необходимости в операторе `goto`, поскольку они предлагают более удобные способы решения поставленной задачи. Оператор `goto` также печально известен тем, что быстро может выйти из-под контроля, а это затрудняет его отслеживание.

Но этот оператор тем не менее может применяться в различных проектах, использующих язык C, — особенно в ядре ОС Linux. Прежде чем клеймить любое использование оператора goto позором, давайте рассмотрим ситуации, в которых лучше всего подойдет именно он.

Простейший, наиболее безобидный вариант применения оператора goto заключается в переходе к метке exit, располагающейся в конце функции:

```
    if ( p == NULL) goto exit;



exit:
    fclose(file1);
    fclose(file2);


    return;
```

Если бы оператор goto можно было использовать только таким способом, проблем при работе с ним было бы гораздо меньше.

Сложности могут возникнуть в ситуации, когда задействуется *несколько* операторов goto, особенно если их «пути» пересекаются. В частности, переход *вверх* с использованием оператора goto может привести к образованию страшно запутанной программы (так называемого «спагетти-кода»). Такой код вполне можно заменить структурированными циклами. В большинстве случаев следует избегать использования оператора goto.

Сокращаем количество вложенного кода

Вложенный код довольно труден для понимания. Каждый уровень вложенности требует наличия дополнительного условия, что усложняет для читателей запоминание кода. Когда читатель видит закрывающую скобку }, ему бывает довольно трудно «вытолкнуть» из стека условие, которое прекращает действовать.

Далее приведен относительно простой пример такой вложенности. Вы заметите, что вам придется возвращаться назад, чтобы перепроверить условия, которые действуют в данном блоке кода:

```
if (user_result == SUCCESS) {
   if (permission_result != SUCCESS) {
       reply.WriteErrors("error reading permissions");
       reply.Done();
       return;
   }
   reply.WriteErrors("");
} else {
   reply.WriteErrors(user_result);

   }
reply.Done();
```

Когда вы видите первую закрывающую скобку, вы можете подумать: «О, блок с условием permission_result != SUCCESS только что закончился, теперь действует условие permission_result == SUCCESS и мы все еще внутри блока с условием user_result == SUCCESS».

В итоге вам все время придется держать в голове значения параметров permission_result и user_result. И как только закончится каждый блок if {}, вам потребуется вспомнить соответствующее значение.

Приведенный выше пример еще больше усугубляет ситуацию, поскольку при чтении вам необходимо будет «переключаться» между успешными (SUCCESS) и неуспешными ситуациями.

Как накапливается вложенный код

Перед тем как мы попытаемся исправить код из предыдущего примера, поговорим о том, к чему может привести множественная вложенность кода. Сначала код был простым:

```
if (user_result == SUCCESS) {
    reply.WriteErrors("");
} else {
    reply.WriteErrors(user_result);
}
reply.Done();
```

Этот код прост для понимания. Он определяет, какую строку, содержащую ошибку, следует записать, а затем выполняет эту операцию при помощи reply.

Но затем программист добавил вторую операцию:

```
if (user_result == SUCCESS) {
    if (permission_result != SUCCESS) {
       reply.WriteErrors("error reading permissions");
       reply.Done();
       return;
    }
    reply.WriteErrors("");
```

Это изменение целесообразно — программист добавил новый фрагмент кода, и он нашел идеальное место для него. Этот новый код был свеж и «четко артикулирован» у него в голове. Кроме того, данный код кажется ему довольно простым изменением.

Однако если позже этот код увидит кто-нибудь еще, у него не будет перед глазами контекста и он окажется в той же ситуации, в которой оказались вы, читая код в начале этого раздела, — вам пришлось разбираться в нем с нуля.

ОСНОВНАЯ ИДЕЯ

Посмотрите на код свежим взглядом, когда вносите в него изменения. Прервите работу и изучите код как единое целое.

Избавляемся от вложенного кода при помощи return

Теперь улучшим рассмотренный ранее код. Подобной вложенности кода можно избежать при помощи оперативной обработки случаев отказа и возврата из функции:

```
if (user_result != SUCCESS) {
    reply.WriteErrors(user_result);
    reply.Done();
    return;
}

if (permission_result != SUCCESS) {
    reply.WriteErrors(permission_result);
    reply.Done();
    return;
}

reply.WriteErrors("");
reply.Done();
```

Этот код имеет лишь один уровень вложенности вместо двух. Но что более важно — читателю не придется восстанавливать в уме воображаемый стек — каждое ключевое слово `if` имеет в паре слово `return`.

Избавляемся от вложенного кода в циклах

Однако подобный прием досрочного возврата из функции не всегда можно применить. Например, рассмотрим вложенный код в следующем цикле:

```
for (int i = 0; i < results.size(); i++) {
    if (results[i] != NULL) {
        non_null_count++;
        if (results[i]->name != "") {
            cout << "Considering candidate..." << endl;


        }
    }
}
```

Внутри цикла аналогом приема с использованием `return` является прием с применением ключевого слова `continue`:

```
for (int i = 0; i < results.size(); i++) {
    if (results[i] == NULL) continue;
    non_null_count++;

    if (results[i]->name == "") continue;
```

```
    cout << "Considering candidate..." << endl;

}
```

Точно как `if (...) return;` играет роль граничного оператора для функции, данные инструкции `if (...)continue;` являются граничными операторами для цикла.

В общем, инструкция `continue` может запутать читателей, поскольку им потребуется перескакивать от одного участка кода к другому (примерно такой же эффект оказало бы использование оператора `goto`). Но в этом случае каждая итерация цикла независима, поэтому читатель может воспринимать инструкцию `continue` как «пропуск нижеследующих элементов».

Можете ли вы отследить порядок выполнения вашей программы?

Игра «Три карты Монте-Карло»[1]

Эта глава посвящена низкоуровневому управлению порядком выполнения программы. Мы рассмотрели вопросы создания удобочитаемых циклов, условных конструкций и прочих переходов. Но следует также подумать о порядке выполнения программы на высоком уровне. В идеале вы должны иметь возможность отследить выполнение всех операций программы. Вы начинаете из функции `main()` и проходите при помощи воображения сквозь весь код. Вы представляете, как одна функция вызывает другую, пока программа не завершится.

---

[1] Аналогична игре в наперстки. — *Примеч. пер.*

Однако на практике языки программирования и библиотеки содержат конструкции, позволяющие коду выполняться «неявно» или же затрудняющие его отслеживание. Приведем несколько примеров (табл. 7.2).

Таблица 7.2

Программная конструкция	Как изменяется порядок выполнения программы на высоком уровне
Многопоточность	Непонятно, какая именно часть кода выполняется в данный момент
Обработчики сигналов/прерываний	Определенный код может выполняться в любой момент времени
Исключения	Исключение может вызываться при работе множества функций
Указатели на функции и анонимные функции	Сложно определить, какая именно часть кода выполняется в данный момент, поскольку это неизвестно во время компиляции
Виртуальные методы	Метод object.virtualMethod() может вызвать выполнение кода неизвестного подкласса

Некоторые из этих конструкций очень важны и даже помогают сделать код более полезным и менее избыточным. Но, как программисты, мы иногда увлекаемся и злоупотребляем этими особенностями, не понимая, что они могут значительно усложнить чтение и понимание кода в дальнейшем. Подобные конструкции также затрудняют обнаружение ошибок.

Возьмите за правило не увлекаться подобными конструкциями в коде. Если использовать их слишком часто, это может привести к тому, что отслеживание порядка выполнения кода превратится в игру в «Три карты Монте-Карло» (как показано в комиксе выше).

Итог

Есть несколько способов сделать поток выполнения операций программы более понятным.

Когда вы сравниваете два параметра (`while (bytes_expected > bytes_received)`), лучше поместить изменяющееся значение слева, а более постоянное — справа (`while (bytes_received < bytes_expected)`).

Можно также поменять местами блоки в условных конструкциях `if/else`. Старайтесь обрабатывать сначала положительный/более простой/интересный случай. Иногда эти критерии могут конфликтовать между собой, но когда этого не происходит, данные критерии следует взять на вооружение.

Использование некоторых программных конструкций, например тернарного оператора (`?:`), цикла `do/while` или оператора `goto`, может усложнить понимание кода. Лучше обойтись без них, поскольку практически всегда существуют более понятные альтернативы.

Применение вложенных блоков кода требует от читателя дополнительной концентрации. Каждый новый уровень вложенности добавляет в «воображаемый стек» человека, читающего ваш код, новый контекст. Следует создавать более линейный код для того, чтобы избежать чрезмерной вложенности.

Преждевременный возврат из функций поможет избавиться от вложенности и упростить чтение кода в целом. Особенно полезны «граничные операторы» (guard statements), обеспечивающие обработку простых ситуаций в начале тела функции.

8 Разбиваем длинные выражения

Гигантская креветка — удивительное и красивое животное, но ее практически идеальное тело имеет один фатальный недостаток: ее мозг, имеющий форму пончика, располагается вокруг пищевода. Поэтому, если она проглотит слишком много пищи за один раз, у нее случится повреждение мозга.

Какое отношение это имеет к коду? С кодом, который пишется большими фрагментами, может случиться нечто похожее. Недавние исследования показали, что большинство из нас может думать только о трех-четырех вещах одновременно[1]. Проще говоря, более длинное выражение, использованное в коде, понять гораздо сложнее.

ОСНОВНАЯ ИДЕЯ

Разбивайте длинные выражения на более удобоваримые аналоги.

В этой главе мы рассмотрим различные подходы, которые помогут вам разбить длинные фрагменты кода на более короткие, в результате чего их будет легче «проглотить».

Поясняющие переменные

Самый простой способ разбить длинное выражение — ввести новую переменную, которая будет содержать меньшее подвыражение. Эту переменную иногда называют поясняющей переменной, поскольку она помогает объяснить, что значит данное подвыражение.

Приведем пример:

```
if line.split(':')[0].strip() == "root":
```

Далее рассмотрим похожий код, содержащий поясняющую переменную:

```
username = line.split(':')[0].strip()
if username == "root":
```

Итоговые переменные

Даже если выражение не *нуждается* в объяснении (поскольку вы и так можете понять, что оно означает), все равно может быть полезно хранить результат вычисления этого выражения в новой переменной. Такая переменная называется *итоговой (summary variable)*, если ее цель заключается лишь в замене большого фрагмента кода небольшим, простым для запоминания именем, которым легче управлять.

Например, рассмотрим следующие выражения:

```
if (request.user.id == document.owner_id) {
    // пользователь может изменять этот документ...
```

[1] Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. Behavioral and Brain Sciences, 24, 97–185.

```
}


if (request.user.id != document.owner_id) {
    // документ доступен только для чтения...
}
```

Выражение `request.user.id == document.owner_id` не так уж и велико, но содержит пять переменных. Поэтому вам потребуется больше времени, чтобы его обдумать.

Основной критерий проверки в этом фрагменте кода — владеет ли пользователь документом? Читателю будет проще разобраться в этой концепции, если вы добавите итоговую переменную:

```
final boolean user_owns_document = (request.user.id == document.owner_id);

if (user_owns_document) {
    // пользователь может изменять этот документ...
}



if (!user_owns_document) {
    // пользователь может изменять этот документ...
}
```

Влияние этой переменной не так уж и велико, однако выражение `if (user_owns_document)` немного проще понять. Объявление переменной `user_owns_document` в верхней части кода также сообщает читателю о том, что «в следующей функции будет использоваться именно эта концепция».

Используем законы де Моргана

Если вы знакомы с электрическими цепями или логикой, то наверняка помните законы де Моргана. Они представляют собой два способа переписать булево выражение в эквивалентное:

- не (*a* или *b* или *c*) ⇔ (не *a*) и (не *b*) и (не *c*);
- не (*a* и *b* и *c*) ⇔ (не *a*) или (не *b*) или (не *c*).

Если вам сложно запомнить эти законы, запомните следующее простое правило, по которому они действуют: распространите отрицание и поменяйте местами и/или (выражаясь иначе, вынесите `not` (отрицание) за скобки).

Иногда можно использовать эти законы для того, чтобы сделать булево выражение более удобочитаемым. Например, данный фрагмент кода:

```
if (!(file_exists && !is_protected)) Error("Sorry, could not read file.");
```

можно заменить следующим:

```
if (!file_exists || is_protected) Error("Sorry, could not read file.");
```

Злоупотребление упрощенной логикой

В большинстве языков программирования булевы операторы выполняют упрощенные вычисления. Например, утверждение `if (a || b)` не вычисляет `b`, если значение `a` равно `true`. Такое поведение довольно удобно, но иногда им злоупотребляют, пытаясь создать более сложную логику.

Рассмотрим пример утверждения, однажды написанного одним из авторов:

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

Если перевести это выражение на русский язык, оно будет обозначать следующее: «Получить сегмент памяти для этого ключа. Если значение не равно нулю, убедитесь, что ключ не занят».

Даже несмотря на то, что это выражение занимает всего одну строку кода, оно заставляет многих программистов прервать чтение и задуматься. Теперь сравните это выражение со следующим фрагментом кода:

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

Он решает точно такую же задачу, и, несмотря на то что занимает две строки, его гораздо проще понять.

Но что послужило причиной создания такого большого выражения? Когда-то это казалось весьма разумным. Можно даже получить определенное удовольствие, сжимая всю логику в небольшой фрагмент кода. Такой подход можно понять — это похоже на решение миниатюрного пазла. Проблема заключается в том, что такой код сродни мысленному «тормозу» для любого человека, который будет читать код.

ОСНОВНАЯ ИДЕЯ

Остерегайтесь «заумных» фрагментов кода — они часто запутывают тех, кто будет читать код позднее.

Означает ли это, что следует избегать использования упрощенного поведения? Нет. Во многих случаях подобный подход довольно прозрачен, например:

```
if (object && object->method()) ...
```

Существует также новая идиома, которую стоит запомнить: в языках вроде Python, JavaScript и Ruby оператор `or` возвращает один из аргументов (он не преобразует их к типу `boolean`), поэтому конструкция типа:

```
x = a || b || c
```

возвращает первое «правдивое» (`true`) значение.

Пример: боремся со сложной логикой

Представьте, что вы реализуете следующий класс `Range`:

```
struct Range {
    int begin;
```

```
    int end;
    // например, [0,5) пересекается [3,8)
    bool OverlapsWith(Range other);
};
```

На рис. 8.1 показаны некоторые примеры числовых промежутков.

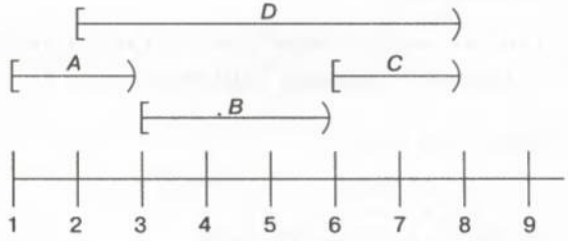


Рис. 8.1

Обратите внимание на то, что параметр end не является включающим. Поэтому промежутки *A*, *B* и *C* не перекрывают друг друга, а промежуток *D* перекрывает их все.

Далее показан пример реализации метода `OverlapsWith()` (пересекается с) — он проверяет, находятся ли конечные точки данного промежутка в промежутке other:

```
bool Range::OverlapsWith(Range other) {
    // Проверяем, входят ли параметры 'begin' и 'end' в промежуток 'other'.
    return (begin >= other.begin && begin <= other.end) ||
           (end >= other.begin && end <= other.end);
}
```

Несмотря на то что этот код длиной всего в две строки, он очень функционален. На рис. 8.2 показана вся его логика.

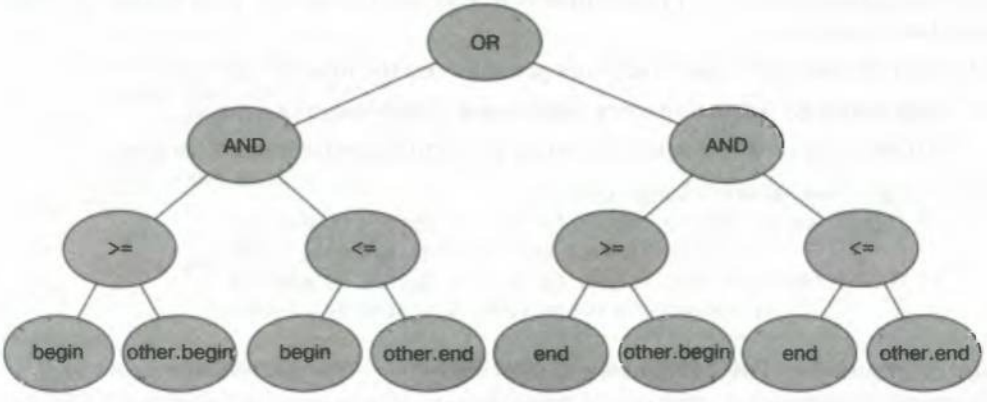


Рис. 8.2

В этом фрагменте кода использовано так много различных условий, что в него легко может закрасться ошибка.

Кстати, здесь действительно *есть* ошибка. Согласно предыдущему коду, промежуток [0,2) перекрывает промежуток [2,4), хотя в действительности это не так.

Проблема заключается в том, что следует внимательней следить за тем, какой оператор используется для сравнения — <= или просто <. Исправим эту проблему:

```
return (begin >= other.begin && begin < other.end) ||
       (end > other.begin && end <= other.end);
```

Теперь все верно, не так ли? На самом деле здесь есть еще одна ошибка. Этот код игнорирует случай, когда совпадают параметры begin и end двух промежутков.

Снова внесем исправления в код:

```
return (begin >= other.begin && begin < other.end) ||
       (end > other.begin && end <= other.end) ||
       (begin <= other.begin && end >= other.end);
```

Ух! Код стал слишком сложным! Теперь нельзя рассчитывать, что любой человек, прочитавший код, поймет все ваши замыслы. Что же делать? Как мы можем разбить это огромное выражение?

Ищем более элегантный подход. Этот случай — один из тех, когда следует остановиться и рассмотреть другие возможные варианты. Решение простой задачи (проверка на перекрытие двух числовых промежутков) обернулось созданием крайне запутанного логического выражения. Обычно это говорит о том, что **есть и более простое решение этой задачи**.

Однако поиск более элегантного решения потребует креативного подхода. Как вы с этим справитесь? Один из возможных приемов — решение задачи от противного. В зависимости от ситуации это может означать проход по массиву или заполнение структуры данных в обратном порядке.

В данном случае антиподом функции OverlapsWith() является функция с именем вроде «не перекрывается». Определение того, что два числовых промежутка не перекрываются, является более простой задачей, поскольку есть только два возможных варианта:

- один промежуток заканчивается раньше, чем начинается другой;
- один промежуток начинается позже, чем заканчивается второй.

Мы без труда можем написать код для реализации подобной логики:

```
bool Range::OverlapsWith(Range other) {
    if (other.end <= begin) return false;  // Другой промежуток
                  // заканчивается до того, как начинается наш.
    if (other.begin >= end) return false;  // Другой промежуток
                 // начинается после того, как закончится наш.

    return true;  // Остается только один вариант - они пересекаются.
}
```

Каждая строка этого фрагмента кода гораздо проще, поскольку содержит лишь по одному сравнению. Благодаря этому читателю несложно проверить, верно ли использован оператор <=.

Разбиваем огромные утверждения

В этом разделе рассматривается вопрос разбиения отдельных выражений, но некоторые приемы могут использоваться и для разделения более объемных конструкций. Например, следующий код, написанный на языке JavaScript, довольно сложно понять с ходу:

```
var update_highlight = function (message_num) {
    if ($("#vote_value" + message_num).html() === "Up") {
        $("#thumbs_up" + message_num).addClass("highlighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    } else if ($("#vote_value" + message_num).html() === "Down") {
        $("#thumbs_up" + message_num).removeClass("highlighted");
        $("#thumbs_down" + message_num).addClass("highlighted");
    } else {
        $("#thumbs_up" + message_num).removeClass("highighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    }
};
```

Отдельные выражения в этом фрагменте кода не так уж и велики, но если записать их вместе, то получится одно гигантское, сложное для понимания выражение.

К счастью, большинство выражений похожи друг на друга. Это означает, что мы можем создать для них итоговые переменные и поместить их в верхней части функции (что также поможет проиллюстрировать принцип разработки ПО «Не повторяйся»):

```
var update_highlight = function (message_num) {
    var thumbs_up = $("#thumbs_up" + message_num);
    var thumbs_down = $("#thumbs_down" + message_num);
    var vote_value = $("#vote_value" + message_num).html();
    var hi = "highlighted";

    if (vote_value === "Up") {
        thumbs_up.addClass(hi);
        thumbs_down.removeClass(hi);
    } else if (vote_value === "Down") {
        thumbs_up.removeClass(hi);
        thumbs_down.addClass(hi);
    } else {
        thumbs_up.removeClass(hi);
        thumbs_down.removeClass(hi);
    }
};
```

Создание переменной `var hi = "highlighted"` не является необходимым, но поскольку у нас было шесть ее копий, то, создав ее, мы получили следующие преимущества:

- она помогает избежать опечаток. (На самом деле вы заметили, что в пятом примере была опечатка — `"highighted"`?);

- она значительно сокращает длину строки, делая код более удобным для просмотра;
- если потребуется изменить имя класса, исправления нужно будет внести только в одно место.

Еще один творческий способ упрощения выражений

Рассмотрим еще один пример кода, в котором множество действий выполняется в каждом выражении. На этот раз код написан на языке C++:

```
void AddStats(const Stats& add_from, Stats* add_to) {
  add_to->set_total_memory(add_from.total_memory() + add_to->total_memory());
  add_to->set_free_memory(add_from.free_memory() + add_to->free_memory());
  add_to->set_swap_memory(add_from.swap_memory() + add_to->swap_memory());
  add_to->set_status_string(add_from.status_string() + add_to->status_string());
  add_to->set_num_processes(add_from.num_processes() + add_to->num_processes());
  ...
}
```

Мы снова столкнулись с длинным кодом, выражения которого похожи друг на друга, но все-таки имеют определенные различия. Внимательно посмотрев на него пару минут, вы можете понять, что каждая строка выполняет одинаковую операцию, но всякий раз делает это с разными полями:

```
add_to->set_XXX(add_from.XXX() + add_to->XXX());
```

В C++ для реализации такого поведения можно использовать макрос:

```
void AddStats(const Stats& add_from, Stats* add_to) {
  #define ADD_FIELD(field) add_to->set_##field
                           (add_from.field() + add_to->field())

  ADD_FIELD(total_memory);
  ADD_FIELD(free_memory);
  ADD_FIELD(swap_memory);
  ADD_FIELD(status_string);
  ADD_FIELD(num_processes);
  ...
  #undef ADD_FIELD
}
```

Теперь, когда мы навели здесь порядок, можно взглянуть на код и мгновенно понять суть всего происходящего. Также стало очевидно, что каждая из строк осуществляет аналогичные операции.

Мы не призываем к частому использованию макросов — на самом деле обычно мы стараемся обходиться без них, поскольку они могут усложнить восприятие кода, а также привнести в него ошибки. Но иногда, как в этом случае, простые макросы могут значительно улучшить читаемость кода.

Итог

Огромные выражения довольно сложно воспринимать. В этой главе были показаны способы разбиения таких выражений на более мелкие, чтобы читатель мог разобраться в них поэтапно.

Можно использовать простой прием, суть которого заключается в применении поясняющих переменных, которые хранят значения большого подвыражения. Этот подход имеет следующие преимущества:

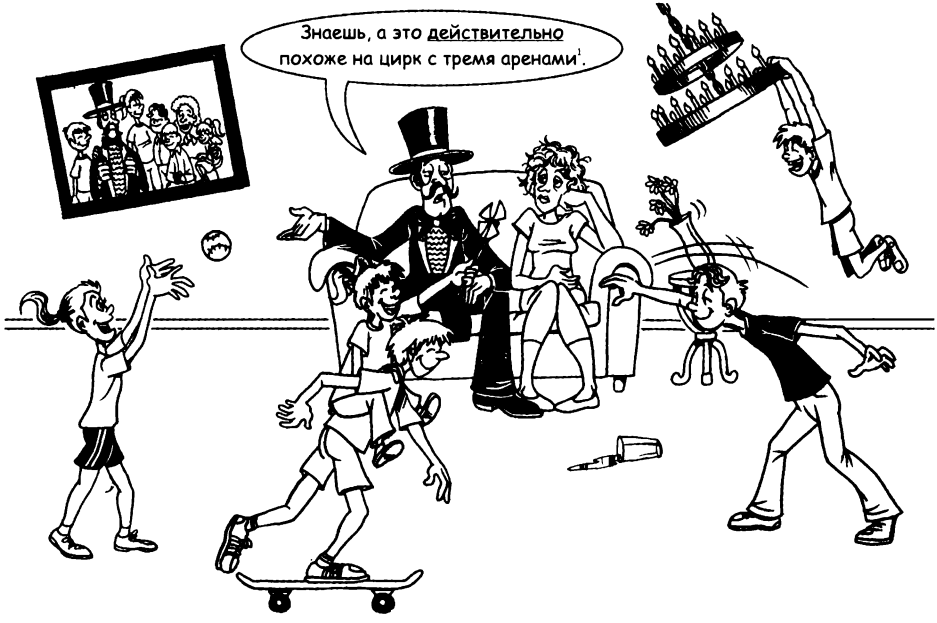
- крупные выражения разбиваются на части;
- код становится более «документированным», поскольку имя выражения исчерпывающе описывает его значение;
- читатель может определить основные концепции вашего кода.

Другим приемом является применение законов Де Моргана, что иногда помогает переписать булево выражение в более прозрачной форме (например, выражение `if (!(a && !b))` превращается в `if(!a || b)`).

Мы также продемонстрировали пример разбиения сложного логического условия на несколько маленьких утверждений вида `if (a < b) ...`. Фактически во *всех* исправленных примерах этой главы в конструкции `if` содержится не более *двух* условий. Такой подход идеален. Конечно, не всегда удается ограничиться двумя условиями — иногда приходится действовать от противного, взглянув на задачу с совершенно противоположной точки зрения.

Наконец, несмотря на то, что эта глава посвящена разбиению отдельных выражений, те же самые приемы могут пригодиться и при разбиении больших блоков кода. Поэтому будьте настойчивы и разбивайте сложные логические выражения всякий раз, когда они вам встречаются.

9 Переменные и читаемость



Дом конферансье цирка

[1] Игра слов: в английском языке выражение «цирк с тремя аренами» также означает «шум, гам, неразбериха». — *Примеч. пер.*

В этой главе мы покажем, как небрежное использование *переменных* может сделать программу трудной для понимания.

Можно выделить три основные проблемы:

- чем больше переменных, тем сложнее отслеживать их все;
- чем больше область видимости переменной, тем дольше ее придется отслеживать;
- чем чаще изменяется значение переменной, тем сложнее отследить ее текущее значение.

В следующих трех разделах мы рассмотрим, как решить подобные проблемы.

Избавляемся от переменных

В гл. 8 мы показали, как использование поясняющих или итоговых переменных позволяет сделать код более читаемым. Эти переменные были полезны, поскольку они разбивали длинные выражения и действовали как своеобразная документация.

В этом разделе мы заинтересованы в том, чтобы избавиться от переменных, которые *не* повышают читаемость. Когда вы удаляете подобную переменную, новый вариант кода становится более сжатым и простым для чтения.

В следующем разделе приведены примеры использования переменных, которые не являются необходимыми.

Бесполезные временные переменные

В следующем коде, написанном на языке Python, использована переменная `now`:

```
now = datetime.datetime.now()
root_message.last_view_time = now
```

Так ли необходима эта переменная? Нет, она не нужна по нескольким причинам:

- она не разбивает сложное выражение;
- она ничего не проясняет — выражение `datetime.datetime.now()` и так довольно прозрачно;
- она использована только однажды, поэтому не помогает избавиться от избыточного кода.

Без этой переменной код является таким же простым для понимания:

```
root_message.last_view_time = datetime.datetime.now()
```

Переменные вроде `now` часто являются «реликтами» — они остаются в коде после того, как тот был отредактирован. Переменная `now` в оригинале могла быть использована несколько раз. Или же программист планировал применять ее несколько раз, но в конечном итоге у него не возникло такой необходимости.

Избавляемся от промежуточных результатов

Рассмотрим код функции, написанной на языке JavaScript, которая удаляет значение из массива:

```
var remove_one = function (array, value_to_remove) {
    var index_to_remove = null;
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            index_to_remove = i;
            break;
        }
    }
    if (index_to_remove !== null) {
        array.splice(index_to_remove, 1);
    }
};
```

Переменная `index_to_remove` используется лишь для того, чтобы хранить *промежуточный результат*. От подобных переменных можно избавиться, обрабатывая результат, как только вы его получите:

```
var remove_one = function (array, value_to_remove) {
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            array.splice(i, 1);
            return;
        }
    }
};
```

Предусмотрев в коде возможность досрочного возврата, мы избавились от переменной `index_to_remove` и несколько упростили код.

В целом хорошей стратегией является **выполнение задачи так быстро, как это возможно**.

Избавляемся от переменных, управляющих потоком команд

Иногда в циклах встречается следующий шаблон:

```
boolean done = false;

while (/* условие */ && !done) {


    if (...) {
        done = true;
        continue;
    }
}
```

Значение переменной done также может задаваться как true в нескольких местах цикла.

Подобный код используется для того, чтобы выполнять какое-нибудь негласное правило, которое нельзя нарушать в середине цикла. Но такого правила нет!

Переменные вроде done называются переменными, управляющими порядком выполнения. Их единственное назначение заключается в том, чтобы управлять выполнением программы, — они не содержат никаких информационных данных. По нашему опыту, от подобных переменных можно избавиться, структурировав ваш код:

```
while (/* условие */) {
    ...
    if (...) {
        break;
    }
}
```

Этот пример довольно легко исправить, но что делать, если вам встретятся *несколько* вложенных циклов, для которых выполнения одной операции break будет недостаточно? В более запутанных случаях решением может быть выделение кода в отдельную функцию (или код внутри цикла, или сам цикл).

ХОТИТЕ ЛИ ВЫ, ЧТОБЫ ВАШИ КОЛЛЕГИ ВСЕ ВРЕМЯ ЧУВСТВОВАЛИ СЕБЯ ТАК, БУДТО ПРИШЛИ НА СОБЕСЕДОВАНИЕ?

Эрик Бречнер, сотрудник компании Microsoft, считает, что хорошая задача на собеседовании должна содержать как минимум три переменные[1]. Это целесообразно, поскольку если нужно держать в уме значения трех переменных, то это не дает кандидату расслабляться! Такая тактика действительно подходит для собеседований, где вы стараетесь максимально проверить соискателей. Но хотите ли вы, чтобы ваши коллеги при чтении кода чувствовали, будто попали на собеседование?

---

[1] Eric Brencher's «I.M. Wright's "Hard Code"» Microsoft Press, 2007, с. 166.

Сокращаем область видимости ваших переменных

Всем нам знаком совет избегать глобальных переменных. Это дельный совет, поскольку довольно трудно отследить, где и как они используются. Из-за «загрязнения пространства имен» (создания нескольких имен, которые могут конфликтовать с вашими локальными переменными) код может изменить глобальную переменную тогда, когда создатель планировал изменить локальную, и наоборот.

На самом деле сокращение области видимости *всех* переменных, а не только глобальных — хорошая идея.

ОСНОВНАЯ ИДЕЯ

Максимально сужайте область видимости переменных.

Многие языки программирования предлагают несколько уровней видимости и доступа, включая область видимости модуля, класса, функции и блока. Ограничение доступа к переменным хорошо помогает в решении этой задачи, поскольку переменные видны лишь в нескольких строках кода.

Зачем это делать? Затем, что так мы эффективно уменьшаем количество переменных, которые читателю придется держать в голове в определенный период времени. Если вы сократите область видимости всех ваших переменных на порядок или два, видимой будет лишь примерно половина переменных из текущего количества.

Например, представьте, что у вас есть очень большой класс, переменная-член которого может использоваться только в двух методах следующим образом:

```
class LargeClass {
    string str_;

    void Method1() {
        str_ = ...;
        Method2();
    }

    void Method2() {
        // используется переменная str_
    }

    // множество прочих методов, которые не используют переменную str_ ...
};
```

В некотором роде переменная, являющаяся членом класса, также представляет собой подобие «мини-глобальной» переменной, определенной только в рамках класса. Довольно сложно отследить все переменные — члены класса, а также изменения, вносимые методами класса. Особенно это касается объемных классов. Чем меньше «мини-глобальных» переменных, тем лучше.

В этом случае оптимальным решением будет «понизить» переменную `str_` до локальной:

```
class LargeClass {
    void Method1() {
        string str = ...;
        Method2(str);
    }
    void Method2(string str) {
        // использует переменную str
    }
    // Теперь другие методы не могут видеть переменную str.
};
```

Еще один способ ограничения доступа к членам классов заключается в **создании максимального количества статических методов**. Статические методы хороши тем, что позволяют читателю узнать, что эти строки кода изолированы от остальных переменных.

Можно также разбить **объемный класс на несколько меньших**. Данный подход целесообразен только в тех случаях, когда мелкие классы изолированы друг от друга. Если вы создаете два класса, которые имеют доступ к членам друг друга, то вы в действительности ничего не добиваетесь.

Тот же прием можно применить к файлам и функциям. Мотивом для такого решения может послужить желание изолировать данные (то есть переменные).

Но в разных языках правила ограничения области видимости различаются. Мы хотели бы привести вам лишь несколько самых интересных правил, которые связаны с использованием области видимости переменных.

Область видимости условной конструкции в языке C++

Представьте, что у вас есть следующий код на языке C++:

```
PaymentInfo* info = database.ReadPaymentInfo();
if (info) {
    cout << "User paid: " << info->amount() << endl;
}

// Далее множество строк кода ...
```

Переменная `info` *останется в области видимости до* конца функции, поэтому человеку, читающему этот код, придется держать ее в уме, ожидая ее изменения или повторного использования.

Но в следующем случае переменная `info` используется лишь внутри утверждения `if`. В языке C++ мы можем определить ее в условной конструкции:

```
if (PaymentInfo* info = database.ReadPaymentInfo()) {
    cout << "User paid: " << info->amount() << endl;
}
```

Теперь читатель может забыть о переменной `info` сразу же после того, как закончится условная конструкция.

Создание «индивидуальных» переменных в языке JavaScript

Представьте, что у вас есть хранимая переменная (persistent variable), которая используется только в одной функции:

```
submitted = false; // Обратите внимание: это глобальная переменная.

var submit_form = function (form_name) {
    if (submitted) {
        return;  // не отправляем форму дважды
    }
    ...
    submitted = true;
};
```

Глобальные переменные вроде `submitted` могут сильно обеспокоить человека, читающего этот код. Кажется, что `submit_form()` — это единственная функция, которая использует эту переменную, но вы не можете быть уверены в этом на 100 %. Фактически другой JavaScript-файл также может применять глобальную переменную `submitted`, но для другой цели!

Это можно предотвратить, поместив переменную `submitted` внутрь *замкнутого выражения (closure)*:

```
var submit_form = (function () {
    var submitted = false; // Обратите внимание: эта переменная
                // может использоваться только функцией, расположенной ниже.

    return function (form_name) {
        if (submitted) {
            return;  // не отправляем форму дважды
        }
        ...
        submitted = true;
    };
}());
```

Обратите внимание на круглые скобки в последней строке: анонимная внешняя функция выполняется немедленно, возвращая внутреннюю функцию.

Если вы не встречали подобного приема ранее, он может показаться странным. Создается «индивидуальная» область видимости, в которую имеет доступ только внутренняя функция. Теперь читатель не будет задаваться вопросом: «Где же еще используется переменная `submitted`?» или беспокоиться о том, что она будет конфликтовать с другими глобальными переменными с таким же именем.

Глобальная область видимости в языке JavaScript

В языке JavaScript, если опустить ключевое слово `var` из определения переменной (например, написать `x = 1` вместо `var x = 1`), переменная будет помещена

в глобальную область видимости, откуда *любой* JavaScript-файл и блок <script> сможет получить к ней доступ. Рассмотрим пример:

```
<script>
    var f = function () {
        // ВНИМАНИЕ: при объявлении переменной 'i' не использовалось
        // ключевое слово 'var'!
        for (i = 0; i < 10; i += 1)
    };

    f();
</script>
```

Этот код помещает переменную i в глобальную область видимости, поэтому последующий блок все еще может ее использовать:

```
<script>
    alert(i);  // Передается значение '10'. .
               // 'i' – это глобальная переменная!
</script>
```

Большинство программистов не подозревают об этом правиле, и эта особенность может привести к возникновению странных ошибок. Обычное проявление данной ошибки — две функции создают локальную переменную с одинаковым именем, но забывают использовать ключевое слово var. Эти функции совершенно непреднамеренно «пересекутся», и бедный программист придет к выводу, что его компьютер рехнулся или испортилась оперативная память.

Беспроигрышный метод для языка JavaScript — **всегда определять переменные, используя ключевое слово** var (например, var x = 1). Такая практика ограничивает область видимости переменной рамками (внутренней) функции, в которой она определена.

В языках Python и JavaScript нет вложенной области видимости

Языки вроде C++ и Java имеют *область видимости блока (block scope)*, где переменные, определенные внутри конструкций if, for, try и др. ограничены вложенной областью видимости этого блока:

```
if (...) {
    int x = 1;
}
x++;  // Ошибка компиляции! Переменная 'x' не определена.
```

Но в языках Python и JavaScript переменные, определенные внутри какого-либо блока, «просачиваются» во всю функцию. Например, обратите внимание на использование переменной example_value в этом идеально корректном коде, написанном на языке Python:

```
# Переменная example_value не использовалась до этого момента.
if request:
    for value in request.values:
        if value > 0:
            example_value = value
            break

for logger in debug.loggers:
    logger.log("Example:", example_value)
```

Это правило обзора может удивить многих программистов, а подобный код действительно трудно читать. В других языках гораздо проще найти то место, где была определена переменная `example_value`, — потребуется лишь отыскать открывающую фигурную скобку.

Предыдущий пример также содержит ошибку: если значение переменной `example_value` не установлено в первой части кода, вторая часть кода выдаст исключение: `NameError: 'example_value' is not defined`. Это можно исправить, а также сделать код более читаемым, определив переменную `example_value` рядом с «ближайшим общим предком» (в контексте вложенности):

```
example_value = None

if request:
    for value in request.values:
        if value > 0:
            example_value = value
            break

if example_value:
    for logger in debug.loggers:
        logger.log("Example:", example_value)
```

Однако это случай, когда можно избавиться от переменной `example_value` в целом. Эта переменная всего лишь хранит промежуточный результат, и, как мы видели в подразделе «Избавляемся от промежуточных результатов» выше, от подобных переменных можно избавиться, выполняя задание «как можно скорее». В этом случае нам следует записывать значение-образец, как только мы обнаружим его.

Так будет выглядеть новый код:

```
def LogExample(value):
    for logger in debug.loggers:
        logger.log("Example:", value)

if request:
    for value in request.values:
        if value > 0:
            LogExample(value)  # мгновенно обрабатываем значение 'value'
            break
```

Смещаем определения вниз

В оригинальном языке программирования C все определения переменных должны были располагаться в верхней части функции или блока. Это требование было очень неудобным, поскольку длинные функции с большим количеством переменных требовали от читателя держать в уме все переменные, даже если те не использовались позже. (В C99 и C++ это ограничение было отменено.)

В следующем примере все переменные определены в верхней части функции:

```
def ViewFilteredReplies(original_id):
    filtered_replies = []
    root_message = Messages.objects.get(original_id)
    all_replies = Messages.objects.select(root_id=original_id)
    root_message.view_count += 1
    root_message.last_view_time = datetime.datetime.now()
    root_message.save()

    for reply in all_replies:
        if reply.spam_votes <= MAX_SPAM_VOTES:
            filtered_replies.append(reply)

    return filtered_replies
```

Проблема этого кода заключается в том, что он требует от читателя помнить сразу три переменные, а также мысленно переключаться между ними.

Поскольку у читателя нет необходимости знать все переменные сразу, можно просто сместить каждое определение к той точке кода, где оно будет использоваться в первый раз:

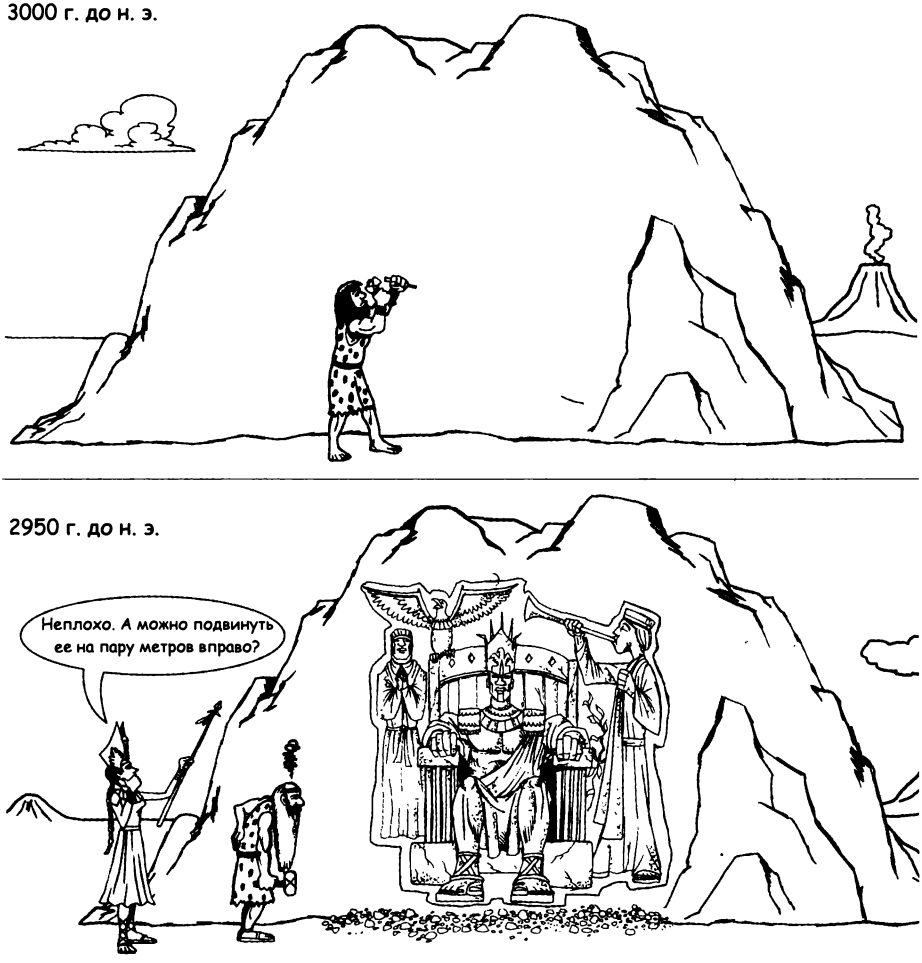
```
def ViewFilteredReplies(original_id):
    root_message = Messages.objects.get(original_id)
    root_message.view_count += 1
    root_message.last_view_time = datetime.datetime.now()
    root_message.save()

    all_replies = Messages.objects.select(root_id=original_id)
    filtered_replies = []
    for reply in all_replies:
        if reply.spam_votes <= MAX_SPAM_VOTES:
            filtered_replies.append(reply)

    return filtered_replies
```

Вы можете спросить: требуется ли многократное употребление переменной `all_replies` или же его можно удалить следующим образом:

```
    for reply in Messages.objects.select(root_id=original_id):
        ...
```

В этом случае `all_replies` — довольно хорошая поясняющая переменная, поэтому мы решили оставить ее.

Используйте переменные, меняющие свое значение однократно



В этой главе мы поговорили о том, насколько труднее становится понимать программу, в которой задействовано много переменных. Однако еще сложнее становится понимать ее в случае, когда эти переменные постоянно изменяются. Отслеживание их значений еще больше усложняет задачу.

Чтобы разобраться с этой проблемой, мы можем дать совет, который поначалу может показаться немного странным: **используйте переменные, меняющие свое значение только один раз**.

Переменные, являющиеся «перманентными», запомнить довольно легко. Конечно же, константы вида:

```
static const int NUM_THREADS = 10;
```

запоминаются довольно легко. По той же причине настоятельно рекомендуется использовать ключевые слова `const` в C++ и `final` в Java.

На самом деле во многих языках (включая Python и Java) некоторые встроенные типы вроде `string` ***неизменны***. Как сказал создатель языка Java Джеймс Гослинг: «[Неизменные типы], как правило, чаще будут работать без ошибок».

Но даже если не удается использовать переменную только один раз, следует стремиться изменять ее как можно реже.

ОСНОВНАЯ ИДЕЯ

Чем больше в коде мест, где изменяется переменная, тем сложнее отследить ее текущее значение.

Итак, что же делать? Как вы можете изменить переменную так, чтобы она использовалась только один раз? В большинстве случаев это требует лишь незначительных изменений в структуре кода, что вы можете увидеть в следующем примере.

Последний пример

В последнем примере этой главы мы хотели бы продемонстрировать большинство принципов, которые мы обсудили.

Представьте, что у вас есть веб-страница, на которой находится несколько текстовых полей, расставленных в следующем порядке:

```
<input type="text" id="input1" value="Dustin">
<input type="text" id="input2" value="Trevor">
<input type="text" id="input3" value="">
<input type="text" id="input4" value="Melissa">
...
```

Как видите, номера начинаются с `input1` и далее увеличиваются.

Ваша задача — написать функцию, которая называется `setFirstEmptyInput()`. Эта функция принимает строку и помещает ее в первое пустое поле `<input>` на странице (в этом примере — `"input3"`). Функция будет возвращать обновленный элемент объектной модели документа (или `null`, если пустых полей на странице не было). Далее приведен фрагмент кода, который решает такую задачу, но ***не*** следует принципам, описанным в этой главе:

```
var setFirstEmptyInput = function (new_value) {
    var found = false;
    var i = 1;
    var elem = document.getElementById('input' + i);
    while (elem !== null) {
        if (elem.value === '') {
            found = true;
            break;
        }
        i++;
        elem = document.getElementById('input' + i);
    }
```

```
    if (found) elem.value = new_value;
    return elem;
};
```

Этот код справляется с задачей, но при этом выглядит неидеально. Что с ним не так? Как его можно улучшить?

Способов улучшения несколько, но мы рассмотрим код с точки зрения переменных, использованных в нем:

- `var found;`
- `var i;`
- `var elem.`

Все эти переменные созданы для всей функции и использованы множество раз. Попробуем оптимизировать применение каждой из них.

Как мы говорили выше в этой главе, промежуточные переменные вроде `found` часто могут быть опущены, если делается досрочный возврат из функции. Оптимизируем код с учетом этого:

```
var setFirstEmptyInput = function (new_value) {
    var i = 1;
    var elem = document.getElementById('input' + i);
    while (elem !== null) {
        if (elem.value === '') {
            elem.value = new_value;
            return elem;
        }
        i++;
        elem = document.getElementById('input' + i);
    }
    return null;
};
```

Далее рассмотрим переменную `elem`. Она используется несколько раз в довольно «зацикленной» манере, поэтому отследить ее значение трудно. Прочитав код, кто-либо может подумать, что `elem` является счетчиком итераций, хотя на самом деле для этой цели мы используем значение переменной `i`. Реструктурируем цикл `while` в цикл `for`, который «управляется» переменной `i`:

```
var setFirstEmptyInput = function (new_value) {
    for (var i = 1; true; i++) {
        var elem = document.getElementById('input' + i);
        if (elem === null)
            return null;  // Поиск не удался. Пустые поля не найдены.

        if (elem.value === '') {
            elem.value = new_value;
            return elem;
        }
    }
};
```

В частности, обратите внимание на то, что переменная `elem` ведет себя так, как будто она используется единожды, и продолжительность ее жизни ограничена длиной цикла. Использование значения `true` в качестве условия цикла `for` — необычный ход, но теперь мы видим, как эта переменная определяется и изменяется в одной строке. (Хотя традиционный цикл `while(true)` в этой ситуации также применим.)

Итог

В этой главе мы рассмотрели ситуации, в которых переменные могут быстро во множестве накопиться в коде и стать слишком сложными для отслеживания. Код можно сделать проще для чтения, применяя меньшее количество переменных и максимально «облегчая» их. Основные принципы, которым вы должны следовать, таковы.

- **Избавляйтесь от переменных**, которые не вписываются в эту концепцию. В частности, мы показали вам несколько примеров того, как можно избавиться от переменных, содержащих промежуточные результаты. Такие результаты лучше обрабатывать немедленно.
- **Уменьшайте область видимости каждой переменной**, делая ее минимальной. Перемещайте каждую переменную в те области кода, где их будет видеть как можно меньшее количество строк программы. О переменных тоже можно сказать: «С глаз долой — из сердца вон».
- **Используйте переменные, значение которых меняется только один раз.** Такие переменные (или же константы `const`, `final` и прочие конструкции) делают код более прозрачным для понимания.

ЧАСТЬ III

Реорганизация кода

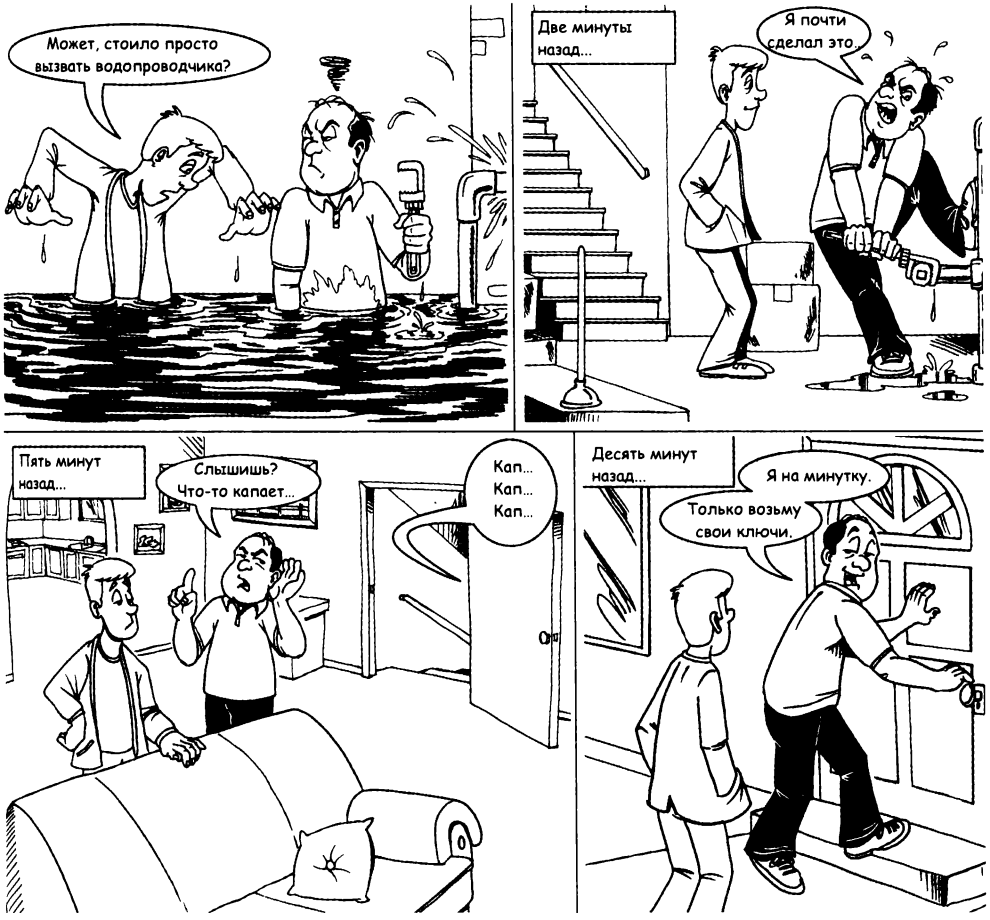
В второй части мы поговорили о том, как можно изменить циклы и логику программы для улучшения читаемости. Мы описали несколько приемов, требующих незначительного изменения структуры программы.

В этой части мы обсудим более значительные изменения, которые можно внести в код на уровне функций. Говоря конкретнее, мы рассмотрим три способа реорганизации кода:

- извлечение побочных подзадач, которые не связаны с первоначальной целью программы;
- перекомпоновку кода таким образом, чтобы он выполнял лишь одну задачу в каждый момент времени;
- поиск более прозрачного решения задачи (для этого мы сначала опишем код словами, а затем используем данное описание в качестве опоры для поиска решения).

Наконец, обсудим ситуации, когда можно целиком удалить код или вообще обойтись без него — ведь чем меньше кода, тем он понятнее.

10 Выделяем побочные подзадачи



Многие инженерные задачи решаются при помощи разбиения больших задач на более мелкие и сведения их решений воедино. Применение такого принципа к коду делает его более надежным и легким для чтения.

Главная рекомендация этой главы — **тщательно выделяйте в коде побочные подзадачи**. Опишем подробнее, что мы имеем в виду.

1. Посмотрите на конкретную функцию или блок и задайтесь вопросом: «Какова главная цель этого кода?»
2. К каждой строке кода поставьте следующий вопрос: «Помогает ли эта строка *непосредственно* достижению цели? Или же она помогает решить *побочную подзадачу*, с которой мы столкнулись в процессе программирования?»
3. Если у вас нашлось достаточно строк, решающих побочную подзадачу, выделите этот код в отдельную функцию.

Возможно, подобным выделением функций вы занимаетесь каждый день. Но в этой главе мы хотели бы остановиться на особенных случаях выделения побочных подзадач, когда отделяемый код будет находиться в блаженном неведении о том, кто его вызывает.

Как видите, это довольно простой прием, однако он может значительно улучшить ваш код. По каким-то причинам многие программисты нечасто используют этот прием. Его суть заключается в том, чтобы активно искать побочные подзадачи.

В данной главе мы рассмотрим множество примеров, иллюстрирующих использование этого приема в различных ситуациях, с которыми можете столкнуться и вы.

Вводный пример: findClosestLocation()

Главная цель следующего кода, написанного на языке JavaScript, заключается в том, *чтобы найти ближайшую к заданной точке локацию* (не увязните в запутанной геометрии, которую мы выделили в коде курсивом):

```
// Возвращаем элемент массива 'array',
// который располагается ближе всего по широте и долготе.
// Моделирует Землю как идеальную сферу.
var findClosestLocation = function (lat, lng, array) {
    var closest;
    var closest_dist = Number.MAX_VALUE;
    for (var i = 0; i < array.length; i += 1) {
        // Преобразуем координаты к радианам.
        var lat_rad = radians(lat);
        var lng_rad = radians(lng);
        var lat2_rad = radians(array[i].latitude);
        var lng2_rad = radians(array[i].longitude);

        // Используем формулу "Сферического закона косинуса".
        var dist = Math.acos(Math.sin(lat_rad) * Math.sin(lat2_rad) +
```

```
                        Math.cos(lat_rad) * Math.cos(lat2_rad) *
                        Math.cos(lng2_rad - lng_rad));

        if (dist < closest_dist) {
            closest = array[i];
            closest_dist = dist;
        }
    }
    return closest;
};
```

Большая часть кода внутри цикла решает побочную подзадачу: вычисление сферического расстояния между двумя точками с использованием широты и долготы. Поскольку для решения этой задачи применяется много строк кода, можно выделить их в отдельную функцию `spherical_distance()`:

```
var spherical_distance = function (lat1, lng1, lat2, lng2) {
    var lat1_rad = radians(lat1);
    var lng1_rad = radians(lng1);
    var lat2_rad = radians(lat2);
    var lng2_rad = radians(lng2);

   // Используем формулу "Сферического закона косинуса".
    return Math.acos(Math.sin(lat1_rad) * Math.sin(lat2_rad) +
                     Math.cos(lat1_rad) * Math.cos(lat2_rad) *
                     Math.cos(lng2_rad - lng1_rad));
};
```

Оставшийся код теперь выглядит так:

```
var findClosestLocation = function (lat, lng, array) {
    var closest;
    var closest_dist = Number.MAX_VALUE;
    for (var i = 0; i < array.length; i += 1) {
        var dist = spherical_distance(lat, lng, array[i].latitude,
                                      array[i].longitude);
        if (dist < closest_dist) {
            closest = array[i];
            closest_dist = dist;
        }
    }
    return closest;
};
```

Этот код воспринимается гораздо лучше, поскольку читатель может сфокусироваться на основной цели, не отвлекаясь на множество геометрических вычислений.

Здесь есть и дополнительное преимущество: функцию `spherical_distance()` проще тестировать, когда она изолирована. Ее также можно будет использовать в будущем. Поэтому она и называется побочной подзадачей — эта функция полностью самодостаточна и «не знает», как ее используют различные приложения.

Чистый вспомогательный код

Есть основной набор простых задач, которые решаются в большинстве программ: например, манипулирование строками, использование хэш-таблиц, а также чтение и запись файлов.

Часто эти «базовые функции» реализуются встроенными библиотеками языка программирования. Например, если вы хотите прочесть все содержимое файла, в языке PHP вы можете вызвать функцию `file_get_contents("filename")`, а в языке Python — `open("filename").read()`.

Но иногда все же приходится «заполнять пробелы» самостоятельно. Например, в языке C++ нет прямого способа прочесть все содержимое файла. Вместо этого придется написать код, подобный следующему:

```
ifstream file(file_name);

// Рассчитываем размер файла и выделяем буфер такого размера.
file.seekg(0, ios::end);
const int file_size = file.tellg();
char* file_buf = new char [file_size];

// Считываем все содержимое файла в этот буфер.
file.seekg(0, ios::beg);
file.read(file_buf, file_size);
file.close();
```

Перед нами классический пример побочной подзадачи, решение которой должно быть выделено в новую функцию, например `ReadFileToString()`. Теперь вся остальная база кода может вести себя так, словно в языке C++ действительно есть такая функция.

В общем, если у вас возникнет мысль вроде: «Я бы хотел, чтобы в нашей библиотеке была функция *XYZ()*», напишите ее сами! (Предполагается, что такая функция еще не существует.) С течением времени у вас на руках будет коллекция вспомогательного кода, который можно будет использовать во множестве проектов.

Прочий универсальный код

При отладке кода, написанного на языке JavaScript, программисты часто используют функцию `alert()` для вызова всплывающего сообщения, которое содержит некоторую информацию, полезную для программиста, так называемую веб-версию «`printf()` для отладки». Например, следующий вызов функции отправляет данные серверу, используя Ajax, а затем отображает ответ, возвращенный сервером:

```
ajax_post({
    url: 'http://example.com/submit',
    data: data,
```

```
    on_success: function (response_data) {
        var str = "{\n";
        for (var key in response_data) {
            str += "  " + key + " = " + response_data[key] + "\n";
        }
        alert(str + "}");

        // Продолжаем обрабатывать 'response_data'
    }
});
```

Основная цель этого кода — *вызвать сервер при помощи Ajax, а затем обработать его отклик.* Но в программе есть немало кода, решающего побочную подзадачу, то есть выводящего *форматированный ответ сервера на экран.* Несложно было бы вынести этот код в отдельную функцию `format_pretty(obj)`:

```
var format_pretty = function (obj) {
    var str = "{\n";
    for (var key in obj) {
        str += "  " + key + " = " + obj[key] + "\n";
    }
    return str + "}";
};
```

Неожиданные преимущества

Существует множество причин для выделения функции `format_pretty()`. Это упрощает код вызова, а использовать саму функцию довольно удобно.

Однако есть еще одна отличная, но неочевидная причина поступать именно так: **функцию** `format_pretty()` **гораздо легче улучшить, когда она автономна**. Если вы работаете с небольшой отдельной функцией, вам проще добавить новые особенности, повысить надежность кода, обработать спорные ситуации и пр.

Функция `format_pretty()` неприменима в ситуациях, если она ожидает, что:

- `obj` — это объект; если вместо объекта была получена простая строка (или `undefined`), текущий код выбросит исключение;
- каждое значение объекта `obj` будет иметь простой тип. Если же объект будет содержать вложенные объекты, то текущий код отобразит их в виде `[object Object]`, что будет уже не так красиво.

Если нам нужно внести в код эти улучшения до того, как мы выделим функцию `format_pretty()`, то нам придется хорошо поработать. (На самом деле очень трудно реализовать рекурсивный вывод на экран вложенных объектов без выделения кода в отдельную функцию.)

Но теперь добавить подобную функциональность стало легко. Вот как выглядит улучшенный код:

```
var format_pretty = function (obj, indent) {
    // Обрабатываем ответы с типами null, undefined, strings,
    // а также ответы, не являющиеся объектами.
```

```
    if (obj === null) return "null";
    if (obj === undefined) return "undefined";
    if (typeof obj === "string") return '"' + obj + '"';
    if (typeof obj !== "object") return String(obj);

    if (indent === undefined) indent = "";

    // Обрабатываем непустые объекты.
    var str = "{\n";
    for (var key in obj) {
        str += indent + "  " + key + " = ";
        str += format_pretty(obj[key], indent + "  ") + "\n";
    }
    return str + indent + "}";
};
```

Этот код справляется с нюансами, перечисленными выше, а на экране отображается следующее:

```
{
    key1 = 1
    key2 = true
    key3 = undefined
    key4 = null
    key5 = {
        key5a = {
            key5a1 = "hello world"
        }
    }
}
```

Создавайте больше универсального кода

Функции `ReadFileToString()` и `format_pretty()` — отличные примеры решения побочных подзадач. Эти подзадачи настолько просты и так часто встречаются, что, скорее всего, вы будете использовать вышеназванные функции во многих своих проектах. Базы кода зачастую имеют специальный каталог для подобного кода (например, `util/`), поэтому его можно без труда применять совместно, одновременно в нескольких ситуациях.

Универсальный код восхитителен, поскольку **он полностью отделен от остального проекта**. Подобный код легче изменять, тестировать и понимать. Эх, если бы весь код был таким!

Подумайте о многих мощных библиотеках и системах, которыми вы пользуетесь. Таковы, например, базы данных SQL, библиотеки JavaScript, а также система шаблонов HTML. Вам не приходится беспокоиться об их содержимом — эти базы кода полностью изолированы от вашего проекта. В результате сам проект остается небольшим.

Чем большую часть проекта вы сможете разбить на изолированные библиотеки, тем лучше, поскольку ваш код будет короче и проще для понимания.

ВОСХОДЯЩЕЕ ИЛИ НИСХОДЯЩЕЕ ПРОГРАММИРОВАНИЕ?

Нисходящим программированием называется стиль, при котором сначала разрабатываются высокоуровневые модули и функции, а низкоуровневые функции реализуются по мере необходимости для их поддержки.

Если вы занимаетесь восходящим программированием, то вы сначала решаете все подзадачи, а затем создаете высокоуровневые компоненты, используя эти фрагменты.

Мы не выступаем за использование того или иного метода. Большинство программистов стараются комбинировать эти методы. Важен только конечный результат — когда отделены и решены все подзадачи.

Функциональность, специфичная для проекта

В идеале подзадачи, которые вы выделяете, не должны быть связаны с проектом. Но даже если это не так — ничего страшного. Выделение подзадач замечательно работает, даже если они специфичны для того или иного проекта.

Вот пример кода веб-сайта, содержащего деловые обзоры. Этот код, написанный на языке Python, создает новый объект класса `Business` и задает значения полей `name`, `url` и `date_created`:

```
business = Business()
business.name = request.POST["name"]

url_path_name = business.name.lower()
url_path_name = re.sub(r"['\.]", "", url_path_name)
url_path_name = re.sub(r"[^a-z0-9]+", "-", url_path_name)
url_path_name = url_path_name.strip("-")
business.url = "/biz/" + url_path_name

business.date_created = datetime.datetime.utcnow()
business.save_to_database()
```

Предполагается, что поле `url` должно содержать прозрачную версию имени объекта. Например, если имя объекта A.C. Joe's Tire & Smog Inc., то его `url` будет иметь значение `"/biz/ac-joes-tire-smog-inc"`.

Побочная подзадача, решаемая в этом коде, — *преобразование имени к корректному URL*. Мы с легкостью можем извлечь код, решающий эту подзадачу. Кроме того, мы можем заранее скомпилировать регулярные выражения (и дать им удобочитаемые имена):

```
CHARS_TO_REMOVE = re.compile(r"['\.]+")
CHARS_TO_DASH = re.compile(r"[^a-z0-9]+")

def make_url_friendly(text):
    text = text.lower()
    text = CHARS_TO_REMOVE.sub('', text)
    text = CHARS_TO_DASH.sub('-', text)
    return text.strip("-")
```

Теперь оригинальный код построен по более «правильному» образцу:

```
business = Business()
business.name = request.POST["name"]
business.url = "/biz/" + make_url_friendly(business.name)
business.date_created = datetime.datetime.utcnow()
business.save_to_database()
```

Прочитать этот код будет гораздо проще, поскольку вы не будете отвлекаться на регулярные выражения и манипуляции со строками.

Куда же стоит поместить код функции `make_url_friendly()`? Она похожа на функцию, решающую общие задачи, поэтому было бы логично поместить ее в каталог `util/`. С другой стороны, эти регулярные выражения создавались для того, чтобы работать с наименованиями юридических лиц США, поэтому, возможно, следует оставить код в том файле, где он и используется. На самом деле расположение кода не имеет особого значения, вы с легкостью можете передвинуть определение подальше. Гораздо важнее то, что эта функция была извлечена.

Упрощаем существующий интерфейс

Всем нравится, когда библиотека имеет ясный интерфейс, особенно если он принимает мало аргументов, не нуждается в прочих библиотеках, а использование интерфейса в целом протекает легко. Код получается элегантным: он одновременно простой и мощный.

Но если интерфейс, который вы используете, не слишком прозрачен, то можете написать собственные функции-обертки, предлагающие такой интерфейс.

Например, работа с браузерными cookies в языке JavaScript далека от идеала. Концептуально cookies представляют собой последовательность пар имя/значение. Но интерфейс, предоставляемый браузером, возвращает единственную строку `document.cookie` со следующим синтаксисом:

```
name1=value1; name2=value2;
```

Чтобы найти необходимый cookie, вам приходится самостоятельно преобразовывать эту гигантскую строку. Рассмотрим пример кода, который считывает значение cookie с именем `"max_results"`:

```
var max_results;
var cookies = document.cookie.split(';');
for (var i = 0; i < cookies.length; i++) {
    var c = cookies[i];
    c = c.replace(/^[ ]+/, '');  // удаляем пробелы
    if (c.indexOf("max_results=") === 0)
        max_results = Number(c.substring(12, c.length));
}
```

Ух, какой некрасивый код. Очевидно напрашивается создание функции `get_cookie()`, поэтому мы просто напишем:

```
var max_results = Number(get_cookie("max_results"));
```

Создание или изменение значения cookie выглядит еще более странно. Вы устанавливаете значение в строке `document.cookie`, используя следующий синтаксис:

```
document.cookie = "max_results=50; expires=Wed, 1 Jan 2020 20:53:47 UTC;
                                                        path=/";
```

По идее это выражение должно перезаписывать все другие существующие cookies, но оно (мистика) этого не делает!

Более близкий к идеалу интерфейс для установки значения cookie выглядит примерно так:

```
set_cookie(name, value, days_to_expire);
```

Стирание значения cookie также выглядит довольно непонятно: в качестве даты истечения срока действия приходится устанавливать прошедшее число. Идеальный же интерфейс должен быть простым:

```
delete_cookie(name);
```

Мораль заключается в том, что **никогда не следует пользоваться интерфейсом, который не является идеальным**. Вы всегда можете создать собственные функции-обертки, которые спрячут некрасивые элементы интерфейса, с которым вы столкнулись.

Изменяем интерфейс под собственные нужды

Большая часть кода программы создана для того, чтобы поддерживать другой код, например настройка параметров, передаваемых в функцию, или обработка возвращаемых параметров. Этот связующий код часто не имеет отношения к реальной логике программы и отлично подходит для вынесения в отдельные функции.

Например, представьте, что вам нужно работать со словарем на языке Python, содержащим конфиденциальную информацию о пользователях в следующем виде: `{ "username": "...", "password": "..." }`, и вам нужно поместить всю эту информацию в URL. Поскольку она конфиденциальна, вы решили сначала зашифровать словарь, используя класс `Cipher`.

Но класс `Cipher` ожидает на входе строку, а не словарь. Кроме того, он возвращает строку байтов (а нам нужно что-нибудь более подходящее для URL) и принимает несколько дополнительных параметров. Таким образом, он довольно неудобен в использовании.

Решение простой задачи обернулось созданием множества связующего кода:

```
user_info = { "username": "...", "password": "..." }
user_str = json.dumps(user_info)
cipher = Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
encrypted_bytes = cipher.update(user_str)
encrypted_bytes += cipher.final()  # flush out the current 128 bit block
url = "http://example.com/?user_info=" +
                          base64.urlsafe_b64encode(encrypted_bytes)
```

Даже несмотря на то, что мы решаем задачу *шифрования информации о пользователях в URL*, большая часть кода решает задачу *преобразования этого объекта языка Python в более подходящую для URL строку*. Довольно легко выделить решение этой подзадачи:

```
def url_safe_encrypt(obj):
    obj_str = json.dumps(obj)
    cipher = Cipher("aes_128_cbc", key=PRIVATE_KEY,
                    init_vector=INIT_VECTOR, op=ENCODE)
    encrypted_bytes = cipher.update(obj_str)
    encrypted_bytes += cipher.final()  # flush out the current 128 bit block
    return base64.urlsafe_b64encode(encrypted_bytes)
```

Тогда результирующий код, реализующий логику *самой программы*, становится довольно простым:

```
user_info = { "username": "...", "password": "..." }
url = "http://example.com/?user_info=" + url_safe_encrypt(user_info)
```

Все хорошо в меру

Как мы говорили в начале этой главы, наша цель — *тщательное* определение и извлечение побочных подзадач. Мы говорим «тщательное», поскольку именно тщательности не хватает многим программистам. Однако есть и другая сторона медали — можно чересчур увлечься.

Например, код из предыдущего раздела можно разбить на еще более мелкие фрагменты так:

```
user_info = { "username": "...", "password": "..." }
url = "http://example.com/?user_info=" + url_safe_encrypt_obj(user_info)

def url_safe_encrypt_obj(obj):
    obj_str = json.dumps(obj)
    return url_safe_encrypt_str(obj_str)

def url_safe_encrypt_str(data):
    encrypted_bytes = encrypt(data)
    return base64.urlsafe_b64encode(encrypted_bytes)

def encrypt(data):
    cipher = make_cipher()
    encrypted_bytes = cipher.update(data)
    encrypted_bytes += cipher.final()  # flush out any remaining bytes
    return encrypted_bytes

def make_cipher():
    return Cipher("aes_128_cbc", key=PRIVATE_KEY,
                  init_vector=INIT_VECTOR, op=ENCODE)
```

Введение большого количества мелких функций вредит читаемости, поскольку *читателю теперь приходится помнить больше информации, а отслеживание* порядка выполнения требует постоянных переходов от функции к функции.

Добавление новых функций в код немного (но ощутимо) влияет на читаемость. В предыдущем случае автор просто пожертвовал удобочитаемостью, не приобретя ничего взамен. Добавлять эти мелкие функции целесообразно только в тех случаях, когда они требуются в других частях вашего проекта. В противном случае без них можно обойтись.

Итог

Основная мысль этой главы довольно проста — **отделяйте код, предназначенный для решения общих задач, от кода вашего проекта**. Оказывается, большая часть кода является универсальной. Создавая множество библиотек и вспомогательных функций, предназначенных для решения общих задач, можно уменьшить размер проекта, что сделает программу уникальной.

Основная цель этого приема — позволить программисту сосредоточиться на небольших, хорошо описанных задачах, которые отделены от остального проекта. В результате решения этих подзадач будут более детальными и верными. Вы также сможете использовать их в дальнейшем.

11 Одна задача в любой момент времени

Код, который решает несколько задач одновременно, труднее понять. Единственный блок кода может инициализировать новые объекты, очищать данные, преобразовывать введенную информацию и использовать бизнес-логику в одно и то же время. Если весь этот код переплетен воедино, то его гораздо сложнее понять, нежели в том случае, когда каждое задание начинается и заканчивается по отдельности.

ОСНОВНАЯ ИДЕЯ

Код должен быть реорганизован таким образом, чтобы он выполнял только одно задание в каждый момент времени.

Говоря простым языком, эта глава посвящена «дефрагментации» кода. Рисунок 11.1 иллюстрирует этот процесс: слева показано выполнение кодом различных задач одновременно, а правая часть демонстрирует выполнение кодом всего одной задачи в любой момент времени.

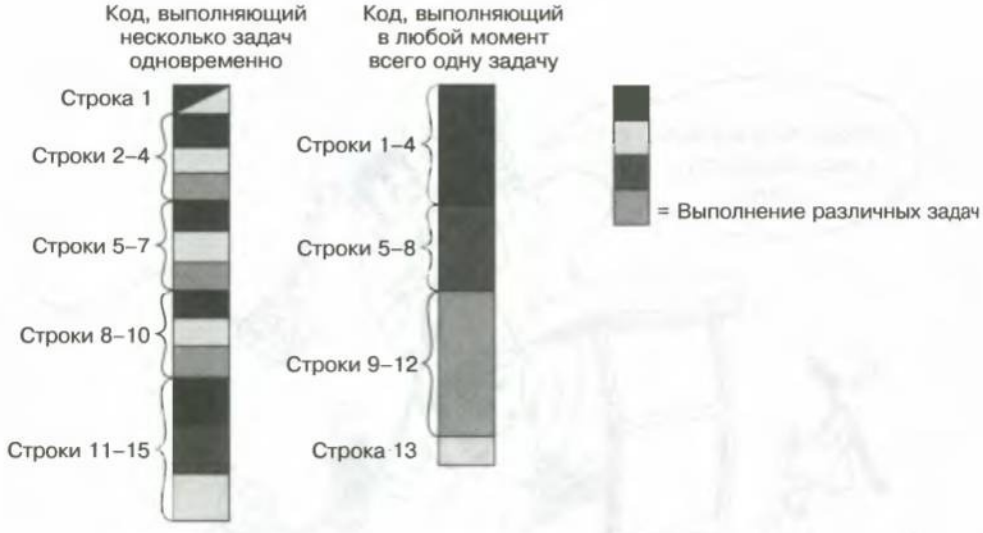


Рис. 11.1

Возможно, вы слышали следующий совет: функции должны быть однозадачными.

Наш совет похож на этот, но он касается не только функций. Конечно, разбиение большой функции на множество мелких может оказаться полезным. Но даже если вы не делаете этого, вы все равно можете организовать код внутри этой функции так, будто в ней есть отдельные логические разделы.

Опишем процесс подобной организации кода.

1. Перечислите все задания, которые выполняет ваш код. Мы используем слово «задание» в очень широком смысле: оно может быть как очень маленьким (например, «убедиться, что объект корректен»), так и распространенным (например, «пройти по каждому узлу дерева»).

2. Попробуйте выделить эти задания в отдельные функции или хотя бы в отдельные разделы кода.

В этой главе будет показано несколько примеров, которые иллюстрируют способы достижения данной цели.

Задания могут быть маленькими

Представьте, что в каком-нибудь блоге есть виджет для голосования, с помощью которого пользователь может проголосовать «За» (Up) или «Против» (Down). Рейтинг комментария (score) получается из суммы всех голосов: +1 за каждый голос «За», −1 за каждый голос «Против».

На рис. 11.2 представлены три состояния, которые может иметь голос пользователя, а также их влияние на итоговый рейтинг.

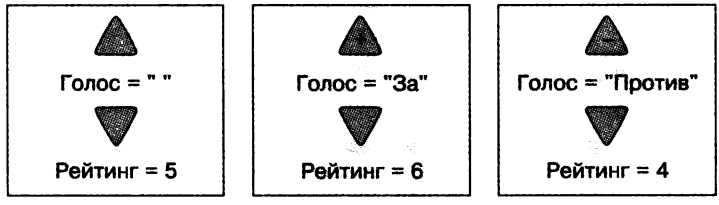


Рис. 11.2

Когда пользователь щелкает на одной из кнопок (чтобы проголосовать/изменить свой голос), вызывается следующая функция, написанная на языке JavaScript:

```
vote_changed(old_vote, new_vote);   // Каждый голос может быть "За",
                                    // "Против" или "".
```

Эта функция обновляет итоговый рейтинг и работает для всех комбинаций параметров old_vote/new_vote:

```
var vote_changed = function (old_vote, new_vote) {
    var score = get_score();

    if (new_vote !== old_vote) {
        if (new_vote === 'Up') {
            score += (old_vote === 'Down' ? 2 : 1);
        } else if (new_vote === 'Down') {
            score -= (old_vote === 'Up' ? 2 : 1);
        } else if (new_vote === '') {
            score += (old_vote === 'Up' ? -1 : 1);
        }
    }

    set_score(score);
};
```

Несмотря на то что этот фрагмент кода довольно короткий, он решает множество задач. Он имеет много замысловатых деталей, и сложно с ходу сказать, есть ли здесь ошибки, связанные с выходом за пределы массива, опечатками и пр.

Может показаться, что код лишь обновляет рейтинг, но на самом деле он выполняет *два* задания одновременно:

- параметры `old_vote` и `new_vote` «преобразуются» в числовые значения;
- обновляется параметр `score`.

Мы можем сделать этот код проще, разделив решения каждой из задач. Следующий фрагмент кода выполняет первое задание — преобразует голос в числовое значение:

```
var vote_value = function (vote) {
    if (vote === 'Up') {
        return +1;
    }
    if (vote === 'Down') {
        return -1;
    }
    return 0;
};
```

Остальной код решает вторую задачу — обновляет параметр `score`:

```
var vote_changed = function (old_vote, new_vote) {
    var score = get_score();

    score -= vote_value(old_vote);  // удаляем прежний голос
    score += vote_value(new_vote);  // добавляем новый голос

    set_score(score);
};
```

Как вы можете видеть, эта версия кода требует от вас меньше усилий, чтобы убедиться в том, что она работает. Такой подход является значительной составляющей упрощения кода для понимания.

Извлекаем значения из объекта

Однажды при работе нам пришлось столкнуться с кодом на JavaScript, который преобразовывал местонахождение пользователя в простую строку вида «Город, страна» (например, Santa Monica, USA или Paris, France). У нас был словарь `location_info`, содержащий структурированную информацию. Все, что оставалось сделать, — извлечь «Город» и «Страну» из его полей и объединить их в одну строку.

На рис. 11.3 показан пример ввода/вывода информации.

Эта задача кажется вполне простой, но ее сложность заключалась в том, что *некоторые или даже все четыре значения могли отсутствовать*. Мы справились с этой проблемой таким образом.

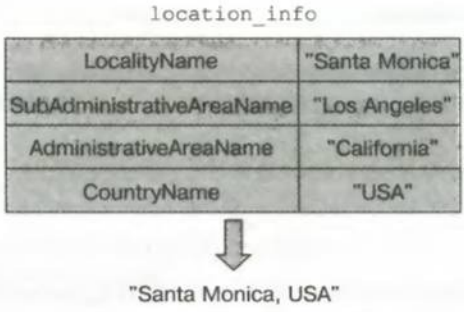


Рис. 11.3

- При выборе «Города» мы сначала использовали параметр `LocalityName` (город), если он доступен, далее `SubAdministrativeAreaName` (крупный город/округ), далее `AdministrativeAreaName` (штат/территория).
- Если отсутствовали все три параметра, «Город» получал название по умолчанию `Middle-of-Nowhere` (посреди Небытия).
- Если же отсутствовал параметр `CountryName`, по умолчанию использовалось значение `Planet Earth` (планета Земля).

Рисунок 11.4 иллюстрирует два примера обработки отсутствующих значений:

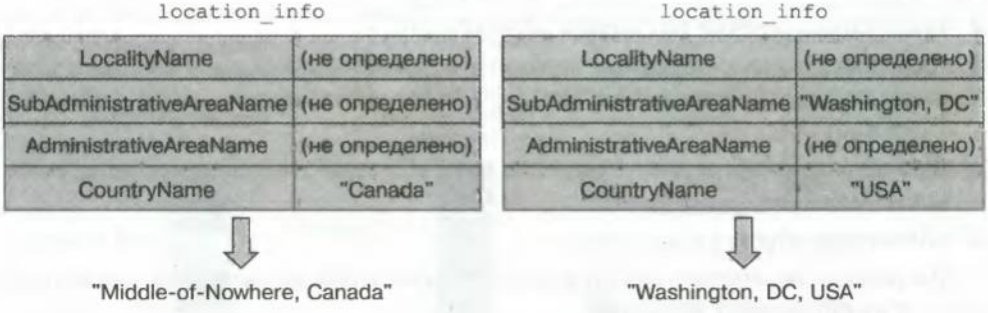


Рис. 11.4

Рассмотрим код, который мы использовали для реализации решения этой задачи:

```
var place = location_info["LocalityName"];  // например, "Santa Monica"
if (!place) {
    place = location_info["SubAdministrativeAreaName"];  // например,
                                                         // "Los Angeles"
}
if (!place) {
    place = location_info["AdministrativeAreaName"];  // например,
                                                      // "California"
}
if (!place) {
```

```
    place = "Middle-of-Nowhere";
}
if (location_info["CountryName"]) {
    place += ", " + location_info["CountryName"];  // например, "USA"
} else {
    place += ", Planet Earth";
}

return place;
```

Конечно, код выглядит немного путано, но зато качественно выполняет задачу.

Но через несколько дней нам понадобилось усовершенствовать его функциональность: для пользователей, находящихся в США, мы хотели отображать *штат* вместо страны (если это было возможно). Например, вместо Santa Monica, USA функция возвращала бы Santa Monica, California.

Добавление этой особенности к предыдущему коду сделало бы его гораздо более некрасивым.

Применяем принцип «Одна задача в любой момент времени»

Вместо того чтобы подчинять этот код своей воле, мы остановились и поняли, что он уже выполнял несколько заданий одновременно:

- извлечение значений из словаря `location_info`;
- последовательное считывание параметров для заполнения поля «Город» и установку стандартного значения `Middle-of-Nowhere` в тех случаях, когда параметры отсутствовали;
- получение параметра для заполнения поля «Страна» и использование по умолчанию значения `Planet Earth`;
- обновление объекта `place`.

Мы решили переписать оригинальный код так, чтобы он выполнял только одну задачу в любой момент времени.

Первое задание (извлечение значений из словаря) решить независимо от других было легко:

```
var town    = location_info["LocalityName"];                // например,
                                                          // "Santa Monica"
var city    = location_info["SubAdministrativeAreaName"]; // например,
                                                            // "Los Angeles"
var state   = location_info["AdministrativeAreaName"];    // например, "CA"
var country = location_info["CountryName"];               // например, "USA"
```

После этого нам уже не нужно было использовать словарь `location_info` и запоминать эти длинные и нелогичные ключи. Вместо этого у нас были четыре простые переменные.

Далее мы определили вторую половину возвращаемого значения:

```
// Начинаем со значения по умолчанию и переписываем его
```

```
// при появлении более точного значения.
var second_half = "Planet Earth";
if (country) {
    second_half = country;
}
if (state && country === "USA") {
    second_half = state;
}
```

Аналогично мы обработали и первую половину:

```
var first_half = "Middle-of-Nowhere";
if (state && country !== "USA") {
    first_half = state;
}
if (city) {
    first_half = city;
}
if (town) {
    first_half = town;
}
```

Наконец, мы объединили полученные строки в одну:

```
return first_half + ", " + second_half;
```

Рисунок с примером «дефрагментации», показанный в начале этой главы, на самом деле представляет собой оригинальное решение и эту новую версию. Приведем эту же иллюстрацию (см. рис. 11.1), но добавим в нее больше деталей (рис. 11.5).

Рис. 11.5

Как видите, четыре задания во втором варианте решения были дефрагментированы в отдельные регионы.

Еще один подход

Рефакторинг кода часто можно провести несколькими способами, и наш пример не исключение. Как только вы отделите некоторые задания, код станет проще для понимания и вы можете придумать более удачный способ его изменения.

Например, те несколько условных конструкций, находившихся в верхней части кода, следует читать довольно аккуратно, чтобы убедиться, что все они работают корректно. В этом коде одновременно выполняются две подзадачи:

- проход по списку всех переменных и выбор той, которая в данном случае подходит больше всего;
- использование другого списка, основываясь на значении поля «Страна» (является ли оно равным USA).

Подводя итог, можно заметить, что логика `if USA` пересекалась с логикой остальной программы. Вместо этого можно обработать ситуации USA и не-USA отдельно:

```
var first_half, second_half;

if (country === "USA") {
    first_half = town || city || "Middle-of-Nowhere";
    second_half = state || "USA";
} else {
    first_half = town || city || state || "Middle-of-Nowhere";
    second_half = country || "Planet Earth";
}

return first_half + ", " + second_half;
```

Если вы не очень хорошо знаете JavaScript, подскажем, что выражение `a || b || c` является идиоматическим и оценивает первое истинное (`true`) значение (в данном случае непустую строку). Достоинство этого кода заключается в том, что список предпочтений довольно просто исследовать и обновлять. Большая часть условных конструкций была удалена из кода, а бизнес-логика программы теперь представлена меньшим количеством строк.

Более объемный пример

В системе поискового робота, которую мы построили, есть функция `UpdateCounts()`. Она вызывается для того, чтобы увеличить некоторые параметры после того, как загружается каждая веб-страница:

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State" ][hd.exit_state()]++;      // например,
                    // "SUCCESS" (УСПЕХ) или "FAILURE" (НЕУДАЧА)
    counts["Http Response"][hd.http_response()]++;    // например,
                    // "404 NOT FOUND" (404 СТРАНИЦА НЕ НАЙДЕНА)
```

```
    counts["Content-Type" ][hd.content_type()]++; // например,
                          // "text/html" (текст/html-страница)
}
```

Ну, по крайней мере, мы *хотим*, чтобы наш код выглядел именно так.

В действительности объект HttpDownload не имеет ни одного из вышеперечисленных методов. Наоборот, HttpDownload — это очень объемный и сложный класс, от которого наследуют многие другие классы, и нам нужно «вылавливать» эти значения самостоятельно. Задачу усложняет тот факт, что иногда отсутствуют все эти значения — в данном случае нам приходится использовать "unknown" (неизвестно) в качестве значения по умолчанию. По этим причинам существующий код выглядит довольно путано:

```
// ПРЕДОСТЕРЕЖЕНИЕ: НЕ ВГЛЯДЫВАЙТЕСЬ В ЭТОТ КОД ПОДОЛГУ.
void UpdateCounts(HttpDownload hd) {
    // Определяем состояние выхода, если оно доступно.
    if (!hd.has_event_log() || !hd.event_log().has_exit_state()) {
        counts["Exit State"]["unknown"]++;
    } else {
        string state_str = ExitStateTypeName(hd.event_log().exit_state());
        counts["Exit State"][state_str]++;
    }

    // Если HTTP-заголовки не используются совсем, устанавливаем
    // для остальных элементов значение "unknown" (неизвестно).
    if (!hd.has_http_headers()) {
        counts["Http Response"]["unknown"]++;
        counts["Content-Type"]["unknown"]++;
        return;
    }

    HttpHeaders headers = hd.http_headers();

    // Запишем ответ сервера, если он нам известен,
    // в противном случае указываем значение "unknown" (неизвестно).
    if (!headers.has_response_code()) {
        counts["Http Response"]["unknown"]++;
    } else {
        string code = StringPrintf("%d", headers.response_code());
        counts["Http Response"][code]++;
    }

    // Записываем тип содержимого, если он известен,
    // в противном случае указываем "unknown" (неизвестно).
    if (!headers.has_content_type()) {
        counts["Content-Type"]["unknown"]++;
    } else {
        string content_type = ContentTypeMime(headers.content_type());
        counts["Content-Type"][content_type]++;
    }
}
```

Как видите, здесь довольно много кода, много логики и даже несколько повторяющихся строк. Читать этот код не так уж и просто.

В частности, данный код переключается между различными задачами. Некоторые задачи, которые выполняются отдельно друг от друга на протяжении всего кода, приведены в следующем списке.

- Использование значения "unknown" как стандартного значения для каждого ключа.
- Определение того, достаточно ли членов объекта HttpDownload.
- Извлечение значения и преобразование его к строке.
- Обновление массива counts[].

Мы можем улучшить код, обособив в отдельные области те его части, которые выполняют различные задачи:

```
void UpdateCounts(HttpDownload hd) {
    // Задача: определить значения по умолчанию для каждого типа значений,
    // который мы хотим извлечь.
    string exit_state = "unknown";
    string http_response = "unknown";
    string content_type = "unknown";

    // Задача: попытаться извлечь каждое значение из объекта HttpDownload,
    // одно за другим.
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
      exit_state = ExitStateTypeName(hd.event_log().exit_state());
    }
    if (hd.has_http_headers() && hd.http_headers().has_response_code()) {
      http_response = StringPrintf("%d", hd.http_headers().response_code());
    }
    if (hd.has_http_headers() && hd.http_headers().has_content_type()) {
      content_type = ContentTypeMime(hd.http_headers().content_type());
    }

    // Задача: обновить массив counts[].
    counts["Exit State"][exit_state]++;
    counts["Http Response"][http_response]++;
    counts["Content-Type"][content_type]++;
}
```

Как видите, в этом коде выделено три области, в которых выполняются следующие задачи:

1. Определение стандартных значений для тех ключей, которые нам необходимы.
2. Извлечение значений для каждого из ключей, если они доступны, и преобразование их к строке.
3. Обновление массива counts[] для каждой пары ключ/значение.

Работать с областями удобно потому, что они *изолированы* друг от друга, — пока вы читаете одну область, вам не нужно отвлекаться на другие.

Обратите внимание на то, что, хотя мы перечислили четыре задачи, нам удалось разделить только три из них. И это хорошо: перечисленные в начале задачи — это всего лишь отправная точка. Отделение даже нескольких из них может, как в этом случае, значительно упростить чтение кода.

Дальнейшие улучшения

Эта новая версия кода значительно улучшена по сравнению с первоначальной гротескной конструкцией. Обратите внимание на то, что мы даже не создавали дополнительных функций, чтобы так исправить код. Как мы упоминали ранее, идея выполнения одной задачи в любой момент времени помогает очистить код независимо от существующих в нем границ функций.

Однако мы можем исправить код и другим способом, добавив в него три вспомогательные функции:

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State"][ExitState(hd)]++;
    counts["Http Response"][HttpResponse(hd)]++;
    counts["Content-Type"][ContentType(hd)]++;
}
```

Эти функции извлекают соответствующее значение или возвращают unknown (неизвестно). Например:

```
string ExitState(HttpDownload hd) {
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
        return ExitStateTypeName(hd.event_log().exit_state());
    } else {
        return "unknown";
    }
}
```

Обратите внимание на то, что в этом альтернативном решении не определяется ни одна переменная! Как мы упоминали в гл. 9, от переменных, хранящих промежуточные результаты, можно полностью избавиться.

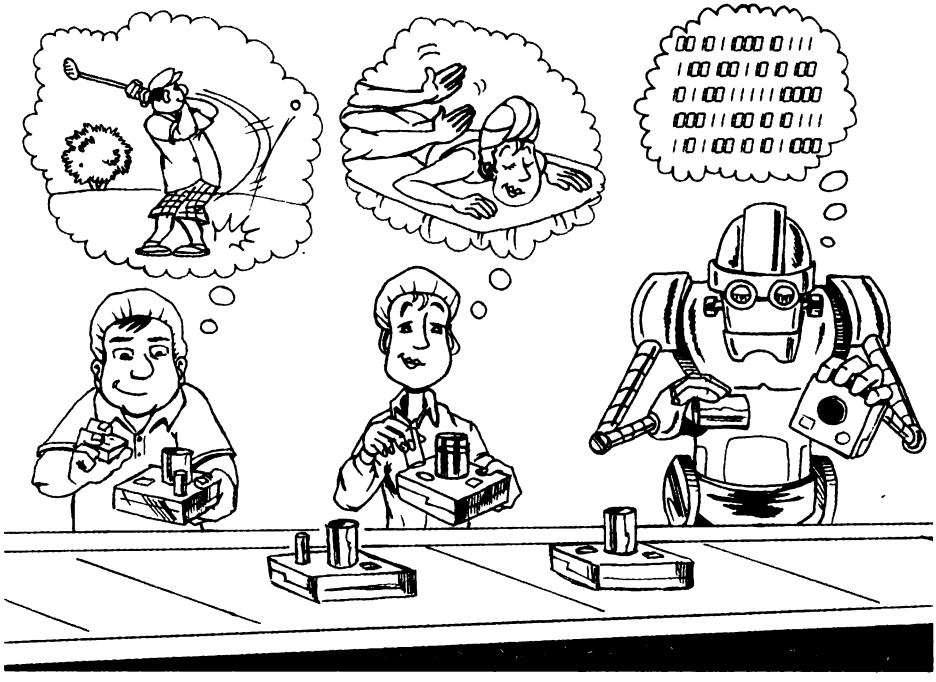
В этом решении мы рассмотрели задачу с другой точки зрения. Оба варианта решения очень удобно читать, поскольку читателю приходится думать только об одной задаче в любой момент времени.

Итог

В этой главе иллюстрируется простой прием организации вашего кода: **выполняйте только одно задание в любой момент**.

Если у вас есть трудночитаемый код, попробуйте перечислить все задания, которые он выполняет. Некоторые из них вполне могут стать отдельными функциями (или классами). Другие же могут стать логическими «абзацами» внутри одной функции. Тонкости того, как вы реализуете подобное разделение, важны не так, как сам факт их разделения. Сложность заключается в том, чтобы описать все мелкие задачи, которые решает программа.

12 Превращаем мысли в код



> Если вы можете что-то объяснить своей *бабушке*, значит, вы действительно это хорошо поняли.
>
> *Альберт Эйнштейн*

Когда вы объясняете кому-нибудь сложную идею, ничего не стоит запутать этого человека обилием мелких деталей. Умение объяснить свою идею простым языком, чтобы менее образованный слушатель мог вас понять, — это очень ценный навык. Для этого требуется выделить в идее лишь самые важные концепции, что позволит не только помочь другому человеку понять ее, но и самому представить ее более четко.

Этот же навык требуется и при представлении вашего кода читателю. Мы считаем, что исходный код — это главный способ объяснить читателю, что именно делает программа. Поэтому код должен быть написан «простым языком».

В этой главе мы рассмотрим несложный процесс, чтобы помочь вам создать более прозрачный код:

- опишем простым языком, что именно должен делать код; так же, как если бы вы объясняли это вашему коллеге;
- обратим внимание на ключевые слова и фразы, использованные в этом описании;
- напишем код, соответствующий описанию.

Четко описываем логику

Рассмотрим фрагмент кода веб-страницы, написанный на языке PHP. Это код верхней части защищенной страницы. Он проверяет, авторизован ли пользователь и имеет ли он право просматривать эту страницу. Если нет, то код перенаправляет посетителя на страницу, которая сообщает ему о том, что он не авторизован:

```
$is_admin = is_admin_request();
if ($document) {
    if (!$is_admin && ($document['username'] != $_SESSION['username'])) {
        return not_authorized();
    }
} else {
    if (!$is_admin) {
        return not_authorized();
       }
}

// продолжаем отображать страницу ...
```

В этом коде довольно много логических конструкций. Как было показано в ч. II данной книги, большие логические деревья, такие как это, сложны для понимания.

Возможно, логику этого кода можно упростить, но каким образом? Опишем ее простым и понятным языком:

```
Есть две ситуации, в которых вы можете быть авторизованы:
1) вы администратор;
2) вы – владелец текущего документа (если таковой существует).
В противном случае вы не авторизованы.
```

Рассмотрим альтернативную реализацию, создание которой было подсказано этим описанием:

```
if (is_admin_request()) {
    // авторизован
} elseif ($document && ($document['username'] == $_SESSION['username'])) {
    // авторизован
} else {
    return not_authorized();
}

// продолжаем отображать страницу ...
```

Эта версия несколько отличается от предыдущей, поскольку в ней есть два пустых блока. Но код стал короче, а логика проще, поскольку мы избавились от отрицания. (Предыдущая реализация содержит три «не».) Кроме того, стала более понятна нижняя строка.

Библиотеки нам помогут

Однажды у нас был веб-сайт, который имел возможность показывать пользователю полезные подсказки вроде:

```
Подсказка: Авторизуйтесь для того, чтобы увидеть ваши последние запросы.
[Покажите мне еще одну подсказку!]
```

Всего имелось несколько десятков подсказок, все они были спрятаны внутри кода HTML:

```
<div id="tip-1" class="tip"> Подсказка: Авторизуйтесь для того,
                    чтобы увидеть ваши последние запросы. </div>
<div id="tip-2" class="tip"> Подсказка: Щелкните на картинке для того,
                    чтобы ее увеличить. </div>
```

Когда пользователь посещал страницу, одна из этих подсказок (выбиралась случайно) становилась видимой, а остальные оставались скрытыми.

Если пользователь переходил по ссылке «Покажите мне еще одну подсказку!», появлялась следующая подсказка. Рассмотрим пример кода, реализующий эту особенность с использованием библиотеки jQuery языка JavaScript:

```
var show_next_tip = function () {
    var num_tips = $('.tip').size();
    var shown_tip = $('.tip:visible');

    var shown_tip_num = Number(shown_tip.attr('id').slice(4));
    if (shown_tip_num === num_tips) {
        $('#tip-1').show();
    } else {
        $('#tip-' + (shown_tip_num + 1)).show();
    }
    shown_tip.hide();
};
```

Этот код хорош, но его можно улучшить. Начнем с того, что опишем словами, что именно должен делать код:

```
Находим подсказку, которая видна в данный момент, и скрываем ее.
Находим следующую за ней подсказку и показываем ее.
Если подсказки закончились, при помощи цикла возвращаемся к первой.
```

Приведем другую реализацию решения задачи, основанную на этом описании:

```
var show_next_tip = function () {
    var cur_tip = $('.tip:visible').hide();  // Находим подсказку, которая
                                             // видна в данный момент, и скрываем ее.
    var next_tip = cur_tip.next('.tip');     // Находим следующую за ней
                                             // подсказку и показываем ее.
    if (next_tip.size() === 0) {             // Если подсказки закончились,
        next_tip = $('.tip:first');          //     при помощи цикла
                                             // возвращаемся к первой.
    }
    next_tip.show();                         // Показываем новую подсказку.
};
```

Этот вариант решения содержит меньше строк кода и не оперирует числовыми значениями напрямую. Он учитывает, что читатель может подумать об этом коде.

В данном случае нам помогло то, что в библиотеке jQuery есть метод `.next()`, который мы можем здесь использовать. Часть написанного кода рассчитана на использование методов, присутствующих в вашей библиотеке.

Применяем этот метод к более объемным задачам

В предыдущих примерах мы применяли наш процесс к небольшим блокам кода. В следующем примере мы используем его в объемной функции. Как видите, этот метод дает возможность разбить код, помогая тем самым определить, от каких элементов можно избавиться.

Представьте, что у нас есть система, которая записывает результаты продаж акций. Каждая транзакция состоит из четырех полей:

- `time` (точные дата и время покупки);
- `ticker_symbol` (символ биржевого тикера, например GOOG, символ Google в номенклатуре «Насдак»);
- `price` (например, $600);
- `number_of_shares` (количество акций, например 100).

По какой-то странной причине данные распределяются между тремя отдельными таблицами, что проиллюстрировано на рис. 12.1. В каждой базе данных уникальным первичным ключом является параметр `time`.

time	ticker_symbol
3:45	IBM
3:59	IBM
4:30	GOOG
5:20	AAPL
6:00	MSFT

time	price
3:45	$120
4:30	$600
5:00	$25
5:20	$200
6:00	$25

time	number_of_shares
3:45	50
3:59	200
4:10	75
4:30	100
5:20	80

Рис. 12.1

Теперь необходимо написать программу, которая объединит три таблицы в одну (результат, эквивалентный операции SQL `JOIN`). Эта задача должна быть простой, поскольку все строки таблицы отсортированы по времени, но, к сожалению, некоторых строк не хватает. Вы хотите найти все тройки строк, в которых совпадает параметр `time`, и проигнорировать остальные, как показано на рис. 12.1.

Рассмотрим код, написанный на языке Python, который находит соответствующие строки:

```
def PrintStockTransactions():
    stock_iter = db_read("SELECT time, ticker_symbol FROM ...")
    price_iter = ...
    num_shares_iter = ...

    # Параллельно проходим по строкам каждой таблицы.
    while stock_iter and price_iter and num_shares_iter:
        stock_time = stock_iter.time
        price_time = price_iter.time
        num_shares_time = num_shares_iter.time

        # Если в трех строках время не одинаково, то пропускаем самую
        # «старую» строку. Обратите внимание: операция "<=" внизу
        # может быть заменена операцией "<" в том случае, если существует
        # два совпадения из трех.
        if stock_time != price_time or stock_time != num_shares_time:
            if stock_time <= price_time and stock_time <= num_shares_time:
                stock_iter.NextRow()
```

```
        elif price_time <= stock_time and price_time <= num_shares_time:
            price_iter.NextRow()
        elif num_shares_time <= stock_time and num_shares_time <=
                                                    price_time:
            num_shares_iter.NextRow()
        else:
            assert False  # невозможно
        continue

    assert stock_time == price_time == num_shares_time

    # Печатаем выровненные строки.
    print "@", stock_time,
    print stock_iter.ticker_symbol,
    print price_iter.price,
    print num_shares_iter.number_of_shares

    stock_iter.NextRow()
    price_iter.NextRow()
    num_shares_iter.NextRow()
```

Этот пример кода работает, однако особенно много действий происходит, когда цикл пропускает неподходящие строки. Возникает вопрос: может ли этот цикл случайно пропустить строки? Может ли он попытаться считать информацию уже после конца потока любого из итераторов?

Как можно сделать этот код более удобочитаемым?

Описание решения задачи на русском языке

Давайте отступим на шаг назад и опишем простым языком то, что мы пытаемся сделать:

```
Мы параллельно читаем значения, на которые указывают три итератора,
проходящие по строкам таблицы.
Если эти строки не выстроены, передвинем их таким образом,
чтобы исправить это.
Далее напечатаем выровненные строки, а затем передвинем их снова.
Повторяем вышеописанные операции, пока не закончатся повторяющиеся строки.
```

Обращаясь к оригинальному коду, можно сказать, что самым неаккуратным фрагментом выглядит блок, решающий задачу «передвинем их таким образом, чтобы исправить это». Чтобы сделать код более понятным, выделим всю эту логику в новую функцию, которая будет называться `AdvanceToMatchingTime()`.

Рассмотрим новую версию кода, в которой используется наша новая функция:

```
def PrintStockTransactions():
    stock_iter = ...
```

```
price_iter = ...
num_shares_iter = ...

while True:
    time = AdvanceToMatchingTime(stock_iter, price_iter,
                                 num_shares_iter)
    if time is None:
        return

    # Печатаем выровненные строки.
    print "@", time,
    print stock_iter.ticker_symbol,
    print price_iter.price,
    print num_shares_iter.number_of_shares

    stock_iter.NextRow()
    price_iter.NextRow()
    num_shares_iter.NextRow()
```

Как видите, этот код гораздо проще для понимания, поскольку мы скрыли все детали процесса выстраивания строк.

Рекурсивное применение методов

Несложно представить, как можно было бы написать код функции AdvanceToMatchingTime(), — в худшем случае он напоминал бы тот некрасивый блок из первой версии:

```
def AdvanceToMatchingTime(stock_iter, price_later, num_shares_iter):
    # Параллельно проходим по строкам каждой таблицы.
    while stock_iter and price_iter and num_shares_iter:
        stock_time = stock_iter.time
        price_time = price_iter.time
        num_shares_time = num_shares_iter.time

        # Если в трех строках время не одинаково, то пропускаем самую
                                                    «старую» строку
        # Обратите внимание: операция "<=" внизу может быть заменена
                        операцией "<" в том случае, если существует
                        два совпадения из трех.
        if stock_time != price_time or stock_time != num_shares_time:
            if stock_time <= price_time and stock_time <= num_shares_time:
                stock_iter.NextRow()
            elif price_time <= stock_time and price_time <= num_shares_time:
                price_iter.NextRow()
            elif num_shares_time <= stock_time and num_shares_time <=
                                                          price_time:
                num_shares_iter.NextRow()
            else:
                assert False  # невозможно
```

```
            continue

        assert stock_time == price_time == num_shares_time
        return stock_time
```

Теперь улучшим этот код, добавив нашу функцию к функции AdvanceToMatching-Tine(). Приведем описание того, что делает наша функция:

```
Сравниваем времена каждой текущей строки: если они совпадают,
 то переходим дальше.
В противном случае передвигаем более «ранние» строки вперед.
Продолжаем, пока не выровняем все строки (или пока не подойдет
 к концу один из циклов).
```

Это описание гораздо более прозрачное и красивое, чем рассмотренный ранее код. Обратите внимание на следующее обстоятельство: в описании не упоминаются stock_iter или другие детали нашей задачи. Это означает, что мы можем переименовать переменные так, чтобы их имена стали звучать более просто и универсально. Рассмотрим получившийся код:

```
def AdvanceToMatchingTime(row_iter1, row_iter2, row_iter3):
    while row_iter1 and row_iter2 and row_iter3:
        t1 = row_iter1.time
        t2 = row_iter2.time
        t3 = row_iter3.time

        if t1 == t2 == t3:
            return t1

        tmax = max(t1, t2, t3)

        # Более «ранние» строки передвигаются вперед.
        # В конечном счете этот цикл упорядочит их.
        if t1 < tmax: row_iter1.NextRow()
        if t2 < tmax: row_iter2.NextRow()
        if t3 < tmax: row_iter3.NextRow()

    return None  # совпадающих строк нет
```

Как видите, этот код после переработки выглядит понятнее. Алгоритм стал проще, и теперь в программе выполняется меньше запутанных сравнений. Также мы использовали более короткие имена вроде t1, и нам больше не нужно учитывать конкретные столбцы базы данных, занятые в решении задачи.

Итог

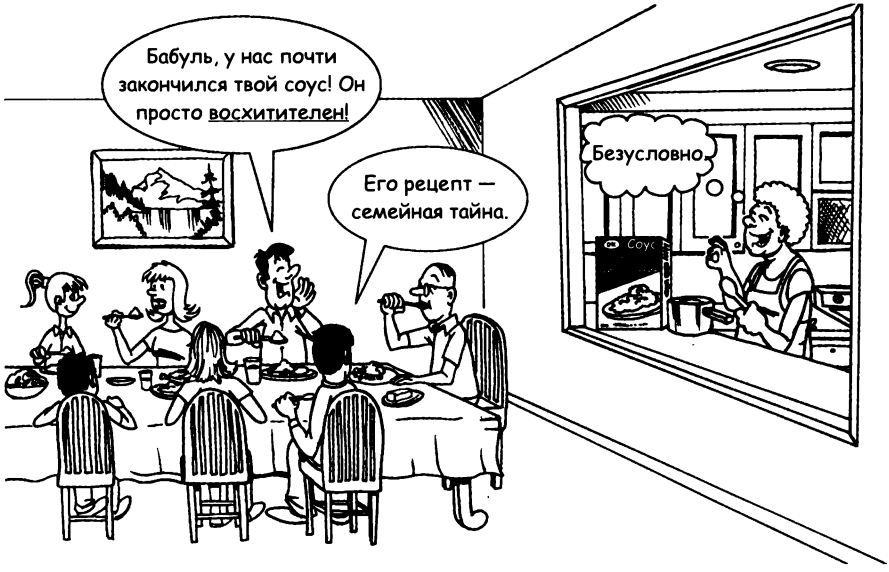
В этой главе был рассмотрен простой метод описания программы естественным языком и использование такого описания для создания более стройного кода. Данный прием обманчиво прост, но потенциал его огромен. Обращая внимание на

слова, присутствующие в вашем описании, обычно легче определить, какие подзадачи следует разбивать на фрагменты.

Процесс описания простым языком можно применять не только при написании кода. Например, политика одной из компьютерных лабораторий гласит, что в тех случаях, когда студенту требуется помощь в отладке программы, он сначала должен объяснить свою проблему плюшевому мишке, который сидит в углу. Удивительно, но простое описание проблемы вслух часто помогает студенту придумать решение. Этот прием называется «метод утенка».

Иначе говоря, если вы не можете описать проблему словами, значит, возможно, вы что-то упустили или не определили. Описание словами программы (или идеи) действительно помогает претворить ее в жизнь.

13 Пишите меньше кода



Понимание того, когда *не нужно* писать код, возможно, самый главный навык, которым может овладеть программист. Каждую строку кода, которую вы пишете, следует тестировать и обслуживать. Повторное использование библиотек или исключение из кода некоторых функций позволяет сэкономить время и сохранять базу кода компактной и эффективной.

ОСНОВНАЯ ИДЕЯ

Наиболее читабельный код — тот, который не написан.

Не беспокойтесь о реализации этой функции — она вам не понадобится

Когда вы начинаете новый проект, вполне естественно думать о различных «изюминках» вашей программы. Но программисты, как правило, переоценивают количество особенностей своего проекта, которые будут *действительно необходимы*. Множество таких наработок часто остаются незаконченными, ненужными, а порой просто усложняют проект.

Программисты также часто недооценивают усилия, которые потребуется приложить, чтобы реализовать ту или иную функцию. Мы оптимистично планируем время, требуемое для реализации сырого прототипа, но забываем о том, сколько времени придется затратить на будущую поддержку, написание документации и работу с разрастанием базы кода.

Критикуйте и разделяйте ваши требования

Не все программы должны быть быстрыми, на 100 % правильными или способными обработать любой мыслимый ввод. Если вы действительно тщательно изучите ваши требования, то сможете иногда «извлечь» из них более простую задачу, которая потребует написания меньшего количества кода. Рассмотрим такую ситуацию на примере.

Пример: поиск магазинов

Представьте, что вам необходимо написать программу для бизнеса, которая ищет магазины. Вы думаете, что требования к программе таковы: для любых заданных пользователем координат по широте и долготе найти ближайший магазин.

Чтобы реализовать это на 100 % правильно, следует обработать следующие ситуации:

- магазин находится за линией перемены даты;
- магазин располагается недалеко от Северного или Южного полюса;
- подстройка под кривизну земли при изменении параметра «градусы широты на километр».

Обработка всех этих ситуаций потребует написания огромного количества кода.

Однако в штате Техас всего 30 магазинов, которые может обнаружить приложение. Для этого менее крупного региона проблемы, описанные выше, не так важны. В результате вы можете сократить все требования до одного: *обнаружить приблизительное местоположение ближайшего магазина для пользователей, живущих в районе Техаса.*

Задача становится проще, поскольку теперь вы можете просто пройти по списку магазинов в цикле и вычислить евклидово расстояние, используя заданные координаты.

Пример: добавление кэша

Однажды мы разрабатывали приложение на языке Java, которое часто считывало объекты с диска. Скорость приложения была ограничена этими считываниями, поэтому мы хотели реализовать кэш для объектов. Типичная последовательность таких считываний выглядела следующим образом:

```
read Object A
read Object A
read Object A
read Object B
read Object B
read Object C
read Object D
read Object D
```

Как видите, в ней есть повторяющиеся обращения к одному и тому же объекту, поэтому кэширование нам определенно бы пригодилось.

Когда мы столкнулись с этой проблемой, то решили реализовать кэш, который удаляет объекты, использовавшиеся наиболее давно. У нас не было его готовой реализации, поэтому нам пришлось писать код кэша самостоятельно. Для нас это не составило труда, поскольку мы реализовали похожую структуру данных ранее (она содержала хэш-таблицу и однонаправленный список — примерно 100 строк кода).

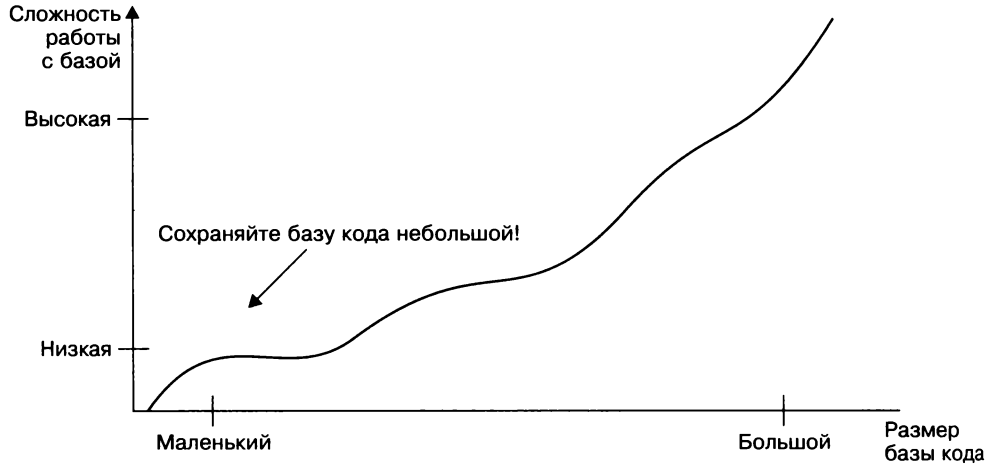
Однако мы заметили, что повторяющиеся запросы всегда следуют друг за другом. Поэтому вместо того, чтобы воплощать наш первоначальный вариант кэша, мы реализуем кэш объемом в один элемент:

```
DiskObject lastUsed;  // член класса

DiskObject lookUp(String key) {
    if (lastUsed == null || !lastUsed.key().equals(key)) {
        lastUsed = loadDiskObject(key);
    }

    return lastUsed;
}
```

Написав меньше кода, мы получили 90 % того, что планировали реализовать, а программа стала занимать меньше места в памяти.

Пользу от уменьшения требований и решения более простых задач трудно переоценить. Требования могут перекрывать друг друга, хотя порой это бывает неочевидно. Это значит, что решение половины задачи может потребовать лишь четверти усилий.

Сохраняйте базу кода небольшой



Когда вы только приступаете к проекту, у вас есть лишь один или два файла исходников, и все просто замечательно. Код компилируется и запускается моментально, в него легко вносить изменения. Запомнить, где находится определение каждой функции и класса, также несложно.

Но с ростом проекта ваш каталог заполняется все большим количеством исходных файлов. Вскоре вам требуется несколько каталогов для того, чтобы организовать все файлы. Становится все сложнее запомнить, какие функции вызываются другими функциями, а отслеживание ошибок требует все больше усилий.

В конечном итоге вы имеете массу исходного кода, разбросанного по множеству различных каталогов. Проект огромен, и никто в отдельности не в состоянии его понять. Добавлять новые функции оказывается крайне сложно, а работа с кодом становится неудобной и неприятной.

Выше мы описали естественный закон Вселенной — с ростом сложности системы сложность скрепляющих ее материалов растет еще быстрее.

Лучший способ избежать этого — **сохранять базу кода максимально компактной**, даже если проект разрастается. По этой причине необходимо:

- писать как можно больше вспомогательного кода, который позволит избавиться от дублирующегося кода (см. гл. 10);
- удалять неиспользуемый код или бесполезные функции (см. далее);
- разделять проект на автономные подпроекты.

В общем, учитывайте «вес» вашей базы кода; сохраняйте ее небольшой и легкой.

УДАЛЯЕМ НЕИСПОЛЬЗУЕМЫЙ КОД

Садовники часто подрезают растения, чтобы они продолжали жить и расти. Подобный принцип можно применить и к неиспользуемому коду.

Если код уже написан, программисты часто неохотно удаляют его, поскольку в него было вложено немало сил. Удалить его означает признать, что время на его создание было потрачено зря. Что же, смиритесь с этим! Вы работаете в творческой области — фотографы, писатели и режиссеры также сохраняют не все свои труды.

Удаление изолированных функций проходит безболезненно, но иногда якобы «неиспользуемый код» на самом деле применяется во всем вашем проекте и вы можете даже не догадываться об этом. Вот несколько примеров.

- Ваша система с недавнего времени поддерживает обработку международных имен файлов, и теперь текст программы засорен кодом, выполняющим эти функции. Однако этот код не полностью функционален, и ваше приложение все равно никогда не используется для работы с международными именами файлов.

  Почему бы не убрать эту функциональность?

- Вы хотели, чтобы ваша программа работала даже в тех случаях, когда в системе закончится память, поэтому реализовали большое количество умной логики, которая пытается восстановить работу программы в таких ситуациях. Эта идея была хороша, но на практике в случаях, когда у системы заканчивается память, программа все равно становится нестабильным зомби — все основные особенности невозможно использовать, а программа готова рухнуть от одного щелчка кнопкой мыши.

  Почему бы просто не завершить работу программы с выводом сообщения: «Извините, в системе закончилась память» и не удалить весь код, на который не хватает памяти?

Старайтесь изучать возможности доступных библиотек

В большинстве случаев программисты не знают о том, что уже существующие библиотеки могут помочь решить их задачу. Или иногда забывают, что делает та или иная библиотека. Важно знать возможности библиотечного кода. Это позволит извлекать максимальную пользу от работы с библиотеками.

Наш совет: **потратьте 15 минут на чтение имен всех функций, модулей и типов вашей стандартной библиотеки, причем делайте это довольно часто**. Сюда относятся стандартная библиотека шаблонов языка C++ (STL), Java API, встроенные модули языка Python и пр.

Цель заключается не в том, чтобы запомнить всю библиотеку. Вам лишь следует знать, какие функции доступны, чтобы в следующий раз при работе над новым кодом у вас могла возникнуть мысль: «Подождите, это звучит очень знакомо. Кажется, я видел что-то подобное в API...» Мы убеждены, что ощутимый результат появится очень быстро и вы уже с самого начала работы будете прибегать к помощи знакомых библиотек.

Пример: списки и множества в языке Python

Допустим, вы программируете на языке Python. У вас есть список (например `[2, 1, 2]`), и вы хотите получить список уникальных элементов (в этом случае `[2, 1]`). Вы можете решить эту задачу, используя словарь, в котором содержится список гарантированно уникальных ключей:

```
def unique(elements):
    temp = {}
    for element in elements:
        temp[element] = None  # The value doesn't matter.
    return temp.keys()

unique_elements = unique([2,1,2])
```

Но вместо этого просто можно использовать менее известный тип `set`:

```
unique_elements = set([2,1,2])  # удалить повторяющиеся значения
```

По данному объекту можно проходить итератором так же, как и по обычному списку. Если вы захотите, чтобы новый список имел тип `list`, то можете написать код следующим образом:

```
unique_elements = list(set([2,1,2]))  # удалить повторяющиеся значения
```

Очевидно, что `set` — это инструмент, отлично подходящий для решения такой задачи. Но если бы вы о нем не знали, вы могли бы написать код вроде функции `unique()`, которую мы рассмотрели выше.

Почему многократное использование библиотек так полезно

Часто приводится следующая статистика: средний инженер-программист пишет в день десять строк кода, готовых к сдаче. Когда программисты впервые слышат это, они недоверчиво говорят: «Десять строк кода? Да я могу написать их за минуту!»

Ключевая фраза здесь — *«готовых к сдаче»*. Каждая строка кода в крупных библиотеках представляет собой результат серьезной работы по проектированию, отладке, рефакторингу, документированию, оптимизации и тестированию. Каждая строка кода, прошедшая такой естественный отбор, очень ценна. Именно поэтому многократное использование библиотек крайне полезно — так можно сэкономить время и написать меньше кода.

Пример: использование инструментов UNIX вместо написания кода

Если веб-сервер часто возвращает коды HTTP-откликов вроде 4xx или 5xx, то это может означать потенциальное наличие проблемы (4xx возвращается при ошибке клиента, 5xx — при ошибке сервера). Допустим, мы хотим написать программу,

которая анализирует журнал доступа к веб-серверу и определяет, какой из URL вызывает наибольшее количество ошибок.

Журналы доступа обычно выглядят примерно так:

```
1.2.3.4 example.com [24/Aug/2010:01:08:34] "GET /index.html HTTP/1.1" 200 ...
2.3.4.5 example.com [24/Aug/2010:01:14:27] "GET /help?topic=8 HTTP/1.1" 500 ...
3.4.5.6 example.com [24/Aug/2010:01:15:54] "GET /favicon.ico HTTP/1.1" 404 ...
```

Часто они содержат строки такой формы:

```
browser-IP host [date] "GET /url-path HTTP/1.1" HTTP-response-code ...
```

Программа для поиска пути `url-path`, в котором наиболее часто встречаются коды ответов вида 4xx или 5xx, будет занимать 20 строк кода на языке C++ или Java.

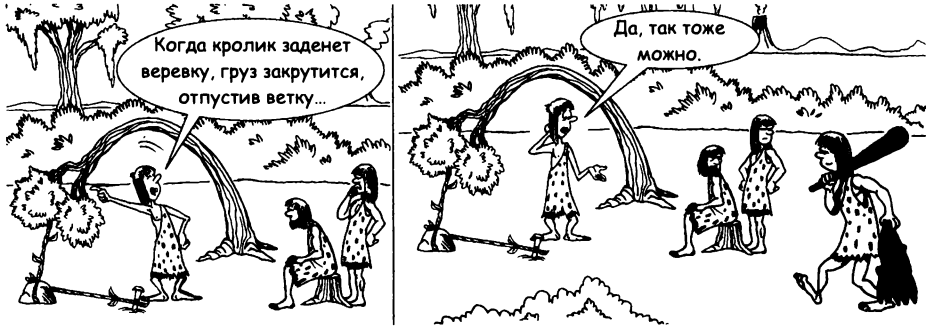
Вместо этого в ОС UNIX можно ввести в командную строку следующую информацию:

```
cat access.log | awk '{ print $5 " " $7 }' | egrep "[45]..$" \
| sort | uniq -c | sort -nr
```

В результате на экране появится следующее:

```
95 /favicon.ico 404
13 /help?topic=8 500
11 /login 403
...
<count> <path> <http response code>
```

Особенно замечательно в этой команде то, что мы избежали написания «реального» кода и его проверки.



Итог

> Приключения, азарт — Джедаи не интересуются такими вещами.
>
> *Мастер Йода*

Цель этой главы — убедить вас писать как можно меньше нового кода. Каждая новая строка кода должна быть протестирована, задокументирована и обслужена.

Чем больше кода в вашей базе, чем «тяжелее» она становится и тем труднее с ней работать.

Избавиться от написания нового кода можно следующими методами:

- исключать некритические особенности из продукта и, так сказать, «не перехимичивать»;
- многократно обдумывать поставленные требования. Это делается для того, чтобы решить самую простую возможную задачу и при этом реализовать первоначальный замысел;
- изучать стандартные библиотеки (и периодически их перечитывать).

ЧАСТЬ IV

Избранные темы

В трех предыдущих частях мы рассмотрели множество приемов, позволяющих сделать код простым для понимания. В этой части мы применим некоторые из них в двух выбранных нами областях.

Сначала обсудим тестирование — научимся писать тесты, которые будут как эффективными, так и читаемыми.

Затем рассмотрим проектирование и реализацию специальной структуры данных (счетчик минут и часов). В этом примере совмещаются производительность, хороший дизайн и читаемость.

14 Тестирование и читаемость

В этой главе мы покажем вам простые приемы, которые позволят написать качественные и эффективные тесты.

Для различных людей тестирование означает разные вещи. В этой главе мы будем использовать слово «тест» для обозначения любого кода, единственным предназначением которого является проверка поведения другого («настоящего») фрагмента кода. Мы сосредоточимся на читаемости тестов, но не будем углубляться в изучение парадигмы, при которой тестирующий код пишется раньше рабочего (такая ситуация называется разработкой через тестирование), а также не станем говорить о других философских аспектах разработки тестов.

Создавайте тесты, которые легко читать и обслуживать

Тестирующий код должен быть читаемым, так же как и нетестирующий. Другие программисты часто рассматривают тестирующий код как неофициальную документацию, в которой описано, как работает настоящий код, а также способ его использования. Если тест легко прочесть, то пользователь сможет лучше понять поведение настоящего кода.

ОСНОВНАЯ ИДЕЯ

Тестирующий код должен быть читаемым. Это упростит другим программистам процесс изменения или добавления тестов.

Когда тестирующий код огромен и выглядит страшно, может случиться следующее.

- **Программисты побоятся изменять настоящий код.** *Мы не хотим связываться с этим кодом — обновление всех его тестов превратится в кошмар!*
- **Программисты не добавят новые тесты при добавлении нового кода.** С течением времени все меньшая и меньшая часть вашего модуля будет протестирована, и вы более не будете уверены в том, что знаете, как работает ваша программа.

Вместо этого вам захочется убедить пользователей вашего кода (а особенно самих себя!) в том, что с тестирующим кодом будет комфортно работать. У пользователей должна быть возможность понять, почему новое изменение кода не согласуется с существующим тестом. Они также должны ощущать, что добавить новые тесты будет несложно.

Что не так с этим тестом?

В нашей базе кода есть функция, созданная для сортировки и фильтрации списка оцененных результатов поиска. Вот ее объявление:

```
// Сортируем 'docs' по оценке (начиная с самой высокой)
// и удаляем документы с отрицательными оценками.
void SortAndFilterDocs(vector<ScoredDocument>* docs);
```

Тест для этой функции поначалу выглядел следующим образом:

```
void Test1() {
    vector<ScoredDocument> docs;
    docs.resize(5);
    docs[0].url = "http://example.com";
    docs[0].score = -5.0;
    docs[1].url = "http://example.com";
    docs[1].score = 1;
    docs[2].url = "http://example.com";
    docs[2].score = 4;
    docs[3].url = "http://example.com";
    docs[3].score = -99998.7;
    docs[4].url = "http://example.com";
    docs[4].score = 3.0;

    SortAndFilterDocs(&docs);

    assert(docs.size() == 3);
    assert(docs[0].score == 4);
    assert(docs[1].score == 3.0);
    assert(docs[2].score == 1);
}
```

В этом тестирующем коде есть как минимум *восемь* различных недочетов. К концу этой главы вы сможете определить и исправить их все.

Приводим тест в читаемый вид

Вооружитесь следующим основным принципом проектирования — **нужно скрывать наименее важные детали от пользователя, а самые важные детали делать наиболее заметными**.

Очевидно, тестирующий код из предыдущего примера противоречит этому принципу. Каждая деталь теста находится в центре внимания. В большей части кода используются свойства `url`, `score` и вектор `docs[]`, хотя они просто описывают подробности процесса, в ходе которого базовые объекты C++ получают свои значения. Эти свойства не позволяют составить целостного впечатления об общей функциональности данного теста.

В качестве первого шага очистки кода мы создадим вспомогательную функцию вида:

```
void MakeScoredDoc(ScoredDocument* sd, double score, string url) {
    sd->score = score;
    sd->url = url;
}
```

С использованием этой функции код станет немного более компактным:

```
void Test1() {
    vector<ScoredDocument> docs;
```

```
    docs.resize(5);
    MakeScoredDoc(&docs[0], -5.0, "http://example.com");
    MakeScoredDoc(&docs[1], 1, "http://example.com");
    MakeScoredDoc(&docs[2], 4, "http://example.com");
    MakeScoredDoc(&docs[3], -99998.7, "http://example.com");

}
```

Однако код все еще недостаточно хорош — остаются заметны малозначимые детали. Например, параметр "http://example.com" лишь усложняет восприятие. Он никогда не изменяется, а точный URL не имеет значения — он нужен лишь для того, чтобы заполнить валидный ScoredDocument.

Мы также вынуждены рассматривать следующие незначительные детали: docs.resize(5), &docs[0], &docs[1] и пр. Расширим функционал нашей вспомогательной функции и назовем ее AddScoreDoc():

```
void AddScoredDoc(vector<ScoredDocument>& docs, double score) {
    ScoredDocument sd;
    sd.score = score;
    sd.url = "http://example.com";
    docs.push_back(sd);
}
```

С использованием этой функции наш код станет еще более компактным:

```
void Test1() {
    vector<ScoredDocument> docs;
    AddScoredDoc(docs, -5.0);
    AddScoredDoc(docs, 1);
    AddScoredDoc(docs, 4);
    AddScoredDoc(docs, -99998.7);

}
```

Этот код выглядит лучше, но все еще не является удобочитаемым и высококачественным тестом. Если вы захотите добавить новый тест с новым набором оцененных документов, то потребуется копировать и вставлять много кода. Как же нам еще улучшить этот тест?

Создание минимального тестового выражения

Чтобы улучшить этот тестирующий код, используем прием, рассмотренный в гл. 12. Опишем, что должен делать наш тест, простым языком:

```
У нас есть список документов с оценками [-5, 1, 4, -99998.7, 3]. После выполнения
функции SortAndFilterDocs() оставшиеся документы должны иметь оценки [4, 3, 1],
именно в таком порядке.
```

Как видите, в этом описании мы не упомянули vector<ScoredDocument>. Самым важным здесь является массив оценок. В идеале наш тестирующий код должен выглядеть примерно так:

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3",  "4, 3, 1");
```

Мы смогли извлечь основную идею этого теста и поместить ее в одну строку кода!

Следует отметить, что это довольно обычная ситуация. Суть большинства тестов сводится к тому, чтобы *проверить присутствие заданного поведения или возвращаемого значения функции в заданной ситуации или заданных входных параметрах*. Во многих случаях эта цель может быть описана одной строкой кода. В дополнение к приведению кода в очень компактный и читаемый вид старайтесь создавать короткие тестовые операторы, что позволит вам с легкостью добавлять новые тестовые ситуации.

Реализуем пользовательские мини-языки

Обратите внимание на то, что функция `CheckScoreBeforeAfter()` принимает два строковых аргумента, описывающих массив оценок. В более поздних версиях C++ можно передавать массивы в функцию следующим образом:

```
CheckScoresBeforeAfter({-5, 1, 4, -99998.7, 3}, {4, 3, 1});
```

Поскольку мы не могли поступить так в то время, помещаем разделенные запятыми оценки внутрь строк. При таком подходе нам придется преобразовывать в функции `CheckScoreBeforeAfter()` подобные строковые аргументы.

Определение пользовательского мини-языка — это мощный способ сообщить обширное количество информации в небольшом пространстве. Другие примеры могут быть связаны с использованием функций `printf()` и библиотек регулярных выражений.

В нашем случае написание вспомогательных функций, преобразующих разделенный запятыми список чисел, — не слишком сложная задача. Вот так будет выглядеть функция `CheckScoreBeforeAfter()`:

```
void CheckScoresBeforeAfter(string input, string expected_output) {
    vector<ScoredDocument> docs = ScoredDocsFromString(input);
    SortAndFilterDocs(&docs);
    string output = ScoredDocsToString(docs);
    assert(output == expected_output);
}
```

Для полноты картины добавим вспомогательные функции, преобразующие строку к типу `vector<ScoredDocument>`:

```
vector<ScoredDocument> ScoredDocsFromString(string scores) {
    vector<ScoredDocument> docs;

    replace(scores.begin(), scores.end(), ',', ' ');

    // Получаем 'docs' из строки, содержащей разделенные пробелами оценки.
    istringstream stream(scores);
    double score;
```

```
    while (stream >> score) {
        AddScoredDoc(docs, score);
    }

    return docs;
}

string ScoredDocsToString(vector<ScoredDocument> docs) {
    ostringstream stream;
    for (int i = 0; i < docs.size(); i++) {
        if (i > 0) stream << ", ";
        stream << docs[i].score;
    }

    return stream.str();
}
```

На первый взгляд может показаться, что здесь слишком много кода, но его возможности очень велики. Поскольку можно провести тест при помощи всего лишь одного вызова функции `CheckScoresBeforeAfter()`, вы можете добавить больше тестов (что мы и сделаем далее в этой главе).

Сообщения об ошибках также должны быть более читаемыми

Предыдущий код был хорош, но что случится, если строка `assert(output == expected_output)` выдаст ошибку? Появится следующее сообщение:

```
Assertion failed: (output == expected_output),
    function CheckScoresBeforeAfter, file test.cc, line 37.
```

Очевидно, если вы увидите эту ошибку, вам захочется узнать, каковы значения параметров `output` и `expected_output`?

Использование оптимизированных версий функции assert()

К счастью, большинство языков и библиотек имеет более сложные версии функции `assert()`, которые вы вполне можете использовать. Поэтому вместо того, чтобы написать:

```
assert(output == expected_output);
```

можно использовать аналог из библиотеки Boost C++:

```
BOOST_REQUIRE_EQUAL(output, expected_output)
```

Теперь, если тест закончится неудачей, вы получите более подробное сообщение:

```
test.cc(37): fatal error in "CheckScoresBeforeAfter": critical check
    output == expected_output failed ["1, 3, 4" != "4, 3, 1"]
```

Это гораздо более полезное сообщение.

Используйте подобные полезные методы с утверждением (assertion methods) при любом удобном случае. Такая практика оправдывается всякий раз, когда тест завершается неудачей.

БОЛЕЕ ПОЛЕЗНЫЕ АНАЛОГИ ФУНКЦИИ ASSERT() В ДРУГИХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

В языке Python встроенное выражение `assert a== b` выводит на экран обычное сообщение об ошибке:

```
File "file.py", line X, in <module>
    assert a == b
AssertionError
```

Вместо этого вы можете использовать метод `assertEqual()`, который находится в модуле `unittest`:

```
import unittest

class MyTestCase(unittest.TestCase):
    def testFunction(self):
        a = 1
        b = 2
        self.assertEqual(a, b)

if __name__ == '__main__':
    unittest.main()
```

Этот метод выводит следующее сообщение об ошибке:

```
File "MyTestCase.py", line 7, in testFunction
       self.assertEqual(a, b)
AssertionError: 1 != 2
```

С каким бы языком вы ни работали, наверняка найдется библиотека или фреймворк (например, XUnit), которые вам пригодятся. Оказывается, разбираться в библиотеках полезно!

Собственные сообщения об ошибках

Используя функцию BOOST_REQUIRE_EQUAL(), мы можем получить более аккуратное сообщение об ошибке:

```
output == expected_output failed ["1, 3, 4" != "4, 3, 1"]
```

Однако подобное сообщение также можно улучшить. Например, было бы полезно узнать, какие входные параметры вызвали подобную неудачу. Идеальное сообщение об ошибке выглядит примерно так:

```
CheckScoresBeforeAfter() failed,
  Input:           "-5, 1, 4, -99998.7, 3"
  Expected Output: "4, 3, 1"
  Actual Output:   "1, 3, 4"
```

Если оно идеально подходит для вас, напишите его самостоятельно!

```
void CheckScoresBeforeAfter(...) {


    if (output != expected_output) {
        cerr << "CheckScoresBeforeAfter() failed," << endl;
        cerr << "Input:           \"" << input << "\"" << endl;
        cerr << "Expected Output: \"" << expected_output << "\"" << endl;
        cerr << "Actual Output:   \"" << output << "\"" << endl;
        abort();
    }
```

Мораль заключается в том, что сообщение об ошибке должно быть максимально полезным. Иногда написание своего собственного сообщения путем построения пользовательского утверждения (custom assert) — лучший способ добиться этого.

Выбираем хорошие входные параметры

Выбор хороших входных параметров для ваших тестов — это в каком-то роде искусство. Те, что имеются у нас в данный момент, выглядят случайными:

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3",  "4, 3, 1");
```

Как же нам выбрать хорошие входные параметры? Подобные значения должны охватывать все возможные ситуации. Но они также должны быть простыми, чтобы их было легко прочесть.

ОСНОВНАЯ ИДЕЯ

Следует выбрать самый простой набор входных параметров, который может проверить код во всех возможных ситуациях.

Например, представьте, что мы выбрали следующие значения:

```
CheckScoresBeforeAfter("1, 2, 3", "3, 2, 1");
```

Хотя этот тест и простой, он не проверяет поведение функции `SortAndFilterDocs()` при появлении отрицательных оценок (не выполняется фильтрация). Если бы в этой части кода была ошибка, данные параметры не позволили бы ее распознать.

Следующий тест — это другая крайность:

```
CheckScoresBeforeAfter("123014, -1082342, 823423, 234205, -235235",
                       "823423, 234205, 123014");
```

Эти значения излишне сложны. (Они также охватывают не все возможные ситуации.)

Упрощаем входные параметры

Как же мы можем улучшить эти входные параметры?

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3",  "4, 3, 1");
```

Возможно, первое, что вы увидели, — очень «яркое» значение `-99998.7`. Это значение подбиралось на роль «какого-нибудь отрицательного числа», поэтому его можно заменить более простым `-1`. (Если бы оно подбиралось как «сверхмалое отрицательное число», лучше было бы использовать что-нибудь четкое, вроде `-1e100`.)

ОСНОВНАЯ ИДЕЯ

Отдавайте предпочтение прозрачным и простым параметрам, которые могут выполнить поставленную перед ними задачу.

Другие значения нашего теста не так плохи, но, поскольку мы взялись за их улучшение, мы можем уменьшить их до сверхмалых чисел. Нам также необходимо лишь одно отрицательное значение для того, чтобы протестировать удаление документов с отрицательными оценками. В результате получим новую версию нашего теста:

```
CheckScoresBeforeAfter("1, 2, -1, 3", "3, 2, 1");
```

Мы упростили тестовые значения, не снизив при этом их эффективности.

ОГРОМНЫЕ ТЕСТЫ

Тестирование кода огромными, безумными параметрами — это, несомненно, полезно. Например, возможно, вы используете тест вроде:

```
CheckScoresBeforeAfter("100, 38, 19, -25, 4, 84, [множество значений] ...",
                       "100, 99, 98, 97, 96, 95, 94, 93, ...");
```

Подобное количество входных параметров помогает отследить множество ошибок, например выход за пределы буфера или какие-либо прочие ошибки, о которых вы даже не догадываетесь.

Но подобный код выглядит пугающе и не совсем эффективен при тестировании кода под нагрузкой. Вместо этого гораздо более эффективным решением было бы создать множество входных параметров, например 100 000 значений.

Множественные тесты функциональности

Вместо того чтобы создать один «идеальный» набор входных параметров, который полностью протестирует код, часто создают несколько меньших по размеру тестов. Это простой, эффективный и более читаемый вариант.

Каждый тест должен указывать коду конкретное направление и пытаться отыскать какую-либо ошибку на отдельном участке логики. Например, рассмотрим четыре теста функции `SortAndFilterDocs()`:

```
CheckScoresBeforeAfter("2, 1, 3", "3, 2, 1");     // базовая сортировка
CheckScoresBeforeAfter("0, -0.1, -10", "0");      // удаляем все значения < 0
CheckScoresBeforeAfter("1, -2, 1, -2", "1, 1");   // входные параметры могут
                                                  // дублироваться
CheckScoresBeforeAfter("", "");                   // входных параметров может
                                                  // не быть
```

Вы можете создать еще больше тестов, если хотите очень уж скрупулезно проверить вашу программу. Написание нескольких отдельных тестов может помочь другому человеку, который будет работать с кодом. Если кто-нибудь случайно найдет ошибку, неудачный тест определить будет гораздо легче.

Называем тестовые функции

Тестирующий код обычно помещают в функции — по одной для каждого метода или ситуации, которые тестируют. Например, код, тестирующий функцию `SortAndFilterDocs()`, находился внутри функции с именем `Test1()`:

```
void Test1() {

}
```

Подбор хорошего имени для тестовой функции может показаться утомительным или незначительным делом, но постарайтесь избегать неинформативных имен вроде `Test1()`, `Test2()` и т. д.

Вместо этого следует использовать имена, описывающие детали теста. В частности, довольно удобно, когда человек, читающий тестирующий код, может быстро определить:

- тестируемый класс (если таковой есть);
- тестируемую функцию;
- тестируемую ситуацию или ошибку.

Самый простой подход к созданию хорошего имени для тестирующей функции — просто объединить всю эту информацию и, возможно, добавить префикс `Test_`.

Например, вместо того, чтобы назвать функцию `Test1()`, можно использовать формат `Test_<ИмяФункции>()`:

```
void Test_SortAndFilterDocs() {

}
```

Основываясь на том, насколько сложен тест, вы можете создать отдельную тестовую функцию для каждой тестируемой ситуации. Можно использовать формат `Test_<ИмяФункции>_Ситуация()`:

```
void Test_SortAndFilterDocs_BasicSorting() {

}
void Test_SortAndFilterDocs_NegativeValues() {

}
...
```

Не бойтесь использовать длинные имена в данном случае. Эти функции не будут вызываться во всей базе кода, поэтому нет причин избегать применения длинных имен функций. Имя тестирующей функции может быть также эффективным комментарием. Кроме того, если этот тест завершится неудачей, большинство тестирующих фреймворков выведет на экран имя функции, в которой утверждение не прошло проверку. Поэтому в таких случаях подробное имя функции особенно полезно.

Обратите внимание: если вы используете тестирующий фреймворк, в нем могут применяться правила и соглашения наименования методов. Например, модуль `unittest` языка Python предполагает, что все имена тестирующих функций будут начинаться с `test`.

Если необходимо назвать *вспомогательную* функцию тестирующего кода, довольно полезно указать, является ли она основной или же она никак не связана непосредственно с тестом. Например, любая вспомогательная функция, приведенная в этой главе, в которой вызывается `assert()`, называется `Check...()`. Но функция `AddScoredDoc()` была названа так же, как и обычная вспомогательная функция.

Что было не так с тем тестом?

В начале главы мы заявляли, что у исходного теста как минимум восемь недостатков:

```
void Test1() {
    vector<ScoredDocument> docs;
    docs.resize(5);
    docs[0].url = "http://example.com";
    docs[0].score = -5.0;
    docs[1].url = "http://example.com";
    docs[1].score = 1;
    docs[2].url = "http://example.com";
    docs[2].score = 4;
    docs[3].url = "http://example.com";
    docs[3].score = -99998.7;
    docs[4].url = "http://example.com";
    docs[4].score = 3.0;

    SortAndFilterDocs(&docs);

    assert(docs.size() == 3);
    assert(docs[0].score == 4);
    assert(docs[1].score == 3.0);
    assert(docs[2].score == 1);
}
```

Теперь, когда мы узнали несколько приемов написания более качественных тестов, определим их.

1. Этот тест очень длинный и изобилует незначительными деталями. Вы можете описать все, что делает этот тест, одним предложением, поэтому тестовое выражение не должно быть длиннее.
2. Добавить еще один тест не так легко. Вам придется копировать, вставлять и изменять, что сделает код еще длиннее. Кроме того, в некоторых местах он будет повторять сам себя.
3. Сообщения, выводимые при неудачном тесте, не очень информативны. Если тест выполняется неудачно, вы увидите лишь `Assertion failed: docs.size() == 3`. Это сообщение не содержит достаточной информации для дальнейшей отладки.

4. Этот тест пытается проверить все сразу — как фильтрацию отрицательных значений, так и упорядочение. Он был бы более читаемым, если разбить его на несколько тестов.
5. Входные параметры теста сложны. В частности, значение `-99998.7` слишком заметное и привлекает внимание, хотя и не является критически важным для теста. Здесь подойдет более простое отрицательное число.
6. Входные параметры не обеспечивают всестороннего тестирования. Например, не проверяется ситуация, когда оценка равна `0`. (Будет ли удален этот документ?)
7. Не тестируются другие ситуации, например пустой вектор, очень длинный вектор или вектор с повторяющимися значениями.
8. Имя `Test1()` не содержит никакой полезной информации — имя должно описывать функцию или ситуацию, которые мы тестируем.

Разработка, ориентированная на тестирование

Некоторые фрагменты кода тестируются легче других. Идеальный код для тестирования имеет качественный интерфейс, не содержит множества состояний и прочих «настраиваемых компонентов», кроме того, в нем немного скрытых данных.

Если вы пишете код, зная, что затем вам придется писать для него тесты, может произойти кое-что интересное: **вы начинаете писать код, который легче тестировать**! Такой способ программирования улучшает и качество создаваемого кода. Разработка с учетом тестирования часто помогает писать хорошо организованный код, отдельные части которого решают отдельные задачи.

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ

Разработка через тестирование (Test-Driven Development (TDD)) — это стиль программирования, следуя которому вы пишете тесты до того, как напишете сам код. Программисты, использующие этот стиль, полагают, что данный процесс позволяет улучшить качество не только тестирующего, но и рабочего кода. Он становится гораздо лучше, чем в ситуациях, когда вы пишете тесты после написания кода.

Но независимо от того, используете вы этот стиль или нет, конечный результат — это код, который тестирует другой код. Цель данной главы заключается в том, чтобы помочь вам сделать ваши тесты более простыми для написания и чтения.

Для всех способов разбить программу на классы и методы действует правило: чем более автономны компоненты, тем проще их протестировать. С другой стороны, предположим, что ваша программа содержит довольно много внутренних связей, множество методов, вызываемых вашими классами, и массу параметров для этих методов. Мало того, что эта программа будет трудна для восприятия, код, тестирующий ее, будет таким же громоздким и сложным для чтения и написания.

Наличие множества «внутренних» компонентов (глобальные переменные, которые нужно инициализировать, библиотеки или конфигурационные файлы, которые необходимо загрузить, и т. д.) также усложняет написание тестов.

Если вы пишете код и понимаете, что его будет сложно протестировать, это хорошая причина, чтобы остановиться и выбрать другой вариант дизайна. В табл. 14.1 показаны типичные проблемы тестирования и дизайна.

Таблица 14.1

Характеристика	Проблема тестирования	Проблема дизайна
Использование глобальных переменных	Все глобальные переменные следует сбрасывать для каждого теста (в противном случае различные тесты могут помешать друг другу)	Сложно определить побочные эффекты, а также то, какие функции их вызывают. Нельзя рассматривать функции по отдельности; необходимо изучать всю программу для того, чтобы понять, работают ли все ее компоненты
Код зависит от множества внешних компонентов	Сложно написать какие-либо тесты, поскольку необходимо выполнить много вспомогательных действий. Тесты писать не так интересно, поэтому программисты пытаются не делать этой работы	При крахе одного внешнего компонента может рухнуть вся система. Сложно понять, какое влияние окажет любое изменение. Осложняется рефакторинг классов. Следует рассмотреть большее количество причин краха и путей восстановления после них
Поведение кода трудно предугадать	Тесты для таких программ хрупки и ненадежны. Неудачные тесты в конечном счете игнорируются	Программа, скорее всего, будет попадать в гонки или допускать другие невоспроизводимые ошибки. Программу становится сложнее исследовать. Кроме того, гораздо сложнее отследить и исправить ошибки

С другой стороны, если дизайн позволяет с легкостью писать тесты, это хороший знак. В табл. 14.2 перечислены положительные характеристики тестирования и дизайна.

Таблица 14.2

Характеристика	Удобство тестирования	Удобство дизайна
У классов нет внутреннего состояния, или оно имеет незначительное влияние	Тесты для таких классов проще писать, поскольку требуется меньше подготовки, чтобы протестировать метод, а также необходимо исследовать меньший объем скрытых данных	Подобные классы просты, в них легко разобраться
Классы и функции решают только одну задачу	Чтобы полностью протестировать класс или функцию,	Меньшие по объему и более простые компоненты

Таблица 14.2, продолжение

Характеристика	Удобство тестирования	Удобство дизайна
	требуется меньшее количество тестов	характеризуются значительной модульностью, система вообще содержит меньше связей
Классы зависят от небольшого количества других классов, высокая степень взаимной независимости	Каждый класс можно протестировать независимо (что гораздо проще тестирования нескольких классов сразу)	Систему можно разрабатывать параллельно. Классы можно легко изменять или удалять, и это не затронет остальную систему
Функции имеют простой и понятный интерфейс	Четко определено поведение, которое необходимо протестировать. Простые интерфейсы легче тестировать	Интерфейсы просты для программистов. Высока вероятность того, что их будут использовать многократно

Не увлекайтесь!

Бывают и такие ситуации, когда программист слишком сосредотачивается на тестировании. Вот несколько примеров.

- **Жертвование читаемостью кода в пользу возможностей тестирования.** Разработка реального кода с заведомо поставленной целью сделать его удобным для тестирования обычно дает двойную выгоду: реальный код становится проще, его компоненты — автономнее, а сами тесты довольно легко писать. Но если вам приходится делать множество уродливых вставок в реальный код лишь для того, чтобы его можно было протестировать, — это признак того, что что-то идет не так.
- **Стремление к стопроцентному тестированию кода.** Тестирование 90 % кода обычно требует меньше усилий, чем тестирование последних 10 %. Эти последние 10 % могут включать в себя пользовательский интерфейс или необычные случаи ошибок, когда ценность найденной ошибки не так высока, а она сама не стоит затраченных на поиск усилий.

  Истина такова, что вы никогда не сможете охватить код тестами на 100 %. Если вы не пропустите ошибку, то, возможно, не реализуете какую-либо особенность или не поймете, что спецификацию следует изменить.

  В зависимости от важности найденных ошибок следует взвешивать, сколько усилий потребуется потратить на разработку тестирующего кода. Если вы создаете прототип веб-сайта, можно совсем не писать тестирующий код. С другой стороны, если вы разрабатываете контроллер для космического корабля или медицинского устройства, то тестирование, вероятно, — основной приоритет.
- **Тестирование становится основополагающей частью проекта.** Нам известны ситуации, когда тестирование, которое должно быть всего лишь одним аспектом проекта, приобретало в нем главенствующую роль. Тестирование становилось как будто идолом, которому следовало поклоняться, и программисты просто выполняли ритуалы и действия, не понимая, что их драгоценное время гораздо лучше было бы потратить на другую работу.

Итог

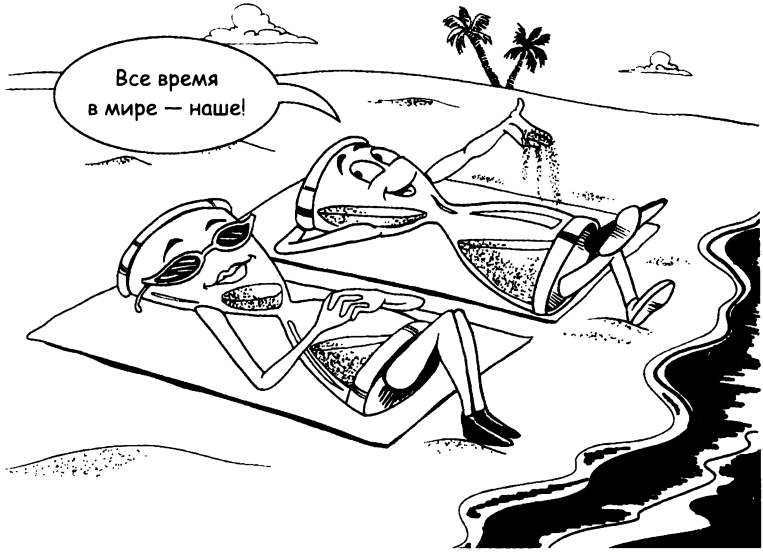
Читаемость тестирующего кода, так или иначе, очень важна. Если тесты удобно читать, то их будет очень легко внедрить в программу и разработчики будут широко пользоваться ими.

Есть некоторые особенности, помогающие улучшить тесты.

- Высший уровень каждого теста должен быть как можно более кратким; в идеале каждый тестовый ввод и вывод должен описываться одной строкой кода.
- В случае неудачи при тестировании тест должен выдавать сообщение об ошибке, которое позволит с легкостью отследить и исправить эту ошибку.
- Желательно использовать самые простые входные параметры, которые будут тестировать все аспекты кода.
- Тестирующим функциям нужно давать очень подробные имена, которые позволят четко понять, что именно тестируется. Вместо имени `Test1()` лучше использовать имя вроде `Test_<ИмяФункции>_<Ситуация>`.

Помимо всего прочего, делайте тестирующие коды простыми и открытыми для добавления новых тестов.

15 Разработка и реализация счетчика минут и часов



Рассмотрим структуру данных, использованную в реальном коде счетчика минут и часов. Мы исследуем естественный мыслительный процесс инженера — сначала решим задачу, а затем улучшим ее производительность и добавим пару особенностей. Самое важное — мы постараемся сделать код максимально простым для чтения, используя принципы, рассмотренные ранее в книге. Мы также разберем несколько ошибок, которые могут встретиться вам в ходе разработки. Посмотрим, сумеете ли вы их отследить.

Постановка задачи

Допустим, необходимо отследить, сколько байт передал веб-сервер за прошедшую минуту и за прошедший час. На рис. 15.1 показано, как именно рассчитываются эти параметры.

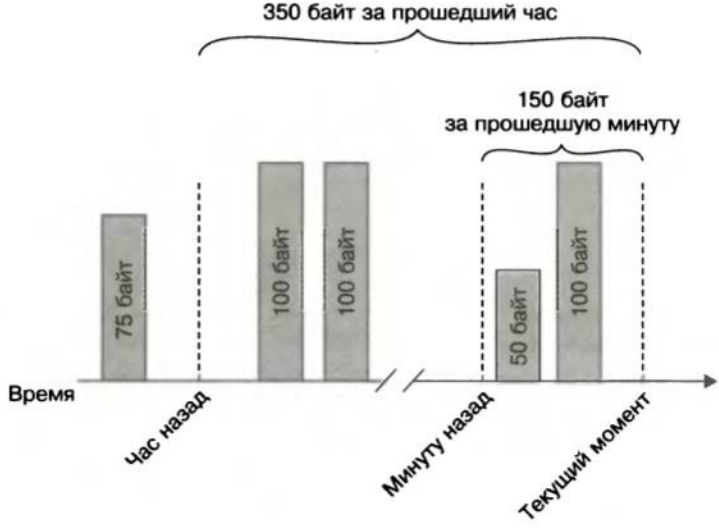


Рис. 15.1

Это довольно простая задача, но, как вы увидите далее, ее эффективное решение не вполне очевидно. Начнем с определения интерфейса класса.

Определение интерфейса класса

Рассмотрим первую версию нашего интерфейса класса на языке C++:

```
class MinuteHourCounter {
  public:
    // добавляем счетчик
    void Count(int num_bytes);

    // возвращаем счетчик за последнюю минуту
```

```
    int MinuteCount();

    // возвращаем счетчик за последний час
    int HourCount();
};
```

Перед тем как мы реализуем этот класс, рассмотрим имена и комментарии, чтобы понять, хотим ли мы что-нибудь изменить.

Улучшаем имена

Имя класса `MinuteHourCounter` подходит хорошо. Оно очень точное, конкретное, и его легко выговорить.

Учитывая имя класса, имена методов `MinuteCount()` и `HourCount()` также хороши. Вы можете назвать их `GetMinuteCount()` и `GetHourCount()`, но это их не улучшит. Как мы говорили в гл. 3, слово get для многих разработчиков означает «средство организации быстрого доступа». И, как вы увидите, реализация будет не самой простой, поэтому лучше обойтись без использования данного слова.

Довольно проблемным является имя метода `Count()`. Мы спрашивали наших коллег о том, что, по их мнению, делает метод `Count()`, и некоторые из них сказали, что он возвращает общее количество отсчетов за все время. Это имя несколько нелогично. Проблема заключается в том, что в английском языке слово `Count` является и существительным, и глаголом и может использоваться в предложениях вида I want a count of the number of samples you have seen (Я хочу узнать количество всех образцов, которые ты видел) и I want you to count this sample (Я хочу, чтобы ты посчитал этот образец).

Рассмотрим другие имена, которые можно было бы применить вместо `Count()`:

- `Increment()` (увеличить);
- `Observe()` (наблюдать);
- `Record()` (записывать);
- `Add()` (добавлять).

Имя `Increment()` обманчиво, поскольку оно говорит о том, что есть значение, которое только увеличивается. (В нашем случае часовой счетчик колеблется с течением времени.)

Имя `Observe()` подходит, но является несколько неопределенным.

Имя `Record()` также может быть понято и как существительное, и как глагол, поэтому оно не годится.

Имя `Add()` — интересный вариант, поскольку оно может означать как «добавить это с помощью цифр», так и «добавить к списку данных». В нашем случае могут быть верны оба варианта, поэтому такое имя подойдет. Поэтому переименуем метод в `void Add(int num_bytes)`.

Однако имя аргумента `num_bytes` чересчур конкретное. Да, его основное применение — подсчет байтов, но классу `MinuteHourCounter` это знать необязательно. Кто-либо еще может применять этот класс для подсчета запросов или транзакций баз данных. Мы могли бы использовать более общее имя вроде `delta`, но этот термин

часто применяется, когда значение может быть отрицательным (что нам не нужно). Имя count должно подойти — оно простое, абстрактное и означает «неотрицательное значение». Кроме того, оно позволяет избежать двойственности контекста.

Улучшаем комментарии

Рассмотрим интерфейс класса, который у нас получился:

```
class MinuteHourCounter {
  public:
    // добавляем счетчик
    void Add(int count);

    // возвращаем счетчик за последнюю минуту
    int MinuteCount();

    // возвращаем счетчик за последний час
    int HourCount();
};
```

Рассмотрим комментарии к каждому методу и улучшим их. Первый метод:

```
// добавляем счетчик
void Add(int count);
```

После переименования метода в подобном комментарии нет необходимости — его следует убрать или улучшить. Вот его улучшенная версия:

```
// Добавляем новую точку данных (count >= 0).
// Через минуту MinuteCount() будет больше на значение count.
// Через час HourCount() будет больше на значение count.
void Add(int count);
```

Теперь рассмотрим комментарий к методу MinuteCount():

```
// возвращаем счетчик за последнюю минуту
int MinuteCount();
```

Когда мы спросили у наших коллег, что мог бы означать подобный комментарий, у них возникли две противоречащие друг другу интерпретации:

- возвращаем счетчик для текущего значения пары параметров «час — минута», например в 12:13;
- возвращаем значение счетчика за прошедшие 60 секунд, независимо от границ часа и минуты.

Вторая интерпретация описывает реальную работу программы. Избавимся от двусмысленности, воспользовавшись более четким и детализированным комментарием:

```
// Возвращаем увеличенное значение счетчика за прошедшие 60 секунд.
int MinuteCount();
```

(Аналогично улучшим комментарий и для метода HourCount().)

Вот описание класса со всеми изменениями, сделанными ранее, а также комментарий, описывающий класс в целом:

```
// Отслеживаем значение кумулятивных счетчиков раз в минуту и раз в час.
// Полезно было бы также отслеживать активность использования полосы
// в последнее время.
class MinuteHourCounter {
  public:
   // Добавляем новую точку данных (count >= 0).
   // Через минуту MinuteCount() будет больше на значение count.
   // Через час HourCount() будет больше на значение count.
   void Add(int count);

   // Возвращаем увеличенное значение счетчика за прошедшие 60 секунд.
   int MinuteCount();

   // Возвращаем увеличенное значение счетчика за прошедшие 3600 секунд.
   int HourCount();
};
```

(Для краткости в дальнейшем будем опускать эти комментарии в листингах программы.)

ВЗГЛЯД СО СТОРОНЫ

Как вы могли заметить, мы в нескольких случаях консультировались с нашими коллегами. Такой подход — отличный способ протестировать код на удобство использования. Попробуйте прислушаться к первым впечатлениям тех, кто смотрит на код со стороны, поскольку другие люди могут прийти к тем же выводам, что и они. Среди этих «других» людей можете оказаться и вы сами спустя шесть месяцев.

Первый подход: простое решение

Перейдем к решению проблемы. Начнем с «лобового» решения: просто создадим список (`list`), в котором будем хранить события с отметками о времени появления:

```
class MinuteHourCounter {
    struct Event {
        Event(int count, time_t time) : count(count), time(time) {}
        int count;
        time_t time;
    };

    list<Event> events;

  public:
    void Add(int count) {
        events.push_back(Event(count, time()));
    }


};
```

Теперь мы можем просто подсчитать количество байтов, переданных за недавнее время:

```
class MinuteHourCounter {


    int MinuteCount() {
        int count = 0;
        const time_t now_secs = time();
        for (list<Event>::reverse_iterator i = events.rbegin();
             i != events.rend() && i->time > now_secs - 60; ++i) {
            count += i->count;
        }
        return count;
    }

    int HourCount() {
        int count = 0;
        const time_t now_secs = time();
        for (list<Event>::reverse_iterator i = events.rbegin();
             i != events.rend() && i->time > now_secs - 3600; ++i) {
            count += i->count;
        }
        return count;
    }
};
```

Насколько понятен этот код?

Несмотря на то что это решение является верным, в нем присутствует несколько проблем, связанных с читаемостью.

- **Циклы for довольно трудно прочесть.** Большинство читателей знакомятся с этой частью кода довольно медленно (по крайней мере читатель замедляется, если проверяет код на наличие ошибок).
- **Методы MinuteCount() и HourCount() практически идентичны.** Если бы эти методы совместно использовали один и тот же код, программа стала бы короче. Данная деталь особенно важна, поскольку избыточный код относительно сложен для понимания. (Лучше всего поместить весь сложный для понимания код в одном месте.)

Более удобочитаемая версия

Код методов `MinuteCount()` и `HourCount()` отличается лишь одной константой (`60` и `3600`). Очевидно, что при рефакторинге следует создать вспомогательный метод, обрабатывающий оба случая:

```
class MinuteHourCounter {
    list<Event> events;
```

```
    int CountSince(time_t cutoff) {
        int count = 0;
        for (list<Event>::reverse_iterator rit = events.rbegin();
             rit != events.rend(); ++rit) {
            if (rit->time <= cutoff) {
                break;
            }
            count += rit->count;
        }
        return count;
    }

  public:
    void Add(int count) {
        events.push_back(Event(count, time()));
    }

    int MinuteCount() {
        return CountSince(time() - 60);
    }

    int HourCount() {
        return CountSince(time() - 3600);
    }
};
```

Стоит отметить в этом новом коде несколько моментов.

Во-первых, обратите внимание на то, что метод `CountSlice()` принимает абсолютный параметр `cutoff` вместо относительного `secs_ago` (`60` или `3600`). Любое из этих решений верное, но теперь методу `CountSlice()` придется выполнять более простую задачу.

Во-вторых, мы переименовали итератор из `i` в `rit`. Имя `i` чаще всего применяется для числовых индексов. Мы решили использовать имя `it`, типичное для итераторов. Но в нашем случае итератор *обратный (reverse)*, и этот факт критически важен для правильности кода. Добавление префикса `r` позволяет обеспечить удобную симметрию в утверждениях вида `rit != events.rend()`.

Наконец, мы извлекли условие `rit->time <= cutoff` из цикла `for` и вынесли его в отдельную условную конструкцию `if`. Почему мы это сделали? Потому что «традиционные» циклы на основе оператора `for` вида `for(begin; end; advance)` (начальное значение переменной, конечное значение переменной, шаг изменения переменной) наиболее просты для чтения. Читатель сразу понимает, что перед ним — «проход по всем элементам», и не задумывается о нем в дальнейшем.

Проблемы с производительностью

Несмотря на то что мы улучшили внешний вид кода, этот вариант дизайна имеет серьезные проблемы с производительностью.

- **Он продолжает разрастаться.** Данный класс хранит абсолютно все произошедшие события, используя при этом неограниченный объем памяти! В идеале класс `MinuteHourCounter` должен автоматически удалять все события старше часа, поскольку в них больше нет необходимости.
- **Методы MinuteCount() и HourCount() слишком медленные.** Метод `CountSlice()` затрачивает $O(n)$ времени, где n — это количество точек данных в заданном временном интервале. Представьте себе высокопроизводительный сервер, в котором метод `Add()` будет вызываться сотни раз за секунду. Каждый вызов функции `HourCount()` будет пересчитывать миллионы точек данных! В идеале класс `MinuteHourCounter` должен иметь отдельные переменные `minute_count` и `hour_count`, которые будут обновляться с каждым вызовом метода `Add()`.

Вторая попытка: реализация конвейерного дизайна

Нам нужен дизайн, способный решить обе предыдущие проблемы:

- удалять данные, в которых нет необходимости;
- постоянно обновлять предварительно рассчитанные суммы байтов `minute_count` и `hour_count`.

Для этого мы используем наш список как конвейер. При появлении новых данных с одного конца мы добавим их к суммам. А затем, когда данные станут слишком старыми, они «выпадут» с другого конца и мы отнимем их от сумм.

Есть несколько способов реализации подобного дизайна. Один из них — создание двух независимых списков, один из которых будет хранить события, произошедшие за последнюю минуту, другой — произошедшие за последний час. Когда происходит новое событие, его копия добавляется в оба списка (рис. 15.2).

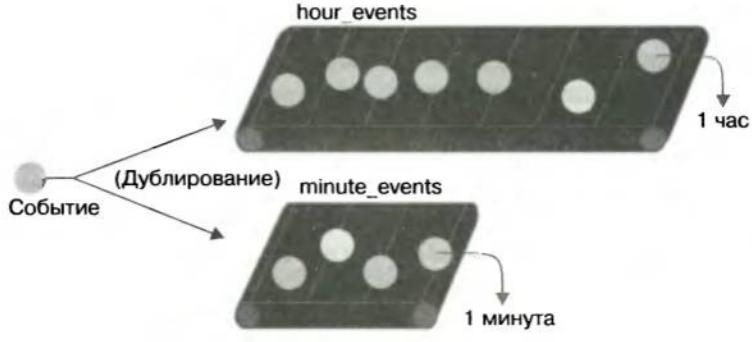


Рис. 15.2

Этот метод прост, но неэффективен, поскольку при таком подходе создаются две копии каждого события.

Другой способ реализации — создание двух списков. При этом событие сначала попадает в первый список (где хранятся события, произошедшие за последнюю минуту), а затем во второй (где хранятся события, произошедшие за последний час, но не за последнюю минуту) (рис. 15.3).

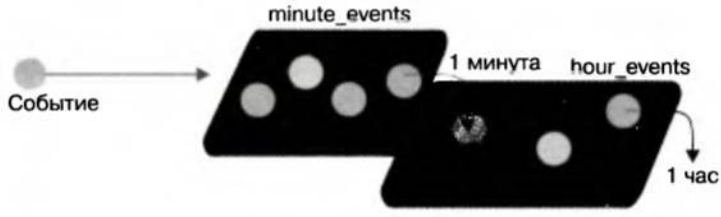


Рис. 15.3

Такой «двухуровневый» проект конвейера кажется более эффективным, поэтому будем реализовывать именно его.

Реализация двухуровневого дизайна конвейера

Начнем с перечисления всех членов нашего класса:

```
class MinuteHourCounter {
    list<Event> minute_events;
    list<Event> hour_events;  // Содержит элементы, НЕ входящие
                              // в список minute_events.

    int minute_count;
    int hour_count;  // Считает ВСЕ элементы за прошедший час,
                     // включая последнюю минуту.
};
```

Основная идея такого дизайна заключается в том, что он способен «смещать» все события по прошествии времени, поэтому события перемещаются из списка `minute_events` в список `hour_events`, а счетчики `minute_count` и `hour_count` обновляются соответственно. Чтобы реализовать его, создадим вспомогательный метод `ShiftOldEvents()`. Имея этот метод, остальную часть класса довольно легко реализовать:

```
void Add(int count) {
    const time_t now_secs = time();
    ShiftOldEvents(now_secs);

    // Добавляем события в список с событиями прошедшей минуты
    // (но не прошедшего часа – это произойдет позже).
    minute_events.push_back(Event(count, now_secs));

    minute_count += count;
    hour_count += count;
}
```

```
int MinuteCount() {
    ShiftOldEvents(time());
    return minute_count;
}

int HourCount() {
    ShiftOldEvents(time());
    return hour_count;
}
```

Как видите, мы передали всю «грязную» работу методу `ShiftOldEvents()`:

```
// Находим и удаляем старые события и, соответственно,
// уменьшаем счетчики hour_count и minute_count.
void ShiftOldEvents(time_t now_secs) {
    const int minute_ago = now_secs - 60;
    const int hour_ago = now_secs - 3600;

    // Перемещаем события старше одной минуты из списка 'minute_events'
    // в список 'hour_events'.
    // (События старше одного часа будут удалены во втором цикле.)
    while (!minute_events.empty() && minute_events.front().time <=
                                                    minute_ago) {
        hour_events.push_back(minute_events.front());

        minute_count -= minute_events.front().count;
        minute_events.pop_front();
    }

    // Удаляем события старше одного часа из списка 'hour_events'.
    while (!hour_events.empty() && hour_events.front().time <= hour_ago) {
        hour_count -= hour_events.front().count;
        hour_events.pop_front();
    }
}
```

Мы уже закончили?

Мы избавились от двух проблем производительности, которые упомянули ранее, и решение работает. Оно замечательно подойдет для многих приложений. Однако в нем есть и некоторые недостатки.

Во-первых, этот дизайн совершенно негибкий. Предположим, что нам понадобится подсчитывать все события за прошедшие 24 часа. Это потребует внесения множества изменений в код. И как вы, наверное, заметили, `ShiftOldEvents()` — это довольно плотная функция, с тонкими взаимодействием между данными за одну минуту и один час.

Во-вторых, этот класс расходует довольно много памяти. Представьте, что у вас есть загруженный сервер, вызывающий функцию `Add()` 100 раз в секунду. Поскольку мы храним все данные за прошедший час, этот код в итоге может потребовать около 5 Мбайт памяти.

В общем, чем чаще вызывается функция `Add()`, тем больше памяти мы задействуем. В промышленной сфере библиотеки, которые используют большие и непредсказуемые объемы памяти, не считаются хорошими. В идеале класс `MinuteHourCounter` должен расходовать фиксированный объем памяти зависимо от того, как часто вызывается метод `Add()`.

Третья попытка: дизайн, при котором время делится на блоки

Вы могли не заметить, но в обеих предыдущих реализациях была небольшая ошибка. Чтобы хранить временную метку, мы использовали переменную типа `time_t`, которая может хранить целое количество секунд. Из-за такого округления метод `MinuteCount()` возвращает данные, расположенные по времени где-то между 59 и 60 секундами, в зависимости от того, когда именно вы его вызвали.

Например, если время происхождения события равно `time = 0.99` секунды, то оно округлится следующим образом — `t = 0` секунд. Если метод `MinuteCount()` будет вызван во время `time = 60.1` секунды, будут возвращены все события, для которых `t = 1, 2, 3, ... 60`. Поэтому первое событие будет пропущено, даже несмотря на то, что технически оно произошло менее минуты назад.

В среднем метод `MinuteCount()` возвращает данные за прошедшие 59,5 секунды, а метод `HourCount()` — за 3599,5 секунды (незначительная ошибка).

Мы можем исправить это, используя другой тип, который способен хранить дробные части секунды. Но, что интересно, большинство приложений, использующих класс `MinuteHourCounter`, не нуждаются в такой точности. Мы применяем этот факт для того, чтобы разработать новый дизайн класса `MinuteHourCounter`, который будет работать гораздо быстрее и потреблять меньше памяти. Обмен точности на производительность оправдает себя.

Основная идея заключается в том, чтобы *объединить* все события, произошедшие внутри небольшого временного окна, и просуммировать эти события как единое целое. Например, все события, произошедшие за последнюю минуту, будут помещены в 60 отдельных блоков, каждый шириной в одну секунду. События, произошедшие за последний час, также могут быть помещены в 60 отдельных блоков шириной в одну минуту (рис. 15.4).

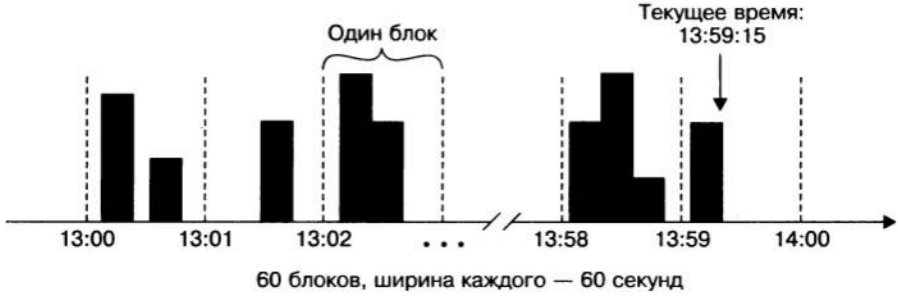


Рис. 15.4

Используя блоки так, как показано на рис. 15.4, методы MinuteCount() и HourCount() будут округлять одну часть из 60, что в данном случае приемлемо[1].

Если необходима большая точность, для обмена следует использовать больше блоков, а значит и больше памяти. Однако важно то, что подобный дизайн использует фиксированное, предсказуемое количество памяти.

Реализация дизайна с временными блоками

Реализация такого дизайна всего в одном классе приведет к созданию массы запутанного кода, который сложно держать в уме. Вместо этого мы будем следовать совету, данному в гл. 11, и создадим отдельные классы для того, чтобы решать различные части этой задачи.

Сначала создадим отдельный класс, который будет отслеживать количество байтов, пришедших в отдельный промежуток времени (например, за последний час). Назовем его TrailingBucketCounter. По сути, это абстрактная версия класса MinuteHourCounter, которая обрабатывает лишь один промежуток времени. Вот его интерфейс:

```
// Класс, учитывающий последние N блоков времени.
class TrailingBucketCounter {
  public:
    // Пример: TrailingBucketCounter(30, 60) отслеживает
    //последние 30 минутных блоков времени.
    TrailingBucketCounter(int num_buckets, int secs_per_bucket);

    void Add(int count, time_t now);

    // Возвращаем сумму байтов, пришедших в последние num_buckets блоков.
    int TrailingCount(time_t now);
};
```

Возможно, у вас возникнет вопрос: почему методы Add() и TrailingCount() требуют передачи текущего времени (time_t now) в качестве параметра? Разве не было бы проще, если бы эти методы высчитывали текущее время самостоятельно?

Хотя это и может показаться странным, передача текущего времени имеет несколько преимуществ. Во-первых, класс TrailingBucketCounter перестает зависеть от времени, поэтому его проще тестировать, а также низка вероятность появления ошибок. Во-вторых, все вызовы функции time() находятся внутри класса MinuteHourCounter. На работе чувствительных ко времени систем положительно скажется тот факт, что функция time() вызывается лишь из одного места.

---

[1] Подобно предыдущим решениям, последний блок в среднем будет лишь наполовину полным. С таким дизайном мы можем исправить это, создав 61 блок вместо 60 и игнорируя блок, который в данный момент находится в обработке. Это приведет к тому, что данные будут частично устаревшими. Лучшим решением в данной ситуации будет комбинирование блока, находящегося в обработке, с дополнительным дроблением самого старого блока, чтобы получить количество блоков, которые являются одновременно объективными и новейшими. Реализацию этой идеи мы оставляем в качестве упражнения для читателей.

Предполагая, что класс TrailingBucketCounter уже реализован, класс MinuteHourCounter также легко реализовать:

```
class MinuteHourCounter {
    TrailingBucketCounter minute_counts;
    TrailingBucketCounter hour_counts;

  public:
    MinuteHourCounter() :
        minute_counts(/* num_buckets = */ 60, /* secs_per_bucket = */ 1),
        hour_counts(  /* num_buckets = */ 60, /* secs_per_bucket = */ 60) {
    }

    void Add(int count) {
        time_t now = time();
        minute_counts.Add(count, now);
        hour_counts.Add(count, now);
    }

    int MinuteCount() {
        time_t now = time();
        return minute_counts.TrailingCount(now);
    }

    int HourCount() {
        time_t now = time();
        return hour_counts.TrailingCount(now);
    }
};
```

Этот код гораздо проще читать, а кроме того, он более гибкий. Если мы захотим увеличить количество блоков (чтобы повысить точность, но при этом увеличить объем используемой памяти), то сделать это будет несложно.

Реализация класса TrailingBucketCounter

Теперь нужно реализовать класс TrailingBucketCounter. Опять же мы собираемся создать вспомогательный класс для того, чтобы разделить эту задачу на еще более мелкие фрагменты.

Мы создадим структуру данных, которую назовем ConveyorQueue. Она должна будет работать с основными показателями времени и их суммами. Класс TrailingBucketCounter может заняться непосредственно задачей передвижения временных рамок для класса ConveyorQueue с течением времени.

Рассмотрим интерфейс класса ConveyorQueue:

```
// Очередь, ограниченная максимальным количеством свободных мест,
// где старые данные «выпадают» с ее конца.
class ConveyorQueue {
    ConveyorQueue(int max_items);
    // Добавляем значение к концу очереди.
```

```
    void AddToBack(int count);

    // Каждое значение в очереди сдвигается вперед на 'num_shifted'.
    // Новые элементы инициализируются нулем.
    // Самые старые элементы будут удалены, поэтому количество элементов
    // в очереди всегда будет <= max_items.
    void Shift(int num_shifted);

    // Возвращаем суммарное значение всех элементов,
    // находящихся в очереди в данный момент.
    int TotalSum();
};
```

Предполагая, что этот класс был реализован, посмотрите, как просто будет реализовать класс TrailingBucketCounter:

```
class TrailingBucketCounter {
    ConveyorQueue buckets;
    const int secs_per_bucket;
    time_t last_update_time;  // время, когда в последний раз
                              // была вызвана функция Update()

    // Рассчитываем, сколько блоков времени было передано и уже прошло
    // через функцию Shift() соответственно.
    void Update(time_t now) {
        int current_bucket = now / secs_per_bucket;
        int last_update_bucket = last_update_time / secs_per_bucket;

        buckets.Shift(current_bucket - last_update_bucket);
        last_update_time = now;
    }

  public:
    TrailingBucketCounter(int num_buckets, int secs_per_bucket) :
        buckets(num_buckets),
        secs_per_bucket(secs_per_bucket) {
    }

    void Add(int count, time_t now) {
        Update(now);
        buckets.AddToBack(count);
    }

    int TrailingCount(time_t now) {
        Update(now);
        return buckets.TotalSum();
    }
};
```

Такое разделение на два класса (TrailingBucketCounter и ConveyorQueue) — это еще один пример использования приема, который мы обсуждали в гл. 11. Мы также можем справиться с этой задачей и без создания класса ConveyorQueue и реализовать

все непосредственно внутри класса TrailingBucketCounter. Но в нашем случае код становится проще для чтения.

Реализация класса ConveyorQueue

Все, что нам осталось сделать, — реализовать класс ConveyorQueue:

```
// Очередь, ограниченная максимальным количеством точек данных,
// где старые данные сдвигаются с конца.
class ConveyorQueue {
    queue<int> q;
    int max_items;
    int total_sum;  // сумма всех элементов в очереди q

  public:
    ConveyorQueue(int max_items) : max_items(max_items), total_sum(0) {
    }

    int TotalSum() {
        return total_sum;
    }

    void Shift(int num_shifted) {
        // Если сдвинуто слишком много элементов, просто очищаем очередь.
        if (num_shifted >= max_items) {
            q = queue<int>(); // очищаем очередь
            total_sum = 0;
            return;
        }

        // Помещаем все необходимые нули.
        while (num_shifted > 0) {
            q.push(0);
            num_shifted--;
        }

        // Позволяем всем избыточным элементам отпасть.
        while (q.size() > max_items) {
            total_sum -= q.front();
            q.pop();
        }
    }

    void AddToBack(int count) {
        if (q.empty()) Shift(1);  // Убеждаемся, что в очереди q
                                  // есть хотя бы один элемент.
        q.back() += count;
        total_sum += count;
    }
};
```

Итак, мы закончили! У нас есть класс `MinuteHourCounter`, одновременно быстрый и эффективно потребляющий память, а также более гибкий класс `TrailingBucketCounter`, который с легкостью можно использовать повторно. Например, довольно легко было бы создать более гибкую версию `RecentCounter`, которая сможет подсчитывать данные в большом количестве различных диапазонов, как за день, так и за последние десять минут.

Сравнение трех решений

Давайте сравним решения, которые мы разобрали в этой главе. В табл. 15.1 приведен размер кода, а также его характеристики производительности (предполагается высокая загруженность сервера, когда метод `Add()` вызывается 100 раз в секунду).

Таблица 15.1

Решение	Количество строк кода	Затраты на вызов метода HourCount()	Объем использованной памяти	Ошибки в классе HourCount()
Простое решение	33	O(#событий-в-час) (~3,6 миллиона)	Не ограничен	Одна часть из 3600
Конвейерный дизайн	55	O(1)	O(#событий-в-час) (~5 Мбайт)	Одна часть из 3600
Дизайн, при котором время разбито на блоки (60 блоков)	98	O(1)	O(#количество-блоков) (~500 байтов)	Одна часть из 60

Обратите внимание, что объем кода в последнем, трехклассовом решении больше, чем в двух других. Однако и производительность при этом значительно выше, а дизайн более гибок. Кроме того, каждый класс довольно просто читать. Такое изменение всегда идет на пользу: лучше иметь 100 простых для чтения строк, чем 50 нечитаемых.

Иногда разбиение задачи на несколько классов может привести к появлению сложного межклассового взаимодействия (которого не будет в одноклассовом решении). Но в нашем случае выстроена простая «линейная» цепочка использования одного класса за другим и пользователи работают только с одним классом в любой момент времени. В итоге выигрыш от разбиения задачи становится определяющим.

Итог

Вспомним все шаги, которые мы сделали перед тем, как создать итоговую версию дизайна класса `MinuteHourCounter`. Примерно по такому же принципу развивались и другие части кода.

Мы начали с создания простого решения. Это помогло нам осознать две проблемы данного дизайна: скорость и объем использованной памяти.

Далее мы попробовали конвейерный дизайн. Он повысил скорость и сократил объем используемой памяти, но оказался недостаточно хорош для высокопроизводительных приложений. Этот дизайн был также очень негибким: адаптация кода к работе с другими временными интервалами потребовала бы много времени.

Наше итоговое решение позволило снять все предыдущие проблемы, разбив их при этом на подзадачи. Далее приведены три класса, которые мы создали. Они расположены в порядке, обратном очередности их реализации. Кроме того, описаны подзадачи, которые они решают.

- `ConveyorQueue` — ограниченная очередь, которая оперирует итоговой суммой и может быть сдвинута.
- `TrailingBucketCounter` — передвигает `ConveyorQueue` в соответствии с прошедшим временем, а также работает с одним (самым поздним) временным интервалом с заданной точностью.
- `MinuteHourCounter` — просто содержит два объекта класса `TrailingBucketCounters`: один для подсчета всех событий за минуту, другой — для подсчета всех событий за час.

Босуэлл Д., Фаучер Т.

Читаемый код, или Программирование как искусство

Перевод с английского О. Сивченко

Заведующая редакцией	*К. Галицкая*
Руководитель проекта	*Д. Виницкий*
Ведущий редактор	*Е. Каляева*
Художник	*Л. Адуевская*
Корректоры	*О. Андросик, Е. Павлович*
Верстка	*А. Барцевич*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 23.03.12. Формат 70×100/16. Усл. п. л. 16,770. Тираж 1500. Заказ № 270.

Отпечатано с готовых диапозитивов в ГППО «Псковская областная типография».
180004, Псков, ул. Ротная, 34.