O'REILLY®



Правила программирования

КАК ПИСАТЬ КАЧЕСТВЕННЫЙ КОД

КРИС ЦИММЕРМАН

astana
international
publishing

O'REILLY®

Книги по программированию

The Rules of Programming

How to Write Better Code

Chris Zimmerman

Правила программирования

Как писать качественный код

Крис Циммерман

УДК 004.414.2
ББК 32.973.202

Ц61

The Rules of Programming: How to Write Better Code
Chris Zimmerman

© 2026 "Astana International Publishing." Authorized Russian translation of the English edition of The Rules of Programming ISBN 9781098133115
© 2023 Chris Zimmerman. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Циммерман, Крис.

Ц61 Правила программирования. Как писать качественный код: [перевод с английского О. Перфильева]. — Алматы: Астана иностранная пресса, 2026. — 368 с. — (O'Reilly. Книги по программированию).

ISBN 978-601-12-7677-1

«Правила программирования» — практическое руководство по созданию качественного, понятного и поддерживаемого кода. Вместо абстрактной теории автор предлагает 21 правило, основанное на десятилетиях работы в Microsoft и студии Sucker Punch Productions, создавшей такие игровые проекты, как Sly Cooper, inFamous и Ghost of Tsushima. На многочисленных примерах автор показывает, как принимать инженерные решения, которые делают код проще, надежнее и удобнее для дальнейшего развития.

Книга помогает сформировать профессиональное мышление программиста и предлагает систему принципов, которые одинаково полезны как начинающим разработчикам, так и опытным инженерам.

УДК 004.414.2
ББК 32.973.202

© Перфильев О.И., перевод на русский язык, 2026
© Издание на русском языке, оформление.
ТОО «Издательство «Астана иностранная пресса», 2026

ISBN 978-601-12-7677-1

Барлық құқықтар қорғалған. Бұл кітапты басып шығарушының рұқсатынсыз онлайн немесе кез келген басқа жолмен сканерлеу, жүктеп алу немесе заңсыз тарату заң бойынша жазаланады / Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме с помощью каких-либо электронных или механических средств, включая изготовление фотокопий, аудиозапись, репродукцию или любой иной способ, или систем поиска и хранения информации без письменного разрешения издателя.

Оглавление

Предисловие ... 7

История Правил ... 11

Как не соглашаться с Правилами ... 14

Правило 1. Максимально просто, но не слишком просто ... 16

Правило 2. Баги заразны ... 31

Правило 3. Хорошее имя — лучшая документация ... 49

Правило 4. Обобщайте после трех примеров ... 62

Правило 5. Первый урок оптимизации — не оптимизировать ... 79

Интермедия, в которой предыдущая глава подвергается критике ... 93

Правило 6. Ревизия кода полезна по трем причинам ... 97

Правило 7. Устраняйте сценарии сбоев ... 104

Правило 8. Код, который не запускается, не работает ... 123

Правило 9. Пишите код, который можно сворачивать ... 135

Правило 10. Локализируйте сложность ... 153

Правило 11. Станет ли от этого вдвойне лучше? ... 171

Правило 12. Большим командам — строгие правила ... 179

Правило 13. Ищите запустивший лавину камушек ... 191

Правило 14. Код бывает четырех типов ... 206

Правило 15. Пропалывайте сорняки ... 222

Правило 16. Двигайтесь от результата к коду, а не от кода к результату ... 229

Правило 17. Большие задачи порой решать проще ... 252

Правило 18. Пусть код сам рассказывает свою историю ... 265

Правило 19. Откройте для себя параллельную переработку ... 274

Правило 20. Не забывайте про математику ... 297

Правило 21. Иногда нужно просто забивать гвозди ... 307

Заключение. Настраивайте правила под себя ... 316

Приложение А. Чтение C++ для программистов на Python ... 320

Приложение Б. Чтение C++ для программистов на JavaScript ... 340

Указатель ... 357

Об авторе ... 366

Отзывы о книге «Правила программирования»

«Правила программирования» — это не только советы для новичков, но и уроки, которые могут научить кое-чему даже экспертов. Легкий и живой язык Циммермана доказывает, что книга может быть одновременно и занимательной, и поучительной.

Марк Серни,
главный системный архитектор PlayStation 4 и 5

«Правила программирования» дают ценные советы как начинающим, так и опытным программистам. Книга написана увлекательным стилем, а перечисленные Циммерманом правила представляют собой очень важный вклад в развитие практик программирования, и это в то время, когда технологии пронизывают почти все сферы бизнеса и общества.

Пол Доэрти,
исполнительный и технический директор компании Accenture

В этой книге описаны практичные и рабочие правила, которые помогут повысить свой уровень любому программисту. Мне повезло — в начале своей карьеры я узнал эти правила от самого Криса, и я успешно применял их в самых разных областях разработки. Теперь с ними можете познакомиться и вы.

Крис Бентцель,
директор по разработке программного обеспечения, Boston Dynamics

Предисловие

Здравствуйте! Перед вами «Правила программирования» — свод легких для запоминания и простых в использовании советов, которые помогут вам научиться составлять более продуманные и качественные программы. Программирование — сложная работа, но правила делают ее чуточку проще.

Перед началом вот вам несколько советов:

- Все Правила самодостаточны. Если вы увидели в оглавлении какое-то Правило, которое показалось вам интересным, и хотите сразу же перейти в середину книги, — пожалуйста, переходите. Такой стиль чтения полностью поддерживается.
- Тем не менее я бы рекомендовал начать с Правила 1 «Максимально просто, но не слишком просто». Оно задает хороший тон для всех последующих Правил.
- Примеры в книге написаны на языке C++. Если вы программируете на Python или JavaScript, то вам стоит для начала ознакомиться с Приложением А («Чтение C++ для программистов на Python») или Приложением Б («Чтение C++ для программистов на JavaScript»). Это своего рода «Розеттский камень», помогающий перевести конструкции C++ в привычные для вас концепции. Если вы работаете с каким-то другим языком и находите примеры на C++ трудными для восприятия, рекомендую посетить замечательный сайт Rosetta Code.
- Тем, кто программирует на C++, следует иметь в виду, что некоторые фрагменты кода в примерах я упростил для удобства тех, кто не работает с C++. Например, в некоторых местах используются знаковые целочисленные типы, тогда как в типичной программе на C++, скорее всего, использовались бы беззнаковые типы. Также я отключил предупреждения о неявном преобразовании между знаковыми и беззнаковыми типами. Кроме того, в примерах не указывалась явно директива «using std», хотя компилировались они с ней. Я исключил ее, чтобы не загромождать код отвлекающими внимание ссылками «std::».
- И наконец, когда я ссылаюсь на какое-то конкретное правило из этой книги, я пишу слово «Правило» с заглавной буквы. Если вы видите слово «правило», написанное со строчной буквы, — это просто

обычное правило, а не какое-то формально описанное в этой книге. Без заглавной буквы могла бы возникнуть путаница, и, надеюсь, вы меня за это простите.

Надеюсь, книга вам понравится и вы откроете для себя несколько полезных мыслей, которые помогут вам отточить свои навыки программирования.

Girls Who Code

Все авторские отчисления от этой книги направляются организации Girls Who Code, прилагающей все усилия ради того, чтобы помочь девушкам открыть увлекательный и полезный мир программирования. Когда я заканчивал колледж, более трети выпускников по компьютерным наукам составляли женщины; сегодня же их доля составляет около одной пятой. Я считаю, что при более сбалансированном гендерном соотношении все мы только выиграли бы. Надеюсь, что вы тоже так считаете. Поддержка Girls Who Code, будь то финансовая или в форме волонтерства, — это шаг к воплощению надежды в реальность.

Условные обозначения

В этой книге используются следующие типографские условные обозначения:

Курсив

Используется для новых терминов, URL, адресов электронной почты, имен файлов и расширения файлов.

`Моноширинный шрифт`

Используется для программного кода, а также в тексте для обозначения таких элементов программы, как имена переменных и функций, базы данных, типы данных, переменные среды, операторы и ключевые слова.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) можно скачать по адресу: *https://oreil.ly/rules-of-programming-code*.

Если у вас возникнут какие-то технические вопросы или если вы столкнетесь с какой-либо проблемой при использовании примеров кода, то вы можете отправить сообщение по адресу электронной почты: *bookquestions@oreilly.com*.

Эта книга призвана помочь вам в вашей работе. Как правило, любой упоминаемый в книге фрагмент кода вы можете свободно использовать в своих

программах и в своей документации. Вам не нужно обращаться к нам за разрешением, если вы не воспроизводите значительную часть кода. Например, если вы захотите включить в свою работу несколько строк из примера, то получать разрешение для этого не нужно. Но если вы хотите продавать или распространять примеры из книг O'Reilly, то для этого требуется разрешение. Если вы отвечаете на какой-то вопрос и приводите пример из этой книги или ссылку на эту книгу, то разрешения не требуется, но если вы собираетесь использовать большое количество примеров из этой книги в документации к своему продукту, то для этого нужно получить разрешение.

Мы признательны всем, кто указывает авторство, но обычно не требуем такого указания. При этом, как правило, указание авторства подразумевает упоминание источника, автора, издателя и ISBN, например: «"Правила программирования", Крис Циммерман (O'Reilly). Copyright 2023 Крис Циммерман, 978-1-098-13311-5».

Если вы считаете, что ваше использование примеров кода выходит за рамки добросовестного использования или данного выше разрешения, пожалуйста, свяжитесь с нами по адресу: *permissions@oreilly.com*.

O'Reilly Online Learning

На протяжении более 40 лет O'Reilly Media предоставляет знания и навыки в области технологий и бизнеса, а также предлагает материалы, помогающие компаниям добиваться успеха.

Наша уникальная сеть экспертов и новаторов делится своим опытом и знаниями в книгах, статьях и на нашей онлайн-платформе обучения. Онлайн-платформа O'Reilly предоставляет по запросу доступ к живым курсам, углубленным учебным программам, интерактивным средам программирования, а также обширной коллекции текстов и видео от O'Reilly и более чем 200 других издателей. Дополнительная информация указана на странице по адресу: *https://oreilly.com*.

Как с нами связаться

Все касающиеся этой книги вопросы и комментарии направляйте издателю:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (США и Канада)
707-829-0515 (международные и местные звонки)
707-829-0104 (факс)

У этой книги есть своя веб-страница, на которой публикуются исправления, примеры и дополнительная информация: *https://oreil.ly/rules-of-programming*.

Если вы хотите поделиться своими впечатлениями, высказать комментарии или задать вопросы автору, зайдите на сайт книги *The Rules of Programming*, где вы найдете нужные указания. Также вы можете написать по адресу: *bookquestions@oreilly.com*, чтобы оставить свои отзывы или задать технические вопросы по поводу этой книги.

Новости и информацию о наших книгах и курсах можно найти по адресу: *https://oreilly.com*.

Благодарности

Прежде всего я благодарю свою прекрасную и талантливую жену Лору, которая убеждала меня тратить время на эту книгу, а не на всякие другие полезные дела, какими я мог бы заняться.

Огромная благодарность всем, кто помогал развивать описанные в этой книге Правила. Среди них — все бывшие и нынешние программисты Sucker Punch, осознанно или неосознанно внесшие свой вклад в книгу; особенно же стоит отметить следующих: Апурва Бансал, Крис Хайдорн, Дэвид Майер, Эрик Блэк, Эван Кристенсен, Джеймс Макнилл, Джасмин Патри, Нейт Слоттоу, Мэтт Дурасофф, Майк Гаффни, Ранджит Раджагопалан, Роб Макдэниел, Сэм Холли, Шон Смит, Уэс Грэндмон и Уильям Росситер.

Я говорю спасибо также всем, кто не связан с Sucker Punch и кто смог посмотреть на книгу свежим взглядом со стороны: Адаму Барру, Андреасу Фредрикссону, Колину Брайару, Дэвиду Оливеру, Максу Шуберту, Майку Гутманну и Сету Файну.

Отдельная особая благодарность отважным читателям, одолевшим все без исключения Правила: Адриану Бентли, Биллу Рокенбеку, Яну Миксовски и Джулиену Мерсерону. Я официально обязан отплатить вам ответной услугой.

И наконец, спасибо всей команде O'Reilly, которая терпеливо поддерживала меня во всех моих неуклюжих попытках написать эту книгу: Чарльзу Румелиотису, Грегори Хайману, Либби Джеймс, Мэри Треселер, Саре Хантер, Сьюзанн Хьюстон и особенно Саре Грей, оказавшей всем вам огромную услугу и выкинувшей из книги все самые несмешные шутки, которые я так и норовил в нее добавить.

История Правил

«Правила программирования» были созданы от отчаяния.

Около десяти лет я руководил командами программистов в Microsoft, а в 1997 году стал соучредителем игровой компании Sucker Punch. Обе компании добились успеха во многом благодаря своей способности привлекать первоклассных программистов и создавать из них великолепные коллективы. Sucker Punch может по праву похвастаться чередой успешных игр, которые выпускала на протяжении 25 лет. Прежде всего стоит отметить три игры из серии Sly Cooper, рассчитанных на детей всех возрастов и позволявших им прикоснуться к полной приключений жизни енота-вора Слая Купера и его друзей. Далее идут пять игр из серии inFamous, в которых игрокам предоставлялся выбор — воспользоваться суперсилами во благо или во зло. И, наконец, наше величайшее творение — это игра Ghost of Tsushima, в которой игроки примеряют на себя роль одинокого самурая, сражающегося против вторгшихся в Японию в 1274 году захватчиков*.

Как в Microsoft, так и в Sucker Punch важной составляющей стратегии найма был прием на работу молодых умных программистов с последующим их обучением профессиональным методам. Практика эта, несомненно, показала свой успех, но она же порождала и особую разновидность разочарования.

Раз за разом я сталкивался с одной и той же проблемой. К нам в команду приходил очередной новый программист, часто только что окончивший колледж. Я проверял какую-нибудь новую функцию, которую он планировал добавить в код — обычно для решения очень простой проблемы, — и оказывалось, что он пытался написать код, решающий гораздо более крупную проблему, которая включала в себя ту самую простую и конкретную задачу в качестве частного случая.

Но нам вовсе не нужно было решать ту самую большую проблему, по крайней мере не в тот момент! Постоянно оказывалось, что решение большой проблемы — это посредственное решение для той простой стоявшей

---

* Самые наблюдательные из вас, вероятно, заметили, что я не упомянул первую игру Sucker Punch, а именно Rocket: Robot on the Wheels. Это потому что в нее играли немногие; если вы находитесь в числе избранных, поздравляю вас.

перед нами задачи. Решение более сложное в использовании, сложное для понимания и потенциально содержащее гораздо большее количество ошибок. Но сколько бы я ни повторял на ревизии кода* о том, что программистам нужно браться только за те задачи, которые им по плечу и которые они понимают, новички продолжали делать всё по-своему.

От отчаяния я пошел на решительные меры. «Ну что ж, — сказал я. — Вот вам новое правило. Пока у вас нет трех примеров решения конкретной проблемы, вы не имеете права придумывать обобщенное решение».

К моему удивлению и радости, это сработало! Превратив общие рассуждения в конкретное правило с четкими критериями, я донес-таки до своих подчиненных нужную мысль. Конечно, большинство наших новых программистов все равно хотя бы раз совершали подобную ошибку чрезмерного обобщения, но сформулированное мной правило помогало им не повторять ее. И оно же помогало им понимать, когда действительно нужно было переходить к обобщению задачи. Меньше трех конкретных примеров? Не обобщай. Три или больше? Пора придумывать что-то большее.

Правило сработало, потому что его было легко запомнить, а ситуации, в которых оно применялось, было легко распознать. Выходя за границы четко определенной текущей задачи, программисты вспоминали правило, подсчитывали, сколько примеров такой задачи они уже выполнили, и решали, что им делать дальше — обобщать или решать конкретную задачу. В результате код у них выходил лучше.

Со временем мы определили и другие важные элементы философии Sucker Punch, которые можно было свести к легким для запоминания фразам, точнее, афоризмам. У афоризмов (или пословиц), то есть коротких и емких высказываний, заключающих в себе некую существенную истину, долгая история. Вы и сами можете с ходу назвать несколько афоризмов по памяти. Да что там — вы бы с легкостью вспомнили несколько пословиц только про птиц. Вот несколько примеров:

- Цыплят по осени считают**.
- Лучше синица в руках, чем журавль в небе.
- Слово не воробей — вылетит, не поймаешь.
- Не клади все яйца в одну корзину.

Афоризмы приживаются, потому что они работают. На современном жаргоне их можно было бы назвать «вирусными». Они «заряжали» людей

---

* Любой код в рамках какого-либо проекта Sucker Punch всегда проходит ревизию. Подробнее об этом говорится в Правиле 6.

** Похожее высказывание встречается еще в «Новых сонетах и прелестных памфлетах» Томаса Хауэлла 1570 года: «Не считай своих цыплят, покуда они не вылупились...» Небольшой пример долговечности хорошего афоризма.

крупицами мудрости на протяжении тысячелетий*. Неудивительно, что они оказались эффективным способом «заразить» новых членов команды философией программирования Sucker Punch.

Итак, шаг за шагом некогда единственное правило разрослось в целый свод описанных в этой книге «Правил программирования». Они отражают многие важнейшие аспекты принятой в Sucker Punch культуры программирования, то есть те принципы, которые, по нашему мнению, привели нас к успеху, и те идеи, которые новые члены коллектива должны усвоить для эффективной работы. И их время от времени нужно вспоминать даже таким опытным программистам, как *я*!

Каждая из следующих глав описывает одно из Правил со множеством примеров, иллюстрирующих стоящую за ним идею. Прочитав главу, вы получите четкое представление о каком-то конкретном практическом методе программирования и о ситуациях, в которых применим этот метод.

Окажутся ли Правила столь же «вирусными» в формате книги? Давайте проверим.

---

* Сам термин *афоризм* ввёл в обращение древнегреческий целитель Гиппократ около 400 года до н. э. Строго говоря, это было слово Ἀφορισμός. Так назывался его свод правил диагностики и лечения, некоторые из которых сохраняют свою силу и спустя тысячелетия, как, например, афоризм 13 из раздела 6: «Приступ чихания излечивает от икоты». Что абсолютно правда.

Как не соглашаться с Правилами

Надеюсь, вы не станете слепо следовать каждому описанному в этой книге 21 Правилу.

Если вы вдруг заметите, что просто вежливо киваете, читая мои Правила с примерами, и говорите: «Ну да, это логично; примеры знакомы, у меня тоже была такая мысль, просто я описывал ее другими словами», — ну что ж, я сочту это провалом.

Я хочу вам дать пищу для размышлений. В идеале хотелось бы, чтобы вы получили одно-два озарения. Возможно, я дам какое-то название тому смутному ощущению, которое у вас уже было, или покажу четкий пример того, чему вы не могли найти определение. Может, я даже предложу вам нечто новое для осмысления.

Но также вероятно, что вы столкнетесь с одной-двумя идеями, с которыми не согласитесь. Вы можете даже подумать, что в чем-то я совершенно не прав и что какое-то из Правил — это плохой совет.

И это прекрасно! Найти Правило, с которым вы не соглашаетесь, — это повод глубже задуматься. Отвергать его тут же, не задумываясь лишний раз, было бы ошибкой.

Я обещаю, что каждое из описанных здесь правил имеет свой смысл, но не утверждаю, что это абсолютный закон. Одно и то же правило может быть хорошим для нас и одновременно неподходящим для вас. Поняв почему, вы лучше поймете себя и свою философию программирования. Вы осознаете различия между компанией Sucker Punch и вашей собственной командой; вы поймете, почему эти различия делают какое-то Правило важной частью нашей культуры, но почему оно плохо совместимо с вашей культурой. В Правилах используются примеры из видеоигр, которые создавала студия Sucker Punch и в которых отражены некоторые наши особенности программирования в сфере видеоигр. Многие другие аспекты затрагиваются в последней главе «Заключение. Настраивайте правила под себя».

Когда я сталкиваюсь с каким-то утверждением о философии программирования, которое противоречит моему собственному опыту, я стараюсь осознать его с помощью следующих шагов:

1. Найти в высказывании не только недостатки, но и какое-то зерно истины. Возможно, я не соглашаюсь с утверждением, исходя из каких-то своих личных соображений. При каких обстоятельствах это утверждение было бы верным?

2. Рассмотреть проблему с другой стороны. При каких обстоятельствах моя точка зрения была бы ошибочной? Какие условия меняют истинность утверждения?

3. Вспомнить о том, что обстоятельства меняются. Утверждение может показаться вам неподходящим именно сейчас, но оказаться верным для другого вашего проекта. Так вы определите, в какой ситуации ваша философия должна будет измениться; в следующий раз вы быстрее распознаете ситуацию, в которой можно было бы и прислушаться к данному утверждению.

Я проходил через этот процесс много раз, и один из примеров иной ситуации — это так называемая «разработка через тестирование» (TDD, test-driven development). Она не сочетается с принятыми у нас в Sucker Punch методами разработки, но мы понимаем, что в каких-то случаях она оправданна. Читая Правила и размышляя над ними, вы тоже поймете, почему наши конкретные условия не подходят для TDD, но это вовсе не значит, что условия эти не могут поменяться. Мы внимательно следим за ситуацией; если что-то изменится, то изменится и наша философия.

Итак, я надеюсь, что вы найдете что-то полезное даже в тех Правилах, с которыми не согласны… но я пойму вас и в том случае, если вы решите последовать другому принципу, описанному Дороти Паркер:

Этот роман не нужно легкомысленно отбрасывать. Его нужно швырнуть прочь изо всех сил.

Если так, советую целиться во что-то мягкое.

Правило 1

Максимально просто, но не слишком просто

Программировать трудно.

Думаю, вы и сами уже пришли к такому выводу. Любой, кто берет в руки и читает книгу «Правила программирования», скорее всего:

- Умеет программировать, по крайней мере немного.
- Испытывает раздражение от того, что это не так просто.

Программировать трудно по многим причинам, и в свое время было разработано немало стратегий с целью облегчить это занятие. В этой книге рассматриваются примеры распространенных ошибок и Правила, помогающие их избежать; все они основаны на моем долгом личном опыте совершения похожих ошибок и попыток исправить ошибки других.

В этих Правилах прослеживается общая закономерность, некая объединяющая большинство из них сквозная тема. Лучше всего ее описывает цитата Альберта Эйнштейна, в которой говорится о принципах работы физика-теоретика: «Максимально просто [как можно проще], но не слишком просто»*. Этим Эйнштейн хотел сказать, что лучшая физическая теория — это самая простая из тех, что полностью описывают все наблюдаемые явления.

Если применить эту идею к сфере программирования, то можно утверждать, что лучшее решение любой задачи — это самое простое решение, которое удовлетворяет всем требованиям этой задачи. Лучший код — это самый простой код.

Предположим, что вы пишете код для подсчета количества битов в целочисленной переменной. Сделать это можно разными способами. Можно вос-

---

* Он почти наверняка не говорил этой фразы именно в таком виде. Потомки оказали Эйнштейну услугу, отточив его афоризмы. В письменных источниках можно найти только следующее наиболее близкое к этой фразе высказывание: «Нельзя отрицать, что высшая цель любой теории — сделать нередуцируемые базовые элементы как можно более простыми и немногочисленными, не отказываясь от адекватного представления любых опытных данных». В целом почти то же самое, просто не так емко. К тому же оригинальная цитата Эйнштейна немного длинновата для названия Правила.

пользоваться «битовыми трюками»* — например, обнулять по одному биту за раз и подсчитывать количество обнуленных битов:

```
int countSetBits(int value)
{
    int count = 0;
    while (value)
    {
        ++count;
        value = value & (value - 1);
    }
    return count;
}
```

Можно также реализовать эту идею без циклов, с использованием битовых сдвигов и масок для параллельного подсчета битов:

```
int countSetBits(int value)
{
    value = ((value & 0xaaaaaaaa) >> 1) + (value & 0x55555555);
    value = ((value & 0xcccccccc) >> 2) + (value & 0x33333333);
    value = ((value & 0xf0f0f0f0) >> 4) + (value & 0x0f0f0f0f);
    value = ((value & 0xff00ff00) >> 8) + (value & 0x00ff00ff);
    value = ((value & 0xffff0000) >> 16) + (value & 0x0000ffff);
    return value;
}
```

А можно просто написать самый очевидный код из возможных:

```
int countSetBits(int value)
{
    int count = 0;
    for (int bit = 0; bit < 32; ++bit)
    {
        if (value & (1 << bit))
            ++count;
    }
```

* Прошу прощения у всех, кто не программирует на C++, за битовые операции в следующих трех примерах. Обещаю, что в остальной части книги будет минимальное количество побитовых операций.

```
    return count;
}
```

Первые два решения можно назвать «умными»... но в данном случае это вовсе не комплимент*. Чтобы понять, как они работают, недостаточно беглого взгляда. В каждом имеется фрагмент кода, на котором посторонний пользователь спотыкается и начинает задавать лишние вопросы. Конечно, приложив некоторые усилия, можно разобраться в их логике, и их «распутывание» может даже немного позабавить. Но распутывание этих решений требует некоторых умственных затрат.

И это даже притом что я заранее сказал, что должны делать функции, а их названия напрямую указывают на их цель. Если бы вы не знали, что код должен подсчитывать биты переменной, разбор первых двух примеров потребовал бы еще больше усилий.

С последним вариантом всё иначе. Совершенно очевидно, что он подсчитывает биты некоего числа. Он прост, насколько это возможно, но не проще, и потому он лучше первых двух решений**.

Как измерять простоту

Измерить «простоту» или «сложность» кода можно разными способами.

Можно судить о простоте на основании того, насколько легко понимают код ваши коллеги. Если случайно выбранный сотрудник-программист может бегло прочитать фрагмент кода и без усилий понять его — значит, код достаточно прост.

Можно также определять степень простоты по тому, насколько легко создавать такой код — с учетом не только времени, потраченного на его написание, но и времени, нужного для полной реализации без ошибок (багов)***. Доводить до ума сложный код труднее, и он требует значительных усилий ума; простой код сделать работоспособным легче.

---

* В правдоподобной альтернативной Вселенной это правило называлось бы «Умничанье — это не добродетель».

** В современных процессорах имеется специальная инструкция для подсчета количества битов в значении переменной — например, popcnt на процессорах x86, которая выполняется за один такт. Можно также воспользоваться SIMD-инструкциями, которые подсчитывают множество битов еще быстрее, чем popcnt. Но все эти способы сложны для понимания и зависят от конкретной модели процессора. Если нет по-настоящему веской причины прибегать к чему-то более сложному, я предпочел бы самый простой вариант countSetBits.

*** Разумеется, без ошибок в пределах погрешности эксперимента. Всегда существуют баги, которые вы еще не обнаружили.

Эти два критерия, конечно, во многом пересекаются. Код, который легко писать, обычно так же легко и читать. Измерять сложность можно и другими критериями:

Объем написанного кода

Более простой код обычно короче, хотя иногда можно втиснуть всё в одну строку, сделав ее невероятно сложной.

Количество вводимых концепций

Простой код обычно основан на концепциях, известных всей вашей команде; он не вводит каких-то новых идей, способов осмысления задач или новой терминологии.

Время, необходимое для объяснения

Простой код легко объяснить — на ревизии кода он настолько очевиден, что проверяющие рассматривают его без задержек. Сложный код требует объяснений.

Код, который кажется простым по одному из этих критериев, скорее всего, будет казаться простым и по другим. Вам просто нужно выбрать, какому критерию удобнее и легче следовать вам самим в своей работе, но я рекомендую начать с легкости создания и легкости понимания. Сосредоточившись на том, чтобы быстро получить работающий и легко читаемый код, вы создадите простой код.

...но не слишком просто

Да, иметь дело с простым кодом, конечно, удобнее, но он при этом все равно должен выполнять поставленную перед ним задачу.

Представьте, что вам нужно посчитать, сколько существует способов подняться по лестнице с заданным количеством ступеней, при условии, что за один шаг можно преодолеть одну, две или три ступени. Например, для лестницы с двумя ступенями существуют только два способа подъема — можно шагнуть на первую ступень или перешагнуть через нее. Аналогично для лестницы с тремя ступенями существуют четыре способа: наступить на первую ступень, наступить на вторую, наступить на первую и вторую или сразу шагнуть на третью. Для четырех ступеней существуют семь способов, для пяти — тринадцать, и т. д.

Можно написать простой код для рекурсивного подсчета:

```
int countStepPatterns(int stepCount)
{
    if (stepCount < 0)
        return 0;
    if (stepCount == 0)
        return 1;
    return countStepPatterns(stepCount - 3) +
           countStepPatterns(stepCount - 2) +
           countStepPatterns(stepCount - 1);
}
```

Основная идея тут в том, что любой подъем по лестнице должен завершаться шагом на верхнюю ступень с одной из трех предыдущих ступеней. Суммируя количество способов подняться на каждую из этих трех ступеней, мы получаем общее количество подъема по лестнице в целом. После этого остается лишь определить базовые случаи. Для упрощения рекурсии в показанном коде для базового случая используется отрицательное количество ступеней.

К сожалению, это решение неработоспособно. Точнее, оно работает для небольших значений **stepCount**, но на вычисление значения **countStepPatterns(20)** требуется примерно вдвое больше времени, чем для вычисления **countStepPatterns(19)**. Компьютеры считают очень быстро, но их скорость быстро нивелирует экспоненциальный рост сложности. Когда я тестировал этот код, то время его выполнения начало заметно увеличиваться при значениях **stepCount** в районе двадцати.

Получается, что если нужно подсчитать количество способов подъема на более длинные лестницы, то этот код оказывается чрезмерно упрощенным. Основная проблема тут в том, что все промежуточные результаты значения **countStepPatterns** пересчитываются раз за разом с самого начала, что приводит к экспоненциальному росту времени выполнения. Стандартное решение этой проблемы — мемоизация (то есть запоминание вычисленных промежуточных значений для их повторного использования), как показано в следующем примере:

```
int countStepPatterns(unordered_map<int, int> * memo, int rungCount)
{
    if (rungCount < 0)
        return 0;
    if (rungCount == 0)
        return 1;
```

```
    auto iter = memo->find(rungCount);
    if (iter != memo->end())
        return iter->second;

    int stepPatternCount = countStepPatterns(memo, rungCount - 3) +
                    countStepPatterns(memo, rungCount - 2) +
                        countStepPatterns(memo, rungCount - 1);
    memo->insert({ rungCount, stepPatternCount });
    return stepPatternCount;
}

int countStepPatterns(int rungCount)
{
    unordered_map<int, int> memo;
    return countStepPatterns(&memo, rungCount);
}
```

С использованием мемоизации каждое значение вычисляется только один раз и сохраняется в хеш-таблице. Последующие вызовы находят вычисленное значение в хеш-таблице примерно за постоянное время, и сложность не растет экспоненциально. Код с мемоизацией становится чуть более сложным, но зато не так сильно ограничивает производительность.

Можно также воспользоваться динамическим программированием, пожертвовав небольшой долей концептуальной простоты ради большей простоты кода:

```
int countStepPatterns(int rungCount)
{
    vector<int> stepPatternCounts = { 0, 0, 1 };
    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
            stepPatternCounts[rungIndex + 1] +
            stepPatternCounts[rungIndex + 2]);
    }
    return stepPatternCounts.back();
}
```

Этот способ также работает достаточно быстро и оказывается даже проще рекурсивной версии с мемоизацией.

Иногда лучше упростить задачу, чем ее решение

Проблемы с оригинальной рекурсивной версией **countStepPatterns** возникли при расчете подъема на длинные лестницы. Самый простой код прекрасно работал для небольшого количества ступеней, но столкнулся с экспоненциальным падением производительности для больших значений. Последующие версии позволили избежать экспоненциального роста ценой некоторого усложнения… но вскоре столкнулись с другой проблемой.

Когда я запустил предыдущий код для вычисления **countStepPatterns(36)**, то получил верный ответ: 2 082 876 103. Но при вызове **countStepPatterns(37)** был возвращен ответ –463 960 867. А это явно неверно!

Дело в том, что используемая мной версия C++ хранит целые числа как 32-битные значения со знаком, и при вычислении **countStepPatterns(37)** разрядная сетка переполнилась. Подняться на лестницу из 37 ступеней можно 3 831 006 429 способами, и это число слишком велико, чтобы его можно было выразить 32-битным целым числом со знаком.

Возможно, код все еще чрезмерно прост. Должен же быть какой-то способ подсчета вариантов подъема на лестницу любой длины? В C++ нет стандартного решения для обработки очень больших целых чисел, но существуют (многочисленные) библиотеки с открытым исходным кодом, реализующие различные варианты целочисленных значений произвольной точности. Можно также написать несколько сотен строк кода и реализовать собственный тип:

```
struct Ordinal
{
public:
    Ordinal() :
        m_words()
        { ; }
    Ordinal(unsigned int value) :
        m_words({ value })
        { ; }

    typedef unsigned int Word;
```

```
    Ordinal operator + (const Ordinal& value) const
    {
        int wordCount = max(m_words.size(), value.m_words.size());

        Ordinal result;
        long long carry = 0;

        for (int wordIndex = 0; wordIndex < wordCount; ++wordIndex)
        {
            long long sum = carry +
                            getWord(wordIndex) +
                            value.getWord(wordIndex);
            result.m_words.push_back(Word(sum));
            carry = sum >> 32;
        }

        if (carry > 0)
            result.m_words.push_back(Word(carry));

        return result;
    }

protected:
    Word getWord(int wordIndex) const
    {
        return (wordIndex < m_words.size()) ? m_words[wordIndex] : 0;
    }

    vector<Word> m_words;
};
```

Подставив в последний пример **Ordinal** вместо **int**, мы получим точные ответы для более длинных лестниц:

```
Ordinal countStepPatterns(int rungCount)
{
    vector<Ordinal> stepPatternCounts = { 0, 0, 1 };

    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
```

```
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
            stepPatternCounts[rungIndex + 1] +
            stepPatternCounts[rungIndex + 2]);
    }
    return stepPatternCounts.back();
}
```

Так что же, проблема решена? Введенный нами тип **`Ordinal`** позволяет вычислять точные значения для гораздо более длинных лестниц. Конечно, добавлять несколько сотен строк кода для реализации **`Ordinal`** — не лучший вариант, особенно притом что сама функция **`countStepPatterns`** занимает всего 14 строк. Но разве это не та цена, которую нужно заплатить за корректное решение задачи?

Пожалуй, нет. При отсутствии простого решения задачи стоит для начала проанализировать собственно задачу и только потом думать о сложном решении. Действительно ли нужно решить именно такую задачу? Или вы сделали какие-то необоснованные предположения о задаче, которые усложняют ее решение?

В данном случае на практике в задаче подсчета вариантов подъема по реальным лестницам длина этих лестниц, скорее всего, будет как-то ограничена. Если в каждой из имеющихся лестниц не более, скажем, 15 ступеней, то адекватным будет любой из показанных в этом разделе вариантов, даже самый первый простой рекурсивный пример. Можно просто добавить в одну из функций **`assert`** с комментарием об имеющихся ограничениях и на этом счесть задачу выполненной:

```
int countStepPatterns(int rungCount)
{
    // ПРИМЕЧАНИЕ (Крис): при количестве ступеней больше 36
    // для int рассчитать результат невозможно...

    assert(rungCount <= 36);
    vector<int> stepPatternCounts = { 0, 0, 1 };
    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
            stepPatternCounts[rungIndex + 1] +
```

```
            stepPatternCounts[rungIndex + 2]);
    }
    return stepPatternCounts.back();
}
```

Или же, если требуется поддержка действительно длинных лестниц (например, лестниц по обслуживанию ветрогенератора), то, возможно, можно будет обойтись лишь приблизительным подсчетом вариантов? По всей видимости — да, и в таком случае легко заменить целые числа на вещественные (числа с плавающей точкой). Настолько легко, что я даже не буду показывать код.

Итак, в конечном счете любая система столкнется с переполнением. Попытка решить крайние случаи задачи всегда приведет к излишне сложному решению. Не попадайтесь в эту ловушку и не ориентируйтесь на самую строгую формулировку задачи. Гораздо лучше получить простое решение для той части задачи, которая действительно требует решения, чем придумывать сложное решение для более широкой формулировки. Если у вас не получается упростить решение, попробуйте упростить саму задачу.

Простые алгоритмы

Иногда код усложняется из-за неудачного выбора алгоритма.

В конце концов, любую конкретную задачу можно решить множеством способов, и некоторые из них сложнее других. Чем проще алгоритм, тем проще код. Проблема здесь только в том, что простой алгоритм не всегда очевиден!

Допустим, вам нужно написать код для тасования колоды карт. Очевидный подход — сымитировать тасование «риффл», которое, вы, вероятно, пытались освоить в детстве: разделить колоду на две стопки, а затем соединить их примерно с равным шансом для каждой карты с каждой стороны оказаться следующей в общей колоде. Повторять до тех пор, пока колода не перемешается*.

Решение может выглядеть примерно так:

```
vector<Card> shuffleOnce(const vector<Card> & cards)
```

---

* Считается, что колода карт достаточно хорошо перемешивается после семи тасовок в стиле «риффл». После четырех или пяти тасовок колода может быть недостаточно перемешанной. И да, мои родные сердятся, когда я слишком долго тасую карты перед очередной партией. «Мы здесь, чтобы играть в карты, Крис, а не смотреть, как ты их перетасовываешь». Хороший пример «горя от ума».

```
{
    vector<Card> shuffledCards;

    int splitIndex = cards.size() / 2;
    int leftIndex = 0;
    int rightIndex = splitIndex;

    while (true)
    {
        if (leftIndex >= splitIndex)
        {
            for (; rightIndex < cards.size(); ++rightIndex)
                shuffledCards.push_back(cards[rightIndex]);
            break;
        }
        else if (rightIndex >= cards.size())
        {
            for (; leftIndex < splitIndex; ++leftIndex)
                shuffledCards.push_back(cards[leftIndex]);
            break;
        }
        else if (rand() & 1)
        {
            shuffledCards.push_back(cards[rightIndex]);
            ++rightIndex;
        }
        else
        {
            shuffledCards.push_back(cards[leftIndex]);
            ++leftIndex;
        }
    }
    return shuffledCards;
}

vector<Card> shuffle(const vector<Card> & cards)
{
    vector<Card> shuffledCards = cards;
    for (int i = 0; i < 7; ++i)
    {
```

```
        shuffledCards = shuffleOnce(shuffledCards);
    }
    return shuffledCards;
}
```

Такой алгоритм имитации тасовки «риффл» работает, и в моем коде он реализован довольно просто. Для проверки корректности всех индексов вам придется потратить немного усилий, но в целом он не так уж плох.

Но существуют и более простые алгоритмы перетасовки карт. Например, можно пополнять перемешанную колоду по одной карте за раз. На каждом прохождении цикла берите новую карту и меняйте ее местами со случайной картой в текущей колоде. Это можно сделать и без дополнительной переменной:

```
vector<Card> shuffle(const vector<Card> & cards)
{
    vector<Card> shuffledCards = cards;

    for (int cardIndex = shuffledCards.size(); --cardIndex >= 0; )
    {
        int swapIndex = rand() % (cardIndex + 1);
        swap(shuffledCards[swapIndex], shuffledCards[cardIndex]);
    }
    return shuffledCards;
}
```

Согласно упомянутым ранее критериям простоты, эта версия превосходит предыдущую. На ее написание ушло меньше времени*. Ее легче читать. Кода здесь меньше. Его проще объяснить. Версия проще и лучше — не из-за самого кода, а благодаря более удачному выбору алгоритма.

Не теряйте суть задачи

Простой код легко читать, а самый простой код можно читать последовательно, сверху вниз, прямо как книгу. Однако программы — это не книги. В погоне за суперпростотой легко получить код, который будет сложно отслеживать, если поток выполнения при этом не так уж прост. Запутанный

* По крайней мере, в моем эксперименте. Я немного «поумничал» с индексами и условиями в реализации «риффла», и он сработал после нескольких попыток. Код со случайным выбором сработал сразу же.

код заставляет вас перескакивать с места на место, и проследить ход его выполнения будет сложнее.

Запутанный код может возникнуть в результате попыток четко реализовывать одну идею только в одном месте. Возьмем, к примеру, первый код тасования карт по типу «риффл». Фрагменты кода для правой и левой стопок карт выглядят довольно похоже. Логику перемещения одной карты или серии карт в перемешанную стопку можно было бы вынести в отдельные функции и затем вызывать их из **shuffleOnce**:

```
void copyCard(
    vector<Card> * destinationCards,
    const vector<Card> & sourceCards,
    int * sourceIndex)
{
    destinationCards->push_back(sourceCards[*sourceIndex]);
    ++(*sourceIndex);
}

void copyCards(
    vector<Card> * destinationCards,
    const vector<Card> & sourceCards,
    int * sourceIndex,
    int endIndex)
{
    while (*sourceIndex < endIndex)
    {
        copyCard(destinationCards, sourceCards, sourceIndex);
    }
}

vector<Card> shuffleOnce(const vector<Card> & cards)
{
    vector<Card> shuffledCards;

    int splitIndex = cards.size() / 2;
    int leftIndex = 0;
    int rightIndex = splitIndex;

    while (true)
    {
```

```
        if (leftIndex >= splitIndex)
        {
            copyCards(&shuffledCards, cards, &rightIndex, cards.
size());
            break;
        }
        else if (rightIndex >= cards.size())
        {
            copyCards(&shuffledCards, cards, &leftIndex, splitIndex);
            break;
        }
        else if (rand() & 1)
        {
            copyCard(&shuffledCards, cards, &rightIndex);
        }
        else
        {
            copyCard(&shuffledCards, cards, &leftIndex);
        }
    }
    return shuffledCards;
}
```

Предыдущую версию **shuffleOnce** можно было читать последовательно сверху вниз; эту — нет, и она сложнее для восприятия. При чтении кода **shuffleOnce** натыкаешься на функции **copyCard** или **copyCards**, в результате чего приходится искать эти функции, разбираться, что они делают, возвращаться к исходной функции и сопоставлять переданные из **shuffleOnce** аргументы с тем, что удалось узнать про **copyCard** или **copyCards**. Всё это сложнее, чем просто читать циклы в оригинальной версии **shuffleOnce**.

Итак, следуя принципу «без повторений», мы потратили больше времени на написание кода, который, к тому же, оказался труднее для восприятия*. И самого кода стало больше! Попытка избавиться от дублирования усложнила код, а не упростила.

Конечно, существует множество веских причин для устранения дублирования. Но важно помнить о том, что у всего есть своя цена, и для неболь-

---

* Опять-таки, судя по эксперименту. Для компиляции потребовалось несколько попыток, так как я выбирал между указателями и ссылками.

шого объема кода, а также для простоты восприятия бывает лучше оставить фрагменты с дубликатами. Такой код будет проще писать и проще читать.

И Правило одно, чтоб править ими всеми

Многие последующие Правила в этой книге будут возвращаться к описанной выше теме простоты и к стремлению делать код максимально простым, но не слишком простым.

В своей основе программирование — это борьба со сложностью. Добавление нового функционала часто влечет за собой усложнение кода, и по мере увеличения сложности кода работать с ним становится всё труднее, а прогресс замедляется. В конечном счете можно достичь своеобразного «горизонта событий», когда любая попытка двигаться вперед — исправить ошибку или добавить какую-то функцию — создает столько же проблем, сколько и решает. Дальнейшее развитие становится практически невозможным.

В конце концов, именно сложность и уничтожает ваш проект.

Отсюда следует, что эффективное программирование — это отсрочка неизбежного. По мере добавления функций и исправления ошибок нужно добавлять как можно меньше сложности. Ищите способы уменьшения сложности или проектируйте программу с таким расчетом, чтобы новые функции не слишком увеличивали общую сложность всей системы. Работайте сообща с коллегами над тем, чтобы сохранять как можно больше простоты.

При должном усердии неизбежный порог можно отдалять бесконечно. Первые строки кода для Sucker Punch я написал 25 лет назад, и с тех пор наша кодовая база постоянно развивалась. Конца этому не видно — сейчас наш код неизмеримо сложнее, чем четверть века назад, но у нас получается контролировать его сложность, и мы по-прежнему эффективно развиваем его.

Мы научились управлять сложностью — научитесь и вы. Будьте начеку, помните о том, что сложность — ваш главный противник, и у вас все получится.

Правило 2

Баги заразны

Среди программистов широко распространено банальное утверждение: чем раньше найти баг, тем проще его будет исправить. Вообще-то так и есть… но, думаю, еще лучше сказать, что чем позже находят баг, тем больше головной боли он доставляет и тем труднее его исправлять.

Как только будет обнаружен какой-то баг, люди сразу же начнут писать код, так или иначе учитывающий этот баг. Иногда эти «костыли» находятся непосредственно рядом с багом, в той же самой системе. В других случаях такие «костыли» могут находиться «ниже по течению» и учитывают некорректные результаты, выдаваемые вашим багом. Либо они находятся «выше по течению», и какой-то фрагмент кода работает только потому, что каким-то хитрым образом вызывает фрагмент с багом.

Всё это вполне естественно, и совсем избежать таких случаев невозможно. Наше внимание в первую очередь привлекает нечто необычное, идущее не так, как нужно, а не то, что работает правильно. Когда что-то ломается, мы ищем причину поломки. Но когда всё работает, никто не старается выяснить почему. Если код работает, или, по крайней мере, выглядит работающим, мы, как правило, с удовлетворением киваем и считаем, что всё в порядке, хотя в нем могут скрываться ошибки. А если их не искать, то и не поймешь, что код заработал не потому что он хорошо написан, а потому что так получилось случайно.

Всё это верно как для вашего собственного кода, так и для кода, написанного другими программистами, который вызываете вы в своем коде. После того, как вы внесете в кодовую базу фрагмент с багом, эта база будет медленно, но неотвратимо пополняться другими фрагментами, полагающимися на ваш баг. Все эти скрытые зависимости становятся заметными только при исправлении очевидного бага и когда загадочным образом перестает работать какая-то другая часть вашего проекта.

Чем раньше вы найдете баг, тем меньше будет времени для разрастания зависимостей. Так вам придется очищать меньше связей, а ведь зачастую это самая трудоемкая часть исправления. Уж слишком распространена ситуация, когда на исправление последствий бага уходит больше времени, чем на исправление самого бага.

Баги удобно воспринимать как своего рода «заразу». Каждый баг в вашей системе стремится породить новые баги — способы косвенного преодоления этого бага (пресловутые «костыли») или фрагменты, полагающиеся на этот баг. Лучший способ остановить эпидемию — устранять баги как можно раньше, пока их пагубное влияние не распространилось на всю систему.

Не полагайтесь на пользователей

Ну ладно, мы поняли, что нужно выявлять проблемы как можно раньше. Но как это сделать?

На что тут точно нельзя полагаться — так это на ваших пользователей. Пользователи — это далеко не лучшая первая линия обороны, будь то пользующиеся вашим кодом коллеги или тестирующие ваш код клиенты. Да, они порой могут сообщить о проблеме, но гораздо чаще подумают, что программа ведет себя так, как вы и задумывали. Именно отсюда берутся те самые запутанные зависимости — не только из-за того, что проблему не заметили, но и из-за того, что ее восприняли как часть замысла.

Можно попытаться улучшить ситуацию. Можно, например, постараться написать более качественную документацию для пользователей. Можно собрать всех коллег в переговорной, чтобы объяснить им новую систему или новый функционал. Можно создать актуальную и поддерживаемую вики с объяснением взаимосвязей между разными фрагментами кода или разместить техническую заметку на сайте поддержки. Все эти усилия действительно оправданны, и они помогают, хотя и требуют немалых затрат и разную степень эффективности, но они не решают проблему кардинально. Ваши пользователи изначально не понимают ваши намерения так же хорошо, как вы сами, поэтому они неизбежно будут принимать баги за «фичи», что бы вы ни делали.

Лучшее решение — это некая разновидность непрерывного автоматизированного тестирования. Большинство программистов согласятся с тем, что автоматизированное тестирование — это хорошо. На худой конец, программисты согласятся с тем, что автоматизированное тестирование — это хорошо, когда им занимаются другие программисты, независимо от того, находится ли у них самих время на такое тестирование.

Что касается автоматизированного тестирования, то на эту тему существует множество самодельных вариаций наряду с более формализованными методами, такими как «разработка через тестирование» (TDD, test-driven development).

В общем смысле идея заключается в том, что ваша система (или, что еще лучше, весь ваш проект) имеет некий набор тестов, которые можно удобно

и быстро запускать для тщательной* проверки системы (или проекта) с сообщением о возможных проблемах. Если тесты действительно быстрые и удобные, их будут запускать постоянно — например, при каждой компиляции или запуске проекта. Любые баги, обнаруженные на столь ранней стадии, легко уничтожить в зародыше. Если тесты быстры и удобны лишь теоретически, то их, как, правило, будут запускать в процессе «коммита» — и это все еще достаточно рано, чтобы не дать разрастись зависимостям, из-за которых исправлять баги так трудно.

Автоматизированное тестирование бывает сложным

Одни проекты и задачи поддаются автоматизированному тестированию лучше других.

Некоторые вещи тестировать сложно — либо потому, что трудно охватить все возможные входные данные, либо потому, что сложно проверить корректность выходных данных. Представьте, что вы разрабатываете новый аудиокодек со сжатием без потерь. Как написать для него автоматизированный тест?

Проверить, что ваш компрессор не вылетает, или измерить степень сжатия на каком-то наборе тестовых файлов легко. Но гораздо сложнее убедиться в том, что сжатый звук действительно похож на оригинал. Если вы действительно занимаетесь аудиокодеками, то у вас, вероятно, достаточно знаний в области цифровой обработки сигналов и вы умеете писать тексты, выявляющие очевидные проблемы, но в какой-то момент вам все равно придется надеть наушники на реальных людей и попросить их выбрать образец из трех сжатых вариантов. И такой тест невозможно запустить и выполнить быстро с максимальным удобством.

Кое-какой код сложно тестировать по самой его природе, потому что некоторые параметры трудно измерять количественно, как и трудно точно определить, в какой степени он удачен или неудачен. Под это определение попадает большая часть кода, который пишется в Sucker Punch. Ведет ли себя персонаж-торговец как настоящий торговец? Действительно ли анимация лица персонажа передает негодование или он просто выглядит так, будто съел что-то не то? Передает ли геймпад ощущение от стрельбы из лука?

---

* ...но, возможно, и не такой уж тщательной. Я не стану выносить категорических утверждений по поводу того, какой должен быть процент покрытия вашей системы автоматизированными тестами. Дальше вы узнаете, что у нас в Sucker Punch этот процент очень низок.

Работая над проектом, содержащим большие фрагменты сложного для тестирования кода, приходится использовать гибридную модель. Тестируйте то, что можно протестировать, контролируйте то, что можно контролировать, и помните, что вы проверяете не всё. Для этого, что не поддается автоматическому тестированию, понадобится ручное тестирование, так что планируйте работу с оглядкой на ручные тесты.

При этом код действительно можно писать так, чтобы его было легче тестировать благодаря самой его структуре.

Допустим, вы собираетесь написать внешний автоматизированный тест для некоторого фрагмента кода, то есть некий тестовый код, отдельный от разрабатываемого, который будет вызывать ваш код, отправлять в него некий набор входных данных, а затем сверять полученные данные с ожидаемыми. Так какой же должна быть структура кода, благодаря которой для него будет легче писать тесты?

Легче тестировать код без состояния

Одна из важных стратегий — это уменьшение количества кода, зависящего от состояния (то есть зависящего от изменяемых данных в памяти). Тестировать код, который не зависит от состояния, гораздо проще. Легко поддается тестированию любая чистая функция, то есть фрагмент кода, который зависит только от непосредственно вводимых данных, не имеет побочных эффектов и дает предсказуемые результаты.

Лучше так:

```
int sumVector(const vector<int> & values)
{
    int sum = 0;
    for (int value : values)
    {
        sum += value;
    }
    return sum;
}
```

Чем так:

```
int reduce(
    int initialValue,
    int (*reduceFunction)(int, int),
```

```
    const vector<int> & values)
{
    int reducedValue = initialValue;
    for (int value : values)
    {
        reducedValue = reduceFunction(reducedValue, value);
    }
    return reducedValue;
}
```

Для тестирования **sumVector** нужен лишь набор входных данных и ожидаемые выходные значения для них. Именно в этом хороши фреймворки разработки через тестирование. Но при необходимости учитывать состояние набора входных данных для тщательной проверки кода становится значительно сложнее.

Тестировать **reduce** сложнее, потому что в кажущемся стремлении к универсальности (и, возможно, как промежуточный шаг к многопоточности) функция многократно вызывает переданную ей функцию для обработки значений вектора **vector**. Она, конечно, справляется со своей обязанностью и суммирует значения вектора:

```
int sum(int value, int otherValue)
{
    return value + otherValue;
}

int vectorSum = reduce(0, sum, values);
```

Но тестирование функции **reduce** сопряжено с проблемами. Кто знает, что делает функция **reduceFunction**? Зависит ли она от какого-то внешнего состояния? Что произойдет, если у нее есть побочные эффекты? Что, если вызов этой функции удалит что-то из вектора **values**, по значениям которого проходят итерации? При тестировании **reduce** придется принимать во внимание и проверять все эти сценарии, а для этого нужен гораздо более сложный набор тестов по сравнению с тем, который потребовался бы для **sumVector**.

Для тщательного тестирования кода нужно передать ему репрезентативный набор всех возможных состояний, которые могут на него повлиять, а затем оценить выходные данные для этих состояний. В случае с чистой функцией единственное имеющее значение состояние — это аргументы этой самой функции. Но когда появляются побочные эффекты, внутреннее состояние

или вызовы произвольных функций, объем потенциально значимого состояния возрастает лавинообразно. В результате приходится идти на компромисс: либо смиряться с тем, что тесты будут охватывать не всё, либо прописывать количество тестовых сценариев, не поддающееся разумному контролю.

Рассмотрим простой пример. Представьте, что вы ведете отсортированный по приоритету список персонажей. У каждого персонажа есть свой приоритет, и по ним легко отсортировать список. Интерфейс прост:

```
class Character
{
public:
    Character(int priority);
    ~Character();

    void setPriority(int priority);
    int getPriority() const;

    static const vector<Character *> & getAllCharacters();

protected:

    int m_priority;
    int m_index;

    static vector<Character *> s_allCharacters;
};
```

Список **s_allCharacters** легко поддерживать в отсортированном состоянии, то есть так, чтобы все персонажи располагались в нем в порядке приоритета. Это можно делать инкрементально: отслеживать позицию каждого персонажа в списке и перемещать его «точечно» при изменении его приоритета. Для этого при создании персонажа нужно вставлять его в правильную позицию:

```
Character::Character(int priority) :
    m_priority(priority),
    m_index(0)
{
    int index = 0;
```

```
    for (; index < s_allCharacters.size(); ++index)
    {
        if (priority <= s_allCharacters[index]->m_priority)
            break;
    }
    s_allCharacters.insert(s_allCharacters.begin() + index, this);
    for (; index < s_allCharacters.size(); ++index)
    {
        s_allCharacters[index]->m_index = index;
    }
}
```

И очищать индексы при уничтожении персонажа:

```
Character::~Character()
{
    s_allCharacters.erase(s_allCharacters.begin() + m_index);
    for (int index = m_index; index < s_allCharacters.size(); ++index)
    {
        s_allCharacters[index]->m_index = index;
    }
}
```

И по минимуму перемещать персонажа в списке при изменении его приоритета:

```
void Character::setPriority(int priority)
{
    if (priority == m_priority)
        return;

    m_priority = priority;

    while (m_index > 0)
    {
        Character * character = s_allCharacters[m_index - 1];
        if (character->m_priority <= priority)
            break;

        s_allCharacters[m_index] = character;
        character->m_index = m_index;
```

```
        --m_index;
    }

    while (m_index + 1 < s_allCharacters.size())
    {
        Character * character = s_allCharacters[m_index + 1];
        if (character->m_priority >= priority)
            break;

        s_allCharacters[m_index] = character;
        character->m_index = m_index;

        ++m_index;
    }

    s_allCharacters[m_index] = this;
}
```

Код рабочий, но тестировать его сложно. В нем присутствует скрытое состояние, недоступное для внешнего теста. Некоторые баги можно обнаружить с помощью теста, создающего набор персонажей с приоритетами и проверяющего, действительно ли **getAllCharacters** возвращает их в правильном порядке, но другие баги останутся незамеченными. Текущий индекс может оказаться неверным, даже если персонажи располагаются в правильном порядке, а проверить это с помощью методов класса **Character** невозможно. Некорректные индексы могут привести к проблемам, но нет гарантии, что эти проблемы проявятся сразу (или вообще когда-либо проявятся). А при наличии трех отдельных фрагментов кода, каждый из которых пытается поддерживать корректность индексов, ошибиться очень легко.

Проще тестировать версию **Character** без внутреннего состояния, которая не пытается поддерживать сложную логику:

```
class Character
{
public:
    Character(int priority) :
        m_priority(priority)
    {
        s_allCharacters.push_back(this);
    }
```

```
    ~Character()
    {
        auto iter = find(
                        s_allCharacters.begin(),
                        s_allCharacters.end(),
                        this);
        s_allCharacters.erase(iter);
    }

    void setPriority(int priority)
    {
        m_priority = priority;
    }

    int getPriority() const
    {
        return m_priority;
    }

    static int sortByPriority(
        Character * left,
        Character * right)
    {
        return left->m_priority < right->m_priority;
    }

    static vector<Character *> getAllCharacters()
    {
        vector<Character *> sortedCharacters = s_allCharacters;

        sort(
            sortedCharacters.begin(),
            sortedCharacters.end(),
            sortByPriority);

        return sortedCharacters;
    }

protected:
```

```
    int m_priority;

    static vector<Character *> s_allCharacters;
};
```

Состояние здесь все еще присутствует, потому что мы отслеживаем всех персонажей в **s_allCharacters**, но оно не скрыто. Составлять тесты для этой версии, возможно, и не так просто, как для чистой функции, но значительно проще, чем для инкрементальной версии **Character**, которую мы рассматривали вначале.

В предыдущей версии, основанной на состоянии, приходилось бы соблюдать маниакальную осторожность с порядком операций. Теперь же, избавившись от лишнего состояния, можно просто проверять ожидаемые выходные данные и чувствовать себя в относительной безопасности.

Код без состояния даже изначально проще написать правильно. Таково скрытое преимущество разработки через тестирование — код, который легче тестировать, обычно и легче писать. Если вы заранее подумаете о том, как будете тестировать код, который собираетесь написать, то в итоге у вас получится нечто более простое.

Проверяйте состояние, от которого нельзя избавиться

Предположим, что обстоятельства вынуждают вас сохранить некоторое состояние. Допустим, таковы требования шаблона вызовов — например, если в вашем отсортированном списке персонажей незначительные изменения приоритетов чередуются с вызовами **getAllCharacters** и постоянная сортировка в реализации без состоянии перегружает кеш памяти.

Если сложно написать внешний тест из-за недоступности какого-то внутреннего состояния, напишите внутренний тест*. Один из простых способов — добавить метод аудита для ваших данных. В данном случае это функция аудита, проверяющая согласованность внутреннего состояния:

```
void Character::audit()
{
```

* Можно также как-нибудь предоставить тестовому коду доступ к внутреннему состоянию — например, ослабить инкапсуляцию с помощью объявления тестового класса «другом» (**friend**). Но по моему опыту удобнее добавлять внутренние тесты.

```
    assert(s_allCharacters[m_index] == this);
}
```

Это очень краткая функция аудита, но лишь потому, что я убрал из класса **Character** все интересное просто ради примера. На практике в настоящем классе **Character**, скорее всего, было бы больше внутреннего состояния и для него потребовалась бы более длинная функция **audit**.

Можно также проверить согласованность всего массива:

```
void Character::auditAll()
{
    for (int index = 0; index < s_allCharacters.size(); ++index)
    {
        Character * character = s_allCharacters[index];

        if (index > 0)
        {
            Character * prevCharacter = s_allCharacters[index - 1];
            assert(character->m_priority >= prevCharacter->m_priority);
        }
        character->audit();
    }
}
```

У такого внутреннего тестирования есть свои преимущества, особенно если рассматривать его как дополнение к внешним тестам. Часто внутренние проверки можно оставлять включенными постоянно, и они будут выполняться на реальных рабочих сценариях, а не на искусственных, созданных специально для модульного теста.

Разумеется, чтобы эти внутренние функции приносили пользу, их кто-то должен вызывать! Хорошее практическое правило заключается в том, чтобы вызывать **Character::audit** в конце любого метода, обновляющего состояние персонажа, и вызывать **Character::auditAll** при каждом изменении списка. Частоту аудита можно увеличивать или уменьшать в зависимости от потребностей.

Не доверяйте вызывающей стороне

В обычной практике вам, как программистам, придется писать код, который будут вызывать другие члены вашей команды. Даже если вы работаете над

одиночными проектами, спустя какое-то время ваш код будете вызывать вы сами, и этого будущего вас в таком случае можно будет назвать почти другим человеком. Будущий вы не вспомнит всех деталей, а другие специалисты и вовсе не будут их знать. Так что даже не думайте, будто вызывающая сторона разберется во всех подробностях!

В ваш код будут передавать несовместимые наборы аргументов. Обращающиеся к нему забудут вызывать ожидаемые функции инициализации и не вспомнят про функции завершения работы. Их вызывающая функция не будет соответствовать базовым требованиям для таких функций. Они всё сделают неправильно… и если вы не обнаружите ошибки, их никто не исправит. Вместо этого возникнет новая запутанность (зависимость) — на этот раз не из-за бага в вашем коде, а из-за ошибки в вызывающем коде.

Возможно, мое утверждение покажется нелогичным, но в таких случаях проще всего обнаружить баги не в вызывающем коде, а в вызываемом. Ошибки может допускать и вызывающая сторона, но именно у вас больше возможностей по их перехвату.

Под хорошим дизайном кода часто подразумевается такой дизайн, который даже не позволяет вызывающей стороне допустить ошибку в деталях, и это тема Правила 7 «Устраняйте сценарии сбоев». Но иногда настолько хороший код сделать не получается. Что же делать в таких случаях?

Предположим для примера, что вы работаете над симулятором физики твердых тел для трех разных разрабатываемых в вашей компании видеоигр. При этом вам нужно отслеживать какое-нибудь внутреннее состояние — например, способы, какими твердые тела контактируют друг с другом, — и это состояние должно где-то храниться. Но вы не можете просто вызывать **`operator new`**, как в стандартном коде на C++. Память ограничена, и у ваших клиентов имеются собственные менеджеры памяти, с которыми вам нужно как-то интегрироваться.

Простое и очевидное решение — попросить клиентов передать функции для выделения и освобождения нужной вам памяти на этапе инициализации. Меня так и подталкивало начать с нескольких плохих примеров инициализации, но я решил перейти сразу к достойному. Если вы и вправду работаете над симулятором физики твердых тел, то в нем, скорее всего, будет не два параметра инициализации, как в следующем фрагменте кода, а больше — например, гравитационная постоянная. Объедините все параметры инициализации в единой структуре, которая затем будет передаваться в единственную функцию инициализации:

```
struct RigidBodySimulator
{
```

```
    struct InitializationParameters
    {
        void * (* m_allocationFunction)(size_t size);
        void (* m_freeFunction)(void * memory);
        float m_gravity;
    };

    void initialize(const InitializationParameters & params);
    void shutDown();
};
```

Добавьте методы для добавления и удаления симулируемых твердых тел в системе, а также для получения и изменения их текущего состояния:

```
struct RigidBodySimulator
{
    struct ObjectDefinition
    {
        float m_mass;
        Matrix<3, 3> m_momentOfInertia;
        vector<Triangle> m_triangles;
    };

    struct ObjectState
    {
        Point m_position;
        Quaternion m_orientation;
        Vector m_velocity;
        Vector m_angularVelocity;
    };

    ObjectID createObject(
        const ObjectDefinition& objectDefinition,
        const ObjectState& objectState);
    void destroyObject(
          ObjectID objectID);
    ObjectState getObjectState(
          ObjectID objectID) const;
    void setObjectState(
         ObjectID objectID,
```

```
        const ObjectState& objectState);
};
```

Предполагаемый сценарий использования вроде бы вполне очевиден. Сначала симулятор инициализируется, затем используется, а после завершения работы корректно закрывается. Добавить объекты, изменить их состояние, удалить, когда они больше не нужны. Ничего сложного.

Однако полагаться на вызывающую сторону нельзя даже в таких простых вещах. Кто-то забудет вызвать **initialize**, кто-то запросит состояние уже удаленного объекта, а кто-то попытается задать состояние для случайного **ObjectID**, которого вы никогда не передавали.

Возникает соблазн просто проигнорировать подобные ситуации — предположить, что пользователи всё сделают правильно и «пусть будет, как будет». Но это огромная ошибка. Если вы не обнаружите и не сообщите об ошибке, всё закончится плачевно. Либо вызывающая сторона не заметит своей ошибки, либо решит, что программа ведет себя как и было задумано.

Предположим, **ObjectID** реализован как оболочка над небольшим целым числом, которое используется в качестве индекса в линейном списке структур **ObjectState**:

```
struct RigidBodySimulator
{
    struct ObjectID
    {
        int m_index;
    };

    ObjectState getObjectState(
        ObjectID objectID) const
    {
        return m_objectStates[objectID.m_index];
    }

    void setObjectState(
        ObjectID objectID,
        const ObjectState & objectState)
    {
        m_objectStates[objectID.m_index] = objectState;
    }
```

```
    vector<ObjectState> m_objectStates;
};
```

Эта конструкция проста и легка для понимания, но довольно ненадежна. Она оставляет незамеченными ошибки, которые может допустить вызывающая сторона. При попытке получить состояние объекта после его удаления результат будет неопределенным.

Строго говоря, такое утверждение немного сбивает с толку, хотя об интерфейсах подобного рода так и говорят. Под «неопределенным результатом» здесь имеется в виду то, что при вызове **getObjectState** для удаленного объекта интерфейс не гарантирует никакого конкретного результата. Но на практике результат вполне определен! В реализации (которую я здесь не показываю) при вызове **getObjectState** сразу после удаления объекта с помощью **destroyObject** будет получено то состояние, которое объект имел непосредственно перед удалением. Вызывающая сторона может осознанно или неосознанно прийти к выводу, что именно так и было задумано... и именно из таких предположений рождается путаница*.

Неопределенные результаты — признак плохо спроектированного интерфейса.

Не оставляйте такое некорректное использование без внимания. Любого, кто будет вызывать **getObjectState** после **destroyObject**, нужно уведомлять об этом, но сначала вам нужно обнаружить проблему. Одно из простых решений — дополнить индекс в **ObjectID** «номером поколения» (generation number):

```
struct RigidBodySimulator
{
    class ObjectID
    {
        friend struct RigidBodySimulator;

    public:

        ObjectID() :
            m_index(-1), m_generation(-1)
        { ; }
```

---

* И это же уязвимость многих систем автоматизированного тестирования, которые редко проверяют неопределенные случаи столь же тщательно, как и определенные. Ваши автоматические тесты вряд ли обнаружат проблему в приведенном примере с последовательностью **destroyObject + getObjectState**.

```
    protected:

        ObjectID(int index, int generation) :
            m_index(index), m_generation(generation)
            { ; }

        int m_index;
        int m_generation;
    };

    bool isObjectIDValid(const ObjectID objectID) const
    {
        return objectID.m_index >= 0 &&
            objectID.m_index < m_indexGenerations.size() &&
            m_indexGenerations[objectID.m_index] == objectID.m_
generation;
    }

    ObjectID createObject(
        const ObjectDefinition & objectDefinition,
        const ObjectState & objectState)
    {
        int index = findUnusedIndex();

        ++m_indexGenerations[index];
        m_objectDefinitions[index] = objectDefinition;
        m_objectStates[index] = objectState;

        return ObjectID(index, m_indexGenerations[index]);
    }

    void destroyObject(ObjectID objectID)
    {
        assert(isObjectIDValid(objectID));
        ++m_indexGenerations[objectID.m_index];
    }

    ObjectState getObjectState(ObjectID objectID) const
    {
```

```
        assert(isObjectIDValid(objectID));
        return m_objectStates[objectID.m_index];
    }

    void setObjectState(
        ObjectID objectID,
        const ObjectState & objectState)
    {
        assert(isObjectIDValid(objectID));
        m_objectStates[objectID.m_index] = objectState;
    }

    vector<int> m_indexGenerations;
    vector<ObjectDefinition> m_objectDefinitions;
    vector<ObjectState> m_objectStates;
};
```

Номер поколения помогает обнаруживать некорректное использование идентификаторов объектов. При создании или уничтожении объекта номер его поколения увеличивается. Если попытаться уничтожить объект, а затем получить его состояние, поколения не совпадут, и это несоответствие будет зафиксировано*. Вызывающая сторона получит уведомление о своей ошибке и сможет исправить ее до того, как та успеет породить путаницу.

Можно без труда добавить проверку и для других упомянутых ранее типичных ошибок использования — например, для тех случаев, когда забыли вызывать метод **initialize** или вызвали его дважды. В приведенном выше коде интерфейс был немного переработан так, чтобы создавать некорректные идентификаторы объектов (**ObjectID**) было труднее: единственный публичный конструктор создает корректный **ObjectID**, и потому вызывающая сторона может получить только правильно созданные и возвращенные идентификаторы.

Вполне допустимо поспорить о том, как именно следует сигнализировать об ошибках такого рода — использовать **assert**, возвращать коды ошибок или выбрасывать исключения. Ответ зависит от принятых в вашей коман-

---

* Один из стандартных способов сигнализации об ошибках в C служит макрос **assert**. Если переданное ему во время выполнения программы условие оказывается ложным (**false**), он выводит сообщение. Формат сообщения зависит от компилятора и операционной системы, но обычно включает в себя номер строки в исходном коде, на которой сработал, и само переданное макросу выражение.

де правил и соглашений. Главное здесь — это обнаруживать ошибку и явно сообщать о ней, а не способ, каким это делается.

«Здоровый код»

Код, который легче тестировать (или, что еще лучше, который постоянно тестирует себя сам), дольше остается «здоровым», то есть стабильным, надежным и работоспособным. Об этом стоит задуматься с самого начала, еще до того, как вы напишете первую строчку кода для нового проекта. Начать можно с составления автоматического теста, как в случае разработки через тестирование (TDD). Можно также выбрать реализацию функциональности без состояний либо добавить постоянный внутренний аудит работы кода.

В результате «заразные баги» будут обнаруживаться на ранних стадиях, еще до того, как они успеют размножиться. Так у вас будет меньше проблем, требующих исправления, а если что-то и понадобится исправить, то сделать это будет несложно.

И в таком подходе есть свое скрытое преимущество! Большинство методов, облегчающих тестирование кода, заодно и упрощают его написание. Когда вам приходится писать тесты для всех случаев использования, то вы волей-неволей задумаетесь о том, чтобы упростить эти случаи и сократить их количество. Избавившись от состояния, вы сделаете код менее запутанным. Снизив вероятность ошибок при использовании интерфейсов, вы сделаете их проще.

Это безусловный выигрыш. Так что вот мой вам совет — следите за простотой и постоянно тестируйте.

Правило 3

Хорошее имя — лучшая документация

По всей видимости, невозможно писать про программирование, не цитируя Шекспира. К нашему времени это уже стало банальностью. Тем не менее вот вам очень краткий пересказ пьесы «Ромео и Джульетта»: Ромео и Джульетта — влюбленные подростки, волею судьбы разлученные враждой своих семей, не позволяющих им встречаться друг с другом. Для всех вовлеченных в эту историю пьеса заканчивается печально.

В сцене II акта II Джульетта горестно жалуется на жизнь и произносит слова, ставшие пятой по известности цитатой из пьесы:

> Что значит имя? Роза пахнет розой,
> Хоть розой назови ее, хоть нет.

Схожие аргументы я слышал от коллег и по поводу кода — обычно от тех, кого во время обсуждения кода раздражало мое придирчивое отношение к названиям тех или иных сущностей. Имена переменных, функций, членов классов, исходных файлов, классов и структур — я готов придраться к любому из них.

Люди, обычно с закатыванием глаз, будут утверждать, что никакие названия не важны, а важна сама именуемая сущность. По их мнению, истинный смысл переменной (или функции, или класса) можно определить, только взглянув на код. Суть переменной в том, для чего она служит и как ею пользуются, а не в том, как она называется. От перемены названий функциональность не меняется.

А потому, заявляют они, достаточно выбрать то, что легко напечатать, и продолжать программировать как ни в чем не бывало.

Эти люди неправы.

Имя сущности — это первая и самая важная имеющаяся у вас документация. Она появляется неизбежно. Она всегда перед глазами. Каждый раз, делая ссылку на ту или иную сущность, вы упоминаете ее по имени. Это прекрасный способ объяснить пользователю кода, что представляет собой эта

сущность, и это объяснение действует каждый раз, когда пользователи видят ее название или имя.

Такую отличную возможность просто нельзя упускать.

Ваша задача при выборе имени для какой-то сущности в коде проста — имя должно отображать все важное для этой сущности и направлять мысли пользователя в нужное русло. Имя переменной должно мгновенно сообщать читателю о том, что эта переменная обозначает. Имя функции должно говорить о том, что делает эта функция.

Довольно просто, не так ли? Так почему же так часто возникают проблемы с именами? По многим причинам, которых примерно столько же, сколько звезд на небе, но ниже я опишу несколько самых распространенных из неудачных способов выбора имен.

Оптимизация — это не нажимать как можно меньше клавиш

Первый шаг к неудаче — излишне краткие имена. Не забывайте о том, что код гораздо чаще читают, а не пишут. Во время написания кода об этом легко забыть и выбрать имена, удобные для печати, вместо того чтобы приложить немного усилий и написать код, удобный для чтения.

В крайних случаях имена получаются сверхкраткими. Чем старше код, тем больше вероятность увидеть такой сверхкраткий стиль для имен переменных. А если вам попадется работа программиста старой закалки, который начал программировать еще в 60-х или 70-х годах, то вы почти наверняка увидите имена из двух-трех букв*.

Я называю такой код «стилем Numerical Recipes»**. Я большой поклонник Numerical Recipes, но описанный в этой работе стиль кодирования довольно неочевиден. Вот пример такого стиля:

```
void cp(
    int n,
```

---

* Я из того поколения, чьим первым языком программирования был Applesoft BASIC, который позволял использовать длинные имена переменных, но учитывал только первые два символа. Вам не показалось: **JUDGE$** и **JUROR$** считались псевдонимами одной и той же строковой переменной. Веселые были времена. Мои переменные в BASIC действительно состояли всего из одного или двух символов, как в следующем примере кода.

** Numerical Recipes — классический труд, в котором объясняются разного рода алгоритмы для математики и прочих точных наук. Кодовая база Sucker Punch пестрит идеями, заимствованными из этой книги. Однозначно рекомендую к прочтению. Последнее третье издание: William H. Press et al., Numerical Recipes: The Art of Scientific Computing, 3rd ed. (Cambridge University Press, 2007).

```
    float rr[],
    float ii[],
    float xr,
    float xi,
    float * yr,
    float * yi)
{
    float rn = 1.0f, in = 0.0f;
    *yr = 0.0f;
    *yi = 0.0f;
    for (int i = 0; i <= n; ++i)
    {
        *yr += rr[i] * rn - ii[i] * in;
        *yi += ii[i] * rn + rr[i] * in;
        float rn2 = rn * xr - in * xi;
        in = in * xr + rn * xi;
        rn = rn2;
    }
}
```

Так сразу и не поймешь, что тут происходит, правда? Конечно, при некотором размышлении можно догадаться, что это вычисление значения многочлена с комплексными числами, но для этого нужно приложить некоторые усилия. Гораздо легче понять следующий код:

```
void evaluateComplexPolynomial(
    int degree,
    float realCoeffs[],
    float imagCoeffs[],
    float realX,
    float imagX,
    float * realY,
    float * imagY)
{
    float realXN = 1.0f, imagXN = 0.0f;
    *realY = 0.0f;
    *imagY = 0.0f;

    for (int n = 0; n <= degree; ++n)
    {
```

```
        *realY += realCoeffs[n] * realXN - imagCoeffs[n] * imagXN;
        *imagY += imagCoeffs[n] * realXN + realCoeffs[n] * imagXN;
        float realTemp = realXN * realX - imagXN * imagX;
        imagXN = imagXN * realX + realXN * imagX;
        realXN = realTemp;
    }
}
```

И, разумеется, этот конкретный фрагмент кода был бы еще проще при использовании отдельного типа данных для комплексных чисел:

```
void evaluateComplexPolynomial(
    vector<complex<float>> & terms,
    complex<float> x,
    complex<float> * y)
{
    complex<float> xN = { 1.0f, 0.0f };
    *y = { 0.0f, 0.0f };

    for (const complex<float> & term : terms)
    {
        *y += xN * term;
        xN *= x;
    }
}
```

Теперь, если вы помните, как работают комплексные числа, структура алгоритма очевидна. Каждое слагаемое умножается на значение x, возведенное в N-ю степень, и результат накапливается в переменной y.

Не смешивайте разные соглашения о наименовании

Вторая причина, по которой имена могут оказаться неудачными, — это несогласованность. Если имена в коде выбираются по разным правилам, то пользователю кода будет легче запутаться.

В большинстве проектов трудно избежать некоторого уровня несогласованности. Если вы пользуетесь внешними библиотеками, то проблемы неизбежны — только если во всех ваших зависимостях вы не придерживаетесь одних и тех же правил именования и если вы сами не готовы подчиняться

принятым правилам. Допустим, вы пишете нативное приложение под Windows и соглашаетесь с принятыми в компании Microsoft рекомендациями по поводу наименований, в таком случае ваш код будет единообразным. Или же вы работаете со стандартной библиотекой шаблонов C++ и соблюдаете ее соглашения. В противном случае в коде появятся видимые несостыковки при столкновении разных стилей наименования.

Предположим, что у вас есть класс **vector**, заранее выделяющий память под фиксированное количество элементов. Это удобно, но стандартная библиотека C++ такого не предоставляет. Допустим, в вашем коллективе принят простой стиль именования методов camelCase («верблюжий стиль») с глаголом в начале. Если опустить некоторые детали, то ваш класс выглядит так:

```
template <class ELEM, int MAX_COUNT = 8>
class FixedVector
{
public:

    FixedVector() :
        m_count(0)
    { ; }

    void append(const ELEM & elem)
    {
        assert(!isFull());
        (void) new (&m_elements[m_count++]) ELEM(elem);
    }

    void empty()
    {
        while (m_count > 0)
        {
            m_elements[--m_count].~ELEM();
        }
        }
    int getCount() const
        { return m_count; }
    bool isEmpty() const
        { return m_count == 0; }
    bool isFull() const
        { return m_count >= MAX_COUNT; }
```

```
protected:

    int m_count;
    union
    {
        ELEM m_elements[0];
        char m_storage[sizeof(ELEM) * MAX_COUNT];
    };
};
```

Всё довольно понятно*. Метод **append** добавляет новый элемент, метод **empty** очищает весь массив, а также есть методы-аксессоры для проверки текущего количества элементов в векторе. Но если вы напишете код, в котором к этому классу **FixedVector** будет добавлен стандартный контейнер C++, картина получится не столь радужной:

```
void reverseToFixedVector(
    vector<int> * source,
    FixedVector<int, 8> * dest)
{
    dest->empty();
    while (!source->empty())
    {
        if (dest->isFull())
            break;

        dest->append(source->back());
        source->pop_back();
    }
}
```

Здесь мы видим две последовательные строки, вызывающие для вектора метод **empty** вектора, но эти два вызова делают совершенно разные вещи. Первый вызов очищает вектор-приемник; второй вызов проверяет, пуст ли вектор-источник. Легко представить, какая может возникнуть путаница!

---

* Если не учитывать, что массив нулевого размера — вещь неочевидная. Его поддержка зависит от компилятора; компилятор, которым я пользуюсь, такие массивы поддерживает, но делает это неохотно. Я мог бы написать код и без массива нулевого размера, но с ним код получается проще для чтения и понимания, так что пусть будет как вышло.

Разумеется, можно было бы применить соглашения Standard Template Library (STL) и для класса `FixedVector` — назвать его `fixed_vector`, а также подобрать соответствующие имена для всех его методов, — но так вы лишь перенесете путаницу в другое место своего проекта. К тому же вам придется не просто просить коллег-программистов учитывать чужие соглашения о стиле, но и писать код в соответствии с этими чужими соглашениями, что уже гораздо более серьезное требование.

В таких случаях возникает так называемая «когнитивная нагрузка», или умственное напряжение, о котором легко забыть. Переключение между стилями требует усилий, ведь приходится постоянно переключаться между разными наборами правил, переосмысливать прочитанное и вспоминать, какие правила действуют в том или ином контексте. В нашем примере это значит, что нужно будет то и дело смотреть в код, чтобы понять, к какому типу относится та или иная переменная и, следовательно, какие соглашения применяются в ее случае, если, конечно, вы вообще запомните, какие соглашения к какому типу относятся!

Мы в Sucker Punch избежали конкретно этой проблемы несогласованности между нашими соглашениями и соглашениями стандартных контейнеров C++ благодаря тому, что написали собственные версии всех контейнерных классов, вместо того чтобы использовать версии STL. Решение довольно радикальное, но оно действительно снижает когнитивную нагрузку: контейнеры ведут себя так же, как и весь остальной наш код, поэтому при пошаговой отладке переход к методу контейнера не превращается в вылазку на чужую территорию. На чужую территорию, вроде джунглей STL, полных макросов и безумной шаблонной магии — я этого не осуждаю, но всё же.

И даже так мы не смогли полностью избавиться от чужих соглашений, ведь нам всё равно приходится пользоваться чужим, написанным не нами кодом — например, библиотеками платформы PlayStation. В большинстве проектов смешение соглашений в какой-то степени неизбежно. Главное — свести это смешение к минимуму. По возможности изолируйте чужой код, чтобы его стиль не «просачивался» в те части проекта, с которыми работает вся команда.

Не «стреляйте себе в ногу»

Не вредите сами себе, если программисты в вашей команде не придерживаются единых правил именования, вы только создаете себе лишние проблемы. Мешанина разных стилей делает трудным для восприятия даже простой код:

```
int split(int min, int max, int index, int count)
{
    return min + (max - min) * index / count;
}

void split(int x0, int x1, int y0, int y1, int & r0, int & r1)
{
    r0 = split(x0, x1, y0, y1);
    r1 = split(x0, x1, y0 + 1, y1);
}

void layoutWindows(vector<HWND> ww, LPRECT rc)
{
    int w = ww.size();
    int rowCount = int(sqrtf(float(w - 1))) + 1;
    int extra = rowCount * rowCount - w;
    int r = 0, c = 0;
    HWND hWndPrev = HWND_TOP;
    for (HWND theWindow : ww)
    {
        int cols = (r < extra) ? rowCount - 1 : rowCount;
        int x0, x1, y0, y1;
        split(rc->left, rc->right, c, cols, x0, x1);
        split(rc->top, rc->bottom, r, rowCount, y0, y1);
        SetWindowPos(
            theWindow,
            hWndPrev,
            x0,
            y0,
            x1 - x0,
            y1 - y0,
            SWP_NOZORDER);
        hWndPrev = theWindow;
        if (++c >= cols)
        {
            c = 0;
            ++r;
        }
    }
}
```

У меня даже голова закружилась, настолько тяжело было набирать этот код. Но я пожертвовал собой ради общего блага.

Алгоритм сам по себе несложный; я просто размещаю окна внутри заданного прямоугольника, разбивая область на строки и столбцы и стараясь сохранить примерно то же соотношение сторон, что и у исходного прямоугольника. Но выбранные имена делают его ненужной головоломкой.

Самая очевидная проблема — это мешанина из трех или четырех разных стилей именования, что уже достаточно плохо. Но есть и другая, более деликатная проблема, которую можно встретить даже в коде с едиными правилами именования: смена имен параметров при передаче их в функции. Например, при первом вызове **split** я передаю **x0** и **x1** как два последних аргумента. Это значения правой и левой сторон границ нового прямоугольника окна. Но внутри функции **split** переменные **x0** и **x1** означают уже нечто совершенно иное.

И это проблема. Если вы будете пошагово отслеживать выполнение **layoutWindows**, то у вас в голове сложится какое-то представление о том, что такое **x0** и **x1**. Зайдя внутрь **split**, вы увидите те же **x0** и **x1**, но теперь они означают что-то совсем другое. В этой функции **x** и **y** используются для абстрактных имен переменных, как это принято в алгебре; они никак не связаны с системой координат, как это было в **layoutWindows**. Алгебра в одной функции, декартовы координаты в другой — всё это дополнительная когнитивная нагрузка, затрудняющая восприятие и порождающая ошибки.

Часть подобных переименований при вызове функций неизбежна: аргументы часто являются результатом операций, а не просто какими-то отдельными переменными. В нашем примере первые два аргумента **split** это **rc->left** и **rc->right**, и их имена нельзя просто так оставить внутри функции **split**. Придется заводить локальные переменные, но если вы достаточно сообразительны, то назовете их **left** и **right**, и тогда при пошаговой отладке будет проще отслеживать, что к чему относится, когда вы войдете внутрь функции.

Вот та же функция, немного переработанная для большей согласованности и читаемости:

```
int divideRange(int min, int max, int index, int count)
{
    return min + (max - min) * index / count;
}

void layoutWindows(vector<HWND> windows, LPRECT rect)
```

```
{
    int windowCount = windows.size();
    int rowCount = int(sqrtf(float(windowCount - 1))) + 1;
    int shortRowCount = rowCount * rowCount - windowCount;

    HWND lastWindow = HWND_TOP;
    int rowIndex = 0, colIndex = 0;

    for (HWND window : windows)
    {
        int colCount = (rowIndex < shortRowCount) ?
                            rowCount - 1 :
                            rowCount;
        int left = divideRange(
                        rect->left,
                        rect->right,
                        colIndex,
                        colCount);
        int right = divideRange(
                        rect->left,
                        rect->right,
                        colIndex + 1,
                        colCount);
        int top = divideRange(
                        rect->top,
                        rect->bottom,
                        rowIndex,
                        rowCount);
        int bottom = divideRange(
                        rect->top,
                        rect->bottom,
                        rowIndex + 1,
                        rowCount);

        SetWindowPos(
            window,
            lastWindow,
            left,
            top,
            right - left,
```

```
            bottom - top,
            SWP_NOACTIVATE);

        lastWindow = window;
        if (++colIndex >= colCount)
        {
            colIndex = 0;
            ++rowIndex;
        }
    }
}
```

Алгоритм тот же, но гораздо более понятный, и единые правила именования помогают его отслеживать. Теперь проще определить, для чего служит та или иная переменная, по одному лишь имени, без тщательного изучения кода — например, можно полностью не знать, что такое **rowIndex**, но можно с уверенностью утверждать, что это индекс строки («row index»). Конечно, неясно, какой именно строки и в каком контексте, но и это уже немало для начала.

Последовательное именование индексов и счетчиков дает дополнительные положительные эффекты. В функции **divideRange** в качестве имен аргументов используются **index** и **count**. С этими аргументами легко сопоставить переменные **colIndex** и **colCount** из **layoutWindows**. Я свел когнитивную нагрузку к минимуму, особенно по сравнению с путаницей **x0/x1** в первой версии этой функции.

Это обычное дело. Если есть последовательный свод правил для именования сущностей, то при переходе между разными функциями или разными частями кодовой базы похожие сущности будут и называться похоже. Идентичные же вещи обычно будут иметь идентичные имена. При пошаговом проходе кода или при попытках понять, как его разные части взаимодействуют друг с другом, не нужно будет жонглировать кучей имен для одной и той же сущности. Существует либо только одно имя, либо небольшое количество очевидным образом тесно связанных между собой имен, как в предыдущем примере с **index + count**.

Не заставляйте меня думать

В правилах, нужных для достижения единообразия, можно пойти и дальше.

Ключ к единообразию — максимальная механистичность всего. Если соглашения вашей команды о том, как называть сущности, подразумевают

какую-то степень субъективности или требуют тщательного обдумывания, то они не сработают. Разные программисты будут принимать разные решения, и имена у всех будут различаться.

Гораздо лучше ситуация, когда все выбирают одни и те же имена для одних и тех же вещей, потому что следуют четко прописанным правилам, не допускающим сомнений или слишком расплывчатого толкования. Это значительно облегчает работу с кодом для каждого программиста. И самый простой способ достичь такого уровня единообразия — ввести механистичные правила, которым будут следовать все.

Правила именования переменных в Sucker Punch подчеркнуто механистичны. В примерах этой книги я их не придерживаюсь, в первую очередь, чтобы читателям было легче воспринимать код. Наши правила хорошо работают для нас, но только потому, что мы все применяем их постоянно, а для посторонних они могут показаться немного странными.

Вместо них для примеров в этой книге я использовал более простые и понятные условные правила, так, например, если класс представляет какого-то персонажа, он называется **`Character`** («персонаж»); переменная, хранящая персонажа, обычно называется **`character`**, а заполненный персонажами вектор будет называться **`characters`***. Они упрощены ради читаемости кода, но используются последовательно.

Кодовая база Sucker Punch организована примерно так же, только с более подробным сводом правил и чуть большей лаконичностью. Для именования переменных мы используем вариант венгерской нотации Microsoft. Это, конечно, спорный вопрос. Не столько для программистов в Sucker Punch, которые довольно быстро приспосабливаются, сколько за пределами экосистемы Microsoft, где венгерская нотация часто становится объектом насмешек**.

Основная идея венгерской нотации заключается в том, что всё имя или часть имени определяется типом (а иногда и назначением) переменной. Так, переменная для индекса массива фракций будет называться **`iFaction`**. При

---

* Естественно, этот класс был бы реализован в файлах с именами *character.h* и *character.cpp*.

** Думаю, негативное отношение к венгерской нотации часто искажает ее реальные достоинства. Изначально она была обходным решением, компенсирующим отсутствие типобезопасного связывания в ранних компиляторах C. Встраивание информации о типах в имена переменных и функций добавляло определенную степень защиты от ошибок, пусть и чисто на уровне соглашений. Сейчас этот исторический аспект уже не имеет значения, но он чаще всего становится предметом насмешек. Кроме того, критики утверждают, что код с венгерской нотацией трудно читать. Но это все равно что сказать, будто исландский язык трудно читать — конечно трудно, если вы его не знаете! Реальная же ценность венгерской нотации для нас сегодня заключается в том, что ее правила позволяют естественным образом создавать одинаковые имена для одних и тех же сущностей. А это, в свою очередь, ведет к созданию более согласованной и удобной для работы кодовой базы.

наличии вектора указателей (vector of pointers) на объекты типа **Character** переменная будет называться **vpCharacter**.

Во многих случаях на этом всё и заканчивается: имя переменной формируется полностью механически, и, как следствие, для одной и той же сущности все используют одно и то же имя. Именно для этого и нужны такие соглашения.

Если же несколько переменных имеют один и тот же тип, к их имени добавляется уточняющий суффикс. Например, два указателя (pointers) на персонажей **Character** могут называться **pCharacter** и **pCharacterOther**. Конечно, такая ситуация заставляет программистов принимать субъективные решения, но принятые дополнительные соглашения об уточнениях сводят возможную непоследовательность к минимуму.

Самое важное — не какие-то конкретные правила именования, а их строгое определение, максимальная механистичность, отражение в документации и соблюдение на практике. Так создается комфортная среда разработки, в которой все для одних и тех же понятий выбирают одни и те же имена, а работа с чужим кодом воспринимается как работа со своим собственным.

Подумайте, какие из ваших собственных соглашений можно сделать более механическими, и сделайте их такими. Польза от такого шага будет ощущаться долгие годы.

Правило 4

Обобщайте после трех примеров

Всех новых программистов учат, что обобщенные решения предпочтительнее частных. Лучше написать одну функцию, которая решает две задачи, чем две отдельные функции для каждой из них.

Мало кто напишет такой код (поиска красного знака):

```
Sign* findRedSign(const vector<Sign*>& signs)
{
    for (Sign* sign : signs)
        if (sign->color() == Color::Red)
            return sign;
    return nullptr;
}
```

если можно легко написать такой (поиска знака любого цвета):

```
Sign* findSignByColor(const vector<Sign*>& signs, Color color)
{
    for (Sign* sign : signs)
        if (sign->color() == color)
            return sign;
    return nullptr;
}
```

Думать в терминах обобщения естественно, особенно в таких простых случаях. Если нужно найти все красные знаки в мире, естественным побуждением программиста будет написать код, который ищет знаки любого цвета, а затем передать в него «красный» как параметр. Природа не терпит пустоты, а программисты не терпят кода, который решает только одну задачу.

Стоит задуматься, почему такое поведение кажется таким естественным. На каком-то уровне побуждение написать **`findSignByColor`** вместо

findRedSign зависит от нашей способности строить предположения. Если вы ищете красный знак, можно с уверенностью предположить, что в какой-то момент вам понадобится искать и синий знак, так что вы решаете заранее написать код, который будет обрабатывать и этот случай.

Но зачем останавливаться на цвете? Почему бы не написать еще более обобщенное решение для поиска любых знаков?

Можно создать более универсальный интерфейс, который позволял бы запрашивать любой аспект знака, такой как цвет, размер, местоположение, текст надписи, и тогда поиск по цвету станет лишь частным случаем общего механизма. А общий механизм можно реализовать, например, с помощью структуры, задающей допустимые значения для каждого из параметров знака:

```
bool matchColors(
    const vector<Color>& colors,
    Color colorMatch)
{
    if (colors.empty())
        return true;
    for (Color color : colors)
        if (color == colorMatch)
            return true;
    return false;
}

bool matchLocation(
    Location location,
    float distanceMax,
    Location locationMatch)
{
    float distance = getDistance(location, locationMatch);
    return distance < distanceMax;
}

struct SignQuery
{
    SignQuery() :
        m_colors(),
        m_location(),
        m_distance(FLT_MAX),
```

```
        m_textExpression(".*")
    {
        ;
    }

    bool matchSign(const Sign* sign) const
    {
        return matchColors(m_colors, sign->color()) &&
               matchLocation(m_location, m_distance, sign->location())
&&
               regex_match(sign->text(), m_textExpression);
    }

    vector<Color> m_colors;
    Location m_location;
    float m_distance;
    regex m_textExpression;
};
```

Параметры запроса заставляют принять до некоторой степени субъективное решение, потому что каждый аспект подразумевает свою модель поиска. В этом примере я сделал следующие допущения:

- Вместо указания одного цвета можно задавать список допустимых цветов. Пустой список означает, что подойдет любой цвет.
- Поскольку **`Location`** («месторасположение») хранит широту и долготу в виде вещественных чисел (чисел с плавающей точкой), поиск точного совпадения не имеет смысла. Вместо этого задается максимальное расстояние от заданной точки.
- Для поиска текста (или его части) можно использовать регулярное выражение, что позволит охватить множество очевидных случаев.

Сам код для поиска подходящего знака довольно прост:

```
Sign * findSign(const SignQuery & query, const vector<Sign *> & signs)
{
    for (Sign * sign : signs)
        if (query.matchSign(sign))
            return sign;
    return nullptr;
}
```

Поиск именно красного знака в этой модели тоже несложен — нужно создать **SignQuery**, указать красный как единственно допустимый цвет и вызвать **findSign**:

```
Sign * findRedSign(const vector<Sign *> & signs)
{
    SignQuery query;
    query.m_colors = { Color::Red };
    return findSign(query, signs);
}
```

Но не будем забывать, что вся конструкция **SignQuery** была создана на основе одного-единственного примера — поиска красного знака. Все остальное было лишь нашими догадками. На данный момент нет никаких других реальных примеров, на которые можно было бы опереться, так что мы просто исходили из гипотетического предположения, что в будущем нам понадобятся другие виды поиска.

И в этом заключается проблема — ваши предсказания, скорее всего, окажутся неверными. Если повезет — лишь немного неверными, но, скорее всего, не повезет.

YAGNI

Скорее всего, вы, исходя из каких-то своих субъективных догадок, попытались решить задачи, которые на практике вообще не встретятся. Предположим, что несколько первых реальных задач по поиску знаков выглядят так:

- Найти красный знак.
- Найти знак рядом с пересечением Мэйн-стрит и Барр-стрит.
- Найти красный знак возле дома 212 на Саут-Уотер-стрит.
- Найти зеленый знак.
- Найти красный знак возле дома 902 на Милл-стрит.

Да, все эти случаи можно обработать с помощью **SignQuery** и **findSign**, так что в этом смысле код неплохо предсказал будущие задачи. Но ни в одном из примеров не нужно искать знаки нескольких цветов или знаки с текстом. Все реальные случаи сводятся к поиску одного цвета, и иногда места. **SignQuery** решает задачи, которых в действительности не существует.

Это типичная ситуация, и она настолько распространена, что в философии экстремального программирования для нее придумали отдельное

название YAGNI — сокращение фразы «You Ain't Gonna Need It», то есть «Вам это не понадобится». Вместо того чтобы искать знаки известного цвета, вы проделали работу для обработки списка всех допустимых цветов, тем самым потратив впустую время и усилия. Эксперименты с классом регулярных выражений C++, попытки разобраться, как отличить полное совпадение от частичного, — это то же время, которое уже не вернуть.

Более того, дополнительная сложность **`SignQuery`** подразумевает, что этим кодом будет сложно пользоваться. Как работает **`findSignByColor`**, интуитивно ясно, а вот **`findSign`** требует дополнительного изучения, ведь он объединяет в себе три разных модели запросов!

В частности, возникают такие вопросы: «Достаточно ли частичного совпадения по регулярному выражению, или нужно, чтобы выражение совпало со всем текстом знака?» Неочевидно и то, как три условия взаимодействуют между собой, — это логическое «и» или «или»? Прочитав код, можно понять, что знак удовлетворяет запросу только при совпадении всех условий, но это становится ясно лишь после изучения реализации. Отсюда возникает новый вопрос: какие поля **`SignQuery`** обязательны? В текущей реализации создаваемый конструктором пустой запрос соответствует любому знаку, поэтому нужно заполнять только те поля, по которым производится фильтрация, но чтобы это понять, приходится разбираться.

С учетом явно намечающегося шаблона в реальных сценариях было бы проще решить текущую задачу:

```
Sign * findSignWithColorNearLocation(
    const vector<Sign *> & signs,
    Color color = Color::Invalid,
    Location location = Location::Invalid,
    float distance = 0.0f)
{
    for (Sign * sign : signs)
    {
        if (isColorValid(color) &&
            sign->color() != color)
        {
            continue;
        }
        if (isLocationValid(location) &&
            getDistance(sign->location(), location) > distance)
        {
            continue;
```

```
        }
        return sign;
    }
    return nullptr;
}
```

Но сейчас вы можете обвинить меня в жульничестве. Конечно, когда у нас есть несколько реальных примеров задач, кажется, что **`findSignWithColorNearLocation`** — более удачное решение, чем **`SignQuery`**. Но ведь после первого примера вы не могли бы прийти к такому предположению. Универсальное решение в виде **`findSignWithColorNearLocation`** после первого случая, казалось бы, не хуже и не лучше удачной реализации **`SignQuery`**. Одна из последующих задач вполне могла бы и в самом деле предполагать поиск по нескольким цветам или по тексту знака.

Но именно в этом и вся суть! После одного примера невозможно было предсказать универсальное решение, а значит, ошибкой была сама попытка его написать. И **`findSignWithColorNearLocation`**, и **`SignQuery`** — всё это неудачное обобщение. Никакого выигрыша, одни лишь проигрыши.

Вот способ найти красный знак получше:

```
Sign * findRedSign(const vector<Sign *> & signs)
{
    for (Sign * sign : signs)
        if (sign->color() == Color::Red)
            return sign;
    return nullptr;
}
```

Да, я серьезно. Да, можно было бы передать нужный цвет в качестве аргумента, но на этом я бы остановился. Если у вас есть только один сценарий использования, пишите код, решающий именно этот сценарий. Не пытайтесь угадывать, каким будет следующий сценарий. Пишите код для тех задач, которые вы понимаете, а не для тех, относительно которых лишь строите предположения.

Очевидное возражение против такого подхода, на которое я отвечаю с удвоенной уверенностью

«Постойте-ка, — скажете вы. — Ведь так мы создаем код, который лишь едва-едва удовлетворяет требованиям текущего сценария. Рано или поздно

возникнут случаи, с которыми он не справится? Что делать, когда нужно будет решить задачу, не подходящую под уже написанное решение? А это будет наверняка.

И разве это не аргумент в пользу создания более общего кода? Да, первые пять реальных задач, в которых использовался **`SignQuery`**, не задействовали всю его функциональность, но что если шестая как раз потребует этого? Разве не будет хорошо, что код уже готов?»

Нет, вовсе нет. Не тратьте силы заранее. Когда появится сценарий, которому ваш код не соответствует, напишите для этого случая новый код. Можно просто скопировать и изменить существующий вариант, адаптировав его под новую задачу. Можно написать всё заново. И то и другое — абсолютно нормально.

Первым реальным сценарием в списке из пяти было «Найти красный знак», и я написал код, который искал именно красный знак и ничего больше. Второй сценарий был «Найти знак рядом с пересечением Мэйн-стрит и Барр-стрит», поэтому теперь я напишу код, который будет делать именно это и ничего больше:

```
Sign * findSignNearLocation(
    const vector<Sign *> & signs,
    Location location,
    float distance)
{
    for (Sign * sign : signs)
    {
        if (getDistance(sign->location(), location) <= distance)
        {
            return sign;
        }
    }
    return nullptr;
}
```

Третий сценарий был «Найти красный знак возле дома 212 на Саут-Уотер-стрит», и он не обрабатывался ни одной из двух написанных мной функций. Это переломный момент, теперь, когда у нас есть три независимых сценария использования, начинает иметь смысл искать обобщение. Имея три реальных примера, мы можем более уверенно предсказать четвертый и пятый.

Но почему три? Что это за волшебное число? На самом деле в нем нет ничего особенного, кроме того, что это не один и не два. Одного примера

недостаточно, чтобы угадать общую закономерность. По моему опыту, двух обычно тоже недостаточно; после двух примеров вы просто утвердитесь в своем неточном обобщающем предположении. А вот три примера уже дают какое-то представление о закономерности, да и ваше стремление строить самые широкие обобщения, скорее всего, немного угаснет. Ничто так не смиряет, как осознание своей неправоты после первого и второго случаев!

Тем не менее никакого строгого требования обобщать код на этом этапе нет! Вполне допустимо написать третью отдельную функцию, не объединяя в ней первые две:

```
Sign * findSignWithColorNearLocation(
    const vector<Sign *> & signs,
    Color color,
    Location location,
    float distance)
{
    for (Sign * sign : signs)
    {
        if (sign->color() == color &&
            getDistance(sign->location(), location) >= distance)
        {
            return sign;
        }
    }
    return nullptr;
}
```

У варианта с тремя отдельными функциями есть важное преимущество — функции эти очень просты. Совершенно очевидно, какую из них вызывать. Если задан цвет и местоположение, вызывайте **`findSignWithColorNearLocation`**. Если только цвет — **`findSignWithColor`**; если только местоположение — **`findSignNearLocation`***.

Если задачи и дальше требуют проверки одного цвета, или местоположения, или того и другого, то этих функций будет достаточно навсегда. Конечно, такой подход плохо масштабируется. Для двух отдельных аргументов и трех функций поиска это еще не катастрофа, но с увеличением числа возможных аргументов он быстро становится абсурдным. Если в какой-то момент

---

* Либо, если вы пользуетесь языком вроде C++, поддерживающим перегрузку функций, можно назвать все три версии **`findSign`**, и пусть компилятор разбирается, какую из них вызывать.

появится задача, требующая проверки текста знака, то вы вряд ли решитесь создавать семь вариаций функции **findSign**.

На этом этапе нет ничего плохого в том, чтобы объединить три функции **findSign** в одну, обрабатывающую все три случая. Строить предположения и обобщать на примере трех реальных сценариев становится безопаснее. Но обобщайте только в том случае, если считаете, что обобщенный код будет проще писать и читать, основываясь исключительно на имеющихся у вас сценариях. Никогда не обобщайте «про запас», исходя лишь из своих догадок о следующих сценариях. Обобщайте только на основе известных вам реальных задач.

Писать общий код на C++ именно для этой задачи немного неудобно, поскольку в C++ нет по-настоящему опциональных аргументов, а только значения по умолчанию. Значит, нам нужно придумать способ помечать аргументы как «не заданные». Одно из решений — добавить **Invalid** («невалидные») значения для цвета и местоположения, которые будут использоваться, когда эти параметры не важны. Повторяю первую версию **findSignWithColorNearLocation**:

```
Sign * findSignWithColorNearLocation(
    const vector<Sign *> & signs,
    Color color = Color::Invalid,
    Location location = Location::Invalid,
    float distance = 0.0f)
{
    for (Sign * sign : signs)
    {
        if (isColorValid(color) &&
            sign->color() != color)
        {
            continue;
        }
        if (isLocationValid(location) &&
            getDistance(sign->location(), location) > distance)
        {
            continue;
        }
        return sign;
    }
    return nullptr;
}
```

Теперь все вызовы **findSignWithColor** и **findSignNearLocation** можно заменить на вызовы **findSignWithColorNearLocation**.

Когда реальность даже хуже YAGNI

До сих пор мы говорили о том, что преждевременное обобщение чаще всего приводит к тому, что вы пишете код, который никогда не будет использоваться, и это плохо. Менее очевидная проблема заключается в том, что преждевременное обобщение затрудняет адаптацию к непредвиденным сценариям использования. Отчасти это объясняется тем, что написанный вами обобщенный код довольно сложен и потому требует больше усилий для изменения, но есть и некие более деликатные обстоятельства. Как только вы напишете некий шаблон для обобщения, вы, скорее всего, для последующих случаев будете просто расширять этот шаблон, вместо того чтобы пересматривать его.

Вернемся немного назад. Представьте, что вы поторопились с обобщением с помощью класса **SignQuery**, но на этот раз первые несколько сценариев использования выглядят так:

- Найти красный знак.
- Найти красный знак «STOP» рядом с углом Мэйн-стрит и Барр-стрит.
- Найти все красные или зеленые знаки на Мэйн-стрит.
- Найти все белые знаки с текстом «MPH» на Уобаш-авеню или Уотер-стрит.
- Найти знак с текстом «Lane» или синего цвета рядом с Милл-стрит, 902.

Первые два сценария в этом списке довольно хорошо подходят для **SignQuery**, но затем все начинает разваливаться.

Третий сценарий, «Найти все красные или зеленые знаки на Мэйн-стрит», добавил два новых требования. Во-первых, код должен возвращать все подходящие знаки, а не один знак. Это несложно:

```
vector<Sign *> findSigns(
    const SignQuery & query,
    const vector<Sign *> & signs)
{
    vector<Sign *> matchedSigns;
    for (Sign * sign : signs)
```

```
    {
        if (query.matchSign(sign))
            matchedSigns.push_back(sign);
    }
    return matchedSigns;
}
```

Второе новое требование — найти все знаки вдоль улицы, и это уже сложнее. Предположим, что улицы можно представить в виде серии отрезков, соединяющих местоположения, и местоположения с улицами можно упаковать в новую структуру **Area** («область»):

```
struct Area
{
    enum class Kind
    {
        Invalid,
        Point,
        Street,
    };

    Kind m_kind;
    vector<Location> m_locations;
    float m_maxDistance;
};

static bool matchArea(const Area & area, Location matchLocation)
{
    switch (area.m_kind)
    {
        case Area::Kind::Invalid:
            return true;

        case Area::Kind::Point:
        {
            float distance = getDistance(
                                area.m_locations[0],
                                matchLocation);
            return distance <= area.m_maxDistance;
        }
```

```
        break;

        case Area::Kind::Street:
        {
            for (int index = 0;
                index < area.m_locations.size() - 1;
                ++index)
            {
                Location location = getClosestLocationOnSegment(
                                    area.m_locations[index + 0],
                                    area.m_locations[index + 1],
                                    matchLocation);
                float distance = getDistance(location, matchLocation);
                if (distance <= area.m_maxDistance)
                    return true;
            }
            return false;
        }
        break;
    }
    return false;
}
```

Эта новая структура **Area** может заменить местоположение и максимальное расстояние в **SignQuery**:

```
struct SignQuery
{
    SignQuery() :
        m_colors(),
        m_area(),
        m_textExpression(".*")
    {
        ;
    }

    bool matchSign(const Sign * sign) const
    {
        return matchColors(m_colors, sign->color()) &&
               matchArea(m_area, sign->location()) &&
```

```
                regex_match(sign->m_text, m_textExpression);
    }

    vector<Color> m_colors;
    Location m_location;
    float m_distance;
    regex m_textExpression;
};
```

Четвертый сценарий использования требует найти все знаки ограничения скорости на одной из двух улиц, что не укладывается в текущую структуру. Добавить поддержку списка областей достаточно легко:

```
bool matchAreas(const vector<Area> & areas, Location matchLocation)
{
    if (areas.empty())
        return true;

    for (const Area & area : areas)
        if (matchArea(area, matchLocation))
            return true;

    return false;
}
```

Затем можно заменить единственную область в **SignQuery** на список:

```
struct SignQuery
{
    SignQuery() :
        m_colors(),
        m_areas(),
        m_textExpression(".*")
    {
        ;
    }

    bool matchSign(const Sign * sign) const
    {
        return matchColors(m_colors, sign->color()) &&
```

```
            matchAreas(m_areas, sign->location()) &&
            regex_match(sign->m_text, m_textExpression);
    }

    vector<Color> m_colors;
    vector<Area> m_areas;
    regex m_textExpression;
};
```

Пятый сценарий использования еще больше всё усложняет. Теперь нужно искать знак, обозначающий историческую достопримечательность. Такие знаки обычно синие, поэтому нужно искать по цвету, но можно найти и зеленый с определенным текстом. Модель **SignQuery** этого не предусматривает.

Опять же, можно постараться решить и эту проблему, добавив в **SignQuery** логические операции:

```
struct SignQuery
{
    SignQuery() :
        m_colors(),
        m_areas(),
        m_textExpression(".*"),
        m_boolean(Boolean::None),
        m_queries()
    {
        ;
    }

    ~SignQuery()
    {
        for (SignQuery * query : m_queries)
            delete query;
    }

    enum class Boolean
    {
        None,
        And,
        Or,
        Not
```

```
};

static bool matchBoolean(
    Boolean boolean,
    const vector<SignQuery *> & queries,
    const Sign * sign)
{
    switch (boolean)
    {
        case Boolean::Not:
            return !queries[0]->matchSign(sign);

        case Boolean::Or:
        {
            for (const SignQuery * query : queries)
                if (query->matchSign(sign))
                    return true;
            return false;
        }
        break;

        case Boolean::And:
        {
            for (const SignQuery * query : queries)
                if (!query->matchSign(sign))
                    return false;
            return true;
        }
        break;
    }
    return true;
}

bool matchSign(const Sign * sign) const
{
    return matchColors(m_colors, sign->color()) &&
           matchAreas(m_areas, sign->location()) &&
           regex_match(sign->m_text, m_textExpression) &&
           matchBoolean(m_boolean, m_queries, sign);
}
```

```
    vector<Color> m_colors;
    vector<Area> m_areas;
    regex m_textExpression;
    Boolean m_boolean;
    vector<SignQuery *> m_queries;
};
```

Уф-ф-ф! Это были задачки посложнее тех, что мы разбирали в начале главы. После множества изменений модель **QuerySign** уже способна обрабатывать широкий диапазон запросов. Конечно, теоретически остаются пока неподдерживаемые запросы, вроде «найти два знака, находящиеся на расстоянии 10 метров друг от друга», но можно считать, что мы охватили все действительно важные случаи. Так что же, победа?

Не так выглядит успех

Сейчас даже непонятно, принесло ли нам пользу расширение **SignQuery** до такой степени или нет, хотя я действовал максимально аккуратно: никаких принципов YAGNI не было нарушено, и код оставался аккуратным и чистым.

Продолжая расширять универсальное решение, легко потерять из виду контекст. Именно это и произошло в нашем случае.

Сравним решение последнего сценария с использованием **SignQuery** и прямой подход. Вот решение через **SignQuery**:

```
SignQuery * blueQuery = new SignQuery;
blueQuery->m_colors = { Color::Blue };

SignQuery * locationQuery = new SignQuery;
locationQuery->m_areas = { mainStreet };

SignQuery query;
query.m_boolean = SignQuery::Boolean::Or;
query.m_queries = { blueQuery, locationQuery };
vector<Sign *> locationSigns = findSigns(query, signs);
```

А вот «прямой» вариант:

```
vector<Sign *> locationSigns;
```

```
for (Sign * sign : signs)
{
    if (sign->color() == Color::Blue ||
        matchArea(mainStreet, sign->location()))
    {
        locationSigns.push_back(sign);
    }
}
```

Прямое решение лучше. Оно проще, понятнее, легче отлаживается и расширяется. Вся работа над **SignQuery** лишь увела нас дальше от наилучшего и простого ответа. И в этом заключается настоящая опасность преждевременного обобщения. Оно опасно не только тем, что вы реализуете функции, которые никогда не будут использоваться, но и тем, что ваша абстракция задает определенное направление расширений, изменить которое потом будет трудно.

Обобщенные решения мешают рассматривать альтернативы. Как только вы создали абстракцию для решения задачи, трудно даже подумать о чем-то другом. После того как вы воспользовались **findSigns** для поиска всех красных знаков, ваш инстинкт подсказывает вам пользоваться **findSigns** для любых других знаков. Ведь на это намекает даже само имя функции!

Когда возникает очередная задача, не подходящая под текущую абстракцию, очевидным решением кажется расширить **SignQuery** и **findSigns** для нового сценария. Так же будет и для последующих задач. Но по мере расширения и обобщения решение становится все более громоздким, и... если не соблюдать осторожность, можно, даже не замечая того, выйти за пределы разумного обобщения.

Когда у вас в руках молоток, всё вокруг кажется гвоздями, правда? Так вот, универсальное решение — это раздача молотков. Не спешите их раздавать, пока не убедитесь, что у вас действительно мешок гвоздей, а не мешок шурупов*.

---

* Кстати, молотком можно вбить и шуруп — просто придется сильнее стучать. Побоюсь сказать банальность, но с кодом происходит то же самое: можно заставить работать неудобную абстракцию — просто придется «стучать» по этой абстракции сильнее.

Правило 5

Первый урок оптимизации — не оптимизировать

Мое любимое занятие в программировании — оптимизация. Обычно под этим подразумевается ускорение работы какой-то системы, хотя иногда приходится оптимизировать использование памяти, сетевой трафик или какой-либо другой ресурс.

Мне оптимизация нравится тем, что успех ее легко измерить. Для большинства задач программирования критерии успеха размыты. В книгах, подобных этой, пытаются найти подходящее определение для «хорошего кода» или «хорошей системы», но понять, что же именно делает строки кода «хорошими», всегда трудно.

С оптимизацией же все иначе. Здесь ответы более четкие. Ускоряя работу чего-то, можно измерить успех напрямую. То же касается цены этого успеха относительно увеличения размера или сложности кода. Никаких расплывчатых долгосрочных выгод, никакой надежды на то, что кто-то, читая ваш код через несколько лет, сразу его оценит и восхитится вашей программистской гениальностью. Только прямые, осязаемые результаты.

Я не одинок в своей любви к оптимизации. Она настолько соблазнительна, что породила известную среди программистов поговорку:

Преждевременная оптимизация — корень всех зол.

Кстати, это не полная цитата. Об этом еще в 1974 году говорил Дональд Кнут, и его высказывание более конкретизированное:

О малых оптимизациях нужно забывать, скажем, примерно в 97% случаев: преждевременная оптимизация — корень всех зол.

Важно отметить контекст*. В 1974 году компиляторы были куда менее совершенными, чем сейчас. Те «малые оптимизации», о которых говорил Кнут, часто представляли собой хитрые фрагменты кода, заставляющие компилятор сгенерировать желаемую машинную последовательность — например, кешировать конечные указатели, чтобы выжать немного больше производительности:

```
int stripNegativeValues(int count, int * data)
{
    int * to = data;
    for (int * from = data, * end = data + count;
         from < end;
         ++from)
    {
        if (*from >= 0)
            *to++ = *from;
    }
    return to - data;
}
```

Или использовать макросы, чтобы сократить затраты на вызов функции**:

```
typedef struct
{
    float x, y, z;
} Vector;

#define dotProduct(A, B) (A.x * B.x + A.y * B.y + A.z * B.z)
```

К счастью, современные компиляторы достаточно умны, чтобы сгенерировать оптимальный код по исходнику, по крайней мере, если он прост и понятен. Чем более «хитро» вы пытаетесь писать, тем меньше шансов на

---

* Происхождение этой цитаты остается спорным, хотя версия Кнута считается первой известной опубликованной формулировкой. Также высказывание приписывают Тони Хоару, который, впрочем, считал, что оно больше похоже на высказывание Эдсгера Дейкстры. Все трое, по всей видимости, не были уверены в истинном источнике, относя его к распространенной «народной мудрости» того времени. Идеальный рецепт для бесконечного спора в интернете.

** Обратите внимание на архаичный синтаксис, который я вставил специально, чтобы поностальгировали ветераны.

то, что компилятор правильно поймет ваш замысел. Поэтому старые приемы оптимизации и слишком «заумная» C++-магия часто приводят к худшему результату по сравнению с прямолинейным, логическим кодом.

Однако умные компиляторы не спасают нас от самих себя. Для программиста совершенно естественно беспокоиться о расходе ресурсов, будь то время, память или пропускная способность, и часто из-за такого беспокойства возникает соблазн решить проблемы производительности еще до того, как они всплывут на самом деле.

Предположим, нужно выбрать случайный элемент из списка, и такая задача часто возникает в играх. При этом элементы имеют разный «вес» (вероятность выбора):

```
template <class T>
T chooseRandomValue(int count, const int * weights, const T * values)
{
    int totalWeight = 0;
    for (int index = 0; index < count; ++index)
    {
        totalWeight += weights[index];
    }

    int selectWeight = randomInRange(0, totalWeight - 1);

    for (int index = 0;; ++index)
    {
        selectWeight -= weights[index];
        if (selectWeight < 0)
            return values[index];
    }

    assert(false);
    return T();
}
```

Код довольно прост: вычисляется сумма всех весов, затем выбирается случайное число в этом диапазоне. Далее веса последовательно вычитаются из выбранного числа, и как только сумма становится отрицательной, выбирается соответствующее значение. Вероятность выбора элемента прямо пропорциональна его весу; готово!

Но, посмотрев на этот код, легко прийти к мысли о том, что его можно ускорить. Вроде бы второй цикл, проходящий по значениям, избыточен. При сохранении суммы весов можно воспользоваться подходом «разделяй и властвуй» и сделать второй цикл гораздо быстрее:

```
template <class T>
T chooseRandomValue(int count, const int * weights, const T * values)
{
    vector<int> weightSums = { 0 };
    for (int index = 0; index < count; ++index)
    {
        weightSums.push_back(weightSums.back() + weights[index]);
    }

    int weight = randomInRange(0, weightSums.back() - 1);

    int minIndex = 0;
    int maxIndex = count;

    while (minIndex + 1 < maxIndex)
    {
        int midIndex = (minIndex + maxIndex) / 2;
        if (weight >= weightSums[midIndex])
            minIndex = midIndex;
        else
            maxIndex = midIndex;
    }

    return values[minIndex];
}
```

Это типичная ошибка новичков. Даже не одна, целая цепочка вложенных ошибок. Да, теперь второй цикл имеет сложность O(log N), а не линейную, но это ничего не меняет, ведь первый цикл по-прежнему линейный. Общая производительность почти не изменилась.

И это еще не главный аргумент. Если выбор с весами производится не очень часто, простой линейный цикл все равно окажется быстрее. Пока количество элементов не достигает 200(!), линейный цикл работает быстрее бинарного поиска, по крайней мере при измерениях на моем компьютере.

Удивительно, правда? До этого порога простота логики и более эффективное обращение к памяти перевешивают алгоритмическую эффективность.

Но это не главная проблема. Не важно, насколько быстро выполняется поиск, важно то, что во второй версии выделяется память, а это намного медленнее всей остальной работы функции. При запуске обеих версий оказывается, что первая выполняется в двадцать раз быстрее второй. В двадцать раз!

Но и это еще не самое главное. Главное в том, что скорость работы **`chooseRandomValue`** вообще не имеет значения, как легко можно убедиться, сделав простое профилирование. Пусть функция и вызывается сотни раз в секунду, профайлер покажет, что на ее долю приходится ничтожная доля общего времени. В движке Sucker Punch имеются функции, которые вызываются миллионы раз в секунду, и если вы работаете над игрой, то у вас тоже есть такие. Вот где действительно важна производительность. А для **`chooseRandomValue`** — нет.

Первый урок оптимизации

Итак, первый урок, или заповедь, оптимизации — «не оптимизируй».

Пишите код как можно проще. Не беспокойтесь о скорости его работы. Он будет работать достаточно быстро. А если окажется, что нет, то простой код можно будет легко ускорить — и это уже второй урок.

Второй урок оптимизации

Представьте, что у вас есть простой, надежный код, написанный с обычным усердием. Ваша часть проекта работает немного медленно, вы измеряете его производительность, и, оказывается, что один небольшой фрагмент потребляет половину времени всех вычислений.

И это замечательно! Если удастся ускорить именно этот кусок, то можно удвоить общую производительность системы.

Такое случается довольно часто. При первом анализе кода, который никогда не оптимизировался, обычно все бывает предельно ясно, и сразу видно, над чем стоит поработать.

Плохо, если ничего на первый взгляд неясно. Но если код еще не проходил через несколько циклов оптимизации, такое бывает редко.

Вот вам практическое правило: если вы никогда не оптимизировали участок кода, то его без особых усилий можно ускорить в 5–10 раз. Утверждение это может показаться слишком оптимистичным, но в неотлаженном коде всегда находится много очевидных возможностей для улучшения.

Проверка второго урока

Проверим практическое правило на деле. Допустим, я ошибался насчет **chooseRandomValue**, и этот метод вызывается так часто и с таким количеством вариантов, что фактически занимает половину времени процессора.

Если бы мы начали со второй реализации (с бинарным поиском и выделением памяти), правило было бы легко доказать: достаточно вернуться к простой версии, и код будет работать в 20 раз быстрее. Правило подтверждается!

Но это слишком простой случай. Предположим теперь, что мы начинаем с первой реализации и легко оптимизировать ее не выйдет. Это немного нереалистичный сценарий, так как обычно профилирование первым делом выявляет случаи чрезмерного выделения памяти, которые легко исправить. Но допустим, вам не повезло, и проблема не так проста.

Вот вам процесс оптимизации из пяти шагов. Я здесь рассуждаю о производительности («процессорном времени», если быть точным), но эти шаги применимы к любому ресурсу: пропускной способности сети, использованию памяти, энергопотреблению или любой другой измеримой величине, которую вы хотите оптимизировать.

ШАГ 1: ИЗМЕРЬТЕ И СООТНЕСИТЕ ВРЕМЯ ПРОЦЕССОРА

То есть измерьте, сколько времени процессора тратится на выполнение вашей задачи, и соотнесите его с функциями, объектами или любыми другими элементами, которые вам удобно отслеживать. Будем считать, что в приведенном примере я это уже проделал, потому что знаю, что **chooseRandomValue** потребляет половину времени процессора.

ШАГ 2: УБЕДИТЕСЬ В ОТСУТСТВИИ ОШИБКИ

Довольно часто оказывается, что проблема с производительностью связана с каким-то багом. В данном случае, если **chooseRandomValue** действительно «съедает» половину циклов, я бы не без основания заподозрил наличие ошибки в каком-то другом месте. Стоит тщательно проверить корректность всех вызовов **chooseRandomValue**.

Возможно, кто-то неправильно написал условие цикла и счетчик всегда проходит до максимального значения. Вместо нескольких итераций он выполняется примерно 2^{32} раз, а это огромное количество вызовов **chooseRandomValue**! (Да, мне на самом деле приходилось исправлять именно такой баг.)

ШАГ 3: ИЗУЧИТЕ ДАННЫЕ

Даже не думайте об оптимизации, пока не поймете, как устроены ваши данные. Сколько раз вызывается **chooseRandomValue**? Сколько имеется вариантов выбора? Выбор производится многократно из небольшого числа взвешенных распределений или же он менее предсказуем? Сколько в списке нулевых весов? Есть ли повторяющиеся значения в списках для выбора?

В большинстве случаев оптимизация опирается на какие-то особенности данных или на способ их обработки. Без понимания структуры данных не получится удачной оптимизации.

ШАГ 4: ПЛАНИРУЙТЕ И ПРОТОТИПИРУЙТЕ

Если бы ваш код работал идеально и свел время процессора практически к нулю, то как бы тогда выглядела общая производительность? В данном случае предположим, что **chooseRandomValue** выполняется за нуль времени. Достигли бы вы своей цели по производительности?

Если нет, ваш план недостаточно хорош. Нужно посмотреть, нельзя ли оптимизировать другие фрагменты кода. Не начинайте работу над оптимизацией, пока не убедитесь в том, что она вписывается в общий план, который действительно может повысить производительность.

Иногда бывает трудно предсказать, как будет выглядеть общая производительность при идеальной оптимизации. Одни фрагменты кода взаимодействуют с другими непредсказуемым образом. Возможно, **chooseRandomValue** загружает значения весов в кеш данных процессора, а эти значения использует еще и какая-то другая функция. В худшем случае вы сведете время выполнения **chooseRandomValue** к нулю, но общая производительность от этого не изменится. Допустим, выяснилось, что основная проблема заключалась в загрузке значений весов в кеш данных, так что виновником плохой оптимизации оказался новый элемент.

Подумайте о прототипировании своей оптимизации. В данном случае, например, можно сделать так, чтобы **chooseRandomValue** каждый раз возвращал только первый элемент списка. Это некорректная работа, но она, пожалуй, даст хорошее представление о производительности при идеально оптимальном решении.

ШАГ 5: ОПТИМИЗИРУЙТЕ И ПОВТОРЯЙТЕ

Выполнив первые четыре шага, переходите к собственно оптимизации. После наблюдений и анализа у вас уже появились какие-то идеи о том, какие фрагменты кода наиболее ресурсозатратны. Возможно, какой-то фрагмент или обращение к памяти можно упростить или пропустить. Если простого способа ускорить код нет, подумайте, что можно сделать с данными.

Например, если большинство передаваемых в **chooseRandovValue** весов равны нулю, этим можно воспользоваться, как можно воспользоваться и повторяющимися значениями.

Не спешите бросаться в атаку на первый подозрительный участок кода. План «найти медленный фрагмент и ускорить его» не сработает. Ваши инстинкты, скорее всего, подсказывают ошибочное решение по поводу фрагмента, по поводу оценки данных или по поводу предполагаемого исправления*.

После завершения пятого шага снова измерьте производительность. Если цель достигнута, отлично! Объявите победу и закончите оптимизацию. В противном случае возвращайтесь к первому шагу. Во второй раз некоторые шаги могут проходить быстрее, но на каждом не стоит торопиться; лучше как можно более тщательно проанализировать полученные знания.

Процесс оптимизации из пяти шагов на практике

Итак, я готов к процессу оптимизации! Сворачиваю текстовый редактор и запускаю среду разработки. Начну с первой реализации **chooseRandomValue**, пройду через пять шагов оптимизации и посмотрю, сколько усилий потребуется для десятикратного (10x) ускорения.

Первая реализация **chooseRandomValue** представляет собой пример аккуратного кода — она оптимизирована под простоту и ясность, а это всегда правильная отправная точка. Если мое практическое правило верно, то я теоретически могу без больших усилий ускорить работу кода в 5–10 раз.

Признаюсь, что сейчас, набирая текст, я немного волнуюсь и ощущаю себя неловко.

Шаг 1 я уже проделал, и я знаю, что трачу половину циклов процессора на **chooseRandomValue**.

На шаге 2 я приложил немало усилий, но не нашел ошибок. Все вызовы функции оправданны, и они не делают ничего явно неправильного.

На шаге 3 я выявляю проблему — **chooseRandomValue** вызывается очень часто, и в большинстве случаев передаются длинные списки весов и значений. Данные выглядят довольно случайными, хотя веса маленькие. Большинство значений меньше 5, и ни одно не превышает 15. Интересно, что вызовов много, но все они обращаются к небольшому, статичному количеству распре-

---

* Если только ваш инстинкт не подсказывает, что где-то выделяется память. Тогда вы, скорее всего, правы.

делений, то есть одни и те же списки из тысяч весов и значений передаются снова и снова.

На шаге 4 я создаю версию **chooseRandomValue** с идеальной производительностью. В данном случае я воспользовался версией, возвращающей случайное значение из списка без учета весов — сложно представить что-то проще. Можно было бы просто возвращать первый элемент списка, но тогда бы не был задействован генератор случайных чисел, который, по всей видимости, в рабочей версии должен присутствовать обязательно, поэтому лучшим прототипом кажется версия с возвратом случайного элемента без учета веса.

И вот я тестирую версию… и она работает примерно в 50 раз быстрее исходной реализации. Похоже, что добиться предположительного ускорения в 5–10 раз вполне реально. Переходим к шагу 5 — делаем код еще быстрее!

Ваша первая реакция на задачу ускорения кода могла бы быть такой: «Просто сделать код быстрее». Проделать те же самые действия, только быстрее: ускорить цикл, воспользоваться мультимедийными инструкциями для обработки нескольких элементов одновременно, написать немного кода на ассемблере, вынести из цикла какую-то математику.

Но это плохой выбор. К такой оптимизации нужно прибегать в последнюю, а не в первую очередь. В двух миллионах строк кода игры Ghost of Tsushima мы прибегали к таким микрооптимизациям всего несколько десятков раз. Это не значит, что мы не тратим много усилий на оптимизацию, дело в том, что все, что мы делаем, должно выполняться за шестидесятую долю секунды*. Мы «до седьмого пота» доводили игру до такой скорости. Но за редким исключением при оптимизации мы не стремимся «делать то же самое, но быстрее».

Правильный способ ускорить выполнение кода — это «делать меньше», а не «делать то же самое, но быстрее». Подумайте о том, что код делает лишнего или что он делает несколько раз, когда достаточно одного. Удалите эти лишние операции, и все будет работать быстрее.

В данном случае очевидный кандидат на оптимизацию — вычисление суммарного веса для распределения. В первой реализации **chooseRandomValue** я делал это при каждом вызове, но, проанализировав данные на третьем шаге, я понял, что генерирую случайные значения из ограниченного числа распределений. Можно легко вычислить суммарный вес один раз для каждого распределения и затем использовать его в **chooseRandomValue**:

---

* Или за тридцатую долю секунду, в зависимости от игры. Учтите, что я не утверждаю ничего конкретного о производительности будущих игр.

```
struct Distribution
{
    Distribution(int count, int * weights, int * values);

    int chooseRandomValue() const;

    vector<int> m_weights;
    vector<int> m_values;
    int m_totalWeight;
};

Distribution::Distribution(int count, int * weights, int * values) :
    m_weights(),
    m_values(),
    m_totalWeight(0)
{
    int totalWeight = 0;

    for (int index = 0; index < count; ++index)
    {
        m_weights.push_back(weights[index]);
        m_values.push_back(values[index]);

        totalWeight += weights[index];
    }
    m_totalWeight = totalWeight;
}

int Distribution::chooseRandomValue() const
{
    int select = randomInRange(0, m_totalWeight - 1);

    for (int index = 0;; ++index)
    {
        select -= m_weights[index];
        if (select < 0)
            return m_values[index];
    }
    assert(false);
```

```
    return 0;
}
```

Выделение памяти — операция затратная, и поэтому неудачная первая попытка оптимизировать **chooseRandomValue** провалилась. В ней память выделялась при каждом вызове, и это полностью определяло затраты на выполнение функции в целом. Здесь же выделение памяти происходит только один раз на распределение, а не при каждом вызове. Если бы новые распределения генерировались постоянно, то такое выделение памяти обернулось бы катастрофой, но на третьем шаге (где я измерял данные) я понял, что у меня относительно небольшой список распределений. Выделять память для каждого распределения в этом коротком списке — нормально.

Я запускаю код снова... и он работает примерно в 1,7 раза быстрее базовой версии. Воодушевляет, но это не полная победа. С точки зрения математики стоило ожидать прироста около 3х. Раньше я проходил список весов в среднем 1,5 раза — полностью один раз для вычисления суммарного веса, затем в среднем до половины, чтобы выбрать случайное значение. Теперь я делаю только сам выбор.

Разница зависит от доступа к памяти. Раньше полный проход по весам загружал их в кеш данных, поэтому на втором проходе доступ к ним был быстрым. Теперь же для извлечения значений на втором проходе требуется больше времени, и прирост составил 1,7х вместо ожидаемых 3х.

Теперь, когда выделение памяти стало приемлемым, следующим очевидным шагом представляется использование бинарного поиска. Это несложно, хотя и требует аккуратности:

```
struct Distribution
{
    Distribution(int count, int * weights, int * values);

    int chooseRandomValue() const;

    vector<int> m_weights;
    vector<int> m_values;
    vector<int> m_weightSums;
};

Distribution::Distribution(int count, int * weights, int * values) :
    m_weights(),
```

```
    m_values(),
    m_weightSums()
{
    int totalWeight = 0;

    for (int index = 0; index < count; ++index)
    {
        m_weights.push_back(weights[index]);
        m_values.push_back(values[index]);
        m_weightSums.push_back(totalWeight);

        totalWeight += weights[index];
    }
    m_weightSums.push_back(totalWeight);
}

int Distribution::chooseRandomValue() const
{
    int select = randomInRange(0, m_weightSums.back() - 1);

    int minIndex = 0;
    int maxIndex = m_weights.size();

    while (minIndex + 1 < maxIndex)
    {
        int midIndex = (minIndex + maxIndex) / 2;
        if (select >= m_weightSums[midIndex])
            minIndex = midIndex;
        else
            maxIndex = midIndex;
    }
    return m_values[minIndex];
}
```

Тест этой реализации показывает, что она примерно в 12 раз быстрее базовой версии. Практическое правило подтверждено! Представьте, с каким выдохом облегчения я вернулся в текстовый редактор.

Для большинства случаев ускорения в 12 раз вполне достаточно. После этого можно переходить к другим задачам. Не поддавайтесь искушению продолжать оптимизацию. На волне энтузиазма можно легко увлечься

и погнаться за дополнительным выигрышем в производительности, который в действительности и не нужен. Функция, которая раньше представляла проблему с точки зрения производительности, уже не представляет проблему. На этом этапе она ничем не отличается от любой другой функции в проекте. Ее больше не нужно оптимизировать.

Я и сам в данный момент сталкиваюсь с таким соблазном. У меня появились идеи о том, как сделать `chooseRandomValue` еще быстрее. Мне интересно, какие из них реально сработают, и я борюсь с желанием проверить их. Но после того как достигнут целевой уровень производительности, самое правильное решение — это добавить свои идеи по поводу дальнейшей оптимизации в виде комментариев к коду и отложить его. Признайте победу и двигайтесь дальше.

Я еще не ответил на один очевидный вопрос. Первый урок, или заповедь, оптимизации это «Не оптимизируй», верно? Просто проявите обычную аккуратность, напишите простой и понятный код; исходите из предположения, что в случае необходимости его можно будет ускорить в 5–10 раз.

Но что, если ускорения в 5–10 раз недостаточно? Что, если с самого начала вы допустили настолько большую ошибку в проекте системы, что теперь нужно ускорить ее работу в 100 или 1000 раз?

Третьего урока оптимизации не существует

Вы могли бы возразить и сказать, что есть третий урок оптимизации: «Не делайте глупостей». Если вам нужно написать приложение для высокочастотного трейдинга, не делайте это на Python. Если вы определяете структуру результата, которую будете передавать по всему коду на C++, не проектируйте ее так, чтобы память выделялась для каждой ее копии.

Лично я считаю, что третьего урока, или заповеди, оптимизации не существует. Просто программисты слишком много переживают из-за производительности, вот и все.

Я и сам понимаю такую озабоченность, у меня такая же слабость. Я могу усложнять код ради производительности без всяких на то оснований. Я ловлю себя на этом все время.

Если бы меня попросили сформулировать третью заповедь оптимизации, то я высказался бы примерно так: «Не переживайте из-за ошибок, потому что не существует ошибки, которую вы не сможете исправить».

Если вы все-таки напишете приложение для высокочастотного трейдинга на Python и столкнетесь с проблемами, выход есть. Перепишите критически важные участки на C++, а то, что может работать медленнее, пусть останется на Python. Конвертация из Python в C++ ускорит работу программы (еще

одно практическое правило) примерно в десять раз, и, судя по нашему опыту, после перехода на C++ можно легко ускорить всё в 5–10 раз. В итоге же получается ускорение от 50x до 100x.

Мы в Sucker Punch довольно часто пользуемся подобным методом: пишем первую версию на нашем любимом, но относительно медленном скриптовом языке, а потом, если возникают проблемы с производительностью, конвертируем ее в C++. Так мы можем быстро тестировать свои идеи, зная, что производительность при случае всегда можно будет улучшить.

Запомните, если вы действительно допустили ошибку, для исправления которой понадобится ускорить производительность в 100 раз, то вы почти сразу узнаете об этом. Такие ошибки не прячутся в тени; они очевидны с самого начала, поэтому вы не зайдете слишком далеко, прежде чем их обнаружите. Так что, опять же, не переживайте из-за них.

Доверьтесь двум урокам оптимизации: пишите простой и понятный код и верьте, что решения для любых проблем с производительностью можно будет найти всегда.

Интермедия, в которой предыдущая глава подвергается критике

Я по-прежнему стою на своем и уверен в том, что первый урок оптимизации действительно заключается в том, чтобы «не оптимизировать», как и писал в предыдущей главе. Тем не менее это мое категоричное высказывание, единственное среди многих во всей книге, сразу же вызвало несогласие у значительной части моих коллег в Sucker Punch.

Будет только справедливо выслушать их аргументированные возражения! Я представлю противоположные точки зрения в форме воображаемого сократического диалога между мной и многими моими оппонентами, которых я для более драматического эффекта объединил в одного персонажа. Всем им была предоставлена возможность ознакомиться с этой главой, чтобы они корректно высказали свои взгляды на нее.

Оппонент: Я официально выражаю несогласие с посылом этой главы*.

Крис: Я думал, что эта глава просто описывает здравый смысл. Разве ты не видел цитату Кнута? «О малых оптимизациях нужно забывать, скажем, примерно в 97% случаев: преждевременная оптимизация — корень всех зол».

Оппонент: Эту цитату используют для оправдания создания всякого рода кода с ужасной производительностью, а ты просто поощряешь такое безответственное поведение.

Крис: Ого! В твоих словах ощущается какая-то эмоциональная подоплека. Возможно, тебе приходилось тратить слишком много времени на переработку чужого кода и исправлять проблемы с производительностью, которых вообще не должно было быть изначально? Или ты часто ждал загрузки популярных видеоигр?

Оппонент: И то и другое.

Крис: Ты часто работаешь с частями кода, для которых критически важна производительность, и, наверное, от этого зависит расстановка приоритетов.

---

* Это точная цитата.

У тех, кто, скажем, занимается логикой пользовательского интерфейса, приоритеты другие.

Оппонент: Верно, хотя мы оба знаем игру*, в которой пользовательский интерфейс был настолько неудачным, что проблемы с производительностью сочли неустранимыми. Весь интерфейс пришлось переписывать заново, из-за чего выпуск игры задержался на полгода.

Крис: Да. К этому случаю применимо Правило 20 «Не забывайте про математику». Проблем не возникло бы, если бы разработчики вовремя поняли, насколько плоха их архитектура, и исправили ее раньше. Серьезные проблемы с производительностью обычно проявляются сразу, но только если ее измерять. Я бы даже сформулировал четвертое правило оптимизации: «Исходи из предположения, что твой код будет достаточно быстрым, но все равно измеряй его».

Оппонент: Ну хорошо, я отчасти готов согласиться. Обычно нам под конец проекта удается более или менее всё оптимизировать, потому что у нас есть точные инструменты профилирования и мы пользуемся ими в своей повседневной работе.

Крис: Да. Думаю, это аналог нашей версии тестирования: во многих командах упор делается на тесты, а у нас — на профилирование. У нас не так много модульных тестов, потому что мы готовы мириться с редкими багами, но не готовы к неожиданным проблемам с производительностью.

Оппонент: И все же, на мой взгляд, в твоих рассуждениях в пользу Правила 5 упущен один важный момент. Его можно легко понять как «Не заботься о производительности», тогда как на самом деле ты хотел сказать: «Простой код легко оптимизировать, поэтому пиши простой код».

Крис: Ну да, ты прав, это соотносится с Правилом 1 и общей темой книги: «Делай код максимально простым, но не проще». Одно из преимуществ такого подхода — то, что код можно будет легко оптимизировать.

Оппонент: Но даже если ты пишешь простой код, ты все равно думаешь о том, как его при необходимости можно было бы ускорить. Я это замечал во время ревизии твоего кода. Или когда ты проверял мой. В общем, в обоих случаях.

Крис: Конечно. Мы все об этом думаем. Возможность оптимизации — не главный приоритет, ведь на первом месте стоят корректность и простота, но продумать путь оптимизации все же бывает полезно, даже если в итоге она не понадобится. Чаще всего не понадобится.

Оппонент: Да, оптимизация не всегда дается даром. Если она усложняет код, требует больше памяти или добавляет какой-то этап предварительной

---

* Которую мы называть не будем.

обработки, то выигрыш в производительности должен того стоить. Более быстрый код — это необязательно более хороший код. В этом мы согласны.

Крис: Отлично!

Оппонент: И еще. Несмотря на то что простой код действительно легче оптимизировать, медленный код от этого простым не становится. Более того, излишнее усложнение — это самый простой способ сделать код медленным.

Крис: Абсолютно верно.

Оппонент: Надо сказать, что это Правило не совсем согласуется с моей работой по оптимизации. Обычно я не улучшаю производительность нового кода, я пытаюсь выжать больше из уже оптимизированного. А это куда труднее.

Крис: Да. Эта глава и вправду больше о том, как писать новый код.

Оппонент: Верно, но даже добавляя новый код в систему, для которой критически важна производительность, я должен думать о производительности с самого начала. Я не могу ограничиться простотой кода и надеяться на лучшее.

Крис: Пожалуй, это справедливо. По крайней мере, для разумно планируемого первого этапа. Но разве ты не согласишься с тем, что излишняя озабоченность производительностью на ранних этапах не раз приводила к тому, что код оказывался оптимизирован сильнее, чем нужно?

Оппонент: С неохотой соглашусь, но, думаю, такие случаи редки. В целом я все равно экономлю время, не тратя его на код, который почти сразу же придется оптимизировать.

Крис: С этим я могу согласиться. Даже Кнут писал, что его правило относится к 97%, верно? Если исходя из своего опыта ты считаешь, что попадаешь в те самые 3%, то да, тебе логично ориентироваться на производительность уже при первой реализации. Просто не увлекайся, пока не измеришь свой код и не убедишься, что проблема реально существует. А если все в команде считают, что именно их участок относится к этим 3%, то вам стоит получше заняться профилированием.

Оппонент: И еще один момент, связанный с оптимизированным кодом. В нем сложнее добиться заметных улучшений. Допустим, я соглашусь, что новый код можно ускорить в 5–10 раз без особых усилий. Но в какой-то момент простые решения заканчиваются, и каждый следующий прирост производительности дается гораздо труднее.

Крис: Да, и на этом этапе правила игры меняются. Скорее всего, ты сократишь время выполнения вдвое, сделав пять небольших улучшений, а не одно большое. Но даже тогда нужно исходить из предположения о том, что существует некое более радикальное, алгоритмическое решение. Например, в первой игре из серии Sly Cooper мы неделями возились с оптимизацией

основного цикла отрисовки, добиваясь мизерных долей процента производительности, и только потом, перейдя на систему пространственного разбиения, добились пятикратного ускорения.

Оппонент: Это было еще до моего прихода. Но история отличная.

Крис: А что скажешь о пятишаговом процессе оптимизации?

Оппонент: Вполне неплохой процесс. С этой частью главы я был согласен.

Крис: Не могу поверить, что никто не отметил мой гениальный пункт «Шаг 2: Убедись в отсутствии ошибки». Я ведь им гордился.

Оппонент: Я и одобрил его тем, что не стал критиковать. Не рассчитывай на похвалу, Крис. Никто из нас не хочет иметь дело с еще более самоуверенной твоей версией.

Крис: Справедливо.

Правило 6

Ревизия кода полезна по трем причинам

Если за последние тридцать с лишним лет профессионального программирования я и изменил к чему-то свое отношение радикальным образом, так это к различным формам ревизии кода.

До начала 1990-х я вообще не слышал о коллегиальных проверках кода или так называемых ревизиях кода («код-ревью»). Не то чтобы их не существовало — конечно, так или иначе код проверяли, — но обычно только в ситуациях, в которых «ошибка недопустима», например, в прошивках медицинских приборов или в управляющем коде ракет. Ну, то есть там, где от багов зависит людская жизнь.

Для большинства же программистов мысль о том, что кто-то другой будет просматривать твой код, лет тридцать назад казалась… вторжением в личное пространство. Понятно, если ты работаешь в команде, тебе приходится хотя бы взглянуть на интерфейс чужого модуля, чтобы понять, как с ним взаимодействовать, и, скорее всего, ты хоть раз проходился по чужому коду в отладчике. Но вот построчно разбирать код и высказывать свое мнение о нем — это воспринималось как нечто странное, почти интимное. Как будто читаешь чей-то дневник или — если перевести на современные реалии — случайно заглянул в чужую историю браузера.

Так или иначе, в начале 90-х я перешел в команду Microsoft, где существовала политика ревизий кода. К счастью для меня, мой проект в масштабах всего коллектива был настолько незначителен, что на меня и моих коллег никто не обращал внимания. Среди прочего это означало, что мы с коллегами сами решали, как проверять код. Я даже не знаю, как это было официально принято в большой команде; мы же просто поступали так, как считали разумным, и никто за нами ничего не перепроверял. Конечно, я и не собирался спрашивать совета из опасения, что меня заставят соблюдать какую-нибудь жуткую бюрократическую процедуру. Лучше уж попросить прощения потом, чем разрешения заранее.

К своему удивлению, я обнаружил, что ревизия кода приносит немедленную и неоспоримую пользу. С тех пор я постоянно провожу их со своими командами, но не совсем по тем причинам, по которым ожидал.

Самая очевидная причина проводить ревизии кода — это выявление ошибок до того, как они попадут в проект. Если процесс ревизии организован разумно, рецензент хорошо понимает проверяемый фрагмент кода. Возможно, он сам принимал участие в реализации этого кода, хорошо знает другую часть системы, от которой зависит новый код, или просто часто пользуется проверяемым кодом. В любом случае он может заметить проблемы, например, нарушение условий, неправильное использование уже существующего кода или изменение поведения системы, из-за чего может сломаться другой фрагмент, над которым работает рецензент.

Происходит ли это на практике? Находят ли во время ревизии кода реальные ошибки? Конечно, по крайней мере, исходя из моего опыта и из того, как ревизии проводятся у нас, в команде Sucker Punch.

Это важная оговорка: польза от ревизии кода напрямую зависит от того, сколько времени и внимания ей уделяют, а также от того, как ее проводят. Вот краткое описание ревизий кода, проводимых у нас в Sucker Punch в настоящее время:

- Ревизия происходит в реальном времени, и два человека сидят за одним компьютером (по крайней мере, так было до пандемии).
- Процесс неформальный. Когда у тебя готов фрагмент кода на проверку, ты подходишь к потенциальному рецензенту и просишь его взглянуть на твой фрагмент. Согласно негласному договору, если кто-то просит тебя проверить его код, то ты должен его проверить, если только у тебя нет по-настоящему неотложных дел.
- Рецензент просматривает изменения в программе сравнения версий (diff), а автор кода комментирует внесенные правки. Обсуждение проводится в форме диалога; рецензент задает вопросы, пока не убедится, что полностью понимает сделанные изменения, предлагает улучшения, высказывает пожелания по поводу улучшения и обсуждает возможные альтернативные решения. Позволять вести беседу проверяемому — это обычно ошибка, потому что слишком легко принять его объяснения на веру, не вдумываясь в детали.
- Автор кода должен записать все предложения по исправлениям и дополнительным тестам. По умолчанию считается, что все предложения нужно реализовать. После большой ревизии остается множество страниц с описанием доработок.

- Обычно одной ревизии бывает достаточно. После внесения изменений и проведения дополнительных тестов автор отправляет код в репозиторий («коммитит» его). Иногда, если изменений много, рецензент может повторно проверить обновленный вариант. Если рецензент не уверен, что понял какие-то изменения полностью, он может попросить посмотреть код другого члена команды. Но чаще всего процесс выглядит так: ревизия кода + внесение изменений + коммит.

В ходе этого процесса мы действительно находим баги… но, опять же, не так, как можно было бы ожидать. Вот способы нахождения багов во время ревизии примерно в порядке убывания частоты:

- Перед тем как попросить кого-то просмотреть ваш код, вы сами просматриваете изменения в diff, чтобы устранить все возможные неловкие моменты. В ходе такой проверки можно найти какой-нибудь баг и исправить его до того, как код увидят другие.
- Во время ревизии вы, как автор кода, объясняете рецензенту, что делает тот или иной конкретный фрагмент кода, и вдруг понимаете, что он ошибочен. Вы сами указываете на баг, начинается обсуждение, вы делаете заметки и продолжаете просматривать код. Если обнаруженный вами дефект достаточно серьезен, ревизия останавливается и начинается заново после внесения всех необходимых изменений.
- Во время ревизии рецензент замечает пропущенное вами проблемное место, либо в ходе беседы, когда вы описываете свои действия, выясняется, что вы неправильно понимаете работу какого-то участка вызываемого кода. Вы с рецензентом обсуждаете возможную проблему, соглашаетесь с тем, что это проблема, и делаете заметку.

Редко бывает, что рецензент находит баг, просто рассматривая код. Обычно ошибки выявляются либо во время подготовки к ревизии, либо в ходе обсуждения внесенных изменений. Вот почему полезно, чтобы ревизии кода проходили в форме диалога, помогающего выявить любые возможные расхождения во мнении между рецензентом и автором кода. Диалог полезен не только для поиска ошибок, но и для добавления комментариев или изменения имен.

Важно отметить очевидные ограничения нашего процесса ревизии кода. В наших версиях кода встречаются тысячи ошибок, а ведь мы не делаем исключений из требований обязательной проверки — каждая строка кода, попадающая в репозиторий, проходит через процесс ревизии. Значит, каждую ошибку перед добавлением кода пропустило несколько человек. Ревизия кода действительно помогает находить ошибки, но, конечно, далеко не все.

Ревизия — неэффективный способ поиска ошибок, и все же мы продолжаем пользоваться ею. Дело в том, что поиск ошибок — это лишь одна из причин, почему мы ею пользуемся, и даже не самая важная.

Ревизия кода — это способ обмена знаниями

Вот более весомая причина проводить ревизию кода: при правильном осуществлении она становится отличным способом распространения знаний внутри команды.

Для коллектива Sucker Punch это особенно важно, потому что у нас гибкая система распределения задач и программисты довольно свободно переходят от одной части кодовой базы к другой. Общая работа идет гораздо лучше, если каждый из них имеет хотя бы базовое представление об устройстве разных частей проекта. Ревизия кода помогает распространять эти знания.

Представьте, что вы условно делите программистов своей команды на две группы по степени знакомства с кодовой базой — «младших» («джуниоров») и «старших» («сеньоров»). Старшие программисты хорошо ее знают, а младшие еще только осваиваются. Так как в ревизии кода участвуют два человека, возможны четыре комбинации по уровню опыта рецензента и рецензируемого (автора кода). Полезными из них оказываются только три, как показано в таблице 6-1.

Таблица 6-1. Таксономия ревизии кода

	Старший рецензент	Младший рецензент
Старший рецензируемый	Полезно	Полезно
Младший рецензируемый	Полезно	ЗАПРЕЩЕНО

Если код младшего программиста просматривает старший, он хорошо замечает проблемы — не только ошибки в самом коде, но и общие заблуждения младшего. Возможно, младший программист не соблюдает принятые в команде стандарты форматирования, слишком рано переходит к обобщениям или пишет излишне сложный код для простой задачи. Формально это не ошибки, но нарушение «Правил программирования» снижает качество кода, поэтому старший программист в ходе ревизии должен указать на такие моменты, чтобы они были исправлены.

Когда младший программист просматривает код старшего, младший с меньшей вероятностью найдет ошибки, но чаще будет задавать вопросы, чтобы понять, что происходит. Отвечая на эти вопросы, автор кода помогает рецензенту разобраться в контексте, и тот лучше начинает понимать, как различные части кодовой базы связаны между собой. Рецензент видит примеры

хорошего кода с правильным форматированием, удачной архитектурой и ясными именами.

Обе эти ситуации — «младший просматривает код старшего» и «старший просматривает код младшего» — можно рассматривать как часть обучения новых разработчиков. Для повышения своей квалификации новичку нужно понимать, как устроен проект, как пишется код в команде и почему все делается именно так. Ревизия — отличный способ передачи этих неформальных знаний новым членам команды.

Третий полезный вариант: старший программист просматривает код другого старшего программиста. Это по-настоящему неплохая возможность найти ошибки, сверить взаимные представления о том, как изменения вписываются в общий контекст, обсудить будущие задачи, наметить дополнительные тесты и убедиться, что вносимые изменения понимают хотя бы два человека в команде.

Запрещенный тип ревизии

Последнее сочетание, когда младший программист проверяет код другого младшего, бесполезное, а порой и вредное. При этом пропадают все перечисленные выше выгоды: обмена знаниями не происходит, не удается понять контекст для поиска ошибок, и ревизию невозможно использовать как основу для обсуждения дальнейшей работы. В худшем случае два новичка начинают обмениваться своими смутными идеями, постепенно воспринимая их как официальные правила команды. Когда в коде Sucker Punch появляются странные приемы и соглашения (а такое иногда случается, несмотря на все наши усилия), то часто так получается из-за того, что младшие программисты обсуждали друг с другом свои фрагменты кода. Поэтому такие ревизии у нас запрещены.

Истинная ценность ревизии

Мы находим ошибки и передаем знания. Этого уже достаточно, чтобы оправдать потраченное на ревизию время, обычно всего 5–10% от времени, потраченного на написание самого кода. Но есть еще одно, пожалуй, самое важное преимущество ревизии кода, лежащее целиком в социальной плоскости.

Люди пишут лучше, когда знают, что кто-то будет просматривать их код.

Разработчики внимательнее следуют правилам форматирования и именования. Они не прибегают к «костылям» и не откладывают исправления на

потом. Они пишут более понятные комментарии. Они решают задачи правильно, а не с помощью временных решений. Они не забывают удалить диагностический код, добавленный для поиска ошибок.

Все это происходит до самой ревизии, потому что программисты испытывают внутреннее давление, они хотят гордиться своей работой, чтобы ее не было стыдно показать коллегам. Это здоровая форма «социального давления». В результате мы пишем более качественный код, и со временем наша кодовая база становится устойчивее, а весь коллектив становится продуктивнее.

Ревизия кода — процесс социальный

Подводя итог, перечислим причины, по которым полезна грамотно организованная ревизия кода:

- Она помогает находить ошибки.
- Участники проекта начинают лучше понимать код.
- Она мотивирует и заставляет писать код, который не стыдно показать другим.

Как и любой процесс, ревизия кода требует осмысленного подхода: если уж вы тратите на нее время, то важно понимать, зачем и ради чего. Избавляйтесь от элементов, не приносящих пользы, и делайте упор на те, что работают. Так вы либо получите больше пользы за то же время, либо сэкономите время без потери результата.

Пишут код и занимаются его отладкой обычно в одиночку. Программист — это одинокий воин за клавиатурой, сражающийся с багами и упрямыми библиотеками.

Ревизия кода — занятие, противоположное по своей природе. Ее ценность заключается в общении между рецензентом и автором кода. Объясняя сроки кода, вы вдруг замечаете в них ошибки; описывая модуль, вы помогаете коллеге понять, как им пользоваться; перед ревизией вы подчищаете «грязь», которую никому не хотите показывать, или узнаёте от коллеги, как можно проще решить ту или иную задачу.

Так как ценность ревизии заключается именно в живом взаимодействии, стоит убедиться, что сам процесс ревизии способствует такому взаимодействию. Если ревизия происходит в тишине, когда рецензент молча листает diff и лишь изредка что-то бормочет себе под нос, то значит, что что-то не так. Формально это еще ревизия кода, но не приносящая особой пользы.

И если каждая ваша ревизия кода превращается в спор, то вы делаете что-то не так! Автор кода, не готовый воспринимать замечания рецензента,

ничему не научится; но и рецензент, не пытающийся понять, почему код был написан именно так, тоже ничего не вынесет из обсуждения. В любом случае ревизия кода это не место для споров о направлении проекта, командных соглашениях или философии разработки. Эти вопросы нужно решать коллективно, в команде, а не в ходе двухсторонних перепалок.

Здоровая ревизия кода не только улучшает кодовую базу, но и укрепляет командный дух. Это профессиональный и открытый диалог, обогащающий обоих участников новыми знаниями.

Правило 7

Устраняйте сценарии сбоев

Уж слишком оптимистичное правило, вам не кажется? Но что вообще оно означает?

Некоторые сценарии сбоев ведь неизбежны, верно? Когда я пытаюсь открыть файл, то этот файл может не существовать или может быть заблокирован другим пользователем. При открытии файла никакие дизайнерские трюки и никакие хитрости интерфейса не помогут устранить возможность собственно ошибки при открытии файла. Так что, вероятно, речь идет не об этом. Пожалуй, тут говорится о том, что нужно устранять такие сбои, которых можно было бы избежать на самом деле, — сбои, не связанные с самой природой операций с файлами, а вызванные ошибками использования вроде записи в файл после закрытия его дескриптора или вызова методов объекта до полной инициализации этого объекта.

Возможно, нужно саму систему проектировать так, чтобы подобные ошибки были невозможны, но это, мягко говоря, непросто. Создать систему, воспользоваться которой неправильно было бы попросту невозможно, — это очень трудная задача. Если вы предлагаете пользователям какой-то вариант действий, то они обязательно найдут способ воспользоваться им так, что все сломается или все пойдет не так, как задумано. Это все равно, что кто-то решит попытаться построить рабочий 8-битный процессор целиком из блоков Minecraft*.

И даже в вашей команде другие программисты рано или поздно воспользуются какой-нибудь возможностью неправильно. Иногда специально, в отчаянной попытке заставить что-то заработать — например, закрыть дескриптор файла после вызова процедуры завершения работы файловой системы, потому что это единственный способ избежать нежелательного обратного вызова. Но чаще такое происходит непреднамеренно, потому что они неправильно понимают, как именно нужно вызывать ваш интерфейс.

Главный вопрос, который обязательно стоит задавать себе при проектировании, звучит так: «Насколько я усложнил пользователям возможность случайно выстрелить себе в ногу?» Конечно, правильным ответом будет:

---

* Я не шучу.

«В очень большой степени». Но на практике мы слишком часто создаем интерфейсы, ошибиться в которых проще простого.

Если функцию или интерфейс легко использовать неправильно, ошибки неизбежны. В каком-то смысле ошибки изначально заложены в эту функцию или интерфейс. Но мы же хотим сделать так, чтобы ошибки были невозможны даже исходя из дизайна программы. Пойдем от обратного и для начала рассмотрим несколько примеров функций, конструкция которых изначально предполагает возможность ошибок.

Функция, облегчающая «выстрел в ногу»

Каждый программист на C знает хотя бы один пример функции, с которой легко ошибиться, — **printf**. В ее конструкции заложена фундаментальная проблема: она ожидает, что переданная ей строка формата будет соответствовать типам переданных аргументов, но если это не так, начинается хаос неопределенности.

Следующий код работает, потому что типы совпадают:

```
void showAuthorRoyalties(const char * authorName, double amount)
{
    printf("%s is due $%.2f this quarter.\n", authorName, amount);
}
```

Но если немного изменить форматную строку, все рушится:

```
void showAuthorRoyalties(const char * authorName, double amount)
{
    printf("remit $%.2f to %s this quarter.\n", authorName, amount);
}
```

Грубо говоря, **printf** пытается интерпретировать **authorName** (строку) как вещественное число (с плавающей точкой). Результат будет непредсказуем. Скорее всего, программа не вылетит с ошибкой, ведь любую комбинацию из 64 бит можно интерпретировать как **double**, — но результат форматирования будет случайным, возможно, даже «NaN». Затем **printf** попытается интерпретировать **amount** (вещественное число) как строку, и это уже, скорее всего, приведет к сбою.

Однако на практике при компиляции и запуске приведенного кода вылета или ошибки не будет. Из-за того, что ошибиться в **printf** легко, в современные компиляторы C добавили специальные проверки. Когда я пытаюсь

скомпилировать неверный пример, компилятор выдает ошибки (errors) для обоих аргументов. Если передаваемая в **printf** форматная строка является константой, компилятор может проверить соответствие типов (и делает это).

Пример как раз подтверждает мою мысль: конструкция **printf** настолько плоха, что для решения проблемы приходится внедрять в компилятор специальные проверки. Для вашего собственного кода, разумеется, такие проверки работать не будут. Если вы в стиле **printf** реализуете свою собственную функцию форматирования, компилятор не станет проверять соответствие типов*.

«Выстрел в ногу» рикошетом

Чтобы все-таки добиться сбоя, я обхожу хак компилятора, проверяющий соответствие типов:

```
void showAuthorRoyalties(const char * authorName, double amount)
{
    printf(
        getLocalizedMessage(MessageID::RoyaltyFormat),
        authorName,
        amount);
}
```

Вместо того чтобы задавать форматную строку напрямую, я беру ее из списка. Наши игры переводятся на множество языков, и потому все видимые пользователю строки берутся из базы локализованных строк. Компилятор не знает, что именно представляет собой эта строка, поэтому не может проверить соответствие типов, как в случае с литеральной строкой.

Результат — катастрофа: плохая идея, обернутая в еще худшую. Начав с ненадежной конструкции **printf**, я усугубил ситуацию, полностью отделив содержимое форматной строки от ее использования, но при этом сохранив скрытую зависимость от порядка параметров. В итоге при локализации игры

---

* Ну ладно, в очень ограниченных случаях это не так. Если пользоваться форматными строками, которые в точности соответствуют спецификации **printf**, и вы готовы углубиться в документацию вашего компилятора, то, вероятно вы сможете придумать, как воспользоваться поддержкой **printf** в компиляторе для вашего случая. Но я бы не советовал вам этим заниматься. В погоне за плохим решением вы просто впустую потратите время и деньги.

какой-нибудь несчастный переводчик, руководствуясь грамматикой своего языка*, неизбежно поменяет порядок параметров, и наш код рухнет.

Отчасти команде **printf** можно простить такое странное поведение, ведь ее дизайн восходит буквально к началу времен**. Но требование, чтобы несколько аргументов совпадали по какому-то скрытому признаку, несмотря на то, что это плохая идея, встречается повсеместно. Например, можно написать функцию, ожидающую передачи двух массивов одинакового размера:

```
void showAuthorRoyalties(
    const vector<string> & titles,
    const vector<double> & royalties)
{
    assert(titles.size() == royalties.size());
    for (int index = 0; index < titles.size(); ++index)
    {
        printf("%s,%f\n", titles[index].c_str(), royalties[index]);
    }
}
```

Можно также добавить аргументы, смысл которых зависит от других аргументов, например, отмечать флагами единичные матрицы в функции преобразования координатного пространства в (возможно, ошибочной) попытке избежать расходов ресурсов на обращение матрицы и несколько операций умножения матриц:

```
Point convertCoordinateSystem(
    const Point & point,
    bool isFromIdentity,
    const Matrix & fromMatrix,
    bool isToIdentity,
    const Matrix & toMatrix)
{
    assert(!isFromIdentity || fromMatrix.isZero());
    assert(!isToIdentity || toMatrix.isZero());
```

---

* Например, судя по одному приложению для перевода, на ирландском это будет так: «Tá $%.2f dlite do %s an ráithe seo».

** И это буквально. Язык C и **printf** были изобретены на заре времен Unix. И сейчас, спустя полвека, **printf** продолжает использоваться, что по моему опыту очень редкое явление для любого кода.

```
    Point convertedPoint = point;
    if (!isFromIdentity)
        convertedPoint *= fromMatrix;
    if (!isToIdentity)
        convertedPoint *= Invert(toMatrix);
    return convertedPoint;
}
```

В лучшем случае подобные проблемы удастся обнаружить только при выполнении программы; компиляция их не выявит.

Варианты действий после обнаружения ошибки неутешительны. Если возвращать ошибку при несогласованных аргументах, придется писать обработку ошибок в коде вызова. А добавление обработки ошибок из-за собственной ошибки при вызове функции — явный признак того, что что-то серьезно сломано.

Альтернатива — добавить проверку `assert`, чтобы убедиться в согласованности аргументов. В зависимости от того, как используются проверки `assert`, результат может быть самым разным, от немедленного аварийного завершения до вывода сообщения, которое на свой страх и риск можно будет проигнорировать. Ни один из этих вариантов нельзя назвать хорошим решением.

Как компилятор помогает «не стрелять себе в ногу»

Лучшее решение — спроектировать интерфейс так, чтобы им нельзя было воспользоваться неправильно, или хотя бы так, чтобы этого не позволял компилятор. Чтобы длина массивов точно не была разной, можно, например, объединить параллельные массивы:

```
void showAuthorRoyalties(const vector<TitleInfo> & titleInfos)
{
    for (const TitleInfo & titleInfo : titleInfos)
    {
        printf("%s,%f\n", titleInfo.m_title.c_str(), titleInfo.m_
royalty);
    }
}
```

Можно также объединить связанные аргументы в один:

```
Point convertCoordinateSystem(
    const Point & point,
    const Matrix & fromMatrix,
    const Matrix & toMatrix)
{
    Point convertedPoint = point;
    if (!fromMatrix.isIdentity())
        convertedPoint *= fromMatrix;
    if (!toMatrix.isIdentity())
        convertedPoint *= Invert(toMatrix);
    return convertedPoint;
}
```

Исправить проблему с локализованным **printf** сложнее. Если вы хотите одновременно проследить за тем, чтобы типы были безопасными, и чтобы при изменении порядка аргументов результат оставался корректным, простого решения в виде передачи всех аргументов как строк будет недостаточно.

Обе проблемы можно решить, создав вспомогательные функции, форматирующие каждый аргумент по отдельности и возвращающие имя поля вместе с отформатированным аргументом:

```
void showAuthorRoyalties(const char * authorName, double amount)
{
    // Например: "{AuthorName} is due {Amount} this quarter."
    // "В этом квартале {AuthorName} должен {Amount}."
    printMessage(
        MessageID::RoyaltyFormat,
        formatStringField("AuthorName", authorName),
        formatCurrencyField("Amount", "#.##", amount));
}
```

Теперь хотя бы можно обнаружить несоответствие между локализованной форматной строкой и передаваемыми аргументами, хотя, к сожалению, и не на этапе компиляции. Форматная строка может задавать аргументы в любом порядке, имеющем смысл для целевого языка, а **printMessage** разберется, что к чему. Если в форматной строке указано имя поля, которое вы не передаете, или, наоборот, вы передаете поле, не упомянутое в строке, это можно зафиксировать в журнале во время выполнения. Еще лучше, если несоответствие будет обнаружено в инструменте, используемом командой

локализации для перевода; в таком случае проблему можно будет исправить еще до запуска программы.

Время решает

Самое главное при создании устойчивых к ошибкам интерфейсов — это обнаружить возможность неправильного использования как можно раньше.

В худшем случае ошибка вообще не обнаруживается, и функция просто выдает неверные результаты. Все зависит от того, заметят ли вызывающие код свою ошибку и разберутся ли они с ней. Но обычно никто ничего не замечает, и остается только удивляться, откуда взялось столько дырок в ногах.

Хуже, если ошибка будет выявлена уже во время выполнения кода; да, это плохо, но все же лучше, чем продолжать работу, не обращая внимания на проблему. В идеале об ошибке нужно сообщать так, чтобы ее невозможно было проигнорировать.

Будет лучше, если ошибку обнаружит компилятор. Пропустить код, который не компилируется, невозможно.

В идеале же вся система должна быть спроектирована так, чтобы ошибочную идею невозможно было даже реализовать в коде!

Более сложный пример

Еще одна сфера, где ошибки чаще «встраиваются» в систему, чем устраняются, — это создание сложных объектов.

Вот вам пример. В Sucker Punch мы часто пишем код для различных визуализаций игрового мира, что помогает в отладке. Например, у нас есть код, отображающий схематичные границы областей, по которым могут передвигаться персонажи. Есть код, рисующий небольшие маркеры над головами NPC (неигровых персонажей), знающих местоположение игрока, то есть тех, которые, согласно системе ИИ, «видят», где находится игрок. Также имеется код, отображающий числа над точками на местности, в которые враг с мечом может потенциально переместиться во время боя.

Такого рода отладочная визуализация устроена гораздо сложнее, чем кажется. Наша система отладочного рендеринга поддерживает 30 различных параметров отрисовки, задаваемых через объект контекста отладочного рисования. Эти 30 параметров задают параметры преобразования простых вызовов отрисовки — например, вызов функции отрисовки треугольника по трем точкам — в реальные графические примитивы. В какой системе координат задавать эти точки? Изображать ли треугольник как каркас или как непрозрачную фигуру? Должен ли треугольник быть

видимым, если он находится за стеной? И тому подобное еще для двадцати семи параметров.

Можно было бы передавать в конструктор все 30 параметров, но это крайне неудобно. Если в целях демонстрации упростить нашу реальную структуру параметров отладочной отрисовки, то код выглядел бы примерно так:

```
struct Params
{
    Params(
        const Matrix & matrix,
        const Sphere & sphereBounds,
        ViewKind viewKind,
        DrawStyle drawStyle,
        TimeStyle timeStyle,
        const Time & timeExpires,
        string tagName,
        const OffsetPolys & offsetPolys,
        const LineWidth & lineWidth,
        const CustomView & customView,
        const BufferStrategy & bufferStrategy,
        const XRay & xRay,
        const HitTestContext * hitTestContext,
        bool exclude,
        bool pulse,
        bool faceCamera);
};
```

Все эти параметры действительно используются в каких-то фрагментах кода Sucker Punch, но в большинстве случаев вызываются лишь один-два из них, а для остальных используются значения по умолчанию. Например, почти всю отладочную отрисовку мы выполняем в той же трехмерной системе координат, которая применяется для самой игры.

Если мы, допустим, хотим нарисовать над головой персонажа простую сферу (как в том сценарии, где отображаются NPC, знающие о местоположении игрока), код может выглядеть так:

```
void markCharacterPosition(const Character * character)
{
    Params params(
        Matrix(Identity),
```

```
        Sphere(),
        ViewKind::World,
        DrawStyle::Wireframe,
        TimeStyle::Update,
        Time(),
        string(),
        OffsetPolys(),
        LineWidth(),
        CustomView(),
        BufferStrategy(),
        XRay(),
        nullptr,
        false,
        false,
        false);

    params.drawSphere(
        character->getPosition() + Vector(0.0, 0.0, 2.0),
        0.015,
        Color(Red));
}
```

И это довольно плохой вариант. Пользоваться таким дизайном неудобно, и он содержит много мест, в которых легко допустить ошибку. Запомнить порядок из 16 аргументов невозможно, и чтобы понять, куда вставлять тот или иной параметр, приходится полностью полагаться на IDE. Может, компилятор и спасет вас в случае с аргументами уникальных типов, но попробуйте понять, за что отвечает каждый из четырех логических аргументов в конце списка. Если просто прочитать функцию **markCharacterPosition**, то ни за что нельзя догадаться об их назначении.

Остается только надеяться на то, что вам никогда не придется добавлять в конструктор аргументы или удалять их из него. Даже в ходе поверхностного анализа кода Sucker Punch можно выявить около 850 мест, в которых определяются параметры для отладочного рендеринга. Не завидую тому, кому придется заниматься их правками.

Особенность функций со множеством параметров заключается в том, что ими неудобно пользоваться, и со временем это неудобство только растет. И все из-за эффекта положительной обратной связи. Функция, которая уже принимает восемь аргументов, с большой вероятностью получит и девятый — ведь именно такие «громоздкие» функции чаще всего и продолжают

обрастать параметрами. С увеличением количества аргументов ситуация только ухудшается. Так что если вам показалось, что функция начинает «распухать», стоит заранее продумать пути к отступлению.

Самое частое решение проблемы перегруженных конструкторов состоит в том, чтобы разбить процесс создания объекта на несколько вызовов. Сам конструктор задает значения по умолчанию, после чего вызываются специальные методы, меняющие нужные параметры и доступные только во время инициализации. Когда все готово, процесс завершается вызовом метода вроде **commit**.

Допустим, мы хотим, чтобы над персонажем отображалась метка, видимая сквозь стены, но с уменьшенной на 50% яркостью. Тогда реализация метода «пошагового конструирования» выглядела бы примерно так:

```
void markCharacterPosition(const Character * character)
{
    Params params;
    params.setXRay(0.5);
    params.commit();

    params.drawSphere(
        character->getPosition() + Vector(0.0, 0.0, 2.0),
        0.015,
       Color(Red));
}
```

Это гораздо лучше, чем вариант с шестнадцатью аргументами, но мы создали новое потенциальное место отказа — зависимость от порядка вызовов. Один набор методов вроде **setXRay** вызывается во время конструирования объекта **params**. Другой набор вроде **drawSphere** вызывается уже после конструирования параметров. Мы нигде не определили, что произойдет, если вызвать их не в ожидаемом порядке — например, вызвать **setXRay** после **commit** или вызвать **drawSphere** до **commit**. Поэтому редактор с компилятором ни о чем не могут нас предупредить, и ошибка обнаружится только во время выполнения — скорее всего, в результате срабатывания **assert** внутри **setXRay** или **drawSphere**.

Ловить такие ошибки на этом этапе — далеко не лучший вариант. Да, так лучше, чем не ловить их совсем, но все-таки хотелось бы либо находить их раньше, либо вовсе не давать им возникнуть.

Можно договориться о правилах, помогающих программистам избегать ошибок, связанных с порядком. Команда могла бы договориться о едином

методе реализации многоэтапных конструкторов: допустим, у конструктора никогда не бывает аргументов, всегда имеется метод **commit**, а для выявления ошибок в порядке вызова используется **assert**. Видишь метод **commit** — сразу же узнаёшь шаблон и понимаешь, как создавать и использовать объект. Это, конечно, лучше, чем полное отсутствие каких бы то ни было соглашений, но далеко не лучший вариант.

В идеале мы вообще не должны полагаться на какие-то условные правила соглашения; компилятор сам должен следить за правильным использованием. Лучше не сводить к минимуму вероятность некорректного порядка вызова, а сделать его вообще невозможным.

Как сделать ошибки порядка вызовов невозможными

Один из способов — разделить обе фазы на разные объекты: сначала создавать объект, задающий параметры, а затем использовать эти параметры для отрисовки. Для пикантности добавим еще парочку дополнительных параметров; нарисуем сплошную сферу вместо стандартной каркасной и сделаем так, чтобы ее размер слегка «пульсировал» для заметности. Разделив фазы на отдельные объекты, получаем примерно следующее:

```
void markCharacterPosition(const Character * character)
{
    Params params;
    params.setXRay(0.5);
    params.setDrawStyle(DrawStyle::Solid);
    params.setPulse(true);

    Draw draw(params);
    draw.drawSphere(
        character->getPosition() + Vector(0.0, 0.0, 2.0),
        0.015,
        Color(Red));
}
```

В такой структуре порядок вызовов становится очевидным. Чтобы создать объект **Draw**, сначала нужен объект **Params**, поэтому естественно сначала инициализировать **Params**, а затем передавать его в **Draw**.

В C++ существует один идиоматический прием, позволяющий сделать код компактнее. Если возвращать ссылки на сам объект из функций set, можно

вызывать их цепочкой. Тем из вас, кто с неодобрением относится к подобным трюкам, лучше отвести взгляд от следующего фрагмента кода:

```
void markCharacterPosition(const Character * character)
{
    const Params params = Params()
                          .setXRay(0.5)
                          .setDrawStyle(DrawStyle::Solid)
                          .setPulse(true);

    Draw draw(params);
    draw.drawSphere(
        character->getPosition() + Vector(0.0, 0.0, 2.0),
        0.015,
        Color(Red));
}
```

Не привыкшим к этому приему код может показаться «не похожим на C++», что вовсе не достоинство. Если придерживаться «принципа наименьшего удивления» (в данном контексте оно подразумевает, что лучше всего подходит алгоритм, вызывающий меньше всего удивления), то синтаксически допустимый, но необычный прием — не лучший вариант.

Вместе с тем он предоставляет и одно серьезное преимущество. Так как все изменения объекта **Params** происходят внутри цепочки вызовов, можно объявить этот объект как **const**. Это значит, что компилятор не позволит изменить **Params** после его создания. Тем самым мы избавимся от одного из оставшихся вопросов, а именно: «Что произойдет, если изменить объект **Params** уже после того, как на его основе создан объект **Draw**?» Объявление объекта **Params** как **const** делает этот вопрос несущественным.

Тем не менее определять два объекта — это хлопотно. Для разработчика из Sucker Punch, пишущего код визуализации отладки, будет очевидно, что не следует вызывать функции set после функций draw... но было бы еще лучше, если бы это было не просто очевидно, а совсем невозможно. С учетом приведенного выше кода можно сделать следующее:

```
void markCharacterPosition(const Character * character)
{
    Draw draw = Params()
                .setXRay(0.5)
                .setDrawStyle(DrawStyle::Solid)
```

```
                .setPulse(true);

    draw.drawSphere(
        character->getPosition() + Vector(0.0, 0.0, 2.0),
        0.015,
        Color(Red));
}
```

Теперь странная «цепочка методов» выглядит логичнее! Мы вообще не раскрываем объект **Params** остальному коду. Он создается лишь на то время, чтобы создать объект **Draw**, а после не остается никакой его копии, к которой можно было бы обратиться. И это довольно элегантное решение — в таком виде мы спроектировали невозможность ошибки.

Как и все странные маленькие идиомы (принятые шаблоны кода), она работает лучше всего, если широко используется во всем проекте. Все участники проекта, встречая эту идиому в коде, узнают ее, она не кажется им странной, и тем самым не нарушается принцип наименьшего удивления. Если же в вашем коллективе такая идиома еще не используется, ее не стоит вводить для решения одной-единственной задачи конструирования объектов. Если бы я увидел такое решение на ревизии кода, то отклонил бы его из принципа, потому что мы в Sucker Punch не пользуемся цепочками методов. Но я легко могу представить себе другую версию Sucker Punch где-нибудь в параллельной Вселенной, для которой цепочка методов — стандартный способ решения подобных задач.

Шаблоны вместо цепочек методов

В реальности мы в Sucker Punch для реализации такого рода типобезопасных необязательных аргументов пользуемся другим приемом — шаблонами C++. Это не единственная задача, в которой у нас встречается множество параметров, из которых часто вызываются лишь немногие, и потому мы выработали на этот счет определенные соглашения. Код, который мы для объекта **Params** написали бы в действительности, выглядел бы примерно так:

```
void markCharacterPosition(const Character * character)
{
    Draw draw(XRay(0.5), DrawStyle::Solid, Pulse());

    draw.drawSphere(
        character->getPosition() + Vector(0.0, 0.0, 2.0),
```

```
        0.015,
        Color(Red));
}
```

Это не лучше и не хуже модели с цепочкой методов, это просто другое решение. Для нас оно лучше, потому что для нас это привычный прием, но для команды, где принято писать цепочки методов, оно показалось бы запутанным и странным. Так или иначе, любой прием, устраняющий ошибки использования, то есть не дающий программисту выстрелить себе в ногу, — это большой шаг вперед по сравнению с предположением о том, что пользователь сам во всем разберется.

Согласованное управление состоянием

В этом правиле уже рассматривались два распространенных примера — согласование аргументов и сложные конструкторы; было показано, как спроектировать их интерфейсы так, чтобы исключить ошибки во время использования. Далее рассмотрим третий пример ситуации, возникающей в играх Sucker Punch неоднократно: координация всего кода, управляющего состоянием персонажей в игре.

Допустим, нам нужно определить, должен ли персонаж реагировать на какое-то повреждение, например на попадание в него стрелы. В общем случае, если в персонажа попадает стрела, он должен как-то реагировать. Но не всегда! Если в игре запущена заскриптованная кат-сцена, в которой игрок подходит к NPC и заговаривает с ним, то лучше просто проигнорировать случайную стрелу, попавшую в игрового персонажа. Да, это может выглядеть глупо, но все же это лучше, чем нанесение урона игровому персонажу, когда им не управляют, — такое в гейм-дизайне считается тяжелым грехом. Кроме того, кат-сцены и без того довольно чувствительны к внешним воздействиям, и получение урона может нарушить все остальное в сцене. Лучше просто сделать так, чтобы стрела отскочила, не нанеся вреда.

Сложность здесь в том, что причин временной неуязвимости персонажа может быть множество, и это не только кат-сцены. Персонаж мог, например, выпить зелье неуязвимости. Его можно сделать временно неуязвимым для стрел, чтобы избежать проблем с анимацией. Иногда при тестировании новых атак мы делаем игрока неуязвимым для удобства, добавляя в меню отладки пункт неуязвимости игрового персонажа. В итоге у нас появляются десятки сценариев, в которых персонажи временно помечаются как неуязвимые.

Самое очевидное решение — сосредоточиться на неуязвимости персонажа. В любой момент он либо неуязвим, либо нет, так почему бы просто

не сделать это свойство доступным напрямую? Вроде бы решение простое и строится по модели push-pop:

```
struct Character
{
    void setInvulnerable(bool invulnerable);
    bool isInvulnerable() const;
};
```

Тогда сделать игрока неуязвимым во время кат-сцены можно так:

```
void playCelebrationCutScene()
{
    Character * player = getPlayer();
    player->setInvulnerable(true);
    playCutScene("where's chewie's medal.cut");
    player->setInvulnerable(false);
}
```

Такое решение работает, но только до тех пор, пока неуязвимость игрока меняется лишь в одном участке кода за один раз. Но мы, скорее всего, напишем похожий код и для зелья неуязвимости*:

```
void chugInvulnerabilityPotion()
{
    Character * player = getPlayer();
    player->setInvulnerable(true);
    sleepUntil(now() + 5.0);
    player->setInvulnerable(false);
}
```

Эти два участка кода могут легко наложиться один на другой, потому что сама их структура повышает вероятность ошибок. Если игрок выпьет зелье неуязвимости во время кат-сцены, то возникнут проблемы. Допустим, кат-сцена запускается, игрок становится неуязвимым посредством метода

* Эти примеры — адаптация аналогичного кода, написанного на нашем скриптовом языке, который имеет встроенную поддержку асинхронного программирования через корутины. Вызов **`sleepUntil`** не блокирует выполнение другого кода; то же самое относится и к **`playCutScene`** в предыдущем примере. Как будет показано чуть ниже, это может привести к осложнениям.

setInvulnerable(true), затем игрок выпивает зелье, и снова вызывается метод **setInvulnerable(true)**. Так как игрок уже неуязвим, то сначала ничего не меняется. Но через пять секунд действие зелья заканчивается и вызывается **setInvulnerable(false)**, пока кат-сцена все еще продолжается. А это плохо.

Если сделать обобщение на основе одного примера, то попытаться исправить проблему можно было бы так:

```
void playCelebrationCutScene()
{
    Character * player = getPlayer();
    bool wasInvulnerable = player->isInvulnerable();
    player->setInvulnerable(true);
    playCutScene("where's chewie's medal.cut");
    player->setInvulnerable(wasInvulnerable);
}

void chugInvulnerabilityPotion()
{
    Character * player = getPlayer();
    bool wasInvulnerable = player->isInvulnerable();
    player->setInvulnerable(true);
    sleepUntil(now() + 5.0);
    player->setInvulnerable(wasInvulnerable);
}
```

Здесь во избежание конфликта код пытается восстановить исходное состояние флага. В некотором смысле он работает, и Sucker Punch выпускала игры с подобными решениями, но с нарушением строгой вложенности, все, опять-таки, рушится. Например, в том случае, когда игрок выпивает зелье неуязвимости прямо перед началом кат-сцены и действие зелья заканчивается уже во время нее.

Как же сделать невозможными такие ошибки использования? Можно рассмотреть вариант с разделением различных участков кода, связанных с неуязвимостью. Если поддерживать разные флаги неуязвимости для каждого участка кода, для которого это имеет смысл, то они не пересекались бы друг с другом:

```
void playCelebrationCutScene()
{
    Character * player = getPlayer();
```

```
    player->setInvulnerable(InvulnerabilityReason::CutScene, true);
    playCutScene("it's anti-fur bias, that's what it is.cut");
    player->setInvulnerable(InvulnerabilityReason::CutScene, false);
}

void chugInvulnerabilityPotion()
{
    Character * player = getPlayer();
    player->setInvulnerable(InvulnerabilityReason::Potion, true);
    sleepUntil(now() + 5.0);
    player->setInvulnerable(InvulnerabilityReason::Potion, false);
}
```

Так вместо одного флага мы проверяем все флаги неуязвимости (**InvulnerabilityReason** — «причина неуязвимости»). Игрок считается неуязвимым, если установлен хотя бы один из них. Пока каждый отдельный флаг выставляется только одним участком кода, нам не нужно беспокоиться о том, что разные участки будут мешать друг другу.

Такое решение работает, но требует внимательности и дисциплины. Если кто-то из лени воспользуется одним и тем же **InvulnerabilityReason** в разных участках кода, все может пойти наперекосяк. И этого человека можно понять, потому что добавление нового значения в перечисление **InvulnerabilityReason** всякий раз при добавлении нового участка кода, подразумевающего неуязвимость, быстро утомляет.

Далее возникает мысль о том, чтобы устранить такие пересекающиеся случаи с помощью подсчета всех причин неуязвимости. Вместо одного флага мы подсчитываем, сколько участков кода хотят сделать персонажа неуязвимым. Если сделать персонажа неуязвимым хочет хотя бы один участок кода, то он таким и будет. В результате получается простая модель push-pop:

```
void playCelebrationCutScene()
{
    Character * player = getPlayer();
    player->pushInvulnerability();
    playCutScene("I'm getting my own ship.cut");
    player->popInvulnerability();
}

void chugInvulnerabilityPotion()
{
```

```
    Character * player = getPlayer();
    player->pushInvulnerability();
    sleepUntil(now() + 5.0);
    player->popInvulnerability();
}
```

Модель push-pop работает. Эту «идиому» легко понять и к ней легко привыкнуть. Она легко расширяется (то есть в код легко вставлять новые участки, требующие неуязвимости), и разные участки кода могут вызывать для счетчика неуязвимости push и pop независимо, не нарушая работу системы.

Да, при этом остаются лазейки для других небольших ошибок использования. Если в вашем коде по невнимательности будет пропущен вызов **popInvulnerability**, персонаж останется неуязвимым навсегда. И сделать такое легко — например, вы можете добавить ранний выход из функции, не понимая, что требуется очистка. Или в попытках исправления этой ситуации вы решите вызвать pop в раннем выходе и случайно вызовете его дважды, что приведет к еще более непредсказуемым последствиям*.

Лучше полностью исключить возможность таких ошибок. Самый простой способ — обернуть push-pop в пару конструктор–деструктор. Теперь сам компилятор становится нашим союзником:

```
void playCelebrationCutScene()
{
    Character * player = getPlayer();
    InvulnerableToken invulnerable(player);
    playCutScene("see you later, losers.cut");
}

void chugInvulnerabilityPotion()
{
    Character * player = getPlayer();
    InvulnerableToken invulnerable(player);

    sleepUntil(now() + 5.0);
}
```

И это довольно надежный способ. Здесь трудно сделать что-то неправильно. Да, напортачить еще можно — например, создать **InvulnerableHandle**

* Признаюсь, что я сам не раз допускал обе эти ошибки.

в месте, где он не будет уничтожен (например, он будет вложен в структуру, размещенную в куче); и все еще возможны ошибки. Впрочем, на практике выяснилось, что программисты обычно правильно управляют временем жизни объектов, и такое решение для надежного управления общим состоянием работает у нас очень хорошо.

Находить ошибки — хорошо, но еще лучше — сделать так, чтобы они не возникали

В этих примерах нам удалось устранить возможность большинства ошибок — обычно с помощью компилятора. Все, что может выявить компилятор, облегчает работу программиста. Мы пользовались несложными приемами. Подобными приемами мы в Sucker Punch пользуемся для решения различных игровых задач. Еще лучше, когда сама структура не допускает возможности совершения ошибки. Впрочем, Дуглас Адамс по этому поводу имеет свое мнение*:

Распространенная ошибка при попытке сделать что-то абсолютно надежным — недооценить изобретательность полных дураков.

И он прав. Идеального решения не существует, и полностью сделать невозможными все ошибки использования невозможно. Можно помешать пользователю «выстрелить себе в ногу» напрямую, но нельзя предусмотреть все возможные рикошеты. Наша цель — не абсолютно надежный дизайн, а такой дизайн, который облегчает правильное использование и затрудняет неправильное.

Даже несмотря на то что нельзя сделать дизайн полностью безошибочным, каждая предотвращенная глупость укрепляет систему в целом и делает ее более надежной. Поэтому старайтесь предотвращать потенциальные ошибки уже на этапе проектирования.

---

* Douglas Adams, Mostly Harmless (Del Rey, 1993).

Правило 8

Код, который не запускается, не работает

Любая крупная кодовая база, особенно существующая уже долгое время, содержит своеобразный «мертвый груз» — строки кода, функции или подсистемы, которые больше не используются. Конечно, их в свое время добавляли не просто так, и когда-то эти строки кода действительно вызывались. Но со временем ситуация меняется, и ранее вызываемый код становится ненужным. Вызовы прекращаются, и код превращается в «сиротский».

Иногда выявить таких «сирот» легко — как, например, функцию, которая нигде больше не вызывается. В достаточно мощных языках и инструментах программирования часто даже предусмотрены специальные предупреждения о таком мертвом грузе.

Но чаще всего «сиротский» код не настолько очевиден. Он может оставаться чисто механическим, как, например, виртуальный метод базового класса, который никогда не вызывается для конкретного производного класса. Статический анализ такое не выявит. Или это код для обработки какого-то особого пограничного случая или сценария, требующего специфической условной обработки. Со временем этот конкретный сценарий перестает встречаться, но код остается, просто не вызывается.

Чем тщательнее вы изучаете зрелую кодовую базу, тем больше находите подобных «сиротских» элементов: значения перечислений, которые определены, но не используются, или код для старой версии библиотеки, которой вы не пользуетесь уже несколько лет.

Такое развитие кода естественно и неизбежно. Кодовая база подобна реке, которая извивается по своей пойме, иногда меняя русло. При слишком резком изменении русла часть реки может оказаться отрезанной и превратиться в озеро — старицу, то есть участок старого русла, который по-прежнему походит на реку, но фактически превратился в изолированное озеро.

Рассмотрим упрощенный пример эволюции кода. Представьте код, отслеживающий всех персонажей в игре. В процессе разработки требования к отслеживанию персонажей меняются, и вместе с этими требованиями меняется и код. Мы рассмотрим четыре точки этого эволюционного пути.

Шаг 1. Простое начало

Начинается все просто. Для каждого персонажа игра создает объект, а этот объект предоставляет несколько простых методов для запроса, в основном о том, считает ли данный персонаж другого врагом или союзником:

```
struct Person
{
    Person(Faction faction, const Point & position);
    ~Person();

    bool isEnemy(const Person * otherPerson) const;

    void findNearbyEnemies(
        float maxDistance,
        vector<Person *> * enemies);
    void findAllies(
        vector<Person *> * enemies);

    Faction m_faction;
    Point m_point;

    static vector<Person *> s_persons;
    static bool s_needsSort;
};
```

При этом поддерживается список всех персонажей в игре:

```
Person::Person(Faction faction, const Point & point) :
    m_faction(faction),
    m_point(point)
{
    s_persons.push_back(this);
    s_needsSort = true;
}

Person::~Person()
{
    eraseByValue(&s_persons, this);
}
```

Каждому персонажу назначается «фракция» (faction), и персонажи из разных фракций считаются врагами:

```
bool Person::isEnemy(const Person * otherPerson) const
{
    return m_faction != otherPerson->m_faction;
}
```

Нередко бывает нужно найти ближайших врагов персонажа, поэтому предусмотрен соответствующий метод:

```
void Person::findNearbyEnemies(
    float maxDistance,
    vector<Person *> * enemies)
{
    for (Person * otherPerson : Person::s_persons)
    {
        float distance = getDistance(m_point, otherPerson->m_point);
        if (distance >= maxDistance)
            continue;
        if (!isEnemy(otherPerson))
            continue;
        enemies->push_back(otherPerson);
    }
}
```

Для определения союзников персонажа также используется отдельный метод. При этом применяется небольшой трюк: персонажи сортируются по фракциям, и таким образом все союзники оказываются рядом. Благодаря этому можно завершить поиск раньше — как только закончится список союзников:

```
bool compareFaction(Person * person, Person * otherPerson)
{
    return person->m_faction < otherPerson->m_faction;
}

void Person::findAllies(vector<Person *> * allies)
{
    if (s_needsSort)
    {
```

```
        s_needsSort = false;
        sort(s_persons.begin(), s_persons.end(), compareFaction);
    }

    int index = 0;

    for (; index < s_persons.size(); ++index)
    {
        if (!isEnemy(s_persons[index]))
            break;
    }

    for (; index < s_persons.size(); ++index)
    {
        Person * otherPerson = s_persons[index];
        if (isEnemy(otherPerson))
            break;

        if (otherPerson != this)
            allies->push_back(otherPerson);
    }
}
```

На этом можно остановиться, так как код работает. Простая модель враждебности, основанная на фракциях, вполне жизнеспособна. Кстати, на таких простых принципах можно построить довольно продвинутые игровые системы: если добавить функцию, определяющую враждебность каждой фракции по отношению к другим, получится та же модель враждебности, что использовалась в серии игр *inFamous* студии Sucker Punch.

Но несмотря на то что модель враждебности кажется достаточной, вскоре оказывается, что двух стартовых функций запроса, **`findNearbyEnemies`** и **`findAllies`**, уже недостаточно. Обе полезны, но появляются новые задачи, которых они не решают. Допустим, нужно найти всех союзников, которых игрок видит по прямой линии обзора. Это можно сделать, сначала получив список союзников игрока, а затем отфильтровав тех, кого игрок не видит:

```
vector<Person *> allies;
player->findAllies(&allies);

vector<Person *> visibleAllies;
```

```
for (Person * person : allies)
{
    if (isClearLineOfSight(player, person))
        visibleAllies.push_back(person);
}
```

Можно также для таких случаев добавить в **Person** дополнительные методы — например, **findVisibleAllies**. Тогда не потребуется промежуточный список **allies**, так как можно было бы сразу получить **visibleAllies**. Но по мере увеличения количества методов поиска такой подход быстро становится неудобным. Добавка в **Person** с десяток еще более специализированных функций поиска вряд ли упростит код, особенно если большинство из них вызывается только из одного места.

Шаг 2. Обобщение шаблона

Как только накопится достаточное количество* случаев такого шаблона («найти персонажей, соответствующих определенному набору условий»), можно уверенно его обобщать. Для этого в класс **Person** добавляется шаблонная функция:

```
template <class COND>
void Person::findPersons(
    COND condition,
    vector<Person *> * persons)
{
    for (Person * person : s_persons)
    {
        if (condition(person))
            persons->push_back(person);
    }
}
```

Такой подход позволяет избежать лишних выделений памяти и сохранить при этом читаемость кода**:

```
struct IsVisibleAlly
```

---

* «Достаточное количество» это три. См. Правило 4 «Обобщайте после трех примеров».

** Разумеется, здесь можно использовать и лямбда-функцию, если, конечно, ваша команда уже из «Поколения Лямбда».

```
{
    IsVisibleAlly(Person * person) :
        m_person(person)
    { ; }

    bool operator () (Person * otherPerson) const
    {
        return otherPerson != m_person &&
               isClearLineOfSight(m_person, otherPerson) &&
               !m_person->isEnemy(otherPerson);
    }

    Person * m_person;
};

player->findPersons(IsVisibleAlly(player), &allies);
```

Когда шаблон отлажен и работает, можно пройтись по всему проекту и заменить все вызовы **Person::findNearbyEnemies** и **Person::findAllies**. В случаях, где ранее использовался многоступенчатый фильтр, как в примере с линией обзора, код переводится на новый шаблон **findPersons**.

В процессе перехода обнаруживается, что во всех местах, где вызывалась **findAllies**, выполнялась дополнительная фильтрация. Поэтому все эти вызовы заменяются на **findPersons**. И это хорошо — код становится проще, быстрее и понятнее. Вы довольны результатом этого шага: кодовая база читается легче, и нет больше многоступенчатых фильтров. В проекте остаются лишь несколько вызовов **findNearbyEnemies**, множество вызовов **findPersons** и ни одного **findAllies**.

Однако со временем выясняется, что простая модель враждебности перестает справляться со своей задачей. Возникает идея реализации маскировки, чтобы, например, игрок мог надеть форму охранника и спокойно пройти через охраняемую территорию, не вызывая враждебности.

Шаг 3. Добавление маскировки

Система маскировки выявляет слабые стороны используемой до этого модели враждебности. Простого деления всех персонажей на союзников и врагов уже недостаточно. Нужно ввести некое промежуточное состояние для отражения неоднозначного отношения друг к другу многих персонажей. Так, для

охранника другой охранник — союзник, но случайный прохожий находится где-то посередине между союзником и врагом.

Такую более проработанную модель враждебности несложно обобщить через виртуальный интерфейс:

```
enum class Hostility
{
    Friendly,
    Neutral,
    Hostile
};

struct Disguise
{
    virtual Hostility getHostility(const Person * otherPerson) const =
0;
};
```

В **Person** добавляется новый метод, позволяющий задавать текущую маскировку персонажа. **nullptr** соответствует отсутствию маскировки:

```
void Person::setDisguise(Disguise * disguise)
{
    m_disguise = disguise;
}
```

Очевидно, что метод **isEnemy** придется немного изменить:

```
bool Person::isEnemy(const Person * otherPerson) const
{
    if (otherPerson == this)
        return false;

    if (m_disguise)
    {
        switch (m_disguise->getHostility(otherPerson))
        {
            case Hostility::Friendly:
                return false;
```

```
            case Hostility::Hostile:
                return true;

            case Hostility::Neutral:
                break;
        }
    }

    return m_faction != otherPerson->m_faction;
}
```

И... вот, собственно, и все. Весь остальной код по-прежнему работает корректно, новых багов не появляется, и маскировка функционирует как задумано.

Но проблема скрывается под поверхностью — старый метод **`Person::findAllies`** больше не работает. И мы даже не знаем об этом, потому что он больше нигде не используется. Добавление маскировки нарушило одно тонкое допущение для **`findAllies`**. Согласно самой задумке метода, если отсортировать список персонажей по фракции, то все союзники должны оказаться в массиве рядом друг с другом. С маскировкой часто тоже получается так... но далеко не всегда.

Рассчитывать на ревизию кода в такой ситуации бесполезно. Ревизия отлично выявляет проблемы в измененном коде, потому что именно ему и уделяется основное внимание. Но она почти не выявляет проблемы в коде, который не менялся, потому что рецензенты обычно просто пролистывают его.

Баг этот особенно неприятен, потому что он необязательно появляется сразу же после реализации маскировки. Пока союзники в списке персонажей непрерывно следуют друг за другом, все работает. И даже когда метод **`findAllies`** не срабатывает, как задумано, он все равно возвращает частичный список союзников, поэтому ошибку можно долго не замечать.

Возможно даже, что этот баг так и останется скрытым навсегда! Я до сих пор нахожу ошибки в коде, который писал лет двадцать пять назад, и уверен, что это далеко не последние баги в том коде. Старый код всегда таит в себе скрытые ошибки, затаившиеся в ожидании подходящего момента для проявления. Но может случиться и так, что условия для их проявления не возникнут никогда... так стоит ли считать это проблемой?

Шаг 4. Старые проблемы возвращаются

В данном случае — да, потому что спустя несколько месяцев происходит очередное изменение. Допустим, кто-то пишет отладочный код, выявляю-

щий всех союзников игрока. Для этой задачи идеальным кажется метод **Person::findAllies**, и его вызывают самым очевидным образом:

```
vector<Person *> allies;
player->findAllies(&allies);
for (Person * ally : allies)
{
    cout << ally->getName() << "\n";
}
```

Код этот выглядит совершенно безобидным! Если не используется маскировка, все работает идеально. И даже для маскировки этот код ломается не очевидным образом. При своей некорректной работе он все равно возвращает частичный список союзников... да, в нем содержатся не все союзники, но заметить пропавшего союзника сложнее, чем появившегося в списке союзников врага. Так что, скорее всего, этот код пройдет и через ревизию.

Но, допустим, за все эти месяцы, пока **findAllies** не вызывали, был написан код, полагающийся на стабильный порядок результатов, возвращаемых методом **findPersons**. И это логично, ведь порядок действительно был стабильным! Новый код выбирал нескольких ближайших союзников, они шли за игроком друг за другом по порядку, и все работало отлично. Но теперь по какой-то странной причине спутники изредка словно впадают в панику и перемещаются на неожиданные позиции. И это уже неприемлемо для релиза.

Проблема, конечно, в этом невинно выглядящем вызове **findAllies**. При добавлении новых персонажей **findAllies** повторно сортирует список персонажей, непредсказуемо нарушая порядок результатов любого вызова **findPersons**. Для выявления возможных ошибок придется очень тщательно проверять все места, где вызывается **findPersons**, что доставит немало хлопот.

Поиски виновника

Так где же все пошло не так?

В поисках источника проблемы легко указать на Шаг 3, ведь именно там мы допустили ошибку. Мы добавили систему маскировки, но не обновили соответствующим образом **findAllies**. И это понятная ошибка: написанный нами код работал отлично, и ничего явно не ломалось. Проблему в **findAllies** не выявили бы никакие тесты продукции, ведь этот метод нигде

не вызывался. Даже в ходе ревизии кода **findAllies** можно было не заметить ошибку, уж слишком неочевидно лежащее в его основе допущение*.

Можно также утверждать, что ошибка возникла на Шаге 4, когда мы написали новый код, «воскресив» тем самым заброшенный **findAllies**. В конце концов, текущее Правило так и называется: «Код, который не выполняется, не работает». Если известно, что **findAllies** не вызывается, следовало бы сразу предположить, что он уже нерабочий.

Но обычно программисты не приходят к такому умозаключению — по крайней мере, работая со «здоровой» кодовой базой. При написании нового кода или исправлении старого приходится опираться на предположение о том, что остальная часть проекта в целом работает. Видя готовый функционал, ожидаешь, что он действует так, как и было задумано. Иначе просто невозможно двигаться вперед.

Настоящая ошибка в этом примере была допущена на Шаге 2. Проблема возникла в тот момент, когда мы сделали «сиротой» метод **findAllies**. Как только его перестали вызывать, он перестал работать.

Но не абсурден ли такой вывод? В момент, когда функция стала «сиротой», она ведь еще полностью выполняла свою задачу. Зачем отказываться от рабочего кода, на который были потрачены усилия? Ведь, насколько кажется, он перестал работать только на Шаге 3, верно?

Возможно. Ситуация похожа на ситуацию с котом Шрёдингера: после Шага 2 код перестал выполняться, и мы можем не знать, работает ли он теперь или нет. В нашем примере все выглядит очевидно — осиротевшая функция перестала работать на Шаге 3, а обнаружилось это (с большими потерями) на Шаге 4. Но в реальной жизни между этими шагами могли пройти десятки шагов, которые я не описал, потому что они не влияли напрямую на наш пример. При этом нарушить работу функции могло любое изменение, и мы об этом не узнали бы.

Проще считать, что едва функция перестает вызываться, она сразу же перестает работать. Со временем такое предположение наверняка станет правдой, просто мы не знаем, когда именно.

Если отталкиваться от такого предположения, то ошибка на Шаге 2 заключалась не в том, что мы сделали метод **findAllies** «сиротой», а в том, что мы не удалили его тогда же. Он перестал работать сразу же, как только мы перестали его вызывать. Тем самым мы на Шаге 2 создали условия для неизбежного проявления Шагов 3 и 4, ведь неиспользуемый код рано или поздно сломается, и его рано или поздно кто-то снова вызовет. Гораздо правильнее было сразу удалить осиротевший код.

---

* Если вы уже мысленно кричите: «А где же ваши модульные тесты?!», не волнуйтесь, до этого мы скоро дойдем.

Пределы тестирования

Но, конечно же, это не стандартное решение проблемы. Если вы работаете в команде, где большое внимание уделяется тестированию, то вы могли задаться вопросом о том, почему не выявили проблему модульные тесты («юнит-тесты»). При полном наборе модульных тестов метод **findAllies** не стал бы по-настоящему осиротевшим. Его продолжали бы вызывать тесты, и нам не пришлось бы сразу же считать его нерабочим.

Дело в том, что модульные тесты несовершенны. Некоторые команды не пишут тесты для каждого фрагмента кода по веским причинам. Во-первых, для разных типов кода эффективность тестов не одинакова. Простые функции без состояния тестировать проще, чем сложные функции с состоянием. Если речь идет о стандартных библиотеках C, то проверять **strcpy** гораздо проще, чем **malloc***. Как только в функциях появляется состояние, модульные тесты с трудом воспроизводят все реальные сценарии использования, и некоторые случаи неизбежно ускользают от внимания.

Тесты для метода **findAllies** были написаны одновременно с его созданием, задолго до того, как мы задумались о реализации идеи маскировки для класса **Person**. Поэтому эти тесты не учитывают маскировку. При отсутствии маскировки функция работает корректно, и проблем не возникает. Допустим, добавивший маскировку сотрудник поймет, что тесты **findAllies** нужно обновить, и добавит проверки с типами маскировок, которые нарушают работу функции и меняют порядок элементов… но это уже серьезное требование и серьезное допущение.

Кроме этого, на поддержку модульных тестов в актуальном состоянии требуются ресурсы. Ради чего нам поддерживать актуальное состояние тестов **findAllies**? Только чтобы убедиться, что функция, которую никто не вызывает, продолжает работать?

Но, скажете вы… разве я не только что утверждал, что Шаг 2 (когда **findAllies** «осиротел») сделал неизбежными Шаг 3 (**findAllies** «сломался») и Шаг 4 (сломанным кодом воспользовались)?

В действительности это не так. Суть вот в чем: если бы мы удалили осиротевший код на Шаге 2, нам вообще не нужно было ни о чем беспокоиться. Никакие баги не могут появиться в коде, которого просто нет. И Шага 3 тоже

---

* Стандартная функция **strcpy** копирует C-строку (нулевой терминатор) в новое место. Она простая и полностью лишена состояния (ее результат зависит только от аргументов). Стандартная функция **malloc** — это универсальный аллокатор памяти в C. Она управляет всеми (более или менее) динамически выделяемыми объектами в коде. Это очень сложная функция, полностью зависящая от состояния.

не будет, потому что нечему ломаться. Удаленную функцию никто не сможет вызвать, а потому не будет и Шага 4.

Вместо этого программисты будут пользоваться идеально работающим методом **findPersons**:

```
struct IsAlly
{
    IsAlly(Person * person) :
        m_person(person)
       { ; }

    bool operator () (Person * otherPerson) const
    {
        return otherPerson != m_person &&
               !m_person->isEnemy(otherPerson);
    }

    Person * m_person;
};

vector<Person *> allies;
player->findPersons(IsAlly(player), &allies);
```

Нужно радоваться тому, что кусок кода осиротел и его можно безопасно удалить*. Я говорю серьезно — это один из самых приятных моментов в работе: количество кода в проекте сокращается, и все становится проще без сокращения функциональности. Все работает быстро, код легко отслеживается, и всем от этого лучше.

---

* Если это вам поможет, напомните себе, что удаленный код всегда можно восстановить из системы контроля версий. Вы, конечно, никогда его не восстановите, но сама мысль о возможности восстановления подтолкнет вас к верным действиям.

Правило 9

Пишите код, который можно сворачивать

Мне часто приходится просматривать код в попытках понять, что он делает. Это может быть код, над отладкой которого я работаю, который я собираюсь вызывать, или код, который вызывает мой собственный. И часто то, что код должен делать по задумке, отличается от того, что он делает на самом деле, и от этого процесс становится по-настоящему увлекательным.

В идеале чтение кода похоже на чтение на любом другом языке. При этом читающий двигается сверху вниз, следуя за так называемыми поворотами сюжета, и в конце полностью понимает, что делает код и зачем он нужен.

На самом простом уровне чтение кода равнозначно чтению одного-единственного слова:

```
int sum = 0;
```

Или:

```
sum = sum + 1;
```

Эти примеры не требуют логических рассуждений, чтобы понять код, достаточно взглянуть на него. То же самое можно сделать и с более крупными фрагментами, если они аккуратно следуют какому-то общему шаблону:

```
Color Flower::getColor() const
{
    return m_color;
}
```

Можно даже окинуть взглядом целый цикл:

```
int sum = 0;
for (int value : values)
```

```
{
    sum += value;
}
```

Хотя это уже ближе к пределу. По мере увеличения фрагментов кода охватывать их взглядом становится труднее, а если вы, как и я, старый циничный программист, то еще труднее доверять своей способности «читать на лету», ведь слишком часто бывало так, что смотришь на код и думаешь, будто понял, а потом оказывается, что это совсем не так.

Когда код становится слишком большим для визуального восприятия, его начинают анализировать. Подумайте, что делает ваш мозг, когда вы смотрите, например, на такой фрагмент:

```
vector<bool> flags(100, false);  ❶
vector<int> results; ❷

for (int value = 2; value < flags.size(); ++value) ❸
{
    if (flags[value]) ❹
        continue; ❹

    results.push_back(value); ❺

    for (int multiple = value; ❻
         multiple < flags.size(); ❻
         multiple += value) ❻
    {
        flags[multiple] = true; ❼
    }
}
```

Вы почти наверняка не прочли код мгновенно, то есть не взглянули на него и не подумали сразу: «А, ну это решето Эратосфена». Вместо этого вы читали код сверху вниз, по строке или две за раз, постепенно выстраивая логику, разбираясь, что делает каждая строка и как она соотносится с предыдущими.

Возможно, ваш мыслительный процесс выглядел примерно так:

❶ **flags** — это вектор из 100 элементов **false**.

❷ Похоже, мы будем собирать какие-то результаты в массив **results**.

❸ Проходим циклом по массиву **flags**, начиная с 2 — странно, почему не с 0 или 1.

❹ Пропускаем текущий элемент, если для него установлен **flags[value]**. Тоже пока неясно.

❺ Добавляем непропущенные значения в **results**. Наверное, это и есть результат.

❻ Второй цикл, на этот раз по кратным текущему значению.

❼ Ага, помечаем все кратные, значит, это решето Эра… Эратосфена! Поэтому и начали с 2, а не с нуля или единицы. Теперь ясно: вектор **results** в итоге содержит список простых чисел.

Такой процесс требует определенного мысленного жонглирования — вы видите нечто непонятное и откладываете его в сторону, пока не поймете, как оно сочетается с дальнейшими элементами, — подобно жонглеру, подбрасывающему мяч в воздух, чтобы поймать его позднее. В данном случае вы «жонглировали» двумя загадками — почему цикл начинается с 2, и зачем пропускаются помеченные значения. Внутренний цикл по кратным разрешил обе — вы поймали мячи и поняли код.

Жонглеры могут удерживать в воздухе лишь ограниченное число мячей, в моем случае три. Количество «мячей», которыми можно жонглировать мысленно, больше, но тоже ограничено. Отложить на потом можно на удивление небольшое количество мыслей. Если попытаться удерживать слишком много, часть их начнет теряться случайным образом.

Так происходит, потому что процесс «мысленного жонглирования», или, говоря более научным языком, когнитивная нагрузка, — это на самом деле просто работа кратковременной памяти*. Упрощая, можно сказать, что есть два типа памяти: долговременная, то есть то, что мы помним надолго, и кратковременная, хранящая то, что нам нужно прямо сейчас. Запоминая список покупок, вы задействуете кратковременную память. Если в списке два-три пункта, их можно просто удержать в голове. Если пунктов более десятка, придется их записать.

Так происходит из-за ограниченности объема кратковременной памяти. Возможно, вы слышали о том, что человек способен удерживать в кратковременной памяти около семи (плюс-минус два) элементов. Если попытаться добавить в нее больше, новые мысли вытесняют старые. И это верно не только для списка покупок, то же самое происходит и при чтении кода.

---

* Или рабочей памяти, если вы приняли сторону тех специалистов, кто считает это более корректным определением. Лично мне больше нравится «рабочая память», но термин *кратковременная память* встречается чаще, поэтому здесь я использую его.

Для большинства из нас это количество элементов — семь плюс-минус два — представляет жесткий предел*. Жонглировать тремя мыслями легко, и при чтении кода не нужно прикладывать никаких усилий, чтобы удерживать в голове пару загадочных моментов. Но если таких загадок с дюжину — пиши пропало. Если выражаться программистскими терминами, твой кеш переполняется, и ты теряешь часть контекста, который пытаешься удержать.

Как выглядит провал

Каждому программисту знакомо чувство, когда ты пытаешься понять работу сложного кода и натыкаешься на совершенно непонятный фрагмент. Чтобы разобраться в нем, ты начинаешь искать, откуда вызывается эта функция, где определена эта структура и есть ли хотя бы какой-нибудь комментарий для контекста. Каждая новая находка тянет за собой очередную загадку. К тому моменту, когда ты доходишь до конца фрагмента, ты уже забываешь, с чего начинал. Мысли вылетают у тебя из головы с той же скоростью, с которой возникают. Ужасное ощущение!

Роль кратковременной памяти

Хороший код не должен порождать описанные выше чувства. Код пишут только однажды, но читают много раз, поэтому если вы хотите научиться писать хороший код, подумайте о тех, кто будет его читать. Не заставляйте их жонглировать многими мыслями одновременно.

Если при чтении кода человеку приходится держать в голове больше семи (плюс-минус два) «мячей», он неизбежно начнет их ронять. Под мячами здесь подразумеваются отдельные удерживаемые в памяти элементы. Это не только загадки, требующие ответов, но и сведения о различных фрагментах кода и связи между ними, с помощью которых человек пытается найти ответы. При превышении определенного предела этих «мячей» (фактов, догадок, связей) читатель неизбежно начинает что-то терять.

Поэтому код бывает так трудно понять. Если читатель забудет какой-то важный факт, нужный для решения загадки, та так и останется неразгаданной. Причем заранее нельзя сказать, какой именно элемент выпадет из памяти, а значит, может пропасть как раз та деталь, без которой развалится вся логика.

Посчитаем количество мыслей, которые приходится удерживать одновременно при чтении следующего кода:

---

* Опытным путем я выяснил, что на этот предел не влияет кофе.

```
vector<bool> flags(100, false); ❶
vector<int> results; ❷

for (int value = 2; value < flags.size(); ++value) ❸
{
    if (flags[value]) ❹
        continue; ❹

    results.push_back(value); ❺

    for (int multiple = value; ❻
         multiple < flags.size(); ❻
         multiple += value) ❻
    {
        flags[multiple] = true; ❼
    }
}
```

❶ **flags** — это вектор из 100 булевых значений, изначально все равны **false** (+1). Мыслей = 1.

❷ **results** выглядит как место, в которое будет записываться результат цикла (+1). Мыслей = 2.

❸ Цикл начинается с индекса **value = 2** (+1). Мыслей = 3.

❹ Некоторые значения пропускаются, но пока неясно почему (+1). Мыслей = 4.

❺ Ага, значения сохраняются в **results**. Гипотеза подтверждается (+0). Мыслей = 4.

❻ Видим еще один цикл, теперь по кратным **value** (+1). Мыслей = 5.

❼ Теперь все сводится к одной идее: «вектор **results** содержит простые числа». Мыслей = 1.

Даже при довольно строгом подсчете количество одновременно удерживаемых в голове мыслей остается в безопасных пределах, и память читателя не перегружается. Причем «счетчик мыслей» не только увеличивается, но и может уменьшаться. Например, при выходе переменной из области видимости о ней можно забыть. Или, что важнее, когда, например, целая группа мыслей «сворачивается», то есть сводится к одной, подводящей к ответу.

В этом примере мысли сворачиваются, когда вы понимаете, что код генерирует список простых чисел. До этого вы жонглировали множеством

деталей, но в какой-то момент увидели, как все складывается воедино. После этого детали уже не требуют внимания, и в памяти удерживается только окончательный вывод. Все отдельные элементы объединяются в одну мысль.

Хороший код облегчает этот процесс. Он «сворачиваемый», то есть он не переполняет кратковременную память и подает идеи небольшими, логически связанными порциями. Каждая такая порция легко укладывается в голове у читателя, где объединяется с другими и порождает логический вывод.

Писать код таким образом помогают некоторые специальные приемы. Вот более длинный пример:

```
void factorValue(
    int unfactoredValue,
    vector<int>* factors)
{
    // Очистка флагов, отмечающих кратные простым числам

    vector<bool> isMultiple;

    for (int value = 0; value < 100; ++value)
        isMultiple.push_back(false);

    // Поиск простых чисел (с пропуском кратных простым)

    vector<int> primes;

    for (int value = 2; value < isMultiple.size(); ++value)
    {
        if (isMultiple[value])
            continue;

        primes.push_back(value);

        for (int multiple = value;
             multiple < isMultiple.size();
             multiple += value)
        {
            isMultiple[multiple] = true;
        }
    }
```

```
    // Поиск простых множителей заданного числа

    int remainder = unfactoredValue;

    for (int prime : primes)
    {
        while (remainder % prime == 0)
        {
            factors->push_back(prime);
            remainder /= prime;
        }
    }
}
```

В средней части функции используется та же логика, что и в предыдущем примере, но этот вариант куда проще для восприятия*. Причина в том, что он лучше сворачивается.

Огромную роль при этом играет хороший подбор имен. Названия **primes** («простые числа») и **isMultiple** («является множителем/кратным») заранее помогают читателю «сжать» смысл цикла: понятно, что в итоге массив содержит простые числа. Если бы массив и в первом примере назывался **primes**, вы, вероятно, распознали бы решето Эратосфена гораздо раньше. Вот она — сила удачного имени.

Кроме того, имя **primes** удобно само по себе и отлично передает идею массива с простыми числами. Если бы переменная называлась как-нибудь вроде **xx**, то одну из ценных «ячеек памяти» пришлось бы тратить на то, чтобы удерживать в голове связь между **xx** и массивом простых чисел. Помнить же о том, что **primes** содержит простые числа, не составляет ни малейшего труда. В худшем варианте нагрузка на память увеличивается, приближая количество элементов не к «семи минус два», а к «семи плюс два»; в лучшем варианте в кратковременной памяти вообще не требуется никакого места.

«Сворачивать» детали помогают и комментарии: они сразу объясняют, что делает каждый участок кода. У них есть и вторая функция: они размечают эти участки. Между каждой парой комментариев размещается небольшой кусочек кода, достаточно компактный для размещения в кратковременной памяти и свертывания в одну мысль. Комментарий в начале блока говорит,

---

* Кстати, это не самый эффективный способ разложения числа на множители, это всего лишь пример. В Правиле 19 показан более эффективный вариант.

во что именно свернется этот кусок кода. А чтение кода лишь подтверждает эту мысль.

Вот в чем сила абстракции, позволяющей понимать сложные концепции. Конечно, человек может удерживать в кратковременной памяти только около семи новых фактов, но их можно объединять в более крупные понятия и создавать новые уровни абстракции на основе уже этих понятий. Вместо того чтобы запоминать все детали, вы запоминаете абстракцию, а простая абстракция занимает всего один «слот», или ячейку, кратковременной памяти.

Где провести грань

Читаемость кода повышается и благодаря абстрагированию с помощью разделения функций. В данном случае можно переделать три снабженных комментариями секции **`getFactors`** в отдельные функции:

```
void clearFlags(
    int count,
    vector<bool> * flags)
{
    flags->clear();
    for (int value = 0; value < count; ++value)
        flags->push_back(false);
}

void getPrimes(
    vector<bool> & isMultiple,
    vector<int> * primes)
{
    for (int value = 2; value < isMultiple.size(); ++value)
    {
        if (isMultiple[value])
            continue;

        primes->push_back(value);

        for (int multiple = value;
             multiple < isMultiple.size();
             multiple += value)
        {
            isMultiple[multiple] = true;
```

```
        }
    }
}

void getFactors(
    int unfactoredValue,
    const vector<int> & primes,
    vector<int> * factors)
{
    int remainder = unfactoredValue;

    for (int prime : primes)
    {
        while (remainder % prime == 0)
        {
            factors->push_back(prime);
            remainder /= prime;
        }
    }
}
```

Перепишем **factorValue** с использованием этих трех функций:

```
void factorValue(
    int unfactoredValue,
    vector<int> * factors)
{
    vector<bool> isMultiple;
    clearFlags(100, &isMultiple);

    vector<int> primes;
    getPrimes(isMultiple, &primes);

    getFactors(unfactoredValue, primes, factors);
}
```

Стало ли проще читать код? И да и нет! Функции задают четкие концепции, и как только поймешь, что именно они делают, имена этих функций помогают зафиксировать в голове соответствующие концепции.

Но вместо того чтобы просто идти сверху вниз и читать линейную последовательность, приходится прыгать от функции к функции. Если начать с **factorValue**, первым нам встречается вызов функции **clearFlags**. Чтобы понять, что она делает, нужно найти эту функцию и прочитать ее. Во время чтения **clearFlags** приходится держать в голове место, откуда мы пришли (**factorValue**), и соответствия между переменными и аргументами, а потом нужно собрать это воедино при выходе из **clearFlags**.

Из-за этого нужно отслеживать больше деталей, и сворачивать концепции становится сложнее. Контекстные сведения тоже отнимают «ячейки» памяти, которых всего семь (плюс-минус два). Перегрузить кратковременную память может даже сама информация о месте, в котором ты находишься в цепочке сложных вызовов.

Согласно распространенному среди программистов мнению, абстракция полезна всегда, то есть всё, что можно вынести в отдельную функцию, следует выносить в отдельную функцию. Чем больше функций, тем лучше. Поскольку абстракция — это инструмент, с помощью которого мы понимаем сложные вещи, то кажется вполне логичным абстрагировать всё, что можно абстрагировать.

Цена абстракции

Было бы глупо отрицать, что абстракция имеет свою цену. Цена выделения логики в отдельные функции может перевесить пользу от такого выделения. Вот пример:

```
int sum = 0;
for (int value : values)
{
    sum += value;
}
```

Это очень простой для понимания код, если уже известно, что **values** — это вектор целых чисел. Код легко сворачивается в одно понятие: пройтись по значениям, сложить всё и получить сумму.

Но иногда можно встретить и такое:

```
int sum = reduce(values, 0, add);
```

Хм. Запись, безусловно, лаконична. По именам **sum** и **add** можно догадаться, что здесь вычисляется некая сумма, но это всего лишь догадка. Чтобы убедиться в этом, нужно для начала разобраться в коде.

Прежде всего, неясно, что именно делает функция **reduce** (по крайней мере, это выглядит как функция), что такое **add** и зачем передается аргумент **0**. Поиск в коде по слову **reduce** выдает массу совпадений, но вот, пожалуй, нужное:

```
template <class T, class D, class F>
D reduce(T & t, D init, F func)
{
    return reduce(t.begin(), t.end(), init, func);
}
```

Так, уже лучше. Похоже, первый аргумент **reduce** — это контейнер, ведь **begin** и **end** — стандартные элементы итерации в C++. Нужно найти версию **reduce** с четырьмя аргументами:

```
template <class T, class D, class F>
D reduce(T begin, T end, D init, F func)
{
    D accum = init;
    for (auto iter = begin; iter != end; ++iter)
    {
        accum = func(accum, *iter);
    }
    return accum;
}
```

Постепенно все проясняется. Функция **reduce** проходит по коллекции, последовательно применяя некую функцию (или объект, ведущий себя как функция) к каждому элементу и накопленному значению. Поднявшись на уровень выше, можно предположить, что **add** просто складывает два значения, и действительно:

```
int add(int a, int b)
{
    return a + b;
}
```

Теперь все складывается. Данный вызов **reduce** вычисляет сумму элементов массива, точно так же, как и простой цикл, с которого мы начали... Но тот простой цикл гораздо понятнее и легче для восприятия. Он «сворачивается» мгновенно, а абстрактная версия требует усилий — понять, что же здесь происходит, получается далеко не сразу. Дополнительные уровни абстракции не помогают, а только затемняют суть. Чтобы разобраться в коде, нам пришлось найти и понять четыре разных фрагмента кода, а для этого приходилось задействовать кратковременную память.

Используйте абстракцию для упрощения понимания

На мой взгляд, отсюда можно вывести хорошее практическое правило относительно того, стоит ли для решения конкретной задачи выделять логику в отдельную функцию или применять обобщенную абстракцию. Правило простое: «Сделает ли это код проще и понятнее?» Будет ли код легче «сворачиваться» при чтении? Если да — выделяйте функцию или используйте абстракцию. Если нет — усложнять не стоит.

Роль долговременной памяти

Пока что все звучит немного мрачно. Ограничение кратковременной памяти в семь (плюс-минус два) мыслей выглядит весьма жестоким. Похоже, что с таким «концептуальным бюджетом» невозможно создать что-нибудь хоть сколько-нибудь по-настоящему серьезное, даже если стараться сворачивать мысли в абстракции при любой возможности.

Но ведь это не вся история! Мы точно знаем, что имеем представление о гораздо большем количестве вещей, чем семь (плюс-минус два) элементов, например, когда речь идет о нашем движке.

Для класса главного персонажа в Sucker Punch определены десятки и десятки методов*, и я знаю, что делает каждый из них. Но почему?

На самом деле все просто. Сведения о наших игровых технологиях хранятся у меня в долговременной памяти, а у нее нет фиксированного лимита. Я не сомневаюсь, что каждый из вас может быстро вспомнить огромное количество подробностей какого-то своего проекта — концепции, факты, имена, историю разработки, ответственных за ту или иную часть, забавные

* Вообще-то их десятки десятков. То есть методов очень много. Нам и вправду нужно как-то получше организовать их.

истории о багах, которые приходилось исправлять в той или иной функции. И все это хранится в вашей долговременной памяти.

Кратковременная память используется, чтобы осознать происходящее, — это рабочее хранилище для рассуждений, где содержатся куски информации, связь между которыми вы пытаетесь выявить. Как только куски складываются в целое, как только рождаются какие-то концепции, свернутые в абстракции, результат может переместиться в долговременную память. Именно в ней живут все детали вашего проекта, и вам не нужно анализировать один и тот же фрагмент кода заново, вы просто вспоминаете сделанные ранее выводы.

Отсюда следует, что, несмотря на очевидное сходство, есть большая разница между этим кодом:

```
sort(
    values.begin(),
    values.end(),
    [](float a, float b) { return a < b; });
```

и этим:

```
processVector(
    values.begin(),
    values.end(),
    [](float a, float b) { return a < b; });
```

Я знаю, что делает **sort** («сортировка»). Я понимаю абстракцию, она хранится в моей долговременной памяти. Я могу почти мгновенно «прочитать» ее, и единственное, на что нужно обратить внимание, так это на функцию сравнения для понимания порядка сортировки. В результате чтение **sort** практически не занимает «ячейку» кратковременной памяти. Я знал, что **values** — это **vector<float>**; теперь я знаю, что **values** — это отсортированный **vector<float>**. Все это остается одной мыслью в моем «кеше» кратковременной памяти.

Реакция же на **processVector** у меня совершенно иная. Я раньше его не видел и не знаю, что он делает. Имя не помогает — это наглядный пример плохо выбранного имени. Единственный способ разобраться — проанализировать код **processVector**, возможно, с пошаговым выполнением*. Мне лишний раз

---

* Пошаговое выполнение кода в отладчике — отличный способ лучше понять код. Рекомендация 10/10. Фундаментально потребности в кратковременной памяти это не отменяет, но отладчик отлично помогает запоминать, какие существуют переменные и какие значения в них хранятся.

приходится задействовать свой крайне ограниченный объем кратковременной памяти (семь плюс-минус две мысли), прежде чем все детали выстроятся в более простую для понимания концепцию.

Общие знания бесплатны; новые концепции стоят дорого

Ваша цель при написании кода — сделать так, чтобы его можно было легко понимать, поэтому стоит держать в уме разницу между **sort** и **processVector**. Обращение к **sort** не перегружает кратковременную память читателя, потому что он уже знает, что делает **sort**. Другое дело **processVector**, чтобы понять код, читателю нужно разобраться в работе этого фрагмента, что загрузит его кратковременную память.

Код, в котором используются общие абстракции или известные всей команде шаблоны, читать гораздо проще, чем код, в котором используются незнакомые шаблоны или вводятся новые абстракции.

Вывод очевиден: при написании кода ориентируйтесь на стандартные абстракции и шаблоны, принятые в вашей команде. Не придумывайте новые абстракции и шаблоны, если только не уверены в их большой эффективности и не считаете, что они могут стать новым стандартом для вашей команды.

Так, в примере с решетом Эратосфена я использовал массив флагов (с говорящим названием **isMultiple**); ими отмечались числа, точно не являющиеся простыми, потому что они кратные другим. Это обычный для C массив типа **bool**. Конечно, его можно было бы абстрагировать в класс «вектор битов», сэкономив немного памяти и слегка улучшив шаблоны доступа к памяти.

С классом **BitVector*** код решета Эратосфена мог бы выглядеть так:

```
vector<int> primes;
BitVector isMultiple(100);

for (int value = 2; value < isMultiple.size(); ++value)
{
    if (isMultiple[value])
        continue;
```

---

* Описанный в этой главе класс **BitVector** — упрощенная версия **vector<bool>** в C++ (по крайней мере, по своей сути).

```
    primes.push_back(value);

    for (int multiple = value;
         multiple < isMultiple.size();
         multiple += value)
    {
        isMultiple[multiple] = true;
    }
}
```

Легко ли это читать? Если класс **BitVector** входит в стандартный инструментарий вашей команды и знаком всем, тогда да! Возможно, эта версия будет даже проще, чем версия с простым массивом **bool**.

Для тех же, кто не знаком с **BitVector**, ситуация другая. Не очень внимательный программист, конечно, может предположить, что это вектор битовых значений, и продолжить работу. Но, как правило, не самые внимательные программисты сами «стреляют себе в ногу». Осторожный программист захочет разобраться, что же представляет собой этот класс **BitVector**, а это не так уж просто, даже в самой упрощенной версии, подходящей под приведенный выше пример использования:

```
class BitVector
{
public:
    BitVector(int size) :
        m_size(size),
        m_values()
    {
        m_values.resize((size + 31) / 32, 0);
    }

    int size() const
    {
        return m_size;
    }

    class Bit
    {
        friend class BitVector;
```

```
    public:
        operator bool () const
        {
            return (*m_value & m_bit) != 0;
        }

        void operator = (bool value)
        {
            if (value)
                *m_value |= m_bit;
            else
                *m_value &= ~m_bit;
        }

        unsigned int * m_value;
        unsigned int m_bit;
    };

    Bit operator [] (int index)
    {
        assert(index >= 0 && index < m_size);
        Bit bit = { &m_values[index / 32], 1U << (index % 32) };
        return bit;
    }

protected:

    int m_size;
    vector<unsigned int> m_values;
};
```

Полноценный же класс **BitVector** был бы еще функциональнее, то есть, разумеется, больше и сложнее! Но даже при таком минимуме возможностей здесь имеются свои хитрости — например, создание временного объекта, который инкапсулирует возможность читать и записывать один бит, полагаясь на перегрузку операторов C++ для доступа и изменения значения*.

---

* Порой кажется, что основная функция C++ — это упаковать в каждой капле кода целую Вселенную сложности. Но я отвлекся.

Это «умный» прием*, позволяющий писать код, который выглядит как простое обращение к элементу массива, хотя на самом деле компилируется он во что-то куда более сложное. Но разгадывание этого трюка займет несколько ячеек кратковременной памяти читателя, тогда как его цель состоит не в том, чтобы разобраться в тонкостях странного класса `BitVector`, а в том, чтобы понять код решета Эратосфена. Пока программисты не усвоят, как работает `BitVector`, не сведут его к абстракции и не закрепят это знание в своей долговременной памяти, код для решета Эратосфена с использованием этого класса будет сложнее, а не проще для понимания.

Так было бы вам проще писать и читать код, если бы при реализации алгоритма решета Эратосфена** для хранения массива флагов был введен новый класс `BitVector`? Почти наверняка вы ответили «нет». Это лишняя работа, делающая код менее читаемым.

Вводить класс `BitVector` стоит только в том случае, если вы уверены, что он будет использоваться настолько широко, что его хорошо запомнят все члены команды, а также что он даст ощутимые преимущества перед уже используемыми решениями. Узнать это можно только в том случае, если вы найдете множество участков кода, где используются массивы битов, выявите при этом (и желательно измерите) преимущества `BitVector`, и у вас будет веская причина не пользоваться `vector<bool>`. Только в этом случае имеет смысл вводить `BitVector`.

У нас в Sucker Punch действительно есть собственный класс для битовых векторов, который используется примерно в 120 местах в нашей довольно большой кодовой базе. Для большинства наших коллег это привычная технология, они усвоили, что делает этот класс, и потому им не нужно разбираться в его деталях. Он стал частью общего багажа знаний, и пользоваться им безопасно. Но мы ввели его не ради одного частного случая, а после того, как накопили множество фрагментов кода, работающего с большими массивами битов.

Подведение итогов

Самый лучший код, а следовательно, и самый простой для чтения и понимания — это такой код, который подразумевает взаимодействие кратковре-

---

* В данном случае «умный» — не комплимент.

** Только не надо на самом деле генерировать простые числа именно так. За прошедшие 2250 лет человечество придумало куда более эффективные методы их получения. Тем не менее я снимаю шляпу перед Эратосфеном! Его «решето» — это, пожалуй, третье или четвертое по значимости достижение в его послужном списке, и я до сих пор знаю, что это такое.

менной и долговременной памяти. Он опирается на стандарты и соглашения команды, уже хранящиеся в долговременной памяти каждого участника. Когда появляются новые идеи, они вводятся небольшими или умеренными порциями, достаточно компактными, чтобы поместиться в кратковременной памяти читателя. Эти порции обладают простой, легко абстрагируемой функциональностью и хорошо подобранными именами, что делает их легкими для «сворачивания» и запоминания.

И что же мы получаем в результате? Кодовую базу, которую легко читать и легко изучать, которая упрощает сведение новых идей к простым абстракциям, а затем рекурсивно делает то же самое с самими абстракциями до тех пор, пока вся система не станет прозрачной и понятной.

Правило 10

Локализируйте сложность

Сложность — враг масштабируемости.

Вы уже знаете, что код должен быть простым — максимально простым, но не слишком простым, как утверждает Правило 1. Но по мере разрастания проекта следовать этому Правилу становится все труднее. Писать простой код для простых задач легко, но по мере развития и обрастания функциональностью он неизбежно усложняется. А чем сложнее код, тем труднее с ним работать, и уже невозможно удерживать в голове все детали. Каждая попытка исправить ошибку или добавить новую функцию сопровождается неожиданными побочными эффектами, и за каждым шагом вперед следует шаг назад.

Отчасти решение заключается в том, чтобы поддерживать простоту кода или упрощать его по мере возможности. Опять же, вспомним Правило 1. Но полностью избавиться от сложности нельзя: любое достаточно функциональное и живущее долго программное обеспечение неизбежно сталкивается с проблемой сложности. Но сложность можно контролировать.

Дело со сложностью обстоит так же, как с некоторыми игроками в американском футболе, о которых говорят: «Его нельзя остановить, его можно только сдержать».

Поэтому полезная стратегия заключается в изоляции сложности, от которой нельзя избавиться. Сложность перестает быть проблемой, когда внутренняя реализация какого-то участка кода сложна, но его внешний интерфейс остается простым. Да, внутри этого кода приходится иметь дело со сложностью, но за его пределами о ней можно не беспокоиться.

Простой пример

Возьмем для примера функции синуса и косинуса в любом языке программирования. Их внешний интерфейс прост: вызываешь функцию — получаешь синус или косинус переданного угла. Однако внутренняя их реализация довольно сложна.

Задумываться о том, как они устроены, я начал далеко не в юном возрасте. До этого момента я считал, что они просто каким-то волшебным образом выдают правильный результат*, — и такое счастливое неведение вполне допустимо! Как бы сложно ни были реализованы функции нахождения синуса и косинуса, их сложность никак не влияет на то, как ими пользуются. Они просто работают так, как от них ожидают.

Можно нарисовать окружность, совершенно не зная, как реализованы функции синуса и косинуса:

```
void drawCircle(Point center, float radius, Color color)
{
    int count = int(ceil(pi / acos((radius - 1.0) / radius)));
    Point previousPoint = center + Vector(radius, 0.0, 0.0);
    for (int index = 1; index <= count; ++index)
    {
        float angle = 2.0 * pi * index / count;
        Point nextPoint = center +
                          radius * Vector(cosf(angle), sinf(angle),
0.0);
        drawLine(previousPoint, nextPoint, color);

        previousPoint = nextPoint;
    }
}
```

Где-то внутри **sinf** и **cosf** (функций синуса и косинуса для 32-битных вещественных чисел из стандартной библиотеки C) скрыта своя сложность, но она не «просачивается» наружу, в остальной код**. Сложность надежно локализована.

---

* Насколько помню, когда я наконец-то заинтересовался этим вопросом, мой наивный первый предположительный ответ сводился к «большим таблицам и линейной интерполяции». Мне немного стыдно за это предположение, ведь я уже тогда знал, что такое ряд Тейлора.

** Функции **sinf** и **cosf** зависят от конкретной реализации и бывают удивительно сложными. Я попытался было написать здесь короткое объяснение, но безуспешно. Главное, что нужно понимать: эти функции не должны вычислять абсолютно точное значение; достаточно, чтобы результат был точным в пределах разрешения числа с плавающей точкой. Кроме того, модульная арифметика позволяет свести угол к удобному для приближенного вычисления диапазону. Любопытно, что в архитектуре процессоров x86 существуют специальные инструкции для вычисления синуса и косинуса, но современные компиляторы их не используют, если только не указать это явно. Инструкции эти были добавлены в 1987 году и имеют известные ошибки, которые невозможно исправить без нарушения обратной совместимости. К сожалению.

Прячем внутренние подробности

То же правило применимо и к вашему коду. По возможности изолируйте сложность, ограничивая ее четко определенными участками программы.

Допустим, у вас есть список записей о клиентах и вы пишете функцию, возвращающую список клиентов, недавно что-то купивших. Записи о клиентах выглядят так:

```
struct Customer
{
    int m_customerID;
    string m_firstName;
    string m_lastName;
    Date m_lastPurchase;
    Date m_validFrom;
    Date m_validUntil;
    bool m_isClosed;
};
```

Сложность здесь в том, что не все записи о клиентах в списке являются валидными, то есть актуальными для использования. Срок действия одних учетных записей истек, другие еще не активированы, а третьи закрыты самими клиентами. Ваша функция должна исключать такие недействительные записи:

```
void findRecentPurchasers(
    const vector<Customer *> & customers,
    Date startingDate,
    vector<Customer *> * recentCustomers)
{
    Date currentDate = getCurrentDate();

    for (Customer * customer : customers)
    {
        if (customer->m_validFrom >= currentDate &&
            customer->m_validUntil <= currentDate &&
            !customer->m_isClosed &&
            customer->m_lastPurchase >= startingDate)
        {
            recentCustomers->push_back(customer);
```

```
        }
    }
}
```

Сложность, вызванная недействительными записями о клиентах, здесь не локализована — она «просочилась» в эту, на первый взгляд, не связанную с ней функцию. Теперь каждый цикл по списку клиентов должен проверять валидность записей. Каждый раз при изменении правил валидности этот цикл придется обновлять.

Вообще говоря, предыдущий фрагмент кода — пример довольно плохого дизайна. Нет никаких причин дублировать проверку валидности клиента в каждом цикле, и одна из целей объектно ориентированного программирования как раз и заключается в том, чтобы скрывать подобного рода сложность. Минимум, что можно здесь сделать, — инкапсулировать правило проверки:

```
struct Customer
{
    bool isValid() const
    {
        Date currentDate = getCurrentDate();
        return m_validFrom >= currentDate &&
               m_validUntil <= currentDate &&
               !m_isClosed;
    }

    int m_customerID;
    string m_firstName;
    string m_lastName;
    Date m_lastPurchase;
    Date m_validFrom;
    Date m_validUntil;
    bool m_isClosed;
};
```

Так цикл становится немного проще:

```
void findRecentPurchasers(
    const vector<Customer *> & customers,
    Date startingDate,
```

```
    vector<Customer *> * recentCustomers)
{
    Date currentDate = getCurrentDate();
    for (Customer * customer : customers)
    {
        if (customer->isValid() &&
            customer->m_lastPurchase >= startingDate)
        {
            recentCustomers->push_back(customer);
        }
    }
}
```

Но это лишь полумера. Лучшее решение находится «выше по течению»: вместо того чтобы перебирать всех клиентов, нужно перебирать только валидных клиентов. Любой участок кода, формирующий список клиентов, заодно должен предоставлять список валидных клиентов, вычисляемый, скорее всего, на основе полного списка. Тогда ваш код станет в надлежащей степени простым:

```
void findRecentPurchasers(
    const vector<Customer *> & validCustomers,
    Date startingDate,
    vector<Customer *> * recentCustomers)
{
    Date currentDate = getCurrentDate();

    for (Customer * customer : validCustomers)
    {
        if (customer->isValid() &&
            customer->m_lastPurchase >= startingDate)
        {
            recentCustomers->push_back(customer);
        }
    }
}
```

Теперь вся сложность локализована в функции, возвращающей список действительных клиентов, а в коде вроде **`findRecentPurchasers`** больше

не нужно заботиться о проверке валидности клиентов, так что писать его и понимать становится проще.

Распределенное состояние и сложность

Объектно ориентированный дизайн помогает локализовать сложность, но это не панацея. Даже с ним легко попасть в неприятности, особенно когда состояние не сосредоточено в одном месте, а распределено между множеством объектов.

Само по себе распределение состояния не обязательно представляет собой проблему. Иногда сама суть системы подталкивает к тому, чтобы создавать несколько объектов, совместно управляющих состоянием системы. Согласно принципам объектно ориентированного дизайна, такая структура должна оставаться управляемой: каждый объект управляет своим состоянием, а взаимодействия между объектами хорошо определены.

Но эти принципы объектно ориентированного дизайна на практике сложно реализовать без продуманного программирования. При попытках сделать нечто зависящее от состояния нескольких объектов легко получить довольно уязвимый код.

Вот вам вымышленный пример. Допустим, вы разрабатываете игру в жанре «стелс», то есть часть удовольствия от игры заключается в том, чтобы прятаться от врагов и тайком перемещаться между ними. Предположим, игра у вас «семейно ориентированная» и игроку нужно подкрадываться к другим персонажам, чтобы прикрепить им на спину табличку «пни меня». Для облегчения задачи на экране отображается маленький значок в виде глаза. Когда ни один из персонажей не видит игрока, глаз этот закрыт, а когда игрок оказывается в пределах видимости какого-то противника, глаз открывается. Итак, закрытый глаз означает, что игрок в безопасности, а открытый глаз означает, что игрока могут заметить.

Для моделирования этой системы у вас имеется несколько объектов и классов: объект игрока, объекты всех остальных персонажей, объект значка глаза и объект, отслеживающий, какие персонажи видят других персонажей. Последний объект, так называемый «менеджер осведомленности», предоставляет способ регистрации функции обратного вызова, вызываемой всякий раз, когда конкретного персонажа замечает другой персонаж, а также регистрации второй функции обратного вызова, вызываемой, когда другой персонаж теряет из виду зарегистрированного персонажа.

Очевидная реализация функционала отталкивается от объекта игрока. В объекте игрока можно реализовать функцию осведомленности и пользоваться ею для подсчета персонажей, заметивших игрока. Когда счет равен

нулю, объект игрока закрывает значок глаза, в противном случае глаз открыт.

Менеджер осведомленности выглядит так:

```
class AwarenessEvents
{
public:
    virtual void OnSpotted(Character * otherCharacter);
    virtual void OnLostSight(Character * otherCharacter);
};

class AwarenessManager
{
public:
    int getSpottedCount(Character * character);
    void subscribe(Character * character, AwarenessEvents * events);
    void unsubscribe(Character * character, AwarenessEvents * events);
};
```

Значок глаза еще проще:

```
class EyeIcon
{
public:
    bool isOpen() const;
    void open();
    void close();
};
```

С такими объектами код для игрока написать легко. При создании объекта игрока из менеджера осведомленности берется начальный счетчик, а для обработки изменений реализуется интерфейс **AwarenessEvents**. Значок глаза можно открывать или закрывать в нужный момент на основе точного учета других персонажей, которые видят игрока:

```
class Player : public Character, public AwarenessEvents
{
public:
    Player();
```

```
    void onSpotted(Character * otherCharacter) override;
    void onLostSight(Character * otherCharacter) override;

protected:

    int m_spottedCount;
};

Player::Player() :
    m_spottedCount(getAwarenessManager()->getSpottedCount(this))
{
    if (m_spottedCount == 0)
        getEyeIcon()->close();

    getAwarenessManager()->subscribe(this, this);
}

void Player::onSpotted(Character * otherCharacter)
{
    if (m_spottedCount == 0)
        getEyeIcon()->open();

    ++m_spottedCount;
}

void Player::onLostSight(Character * otherCharacter)
{
    --m_spottedCount;

    if (m_spottedCount == 0)
        getEyeIcon()->close();
}
```

И это неплохой код. Он не слишком обширный и достаточно хорошо читается. В нем имеется одна небольшая тонкость — выбор для сравнения **m_spottedCount** с нулем, но понять это несложно. Считаю такой код оправданным.

Недееспособность?

Как и любой код, этот код со временем развивается. Чтобы добавить в игру немного сложности, мы вносим следующее изменение: значок глаза должен открываться, когда игрок «недееспособен», то есть когда он не может действовать и выполнять свои обычные задачи в игре. Или же, выражаясь иначе, глаз должен закрываться, когда ни один из врагов не замечает игрока и игрок в состоянии действовать.

В данном случае у класса **Player** появляется метод **setStatus**, вызываемый для изменения состояния игрока. Написать код для регистрации случаев, когда игрок становится «недееспособным» или, наоборот, восстанавливается и снова становится «дееспособным», легко. Интерес представляют только случаи, когда **m_spottedCount** становится равным нулю, так как в противном случае глаз уже открыт. Аналогично, когда счетчик **m_spottedCount** становится не равным нулю, глаз нужно обновлять, только когда игрок не недееспособен:

```
enum class STATUS
{
    Normal,
    Blindfolded
};

class Player : public Character, public AwarenessEvents
{
public:
    Player();

    void setStatus(STATUS status);

    void onSpotted(Character * otherCharacter) override;
    void onLostSight(Character * otherCharacter) override;

protected:

    STATUS m_status;
    int m_spottedCount;
};

Player::Player() :
```

```
    m_status(STATUS::Normal),
    m_spottedCount(getAwarenessManager()->getSpottedCount(this))
{
    if (m_spottedCount == 0)
        getEyeIcon()->close();

    getAwarenessManager()->subscribe(this, this);
}

void Player::setStatus(STATUS status)
{
    if (status == m_status)
        return;

    if (m_spottedCount == 0)
    {
        if (status == STATUS::Normal)
            getEyeIcon()->close();
        else if (m_status == STATUS::Normal)
            getEyeIcon()->open();
    }

    m_status = status;
}

void Player::onSpotted(Character * otherCharacter)
{
    if (m_spottedCount == 0 && m_status == STATUS::Normal)
        getEyeIcon()->open();

    ++m_spottedCount;
}

void Player::onLostSight(Character * otherCharacter)
{
    --m_spottedCount;

    if (m_spottedCount == 0 && m_status == STATUS::Normal)
        getEyeIcon()->close();
}
```

Описанные выше правила добавили некоторую сложность, особенно в случаях пересечения условий для отображения индикатора состояния здоровья. Тем не менее код еще не выглядит катастрофически.

Можно заметить, что код базируется на более тонких допущениях — например, значок глаза в **`CPlayer::setStatus`** обновляется только тогда, когда счетчик заметивших равен нулю. Понять происходящее не так уж сложно, но небольшая оптимизация достигается за счет некоторого увеличения сложности.

Начинается путаница

Дизайн продолжает развиваться, что никого не удивляет. На этот раз добавляются погодные эффекты. Если погода туманная (foggy), значок глаза должен оставаться открытым — точно так же, как когда игрока замечают или когда он недееспособен.

Система погоды (weather), как и система осведомленности, предлагает простой интерфейс с запросами и функциями обратного вызова:

```
enum class WEATHER
{
    Clear,
    Foggy
};

class WeatherEvents
{
public:
    virtual void onWeatherChanged(WEATHER oldWeather, WEATHER
newWeather);
};

class WeatherManager
{
public:
    WEATHER getCurrentWeather() const;
    void subscribe(WeatherEvents * events);
};
```

Он вписывается в ту же схему, что и система осведомленности. Реализуем обратный вызов для погоды, добавим инициализацию, внесем в существующие проверки новую логику, и система заработает:

```
class Player :
    public Character,
    public AwarenessEvents,
    public WeatherEvents
{
public:
    Player();

    void setStatus(STATUS status);

    void onSpotted(Character * otherCharacter) override;
    void onLostSight(Character * otherCharacter) override;

    void onWeatherChanged(WEATHER oldWeather, WEATHER newWeather)
override;

protected:
    STATUS m_status;
    int m_spottedCount;
};

Player::Player() :
    m_status(STATUS::Normal),
    m_spottedCount(getAwarenessManager()->getSpottedCount(this))
{
    if (m_spottedCount == 0 &&
        getWeatherManager()->getCurrentWeather() != WEATHER::Foggy)
    {
        getEyeIcon()->close();
    }

    getAwarenessManager()->subscribe(this, this);
    getWeatherManager()->subscribe(this);
}

void Player::setStatus(STATUS status)
```

```
{
    if (status == m_status)
        return;

    if (m_spottedCount == 0 &&
        getWeatherManager()->getCurrentWeather() != WEATHER::Foggy)
    {
        if (status == STATUS::Normal)
            getEyeIcon()->close();
        else if (m_status == STATUS::Normal)
            getEyeIcon()->open();
    }

    m_status = status;
}

void Player::onSpotted(Character * otherCharacter)
{
    if (m_spottedCount == 0 &&
        m_status == STATUS::Normal &&
        getWeatherManager()->getCurrentWeather() != WEATHER::Foggy)
    {
        getEyeIcon()->open();
    }

    ++m_spottedCount;
}

void Player::onLostSight(Character * otherCharacter)
{
    --m_spottedCount;

    if (m_spottedCount == 0 &&
        m_status == STATUS::Normal &&
        getWeatherManager()->getCurrentWeather() != WEATHER::Foggy)
    {
        getEyeIcon()->close();
    }
}
```

```
void Player::onWeatherChanged(WEATHER oldWeather, WEATHER newWeather)
{
    if (m_spottedCount == 0 &&
        m_status == STATUS::Normal)
    {
        if (oldWeather == WEATHER::Foggy)
            getEyeIcon()->close();
        else if (newWeather == WEATHER::Foggy)
            getEyeIcon()->open();
    }
}
```

Сразу после написания такой код может показаться разумным. Я и сам писал подобное без всяких раздумий и волнений!

Код развивается по заложенному еще в первой версии пути: сначала при инициализации объекта игрока проверяется состояние мира, затем отслеживаются изменения, влияющие на значок глаза.

С концептуальной точки зрения лежащая в основе идея остается той же, что и в первой реализации: если состояние глаза не меняется, обновлять значок глаза не нужно. Например, при смене погоды менять значок глаза имеет смысл только в тех случаях, когда счетчик заметивших игрока равен нулю и игрок не недееспособен. Если же счетчик заметивших игрока не равен нулю или если игрок недееспособен, значок глаза уже открыт и должен оставаться открытым.

В коде эта идея проявляется в трех вариантах: для линии видимости, для состояния игрока и для текущей погоды. В каждом случае идея реализована немного по-разному, но суть ее одинакова, поэтому на первый взгляд она кажется несложной.

Но здесь стоит подумать о том, почему и кому она кажется несложной. Если код рассматриваете вы, то общая идея действительно кажется вам несложной — ведь вы же сами и написали этот код! Но если его код будет просматривать ваш коллега, будет ли код понятен ему?

Если общая идея — «не обновлять глаз без необходимости» — знакома, то можно заметить, как она реализуется при каждом повторении. Но если пытаться вывести общую концепцию, только глядя на код с его вариантами… то это уже не так очевидно.

Смена подхода

Но есть проблема и посерьезнее. Визуально для пользователя функция довольно проста — значок глаза должен быть закрыт, когда выполняются три условия:

- Ни один враг не заметил игрока.
- Игрок не недееспособен.
- Погода не туманная.

В примере кода выше эта идея реализована в пяти (!) различных вариантах, в каждом из которых эти три условия выражаются по-своему. Код получился запутанным. И в нем нет простого места, в котором можно было бы одновременно проверить все три этих условия. Все пять вариантов реализуют идею по-своему, и каждый для минимизации выполняемой работы полагается на контекст.

При смене дизайна придется обновлять каждую из пяти реализаций. Например, при добавлении нового погодного состояния **`WEATHER::HeavyFog`** (густой туман) придется добавлять проверки для него во всех местах, где проверялось состояние **`WEATHER::Foggy`**.

Еще опаснее вносить какие-то другие изменения в добавленные методы. Допустим, мы решили, что модель игрока должна поворачивать голову и смотреть на врага, который заметил игрока, а это значит, что в **`CPlayer::onSpotted`** появится дополнительный код. Нужно будет убедиться в том, что в результате такого изменения не поломается индикатор стелса.

Основная проблема этого кода в том, что сложность дизайна не локализована. Сам по себе дизайн прост и подразумевает три описанные в самом начале условия, но его реализация распределена по пяти отдельным блокам, написанным немного по-разному. Каждое условие добавляет свою сложность, и каждая порция сложности взаимодействует со всеми остальными. Всё быстро запутывается.

Если у вас имеется какая-то сложная идея, например правила смены значка с открытым глазом на значок с закрытым глазом, то лучше выразить эту идею в одном месте.

В данном случае речь идет о единой и простой реализации трех условий, на основе которой строится остальной код. При этом остальная система остается без изменений, и в результате может получиться примерно такое решение:

```
enum class STATUS
{
    Normal,
    Blindfolded
};

class Player :
```

```
    public Character,
    public AwarenessEvents,
    public WeatherEvents
{
public:

    Player();

    void setStatus(STATUS status);

    void onSpotted(Character * otherCharacter) override;
    void onLostSight(Character * otherCharacter) override;

    void onWeatherChanged(WEATHER oldWeather, WEATHER newWeather)
override;

protected:

    void refreshStealthIndicator();

    STATUS m_status;
};

Player::Player() :
    m_status(STATUS::Normal)
{
    refreshStealthIndicator();

    getAwarenessManager()->subscribe(this, this);
    getWeatherManager()->subscribe(this);
}

void Player::setStatus(STATUS status)
{
    m_status = status;
    refreshStealthIndicator();
}

void Player::onSpotted(Character * otherCharacter)
{
```

```
    refreshStealthIndicator();
}

void Player::onLostSight(Character * otherCharacter)
{
    refreshStealthIndicator();
}

void Player::onWeatherChanged(WEATHER oldWeather, WEATHER newWeather)
{
    refreshStealthIndicator();
}

void Player::refreshStealthIndicator()
{
    if (m_status == STATUS::Normal &&
        getAwarenessManager()->getSpottedCount(this) == 0 &&
        getWeatherManager()->getCurrentWeather() != WEATHER::Foggy)
    {
        getEyeIcon()->close();
    }
    else
    {
        getEyeIcon()->open();
    }
}
```

Здесь все пять реализаций проверки условий объединены в один метод **`refreshStealthIndicator`**. Этот метод вызывается всякий раз, когда изменяются условия, которые он проверяет.

Да, сложность сохраняется — не совсем очевидна связь между условиями и функциями обратного вызова, отслеживающими их изменения, — но теперь ее гораздо меньше, чем раньше.

Эта реализация линейна по количеству условий. При добавлении нового условия достаточно добавить проверку в **`refreshStealthIndicator`**, написать немного кода для инициализации и в одном-двух местах для отслеживания изменений. При добавлении десяти условий код вырастет в 10 раз.

Раньше, когда сложность не была локализована, сложность кода была квадратичной: при добавлении нового условия приходилось добавлять новые проверки для всех существующих реализаций логики. Как следствие,

количество строк кода росло пропорционально квадрату числа условий, что плохо, потому что при дальнейшем росте сложности можно быстро «упереться в стену».

Локализованная сложность, простые взаимодействия

Главный вывод отсюда в том, что нужно избегать сложных взаимодействий между разными частями системы. Какие-то детали можно делать сложными, но только при локализованной сложности. Даже если внутри какого-то компонента и есть сложные детали, но если у него простая внешняя интерфейсная часть и для него определены простые взаимодействия, проект он не разрушит. А вот компонент со сложным интерфейсом и сложными взаимодействиями грозит катастрофой, даже если внутри он прост.

Система, построенная из компонентов с простыми взаимодействиями, обычно имеет линейную сложность. Каждый новый компонент немного повышает сложность, но в целом система остается управляемой.

Если же взаимодействия между компонентами сложны, то всё быстро выходит из-под контроля.

Если при добавлении какой-то новой функции нужно вносить правки во множестве мест, то это признак плохого кода. В лучшем случае это значит, что новая функция не вписывается в существующий код. Если же вносить изменения во множество мест приходится при добавлении каждой функции, то, скорее всего, вам не удалось локализировать сложность. А это добром не кончится.

Правило 11

Станет ли от этого вдвойне лучше?

Любой проект рано или поздно достигает естественных пределов своей архитектуры, когда возникают задачи, которые просто не укладываются в существующую схему. Например, нужно добавить некую функцию, которую невозможно выразить в текущей парадигме, — допустим, у вас есть механизм фильтрации по критериям и вдруг возникает случай, когда некоторые критерии нужно выбирать по логическому оператору ИЛИ, а не И.

Либо меняется структура данных. Систему могли построить для решения задач определенного масштаба, а со временем ее стали применять к задачам другого масштаба или формы, и теперь всплывают фундаментальные проблемы производительности.

Бывает и так, что все просто запутывается. Ваша система позволяла подстраивать основное поведение системы под особые случаи, и потому она продержалась так долго. Но теперь механизм расширения используется не только в особых случаях, а во всех. Каждый вызов системы превращается в прыжки между разными исключениями, и специалистам становится трудно понимать, что и как работает, а что не работает.

Иногда какой-то фрагмент кода настолько устаревает, что уже больше не вписывается в общую кодовую базу. Допустим, одна из частей вашего достаточно современного проекта написана на C, а не на C++, и вы с неодобрением взираете на все эти ручные манипуляции с указателями. Эта часть соответствует устаревшему образу мышления, и все хотят переписать ее в более современном стиле.

Такое развитие событий естественно и неизбежно, и оно не должно вызывать паники. Это даже не проблема, а просто естественный ход вещей.

Если вы думаете, что нужно было лучше стараться с самого начала, продумай вы дизайн всей системы получше, то проблем можно было бы избежать, то вспомните Правило 4 «Обобщайте после трех примеров». Возможно, ваш дизайн изначально был неудачным. Но, скорее всего, вам не удалось бы предугадать будущее и даже при более тщательных усилиях вы создали бы не

менее плохой и при этом более сложный дизайн, который достиг бы своих пределов еще раньше.

Не на каждом участке проекта вы упретесь в ограничения его архитектуры. Некоторые части могут спокойно работать без каких-либо переделок годами, и это не случайность: если при первоначальном проектировании вы приняли удачные решения, если выполняемые задачи со временем не слишком меняются, если вы с коллегами поддерживаете порядок в коде и если возникающие исключения удается локализовать и легко устранить, то вся система может работать практически бесконечно.

Вот почему так опасны попытки «подстраховаться на будущее» — иногда в этом просто нет необходимости, а в остальных случаях они все равно не срабатывают.

Три пути вперед: игнорирование, мелкие правки, рефакторинг

Итак, вы столкнулись с каким-то естественным пределом, но это еще не значит, что нужно выкидывать весь код и переписывать его с нуля.

Можно просто смириться с ограничением. Например, не разрешать использовать оператор ИЛИ в фильтрах, купить более мощное оборудование для увеличения производительности или смириться с какой-то степенью сложности и с устаревшим кодом.

Могут сработать небольшие правки и обработка исключений. Допустим, ваш фильтр представляет собой сочетание условий И и ИЛИ, как на большинстве веб-сайтов, и дополнительный фильтр ИЛИ можно реализовать, просто объединив в интерфейсе две категории. Иногда проблема с производительностью кроется в дублировании некоторых вычислений, и проблему почти полностью решит небольшое кеширование. Может, система действительно каждый раз проходит через механизм исключений, но большинство из них однотипны, и их можно встроить в основной код, чтобы избавиться от многочисленных особых случаев. Или же единственная действительно устаревшая часть — это набор плохо продуманных макросов для управления массивами на C, и их несложно заменить на `std::vector`.

В некоторых ситуациях все же приходится идти на радикальную переделку. Вы разрабатывали архитектуру системы исходя из потребностей, существовавших на момент начала ее создания, но теперь задачи изменились, или же вы лучше поняли их суть. Текущая архитектура не способна решать задачи в их новом виде, и вы придумали более правильный способ решения.

Итак, какой же из трех путей выбрать: игнорирование, мелкие правки или рефакторинг?

Постепенная эволюция или постоянное изобретение заново

Программисты, как правило, плохо справляются с решением этого вопроса. Часто они идут на радикальные перемены тогда, когда это не нужно, меняют код не по тем причинам, по каким его действительно надо менять, и делают это не вовремя, из-за чего проблем становится еще больше.

Если более конкретно, то программистов условно можно разделить на два типа. Первый тип мыслит «кусками», то есть они рассматривают каждую новую задачу через призму текущего решения и дорабатывают текущий дизайн. Второй тип отталкивается от задачи и решения как единого целого; им хочется исправить не что-то отдельное, а всю систему целиком, и они охотно берутся переписывать всё с нуля.

Губительны обе крайности. Если постоянно вносить лишь небольшие исправления, проект со временем разваливается. Улучшения откладываются, а исключения с «костылями» накапливаются, и система под их тяжестью постепенно глохнет. Если же каждый раз подвергать систему радикальной перестройке, вы начнете буксовать на месте. Тем самым вы отказываетесь от накопленного опыта и заново наступаете на одни и те же грабли. Каждая новая архитектура влечет за собой новые проблемы, и прогресса не наблюдается.

Как и во многих других сферах, наилучший результат дает баланс. Да, выбирать между мелкими правками и рефакторингом непросто, но вы можете принять более взвешенное решение, советуясь с коллегами и отталкиваясь от своих склонностей. Если в случае неопределенности вы, как правило, выбираете вариант, от которого вам комфортнее, то со временем вы будете поступать так все чаще и чаще. Для программиста первого типа комфортный выбор — мелкие правки или игнорирование; для программиста второго типа комфортный выбор — переделка всей системы. И так они поступают в большинстве случаев, что плохо, потому что нужно соблюдать равновесие между этими двумя крайностями.

Вот некоторые тревожные признаки того, что ситуация выходит из-под контроля для первого типа:

- Проблемы описываются исключительно в терминах текущей архитектуры. Проявляться это может в мелочах, даже в том, что для описания проблемы используются свои внутренние термины, а не термины из той сферы, к которой относится проблема. Иногда программистам

первого типа бывает трудно даже осмыслить проблему вне контекста существующей архитектуры, и это заметно по их языку.

- При описании проблемы они часто используют слова вроде «невозможно». И почти всегда это не так — в худшем случае проблему бывает очень трудно решить без внесения масштабных изменений в общую архитектуру системы.
- Закрывая дискуссию, они ссылаются на график проекта. Это не значит, что сроки не важны, — они, разумеется, важны. Но если единственный ваш аргумент звучит как: «Мы не успеваем в сроки», то вы, возможно, застряли в мышлении первого типа.
- С момента последнего крупного пересмотра архитектуры прошло уже много лет, хотя мелкие изменения вносились регулярно.

Тревожные признаки для второго типа:

- Главный аргумент в пользу переделки всей системы звучит как: «Нам наконец-то нужно привести этот участок кода в порядок».
- Решение о пересмотре всей системы принимается на основе одного частного случая — например, когда сложно реализовать какую-то отдельную функцию или когда какой-то отдельный набор данных влечет за собой низкую производительность.
- Необходимость общей перестройки объясняется в терминах производительности или экономии ресурсов, но при этом никто не проводил профилирование, которое выявило бы настоящий источник проблем.
- Предложение о переделке преподносится как возможность, а не как решение какой-то конкретной проблемы. Без опоры на реальную задачу любое предложение о радикальной переделке сомнительно.
- Главный упор в предложении делается на некий «новый замечательный инструмент»: на новый язык, новую библиотеку, новый технический прием.

Возможно, в этих описаниях вы узнали какие-то собственные модели поведения. Я точно узнаю свои: по природе я программист второго типа, и, когда я принимаю решения, мне постоянно приходится об этом себе напоминать. К счастью, в моей команде хватает представителей первого типа, и на них можно положиться при обсуждении сбалансированного подхода.

Также описанные выше признаки обоих типов часто наблюдаются в одной и той же ситуации. Например, вы размышляете над радикальной переделкой всей системы, которая в своей основе не менялась уже много лет, и это признак первого типа; побуждением же к действию служит восторженное впечатление от какой-то новомодной технологии баз данных, и это признак

второго типа. Здесь мы наблюдаем и сигнал первого типа («Пора бы уж поменять систему!») и сигнал второго типа («Ух ты! Новая база данных!»).

Умение распознавать эти шаблоны поведения помогает принимать более осознанные решения, но само по себе оно не дает ответа. То, что вы заметили признаки мышления более консервативного первого типа, еще не значит, что мелкие правки — это неправильно. Точно так же наличие признаков второго типа не делает масштабный рефакторинг ошибкой. Чтобы принять важное решение — стоит ли серьезно перерабатывать систему или ограничиться малыми шагами — нужен дополнительный критерий.

Простое практическое правило

Принимая решения о том, следует ли серьезно менять всю систему, я задаю себе простой вопрос: «Станет ли от этого система вдвойне лучше?»

Если вы уверены в том, что обновленная система действительно будет как минимум вдвое лучше текущей, то рефакторинг стоит всех нарушений установленного процесса и неизбежных новых проблем. Если же новая система вряд ли будет как минимум вдвое лучше уже существующей, то лучше решать проблему постепенно, мелкими шагами.

Иногда ответ очевиден. Если существующая архитектура совершенно не способна справляться с текущими задачами, то вам точно нужно что-то с ней делать. Например, нужно переработать серверный код, чтобы поддерживать установленные новым законом требования к конфиденциальности. Эти требования включают «право на забвение», подразумевающее возможность удаления старых данных конкретного пользователя. Но текущая архитектура физически не поддерживает такую возможность, потому что смешивает данные так, что удалить какую-то отдельную запись из них невозможно.

Будет ли новая система в два раза лучше? В этом случае — да, и даже более того: новая версия сможет делать то, чего старая не могла вовсе. В каком-то смысле новая система будет бесконечно лучше старой. А так как «бесконечно» уж точно больше, чем «в два раза», решение очевидно — нужно менять систему.

Однако чаще ответ не так очевиден, так что приходится прибегать либо к измерениям (если это возможно), либо хотя бы к субъективным оценочным суждениям (при невозможности измерений). В таких случаях остается только примерно прикидывать, насколько новая система будет лучше старой.

Так, например, когда наша студия Sucker Punch приступила к работе над игрой Ghost of Tsushima, стало ясно, что существующий способ физического моделирования земли не справится с масштабом новой игры. В предыдущих проектах поверхность земли изображалась как набор вручную создаваемых

треугольников, но карта новой игры была в сорок раз больше. Поверхность острова Цусима создавалась из набора карт высот, каждая из которых представляла собой битовую карту 512 × 512, описывающую равномерную сетку высот на участке в 200 квадратных метров.

Частичное улучшение заключалось в том, чтобы преобразовывать эти карты высот в треугольники, прогоняя их затем через стандартный физический конвейер. Решение сработало, но оказалось слишком громоздким — всего получалось около полумиллиона треугольников, и даже после некоторой оптимизации только для хранения этой структуры требовалось несколько мегабайт памяти.

Альтернативным вариантом была переработка физического движка таким образом, чтобы он напрямую поддерживал карты высот. Но это означало большой объем работ: пришлось бы реализовывать взаимодействие карт высот со всеми остальными базовыми физическими примитивами, продумывать способ кодирования дополнительной информации о типах поверхности для физического движка, добавлять поддержку карт высот во все наши отладочные инструменты и т. п. В целом на это ушло бы около трех человеко-месяцев разработки.

Но стоило только задать практический вопрос: «Станет ли новая система вдвое лучше?», как сразу по нескольким параметрам напрашивался однозначный ответ: «Да!» Карты высот уже использовались в графическом движке, и из них можно было получить все необходимые данные. Вместо двадцати с лишним мегабайт для физического моделирования каждого участка в двести квадратных метров понадобилось бы всего несколько сотен байт для интеграции карты высот*.

Кроме того, базовые физические операции (например, проверка пересечения короткого отрезка) в новой системе требовали бы обращения всего к одной-двум ячейкам карты высот, а не к десяткам уровней бинарного дерева пространственного разбиения, которое применялось для представления произвольной геометрии. Это не просто улучшение в два раза, а гораздо более значительное.

Итак, правило «вдвое лучше» доказало оправданность крупной переделки. Новая система значительно превосходила старую по важным показателям, и мы решили, что стоит заплатить цену за ее реализацию, за переход на новые рабочие процессы и за новые баги.

---

* По целому ряду сложных причин мы в итоге сделали отдельную копию карты высот для рендеринга и при копировании преобразовали ее в формат с плавающей точкой. Общий объем данных при этом составил около одного мегабайта на каждый участок размером в 200 квадратных метров.

Когда преимущества трудно измерить

Пользу от радикальной переделки далеко не всегда можно выразить в точных цифрах, но это не повод отказываться от правила «вдвое лучше». Если речь идет о нематериальных улучшениях — например, если после изменений программистам станет легче работать или если новые инструменты позволят дизайнерам создавать более удобный и приятный для пользователей интерфейс, — то подумайте о том, как найти способы оценки таких улучшений.

В противном случае вы рискуете принять самое комфортное для себя решение, то есть соответствующее вашим естественным склонностям. А если слишком часто выбирать комфортный вариант, проблемы не заставят себя ждать.

Допустим, цель каких-то изменений — просто облегчить работу программистам, как в примере с заменой устаревшего кода. Но почему им станет легче? В чем измеряется их удовлетворение от работы? Может, они станут продуктивнее, потому что перестанут бороться с багами, неизбежно возникающими из-за текущей путаницы в коде? Если так, то насколько продуктивнее? В два раза?

Или, скажем, вы раздумываете над тем, стоит ли перестраивать систему пользовательского интерфейса. Насколько лучше будут новые инструменты интерфейса? Почему пользователям вашей продукции будет удобнее ими пользоваться и как оценить разницу? Будут ли пользователи проводить больше времени, получая приятные впечатления от вашего продукта, что и является целью игр? Или, если вы работаете над более традиционным программным обеспечением, смогут ли пользователи выполнять свои задачи быстрее и эффективнее?

Переделка — хороший способ исправления мелких проблем

Прежде чем возвращаться к главам с реальным кодом, обозначу еще одну мысль.

Раз уж вы решили переработать всю систему, стоит заодно привести в порядок и все мелкие недочеты. Вряд ли стоит затевать крупномасштабную переделку только ради замены куска устаревшего кода — выгода от этого не стоит тех проблем, которые вы неизбежно создадите. Но если вы уж решили переписать этот участок кода, то вы готовы к возможным затратам, и в любом случае будете тщательно тестировать результаты, так что заодно можно было бы устранить несколько мелких проблем — например, заменить устаревший код в одном фрагменте на более современный.

Это довольно продуктивный подход. Держите на заметке мелкие проблемы в своем коде, пока не подвернется случай устранить их в ходе более крупной переработки общей системы.

Конечно, это вовсе не значит, что мелкие правки не нужны. Занимайтесь ими по мере возможностей. Просто со временем мелких вопросов может накопиться столько, что можно будет оправдать более серьезную переделку. Такое бывает достаточно часто, особенно когда начинают проявляться какие-то повторяющиеся схемы. Если вы заметили, что с полдесятка мелких проблем в текущей системе можно устранить одним улучшением, то вы нашли ту самую «точку перелома», оправдывающую затраты на улучшение. В своей совокупности мелкие проблемы становятся более значимыми и оправдывают крупные переделки.

«Вдвое лучше» — это удобное правило, говорящее о том, что серьезные изменения должны оправдываться серьезными улучшениями. Не ломайте систему только ради какой-то незначительной выгоды, это плохая стратегия. Ломайте систему только тогда, когда новое решение значительно лучше старого. Вдвое лучше.

Правило 12

Большим командам — строгие правила

Основная идея этой книги заключается в том, что программирование — само по себе сложное занятие, и вашу продуктивность, как индивидуальную, так и коллективную, ограничивает его сложность. Чем сложнее система, над которой вы работаете, — или чем сложнее вы ее делаете, — тем меньше у вас шансов на успех. Чем проще система, чем проще ее делать, тем больше шансов на успех, так что старайтесь сохранять простоту!

Совет этот справедлив для любого проекта, но для некоторых проектов он особенно актуален.

Бывают достаточно небольшие и простые проекты, сложность которых не играет особой роли. Если вы работаете один и пишете в день около сотни строк кода, который потом все равно придется заменить или выбросить, то писать этот код можно практически как угодно — сойдет и так, как говорится.

Но в команде, пусть даже из двух человек, это правило уже работает не так. Можно попытаться провести некую границу и сказать: «Над этим куском работаешь ты, а над этим буду работать я!» Но это плохое решение. Если эти две части не разграничены концептуально, если вы выделили их, скорее, просто из-за того, что не хотите мешать друг другу, то вам придется регулярно пересекать эту границу. Проблемой станет даже интерфейс между двумя кусками — например, кто будет решать, как именовать элементы интерфейса, если одна часть его на одной стороне, а другая на другой?

По-хорошему разграничить сферу ответственности в больших командах нельзя, таких способов не существует, но люди часто пытаются это сделать. Некоторым программистам нравится, когда все решения о «своем» коде принимают исключительно они сами. Очень легко сказать, что программирование — это творческий процесс, и что ограничения мешают творчеству. Легко утверждать, что для разных частей проекта разные требования и потому они должны разрабатываться по-разному. Легко привыкнуть к какому-то своему стилю кода и в результате с жаром спорить по поводу места каждой фигурной скобки.

Все эти аргументы ошибочны. Не полностью, но любая крупица истины в них перевешивается реальностью работы в большой команде над сложным проектом. Различия в стиле кодирования только усложняют работу каждого.

Форматирование кода

Каждый фрагмент кода отражает определенный стиль и свою философию. Разбираться в коде, написанном в чужом стиле или с чужой философией, труднее, и так проще допускать ошибки. Это все равно что читать книгу на иностранном языке, который вы понимаете лишь частично; все дается с трудом.

Если вы привыкли к коду вроде этого:

```
/// |struct TREE
/// |brief Бинарное дерево целых чисел
/// |var TREE::l
/// Левое поддерево
/// |var TREE::r
/// Правое поддерево
/// |var TREE::n
/// Данные

/// |fn sum(Tree * t)
/// |brief Возвращает сумму всех целых чисел дерева
/// |param t Корень дерева(или поддерева) для суммирования
/// |возвращает Sum сумму целых чисел дерева

struct TREE { typedef TREE self; self * l; self * r; int n; };
int sum(TREE * t) { return (!t) ? 0 : t->n + sum(t->l) + sum(t->r); }
```

то код вроде следующего будет для вас сложнее для чтения, и наоборот:

```
// Узел дерева целых чисел
struct STree
{
    STree *     m_leftTree;
    STree *     m_rightTree;
    int         m_data;
};
```

```
// Возвращает сумму всех целых чисел дерева

int sumTree(STree * tree)
{
    if (!tree)
        return 0;

    return tree->m_data +
           sumTree(tree->m_leftTree) +
           sumTree(tree->m_rightTree);
}
```

Я не выношу здесь никаких своих личных оценок — в каком-то смысле это один и тот же код; разница только в именах и форматировании. Я, допустим, привык ко второму стилю, и поэтому первый кажется мне странным. Для его чтения и перевода в привычный мне формат мне приходится прикладывать определенное мысленное усилие. Если же вы привыкли к первому стилю, ваша реакция будет прямо противоположной.

Проблема здесь не в самом стиле кодирования, а в том, что иногда эти стили смешиваются. Если вы не придерживаетесь какого-то одного стиля, а смешиваете их между собой, вам будет тяжело переключаться между ними. Попытка поддерживать разные стили для разных частей кода — плохая идея.

Языковые соглашения

То же самое относится и к языковым конструкциям. Если вы привыкли к «базовому» коду на C++, как в предыдущем примере, то более «современный» C++, вроде последующего примера, будете воспринимать с трудом:

```
int sumTree(const Tree * tree) {
    int sum = 0;
    visitInOrder(tree, [&sum](const Tree * tree) {
        sum += tree->m_data;
    });
    return sum;
}
```

Те, кто привык к более старым версиям C++, могут даже сказать, что это некорректный код! Здесь присутствует определение лямбда-функции (фрагмент, начинающийся с [&sum]), а лямбда-функции были добавлены только в C++ 11.

Повторюсь, я здесь не выношу оценок тем или иным конструкциям. Лямбда-функции бывают полезны, и я понимаю, почему они в C++ реализованы именно таким образом. Если вы в своем проекте со своей командой регулярно пользуетесь лямбда-функциями и вы все имеете четкое представление о том, как и где ими пользоваться, то в показанном выше примере кода нет ничего плохого. Но если вы в команде единственный сторонник лямбда-функций, то этот фрагмент грозит катастрофой.

Опять же, проблема здесь не в том, какие возможности и какие конструкции языка использовать в том или ином случае, а в том, что они часто смешиваются. Если вы привыкли к одному набору конструкций, то вам будет тяжело перестроиться на другой набор. Особенно верно это в том случае, когда переходить с одного стиля на другой приходится неожиданно и когда вы точно не знаете, какие языковые соглашения приняты для того кода, который вы рассматриваете.

Соглашения для решения задач

У программистов редко бывают задачи с единственным решением, поэтому у каждого со временем вырабатываются собственные методы решения той или иной задачи при написании кода. Когда ваш подход не совпадает с подходом коллег, когда вы одну и ту же задачу решаете разными способами, возникают определенные трудности. В лучшем случае увеличивается когнитивная нагрузка при чтении чужого кода. Чаще же все заканчивается изобретением велосипеда: все по очереди решают одну и ту же задачу несколькими способами, не осознавая, что она уже была решена.

Возьмем, к примеру, обработку ошибок. Существует множество способов ее организации: некоторые способы заложены в самом языке и стандартных библиотеках, а другие способы команды разрабатывают под свои нужды. Даже если ограничиваться встроенными средствами C++, можно насчитать как минимум три различных модели обработки ошибок, каждая из которых когда-то доминировала на определенном этапе полувековой эволюции языка.

Трактовать по-разному можно даже само понятие ошибки (error)! С одной стороны, можно вполне разумно считать ошибкой так называемые «ошибки использования» — именно так поступают операционная система и большинство библиотек. Но можно считать ошибками только неизбежные проблемы

и сбои, вроде отсутствующего файла, выделяя логические погрешности в отдельную категорию (mistakes).

Мы в Sucker Punch, например, рассматриваем ошибки использования как утверждения (assert), а не как ошибки (errors). Таково наше соглашение. Из множества допустимых подходов к обработке ошибок мы выбрали один и всегда его придерживаемся.

Придерживаться единых соглашений бывает сложно, хотя бы потому, что каждая библиотека или зависимость подразумевает свою модель обработки ошибок. Как минимум приходится иметь дело с ошибками, возвращаемыми библиотекой, после чего нужно решать, как поступать с ними дальше. Если вы придерживаетесь старомодного подхода к обработке файлов в стиле C, то он может привести к довольно неуклюжему коду:

```
string getFileAsString(string fileName)
{
    errno = 0;
    string s;
    FILE * file = fopen(fileName.c_str(), "r");
    if (file)
    {
        while (true)
        {
            int c = getc(file);
            if (c == EOF)
            {
                if (ferror(file))
                    s.clear();
                break;
            }
            s += c;
        }
        fclose(file);
    }
    return s;
}
```

В этом коде, написанном в стиле 1980-х годов, ошибки возвращаются комбинацией глобального состояния и специальных значений возврата. Подробности не так важны, но бросается в глаза отсутствие единых соглашений: каждая функция сообщает об ошибке по-своему: **fopen** при сбое возвращает

nullptr, **getc** возвращает **EOF**, но при этом устанавливает глобальный флаг и т. д. При работе с такой моделью нужно запоминать множество произвольных деталей.

Ситуацию немного улучшает перенос обработки ошибок внутрь самих объектов; при этом можно ввести новые, более строгие соглашения:

```
bool tryGetFileAsString(string fileName, string * result)
{
    ifstream file;
    file.open(fileName.c_str(), ifstream::in);
    if (!file.good())
    {
        log(
            "failed to open file %s: %s",
            fileName.c_str(),
            strerror(errno));

        return false;
    }

    string s;
    while (true)
    {
        char c = file.get();
        if (c == EOF)
        {
            if (file.bad())
            {
                log(
                    "error reading file %s: %s",
                    fileName.c_str(),
                    strerror(errno));
                return false;
            }

            break;
        }
        s += c;
    }
```

```
    *result = s;
    return true;
}
```

Соглашение здесь такое: функции, имена которых начинаются с **try**, могут завершиться неудачей. Они возвращают **true** при успешном выполнении и **false** при сбое, а все подробности о сбое записываются в системный журнал ошибок. Видя функцию с префиксом **try**, сразу понимаешь, чего от нее ожидать. В этом и состоит сила соглашений — они делают код понятнее и освобождают от необходимости выяснять детали при разборе реализации. Как и в данном примере, для любой библиотеки, не соответствующей принятому соглашению, приходится писать дополнительный код для согласования, но это работа для одного программиста, тогда как вся остальная команда пользуется единым и понятным подходом к обработке ошибок.

Мне доводилось работать над проектами, где вместо простого булева значения для обозначения успеха или неудачи был определен особый, более подробный тип ошибки:

```
struct Result
{
    Result(ErrorCode errorCode);
    Result(const char * format, …);
    operator bool () const
    { return m_errorCode == ErrorCode::None; }

    ErrorCode m_errorCode;
    string m_error;
};
```

Такой способ обработки ошибок позволяет передавать их дальше по коду с сохранением подробной контекстной информации. Это полезная модель для проектов, в которых приходится иметь дело с большим количеством ошибок, особенно если сочетать ее с соглашением об именовании (вроде **try**):

```
Result tryGetFileAsString(string fileName, string * result)
{
    result->clear();

    ifstream file;
    file.open(fileName.c_str(), ifstream::in);
```

```
    if (!file.good())
    {
        return Result(
            "failed to open file %s: %s",
            fileName.c_str(),
            strerror(errno));
    }

    string s;
    while (true)
    {
        int c = file.get();
        if (c == EOF)
            break;

        if (file.bad())
            return Result(
                        "error reading file %s: %s",
                        fileName.c_str(),
                        strerror(errno));
        s += c;
    }
    *result = s;
    return ErrorCode::None;
}
```

Можно использовать и исключения — третий базовый механизм обработки ошибок, предусмотренный в библиотеках C++:

```
string getFileAsString(string fileName)
{
    ifstream file;
    file.exceptions(ifstream::failbit | ifstream::badbit);
    file.open(fileName.c_str(), ifstream::in);

    string s;
    file.exceptions(ifstream::badbit);
    while (true)
    {
        int c = file.get();
```

```
        if (c == EOF)
            break;
        s += c;
    }

    return s;
}
```

Хорошо это или плохо, но такая функция скрывает сами исключения, которые выбрасываются при вызовах **file.open** или **file.get**. Преимущество этого подхода заключается в том, что основной поток выполнения не перегружается логикой обработки ошибок; недостаток же в том, что сама сложность обработки ошибок оказывается скрытой и рассредоточенной по множеству функций*.

Все четыре приведенные модели в целом работоспособны, как и многие другие. В качестве модели для команды можно выбрать любую из них — ну, пожалуй, кроме первой, уже совершенно устаревшей. А вот любая из оставшихся трех послужит разумным выбором в зависимости от особенностей проекта.

Что точно не имеет смысла — так это смешивать разные модели обработки ошибок в одном проекте. Несогласованные правила будут постоянно вносить в код путаницу, а не определившиеся с ними программисты будут постоянно допускать ошибки.

Я специально спрятал пример подобной несогласованности в двух вариантах функций с префиксом **try**. Обе возвращают «реальное» значение результата через указатель. Но в первом случае функция, если она завершается ошибкой, не трогает значение, а во втором она очищает его при сбое.

Можно привести убедительные аргументы в пользу любого из этих подходов, но нельзя использовать их вперемешку в одном проекте, потому что это гарантированно приведет к катастрофе. То же самое относится и к смешению различных исключений с кодами ошибок — в таком случае жди беды.

Вы могли бы возразить и сказать, что в программе вообще не должно быть обработки ошибок. Да, именно так — никакой обработки ошибок! В общем-то, именно такого подхода мы и придерживаемся в коде большинства наших игр: мы просто не определяем ошибки, а значит, и обрабатывать нечего.

---

* При знакомстве с другими Правилами этой книги — в частности, с Правилом 10 «Локализуйте сложность» — вы не должны удивляться тому, что в студии Sucker Punch исключения C++ не используются, и именно по этой причине. В кодовой базе имеется несколько **try**, но только там, где этого требуют внешние библиотеки инструментальной цепочки.

Ошибки использования (логические) обрабатываются с помощью `assert`. Катастрофические сбои, вроде нехватки памяти, просто приводят к остановке игры. В пограничных случаях вместо возврата ошибок выполняется поведение по умолчанию.

Разумеется, на стыках кода от ошибок никуда не деться — например, сетевой модуль должен как-то обрабатывать потерю пакетов. Но программисты Sucker Punch могут месяцами писать код, ни разу не создавая и не обрабатывая ни одной ошибки.

В эффективной команде все думают одинаково

Цель вашей команды (не отдельных людей, а именно команды) должна заключаться в том, чтобы думать как единое целое. В идеальном случае каждый участник настолько синхронизирован с остальными, что при решении конкретной задачи все теоретически могли бы написать абсолютно одинаковый код. Под словом «абсолютно» я имею в виду «совершенно одинаковый», с одинаковыми алгоритмами, с одинаковым форматированием, с одинаковыми именами для всего.

Все знают, что читать свой код и работать с ним проще, чем с чужим. В любом фрагменте кода неявным образом заложены предположения о том, как нужно писать код. Когда читаешь свой собственный код, эти предположения кажутся естественными, и на них не обращаешь внимания. При чтении же чужого кода спотыкаешься о каждое допущение, не совпадающее с твоим.

Когда соглашения об именовании не совпадают с привычными, читать код становится труднее. Конечно, можно разобраться во всем, как и привыкнуть ко всему, но для этого потребуются время и усилия. То же самое касается расположения скобок в непривычных для вас местах, непривычных для вас языковых конструкций или непривычных способов решения типовых задач (как в примере с конструкторами выше).

Решение очевидно. Если вы хотите, чтобы работа команды шла гладко и слаженно, — согласуйте свои правила! Пользуйтесь одними и теми же условностями! Не делайте ничего вразнобой!

При этом сами по себе конкретные правила или соглашения — например, о том, где ставить фигурную скобку, — почти никогда не играют особой роли. Да, по поводу размещения скобок можно устроить целую дискуссию, но существует множество совершенно приемлемых вариантов. Конкретный стиль оформления не имеет значения, но важно, чтобы вы все выбрали один и тот же стиль и последовательно его придерживались.

У нас в Sucker Punch мы решаем этот вопрос следующим образом. У нас имеется определенный набор стандартов кодирования с довольно строгими правилами для всего, что только можно придумать, включая:

- Правила именования всех сущностей, от переменных до классов.
- Форматирование кода. Если вы достаточно умны, то ваши правила форматирования должны в точности совпадать с форматом вывода какого-нибудь одного из множества инструментов автоматического форматирования. Поддерживать единый стиль в таком случае будет легко — достаточно просто запустить средство форматирования*.
- Языковые конструкции, в том числе какие используются, а какие нет.
- Типовые задачи, которые нам приходится решать. Например, у нас приняты строгие стандарты для реализации конечных автоматов, так как в игровом коде их бывает очень много.
- Границы между файлами, порядок их группировки, группировка кода внутри каждого файла.
- Представление констант в коде. Недостаточно просто заявить, что вместо «магических чисел» нужно использовать `#define` или `const`. Важную роль играют имена и место их определения — рядом с местом использования, в начале файла вместе с другими константами или же заголовочный файл всего проекта.

Этим соглашениям следуют все, и на их соблюдении мягко, но неукоснительно настаивают во время ревизии кода. Многим новичкам поначалу бывает непривычно работать в таких жестких рамках, но вскоре преимущества строгого следования стандартам становятся очевидными.

В начале каждого проекта любой участник команды может внести предложение о внесении изменений в правила написания кода. Каждое предложение обсуждается, после чего проводится голосование. Если предложение получает большинство голосов, оно вступает в силу для нового проекта. Например, в последнем голосовании мы приняли правило об использовании ключевого слова `auto` в некоторых случаях, что, в зависимости от личных предпочтений, может показаться либо чрезмерно строгим, либо, наоборот, слишком либеральным правилом**.

После утверждения всех изменений мы делим наш довольно большой код на части и прочесываем его всем составом, словно рой прожорливой саранчи по полю, приводя его к единому стандарту. На это тратятся определенные ресурсы, но такой процесс занимает меньше недели, а в итоге мы получаем

---

* Мы, к сожалению, не настолько умны. Наши правила форматирования довольно… своеобразные.

** Вот почему нужны общие правила.

чистое, полностью соответствующее всем командным правилам и соглашениям, кодовое пространство.

Помните нашу цель: любой программист студии Sucker Punch, столкнувшийся с конкретной задачей, должен написать точно такое же решение, как любой другой программист команды. Чем ближе мы к такому идеальному случаю, тем ближе та идеальная ситуация, когда работать с чужим кодом так же легко, как и с собственным. Если я смотрю на какой-то фрагмент кода студии Sucker Punch и не могу точно сказать, кто его написал, значит мы приближаемся к этой цели и создаем условия для спокойной работы без стресса*.

* Ну ладно, не совсем, но почти без стресса.

Правило 13

Ищите запустивший лавину камушек

Если я скажу, что программирование — это на самом деле отладка, вы, вероятно, печально покачаете головой и пробормочете нечто вроде: «Ну да, это правда, дружище, это правда...»

Вообще-то никто так не говорит, но вы, скорее всего, готовы согласиться с таким определением. Когда какую-то идею нужно довести до рабочей реализации, почти всегда времени на «заставить это работать» уходит несоизмеримо больше, чем времени на просто «набрать код». За редкими исключениями (сочетание простоты идеи и невероятной удачи) вы больше времени будете проводить за отладкой, чем собственно за написанием кода. Это настолько очевидно, что даже необязательно об этом говорить.

Но тут есть один нюанс.

Как очевидное предположение о том, что программирование — это на самом деле отладка, влияет на ваши действия? Как вы решаете поступать, зная о том, что вам неизбежно придется заниматься отладкой? Как пишете код в свете этого знания?

Очевидный ответ — писать код с меньшим количеством ошибок. Именно этому посвящена остальная часть книги, так что отложим пока эту тему. Текущее Правило посвящено немного другому: как писать код, удобный для отладки.

Жизненный цикл бага

Давайте отвлечемся и подумаем, что же такое, собственно, отладка. В жизненном цикле бага выделяются четыре основных этапа.

1. Баг обнаружен — вы выявили проблему.
2. Затем он диагностируется — вы исследуете ситуацию и выясняете, чем именно вызвано некорректное поведение.
3. Потом вы исправляете баг, меняя реализацию так, чтобы устранить ошибочное поведение.

4. Наконец, вы тестируете код, чтобы убедиться в том, что баг действительно исправлен и что исправление не вызвало новых проблем, после чего фиксируете изменения (отправляете «коммит»).

Фаза диагностики часто самая длительная и самая раздражающая. Дело в том, что большинство багов не сопровождаются какими-то указаниями на детали: программа просто вылетает, кнопка «OK» в диалоге почему-то все время отключена, в списке пользователей у четверти записей перепутаны имя и фамилия. Если повезет, отчет о баге будет содержать некоторый контекст — например сообщение о том, что делал пользователь перед вылетом программы.

Но из сообщения об ошибке вы не узнаете ее причин. Что породило симптомы? Что именно пошло не так? Диагностика — это процесс ответов на подобные вопросы. А пока вы не поймете, что пошло не так, вы не сможете исправить проблему.

Здесь здорово помогает тот факт, что компьютеры детерминированы. Если воссоздать точно ту же ситуацию, компьютер выдаст тот же самый результат. Если результат отличается, то ситуация была воспроизведена не совсем точно.

Заниматься диагностикой багов было бы легко, если бы можно было путешествовать назад во времени. Тогда задача была бы проще некуда: вернуться в точку за мгновение до того, как все пошло не так, и пройтись по коду шаг за шагом. А если случайно проскочил момент сбоя или начал не с той точки — не беда, включил снова машину времени и возвращаешься чуть дальше назад.

Разумеется, на самом деле путешествовать во времени невозможно, а даже если бы и было возможно, то у вас в прошлом, пожалуй, нашлись бы дела поважнее, чем исправлять баги в коде. Вместо этого можно сымитировать путешествие во времени: вернуть код в ту же ситуацию, которая вызывает проблему, и войти в отладчик прямо перед тем, как все пойдет наперекосяк.

Между ошибкой в коде и последствиями этой ошибки для выполнения программы обычно наблюдается некий разрыв во времени, поэтому понять, с какого именно места в отладчике нужно начинать, — это своего рода магия. Если очень повезет, сама проблема и ее проявления оказываются в одном месте:

```
void showObjectDetails(const Character * character)
{
    trace(
        "character %s [%s] %s",
        (character) ? character->name() : "",
```

```
        character,
        (character->sourceFile()) ? character->sourceFile() : "");
}
```

Здесь легко диагностировать вылет из-за нулевого объекта. Симптом (вылет) происходит из-за той же строки, в которой находится реальная проблема (мы проверяем **character** на **null**, что подразумевает поддержку null-объектов, но две строки спустя разыменовываем **character** без дополнительной проверки на **null**). Без разрыва между симптомом и проблемой диагностировать баг легко.

При чуть меньшем везении симптом и проблема будут находиться по соседству:

```
int calculateHighestCharacterPriority()
{
    Character * bestCharacter = nullptr;

    for (Character * character : g_allCharacters)
    {
        if (!bestCharacter ||
            character->priority() > bestCharacter->priority())
        {
            bestCharacter = character;
        }
    }

    return bestCharacter->priority();
}
```

Еще один вылет из-за нулевого объекта, который произойдет, если вызывать **calculateHighestCharacterPriority** при отсутствии персонажей. Симптом (вылет из-за того, что значение **bestCharacter** остается **null**) отделен от проблемы несколькими строками, и логика предыдущего цикла не учитывает того, что список персонажей может быть пустым.

Здесь мы впервые приближаемся к самому процессу диагностики бага. Ранее мы говорили, что если бы можно было перемотать время до момента, когда все пошло наперекосяк, диагностика была бы простой. Это правда, и именно этим мы во многом занимаемся при диагностике, но редко когда удается за один шаг откатиться прямо к изначальной причине — к той самой точке, где все началось.

Чаще всего ошибка проявляется постепенно. Вместо того чтобы прыгать сразу к исходной точке ошибки, вы двигаетесь назад шаг за шагом. Находите нечто, что выглядит неправильно, затем снова шагаете назад, чтобы понять, когда это начало выглядеть неправильно. Часто при этом вы замечаете другую странность, и процесс повторяется (еще шаг назад, еще одна проверка), пока вы не натыкаетесь на тот самый камешек, который и запустил всю лавину, приведшую к наблюдаемому симптому. Вот что такое диагностика бага.

Я понимаю, если попытка разложить процесс отладки на этапы кажется вам бесполезной. У вас уже имеется какое-то представление об отладке, ведь вы же программист, а программирование — это отладка. Значит, вы уже занимались отладкой кода. Зачем столько усилий для описания процесса, который и так очевиден?

Дело в том, что наша цель — сделать отладку проще, а для этого нужно хорошо представлять себе, что же такое отладка.

Если определить процесс отладки как процесс пошагового движения назад во времени и восстановления цепочки сбоев, которые в итоге привели к проявившемуся симптому, то облегчить отладку можно, упростив это обратное движение. В конце концов мы двигаемся обратно к тому самому камешку, который запустил лавину, и именно там нужно исправлять ошибку. Чем легче двигаться назад, тем больше шансов дойти по цепочке причинно-следственных связей до источника ошибки.

Но вот в чем дело с лавинами: добираться до самого первого камешка необязательно. Можно просто устранить симптом, не пытаясь проследить его причину. Во втором примере после сбоя можно просто добавить проверку на нулевой указатель и тем самым устранить сбой функции при отсутствии персонажей:

```
int calculateHighestCharacterPriority()
{
    Character * bestCharacter = nullptr;

    for (Character * character : g_allCharacters)
    {
        if (!bestCharacter ||
            character->priority() > bestCharacter->priority())
        {
            bestCharacter = character;
        }
    }
```

```
    return (bestCharacter) ? bestCharacter->priority() : 0;
}
```

Когда двигаться назад сложно, велик соблазн поступить именно так — устранить симптом, не разбираясь в его причине. Соблазн велик, потому что в каком-то смысле решение действительно работает: раньше программа вылетала, а теперь не вылетает. Работа вроде бы сделана.

Если бы мы чуть-чуть продвинулись назад по цепочке событий, то, вероятно, поняли бы, что правильнее избавиться от указателя **bestCharacter**:

```
int calculateHighestCharacterPriority()
{
    int highestPriority = 0;
    for (Character * character : g_allCharacters)
    {
        highestPriority = max(
                          highestPriority,
                          character->priority());
    }
    return highestPriority;
}
```

В большинстве случаев ошибки не настолько просты, как в этом примере, и если исправлять только симптомы, не отслеживая источник проблемы, исходная ошибка остается в коде, и она всегда может вызвать новую лавину.

В данном случае «камешек» — это указатель со значением **null**, но он получает это значение только в особом случае. Код не учитывает этот особый случай, и, скорее всего, рано или поздно мы снова его упустим. Лучше избавиться от «камешка», устранив сам указатель.

Искушение бороться с симптомом, а не с причиной, возникает на каждом этапе движения вверх по цепочке причинно-следственных связей. Шагая назад, от симптома к причине, от причины симптома к причине причины и т. д., в любой момент можно заглушить проблему и объявить о победе. В каком-то смысле это действительно победа, ведь симптом исчезает.

Но если объявить победу на середине пути, «камешек» останется на своем месте. Рано или поздно он вызовет новую лавину — и не важно, накроет ли она вас или кого-то другого. Чем проще шагать назад во времени, тем легче сопротивляться искушению подавлять симптомы, вместо того чтобы устра-

нять корень проблем. И тем проще удалять «камешки», а не преграждать путь лавинам*.

Минимизация состояния

Исходя из данного выше определения отладки, можно выделить несколько направлений для улучшения этого процесса:

Чем ближе симптомы к причинам, тем легче находить источник проблемы.

Если причина находится в исходном коде рядом или если какое-то событие, вызвавшее проблему, произошло недавно, установить связь между ними проще.

Чем короче цепочка причинно-следственных связей, тем короче процесс отладки.

Симптом, возникающий из-за одной причины, исправить легче, чем симптом, за которым тянется длинная цепочка других симптомов и причин.

Чем легче шагать назад, тем легче прослеживать связи каждого звена цепочки.

Если воспроизвести состояние, ставшее причиной каждого симптома, легко, то и исследовать всю цепочку связей становится проще.

Самая очевидная цель здесь — последняя. Слишком объемное состояние воспроизводить сложно. При сокращении количества состояний шагать назад по цепочке причинно-следственных связей будет проще.

Проще всего отлаживать проблему в чистой функции, то есть в функции, не имеющей побочных эффектов и зависящей только от своих входных данных. Если для некоторого набора входных данных функция возвращает неправильное значение, достаточно вызвать ее снова с теми же входными данными, и она вернет тот же результат. Повторять можно сколько угодно раз.

Допустим, мы вычисляем последовательность Фибоначчи, и в коде есть ошибка. Такая задача встречается разве что в тестовых заданиях и на собе-

---

* Как понять, что вы действительно нашли «камешек», а не очередной симптом? Если вы не уверены, почему или при каких условиях возникает проблема, значит, вы его еще не нашли и стоит продолжить расследование. Но не слишком озабочивайтесь поисками — любой шаг вверх, в сторону настоящего источника проблемы, приносит пользу.

седованиях, но все же потерпите немного. Вот вам код*. Согласно отчету об ошибке, функция **getFibonacci** возвращает неправильное значение:

```
int getFibonacci(int n)
{
    static vector<int> values = { 0, 1, 1, 2, 3, 5, 8, 13, 23, 34, 55
};
    for (int i = values.size(); i <= n; ++i)
    {
        values.push_back(values[i - 2] + values[i - 1]);
    }
    return values[n];
}
```

Это чистая функция, поэтому воспроизвести проблему легко. Она зависит от единственного состояния — аргумента функции. Вызывая **getFibonacci(8)**, мы каждый раз получаем один и тот же неверный результат: 23 вместо 21. Стоит лишь зайти внутрь функции, и причина ошибки становится очевидной: мы заранее вставили в массив **values** неправильное значение. Диагностика завершена.

Вот наш первый вывод: если строить код на основе чистых функций, воспроизводить состояние будет проще, а значит, и отлаживать его тоже будет проще.

Рассмотрим более сложный сценарий. Допустим, у нас есть метод **Character**, возвращающий значение «угрозы» (threat), зависящее от текущего оружия персонажа, брони, уровня здоровья, действующих эффектов состояния и т. д. Можно написать код, который будет поддерживать это значение угрозы как состояние персонажа:

```
struct Character
{
    void setArmor(Armor * armor)
    {
        m_threat -= m_armor->getThreat();
        m_threat += armor->getThreat();
         m_armor = armor;
    }
```

* Не самый лучший код для вычисления последовательности Фибоначчи. Не пользуйтесь им для выполнения тестовых заданий.

```
    void setWeapon(Weapon * weapon)
    {
        m_threat -= weapon->getThreat();
        m_threat += weapon->getThreat();
        m_weapon = weapon;
    }

    void setHitPoint(float hitPoints)
    {
        m_threat -= getThreatFromHitPoints(m_hitPoints);
        m_threat += getThreatFromHitPoints(hitPoints);
        m_hitPoints = hitPoints;
    }

    int getThreat() const
    {
        return m_threat;
    }

protected:
    int m_threat;
    Armor * m_armor;
    Weapon * m_weapon;
    float m_hitPoints;
};
```

В этом коде есть ошибка, которая проявляется, например, так: «Игрок не реагирует на врага с Мечом Жестоких Ран (+1 к урону)». К счастью, баг этот легко воспроизвести вручную: если подойти к противнику, держащему волшебный меч, персонаж игрока сохраняет анимацию покоя вместо того, чтобы принимать боевую стойку.

Разумеется, сам симптом проявляется не при выполнении этого фрагмента кода. В данном случае симптом — это сохранение анимации покоя вместо перехода к анимации готовности к бою. Мы уже проследили несколько шагов вверх по причинно-следственной цепочке и добрались до этого фрагмента кода, выяснив, что переменная **`m_threat`** содержит неверное значение.

Теперь нужно понять, почему оно неверное. Проделаем «магический трюк», то есть вернемся во времени и воспроизведем состояние, при котором **`m_threat`** было присвоено неправильное значение.

Это непросто. В отличие от предыдущих примеров, фрагмент кода не находится рядом с тем фрагментом, во время выполнения которого проявился симптом. Ошибка не «недавняя». В какой-то момент в прошлом переменной **m_threat** было присвоено неверное значение, но когда именно — непонятно.

Проблема кода с состоянием заключается в следующем: ошибка проявляется лишь спустя некоторое время после того, как что-то пошло не так, и этот разрыв между причиной и симптомом усложняет диагностику. В данном случае мы знаем, что **m_threat** имеет неверное значение, но не знаем, почему и когда оно стало таким.

Если следовать совету о функциях проверки, или функциях аудита (audit functions) из Правила 2, диагностика проблемы превращается в пустяковое дело. Для этого достаточно добавлять вызов функции проверки всякий раз, когда изменяется состояние персонажа:

```
struct Character
{
    void setWeapon(Weapon * weapon)
    {
        m_threat -= weapon->getThreat();
        m_threat += weapon->getThreat();
        m_weapon = weapon;
        audit();
    }

    void audit() const
    {
        int expectedThreat = m_armor->getThreat() +
                             m_weapon->getThreat() +
                             getThreatFromHitPoints(m_hitPoints);
        assert(m_threat == expectedThreat);
    }
};
```

Так функция **audit()** в конце выполнения **setWeapon** вызовет **assert**. И выяснится, что мы вычитаем значение угрозы старого оружия перед тем, как добавить значение угрозы нового. Неудивительно, что персонаж игрока оставался настолько хладнокровным.

Без функции проверки диагностика была бы совсем не пустяковой. Пришлось бы ставить точки останова на всех строках, где изменяется **m_threat**, запускать код и каждый раз при срабатывании точки останова проверять

состояние. Занятие это утомительное, и в данном случае совершенно лишнее: хранить **m_threat** как часть состояния вообще не следовало. Не добавляйте состояние, если в нем нет острой необходимости.

Для сравнения рассмотрим похожий баг в коде без состояний:

```
struct Character
{
    void setArmor(Armor * armor)
    {
        m_armor = armor;
    }

    void setWeapon(Weapon * weapon)
    {
        m_weapon = weapon;
    }

    void setHitPoint(float hitPoints)
    {
        m_hitPoints = hitPoints;
    }

    int getThreat() const
    {
        return m_armor->getThreat() -
               m_weapon->getThreat() +
               getThreatFromHitPoints(m_hitPoints);
    }

protected:
    Armor * m_armor;
    Weapon * m_weapon;
    float m_hitPoints;
};
```

Для кода без состояния можно продумать четкий план действий. Обнаружив, что неверное значение возвращает **Character::getThreat**, мы переходим к врагу с волшебным мечом и ставим точку останова в методе **getThreat**. Диагноз прост — в том месте, где должен быть плюс, по ошибке поставлен минус. Сокращение состояний значительно упростило отладку.

Полностью мы состояние из `Character` не устранили. Оставшееся состояние — это доспехи персонажа, оружие и текущее количество очков здоровья, то есть то, что, собственно, и определяет суть объекта `Character`. Это состояние нельзя исключить.

Во многих игровых системах, в которых мы моделируем реальные объекты с помощью их виртуальных аналогов, такие объекты неизбежно имеют какое-то состояние: положение в пространстве, скорость, текущее здоровье или сила магических камней, вставленных в его волшебный меч. Все это — состояние, и от него не так-то просто избавиться.

Но во многих других случаях можно обойтись и без состояния. Состояние усложняет отладку, а отладка, по сути, и есть программирование. Моделируйте поведение на основе чистых функций, насколько это возможно. Так проще поддерживать корректность различных деталей, а когда что-то пойдет не так, будет проще выявить причину

Когда состояние неизбежно

Когда в коде неизбежно должно быть состояние, диагностика проблем усложняется. Представьте, что в игре обнаружился следующий баг: персонажи реагируют на попадание в них стрелы не всегда так, как положено. Предполагается, что они должны отшатываться назад*, но иногда вместо этого шагают вперед.

Так, подумаем... в описании бага имеется замечательное слово «иногда». Оно подсказывает, что проблема, по всей видимости, связана с состоянием взаимодействующих объектов. Я бы предположил, что баг связан с персонажем, у которого больше всего состояний, хотя источником ошибки может быть и стрела. Для диагностики придется воспроизвести нужное состояние.

В некоторых случаях диагностика оказывается простой. Если баг проявляется в 100% модульных тестов, то, считайте, вам повезло. Модульный тест воссоздает то же самое состояние, которое приводит к неправильному поведению, а дальше все идет по прямой. Установите точку останова, срабатывающую в момент попадания в персонажа злополучной стрелы, и начинайте отладку. Возможно, вам придется пройти немного вверх по цепи причинно-следственных связей, чтобы найти запустивший лавину «камешек», но самую трудную часть каждого шага — воспроизведение состояния — за вас уже делает модульный тест.

---

* Это кинематографическая логика. У стрел не настолько большая энергия, и в реальности они могут отбросить назад разве что белку. Но такого поведения ожидают все игроки в видеоигры, поэтому приходится подчиняться их ожиданиям.

Но вам может повезти меньше. У вас нет автоматического теста, но после нескольких попыток вам удается воспроизвести проблему вручную, и вы определяете момент, в который она возникает.

В движке Sucker Punch есть объект, сопоставляющий каждую запись о полученном уроне с соответствующей реакцией — именно благодаря этому код определяет, что делает персонаж, когда в него втыкается стрела. Проблему можно выявить в этом коде сопоставления — для этого нужно просто добавить код, проверяющий совпадение направления вектора скорости стрелы и направления отклонения персонажа:

```
void DamageArbiter::getDamageReaction(
    const Damage * damage,
    Reaction * reaction) const
{
    // Здесь находится вся логика сопоставления повреждений реакциям.
    // В движке Sucker Punch это реализовано в одной функции длиной
    // почти в 3000 строк, и это, мягко говоря, не самый чистый
    // пример соблюдения Правил, хотя, надо признать, задача там
    // весьма нетривиальная.

    if (damage->isArrow())
    {
        assert(reaction->isStumble());
        Vector arrowVelocity = damage->impactVelocity();
        Vector stumbleDirection = reaction->stumbleDirection();
        assert(dotProduct(arrowVelocity, stumbleDirection) > 0.0f);
    }
}
```

Тот момент, когда срабатывает этот **assert**, можно считать хорошей точкой для диагностики проблемы.

Функция **getDamageReaction** относительно чиста — для любого урона **Damage** она возвращает одну и ту же реакцию **Reaction**, и у нее нет побочных эффектов, но она принимает решения на основе состояния произвольных объектов в игровом мире. Звучит пугающе — неужели для воспроизведения проблемы нам придется воспроизводить состояние каждого объекта в мире?

Вот почему так важно обнаружить проблему на ранней стадии, еще до выхода из **getDamageReaction**. Функция не имеет побочных эффектов, поэтому состояние всех объектов в мире не изменилось. Если немедленно вызвать **getDamageReaction** снова, мы должны получить тот же результат.

Когда-то давно для обработки подобных случаев я вставлял специальный код. Когда проблема обнаружена, можно войти в отладчик для пошагового выполнения и вызывать чистую функцию рекурсивно:

```
void DamageArbiter::getDamageReaction(
    const Damage * damage,
    Reaction * reaction) const
{
    // Здесь находится вся логика сопоставления повреждений реакциям.
    // В движке Sucker Punch это реализовано в одной функции длиной
    // почти в 3000 строк, и это, мягко говоря, не самый чистый
    // пример соблюдения Правил, хотя, надо признать, задача там
    // весьма нетривиальная.

    if (damage->isArrow())
    {
        assert(reaction->isStumble());
        Vector arrowVelocity = damage->impactVelocity();
        Vector stumbleDirection = reaction->stumbleDirection();
        if (dotProduct(arrowVelocity, stumbleDirection) <= 0.0f)
        {
            assert(false);

            static bool s_debugProblem = CHRISZ;
            if (s_debugProblem)
            {
                getDamageReaction(damage, reaction);
            }
        }
    }
}
```

Сейчас я больше импровизирую. В Sucker Punch мы пользуемся IDE, позволяющим задавать следующую строку для выполнения прямо с того момента, на котором программа остановилась в отладчике. Это, конечно, не лишено риска, так как произвольные переходы по коду могут вызвать новые проблемы, но при должной осторожности метод работает отлично. Наткнувшись на проблемный фрагмент, особенно если это чистая функция, я могу «отмотать» выполнение кода и выявить причину. Такой инструмент полностью меняет процесс диагностики, и шаг во времени назад, который раньше

было выполнить трудно, теперь выполняется легко. Если корень проблемы находится где-то рядом — в той же области кода и выполнился недавно, — то найти его легко.

Когда я говорил о том, что по возможности лучше избавляться от состояния, я не имел в виду радикальный принцип «все или ничего». Избавившись от состояния хотя бы на небольшом участке кода, вы значительно облегчите отладку — даже если на других участках состояние сохраняется. В этом помогает каждая мелочь.

Когда неизбежна задержка

Во всех рассмотренных выше примерах симптом можно было обнаружить механически. Вылет программы говорил о том, что проблема, так сказать, «обнаружила себя сама». Если код контролировал себя с помощью **assert**, то он сам и выявлял ошибки. В случаях вроде примера со стрелой мы замечали проблему во время ручного тестирования, но могли бы обнаружить и с помощью **assert** в коде.

Но так бывает не всегда, что осложняет диагностику. Иногда симптомы проблемы проявляются не сразу.

Приведу пример из практики Sucker Punch, связанный с отладкой проблем в сфере анимации. Персонажи в наших играх двигаются благодаря анимациям (эффектам анимации), которые создают наши аниматоры. Каждая анимация описывает движение частей тела персонажей, то есть изменение их положения со временем. Например, в момент времени 1,5 секунды левая рука находится в определенной точке и развернута назад; в момент времени 1,53 секунды рука находится чуть-чуть выше и развернулась немного вперёд. И так далее — всего мы отслеживаем около шестисот частей тела персонажа с частотой 60 раз в секунду.

Сама по себе отдельная анимация представляет собой чистую функцию, не зависящую от внешнего состояния и не имеющую побочных эффектов. Ее интересует только одно входное значение — точка времени внутри шкалы анимации. При повторном вычислении анимации для одного и того же момента времени мы каждый раз получаем одну и ту же позу тела.

Но все не так просто. При смене анимаций — например, когда бегущий персонаж решил прыгнуть — нельзя переключиться с одной анимации на другую мгновенно, иначе движение выйдет слишком резким и «рваным», что плохо. Вместо этого выполняется плавный переход между старой и новой анимацией.

Сглаживание усложняет задачу, потому что теперь анимация зависит не только от точки на шкале времени, но и от текущего положения частей тела

персонажа. Чтобы воспроизвести проблему с плавным переходом, наряду с моментом времени нужно знать положение и ориентацию всех примерно шестисот частей тела персонажа.

И это еще не все. Наш мозг хорошо улавливает сбои в анимации, но для осознания ошибки требуется некоторое время, тогда как анимация пересчитывается со скоростью 60 раз в секунду. К тому моменту, как мы замечаем, что что-то пошло не так, анимация уже была пересчитана десятки раз и вызвавшее сбой состояние пропало.

Решение существует, хотя и очень затратное. Плавное сглаживание анимаций действительно зависит от большого количества состояний, но, по крайней мере, все состояния можно определить. Если каждый раз при пересчете анимации сохранять общее состояние для определенного момента времени, то этими сохраненными состояниями можно впоследствии воспользоваться для диагностики проблем.

Собственно говоря, именно так мы и поступаем. Так как плавность анимации для нас — это безусловное требование, мы потратили немало ресурсов на инструменты отладки. Мы сохраняем все влияющее на анимацию состояние персонажа для каждого кадра, и у нас есть инструмент отладки, позволяющий перемещаться назад и вперед по недавним кадрам. Замечая неполадки с анимацией, мы останавливаем игру, запускаем инструмент отладки анимации, отматываем время до момента сбоя, а затем запускаем отладчик и отслеживаем путь от симптомов к причине вверх по цепочке причинно-следственных связей.

С помощью этого инструмента мы автоматизировали самую сложную часть отладки — воспроизведение состояния, приведшего к проблеме. Если ваш код зависит от состояния, но это состояние можно ограничить какими-то рамками, то можно значительно упростить отладку благодаря сохранению состояния.

Такую технику можно рассматривать и как создание исполняемого журнала событий — журнала, который не просто описывает то, что произошло, а содержит все данные для повторного воспроизведения тех же событий. Если система построена на основе чистых функций, то создать такой журнал вполне реально. Для этого достаточно сохранять все входные данные и иметь возможность их воспроизведения.

Реализовать такой журнал, конечно, непросто, но для отладки по-настоящему важных и непростых для отладки задач, таких как поддержание качества анимации, он стоит усилий.

Правило 14

Код бывает четырех типов

Здесь я описываю крайне упрощенную, но все же полезную модель восприятия кода. Представьте, что существуют всего два типа программных задач: Легкие и Трудные.

Наверняка вы уже имеете какое-то свое представление о том, что такое Легкая задача, но приведу несколько примеров: поиск наибольшего и наименьшего значения в массиве чисел; вставка узла в отсортированное двоичное дерево; удаление нечетных значений из массива.

Назвать Трудные задачи тоже легко: выделение памяти, например — реализация функций `malloc` и `free` из стандартной библиотеки C; разбор (парсинг) скриптового языка; написание решателя задач с линейными ограничениями.

Так вот, Легкие и Трудные задачи, как я их здесь называю, — это лишь две точки спектра сложности, и даже не самые крайние. Существуют задачи «тривиальные», то есть такие, какие даже проще приведенных примеров Легких задач. И существуют задачи гораздо сложнее приведенных примеров Трудных задач — например, создание с нуля файловой системы с журналированием.

Но эти две точки полезны для восприятия всего спектра задач. Большинство задач, с которыми сталкиваются программисты в своей повседневной практике, располагаются между Легкими и Трудными. Мне, кстати, приходилось писать решения для всех этих примеров, кроме создания с нуля файловой системы с журналированием, хотя звучит это интересно.

Казалось бы, очевидно, что для решения Трудной задачи нужно будет написать больше кода, чем для решения Легкой, и что код этот будет более сложным. Чаще всего так и бывает: расписывать решение Трудных задач обычно труднее, и код в итоге получается длиннее и сложнее, чем для решения Легких задач.

Отсюда мы приходим к еще одной упрощенной модели. На этот раз представим, что существуют два типа решений задач: Простые и Сложные. Простые решения короткие и легко понимаются. Сложные решения длинные и трудны для понимания. Опять же, это всего лишь две точки на спектре.

Понятно, что бывают решения Умеренно сложные или Относительно простые, но полезно рассматривать крайние точки: Простое и Сложное решение.

К этому моменту вы, как программист, наверняка уже догадались, как можно получить четыре типа кода, что и соответствует названию описываемого здесь Правила (таблица 14-1).

Таблица 14-1. Простые и Сложные решения для Легких и Трудных задач

	Легкая задача	Трудная задача
Простое решение	Ожидаемо	К чему нужно стремиться
Сложное решение	Очень, очень плохо	Приемлемо

Очевидно, что для Легких задач существуют Простые решения, а для Трудных задач — Сложные решения. Из личного опыта всем нам известно, что для Легкой задачи бывает на удивление легко написать Сложное решение. А в некоторых случаях можно написать Простое решение для Трудной задачи.

Согласно Правилу 1, нам хотелось бы находить как можно более простые решения для задач, так что, казалось бы, понятно, к чему я клоню, описывая текущее Правило. Но давайте рассмотрим несколько примеров.

Легкая задача, Простое решение

Начнем с примера Легкой задачи — найти наибольшее и наименьшее значения в массиве. Вот Простое решение этой Легкой задачи:

```
struct Bounds
{
    Bounds(int minValue, int maxValue)
    : m_minValue(minValue), m_maxValue(maxValue)
        { ; }

    int m_minValue;
    int m_maxValue;
};

Bounds findBounds(const vector<int> & values)
{
    int minValue = INT_MAX;
    int maxValue = INT_MIN;
```

```
    for (int value : values)
    {
        minValue = min(minValue, value);
        maxValue = max(maxValue, value);
    }

    return Bounds(minValue, maxValue);
}
```

Алгоритм прост — пройтись по всем значениям, отслеживая наибольшее и наименьшее. Небольшая тонкость заключается в том, с чего начать; я воспользовался довольно стандартным приемом, чтобы значение первого элемента было одновременно присвоено и **minValue**, и **maxValue**. В остальном код легко читать и понимать. Он адекватно прост.

Легкая задача, три Сложных решения

Нет ничего удивительного в том, чтобы взять тот же алгоритм и сделать на его основе гораздо более сложный код. Все мы видели примеры, когда простой алгоритм прячут под несколькими слоями абстракций, например:

```
enum EmptyTag
{
    kEmpty
};

template <typename T> T MinValue() { return 0; }
template <typename T> T MaxValue() { return 0; }

template <> int MinValue<int>() { return INT_MIN; }
template <> int MaxValue<int>() { return INT_MAX; }

template <class T>
struct Bounds
{
    Bounds(const T & value)
    : m_minValue(value), m_maxValue(value)
        { ; }
    Bounds(const T & minValue, const T & maxValue)
    : m_minValue(minValue), m_maxValue(maxValue)
```

```
        { ; }

    Bounds(EmptyTag)
    : m_minValue(MaxValue<T>()), m_maxValue(MinValue<T>())
        { ; }

    Bounds & operator |= (const T & value)
    {
        m_minValue = min(m_minValue, value);
        m_maxValue = max(m_maxValue, value);
        return *this;
    }

    T m_minValue;
    T m_maxValue;
};

template <class T>
struct Range
{
    Range(const T::iterator & begin, const T::iterator & end)
    : m_begin(begin), m_end(end)
        { ; }

    const T & begin() const
    { return m_begin; }

    const T & end() const
    { return m_end; }

    T m_begin;
    T m_end;
};

template <class T>
Range<typename vector<T>::iterator> getVectorRange(
    const vector<T> & values,
    int beginIndex,
    int endIndex)
{
```

```
    return Range<vector<T>::const_iterator>(
                values.begin() + beginIndex,
                values.begin() + endIndex);
}

template <class T, class I>
T iterateAndMerge(const T & init, const I & iterable)
{
    T merge(init);

    for (const auto & value : iterable)
    {
        merge |= value;
    }
    return merge;
}

void findBounds(const vector<int> & values, Bounds<int> * bounds)
{
    *bounds = iterateAndMerge(
                    Bounds<int>(kEmpty),
                    getVectorRange(values, 0, values.size()));
}
```

Это тот же самый алгоритм, хотя, чтобы убедиться в этом, придется немного покопаться в коде. По крайней мере, намерения у автора были благие. Здесь ничего вопиюще плохого нет, никаких странных трюков C++, и самое «хитрое» — это специализация шаблонов. Все имена осмысленные. При желании можно найти объяснение для каждой строки.

И все же... кода стало в четыре раза больше, и разобраться в нем куда труднее, чем в исходном простом варианте. Это Сложный код — по крайней мере, по сравнению с решаемой задачей. Предыдущий пример был разумно прост, а этот — неоправданно усложнен.

Конечно же, это только один из способов излишне усложнить решение. Всем нам знакомы примеры, когда код выходит за рамки поставленной задачи:

```
struct Bounds
{
```

```
    Bounds(int minValue, int maxValue)
    : m_minValue(minValue), m_maxValue(maxValue)
        { ; }

    int m_minValue;
    int m_maxValue;
};

template <class COMPARE>
int findNth(const vector<int> & values, int n)
{
    priority_queue<int, vector<int>, COMPARE> queue;
    COMPARE compare;

    for (int value : values)
    {
        if (queue.size() < n)
        {
            queue.push(value);
        }
        else if (compare(value, queue.top()))
        {
            queue.pop();
            queue.push(value);
        }
    }

    return queue.top();
}

void findBounds(const vector<int> & values, Bounds * bounds)
{
    bounds->m_minValue = findNth<less<int>>(values, 1);
    bounds->m_maxValue = findNth<greater<int>>(values, 1);
}
```

Здесь решается более общая задача — поиск n-го по возрастанию (или n-го по убыванию) числа в массиве, а затем с ее помощью решается поиск минимального и максимального значений как частный случай. Такой чрез-

мерно обобщенный подход почти всегда ошибочен. Да, кода добавилось не так уж много, и писать что-то «умное» вроде этого интереснее, чем простое решение, но читать такой код гораздо труднее*.

Наконец, можно усложнить задачу, выбрав неверный алгоритм. В данном случае выбрать неверный алгоритм трудно, так как простое решение лежит на поверхности, но всем нам знаком код, упускающий очевидный путь:

```
struct Bounds
{
    Bounds(int minValue, int maxValue)
    : m_minValue(minValue), m_maxValue(maxValue)
        { ; }

    int m_minValue;
    int m_maxValue;
};

int findExtreme(const vector<int> & values, int sign)
{
    for (int index = 0; index < values.size(); ++index)
    {
        for (int otherIndex = 0;; ++otherIndex)
        {
            if (otherIndex >= values.size())
                return values[index];

            if (sign * values[index] < sign * values[otherIndex])
                break;
        }
    }
    assert(false);
    return 0;
}

void findBounds(const vector<int> & values, Bounds * bounds)
```

---

* Его производительность ужасна, но так как в Правиле 5 я говорил, что не следует беспокоиться об оптимизации, то считаю своим долгом упомянуть об этом в сноске. В свою защиту отмечу, что самое простое решение здесь и самое быстрое, что не редкость.

```
{
    bounds->m_minValue = findExtreme(values, -1);
    bounds->m_maxValue = findExtreme(values, +1);
}
```

Итак, сделать решение сложнее, чем нужно, можно тремя типичными способами: чрезмерное абстрагирование, избыточное обобщение и выбор неправильного алгоритма.

Цена сложности

Сложность имеет вполне реальную цену. Писать сложный код приходится дольше, а его отладка занимает еще больше времени. Любому, кто читает такой код, приходится буквально пробираться сквозь нагромождение деталей, чтобы понять, что происходит. Простое решение лишено этих проблем: его легко реализовать с первого раза, и достаточно одного взгляда, чтобы понять, как оно работает, и что оно правильно.

По сути, именно этот фактор — решаете ли вы Легкие задачи Простыми способами — и отличает лучше всего посредственных программистов от хороших. На собеседованиях в Sucker Punch мы обращаем внимание на две вещи: способен ли кандидат решать Трудные задачи и находит ли он Простые решения для Легких задач. Если хотя бы на один из этих вопросов ответ отрицательный, нас такой человек не интересует.

Тот, кто пишет Сложные решения для Легких задач, не только усложняет жизнь себе, но и делает ее труднее для всей команды. Такие решения требуют больше времени на разработку, вносят больше ошибок в кодовую базу и становятся источником раздражения для остальных разработчиков, которым потом приходится с ними работать. Мы не можем себе этого позволить.

Четыре (на самом деле три) типа программистов

Отталкиваясь от четырех типов кода (Легкие и Трудные задачи, Простые и Сложные решения), можно выделить четыре типа программистов. Напишете ли вы Простое или Трудное решение, столкнувшись с Легкой задачей? А если задача Трудная, разработаете ли вы Простое решение или Сложное?

На деле программистов, которые пишут Простые решения для Трудных задач, но Сложные для Легких, почти не бывает. Поэтому остаются три типа, как показано в таблице 14-2.

Таблица 14-2. Три типа программистов

Типы программистов	Легкая задача	Трудная задача
Посредственный	Сложное решение	Сложное решение
Хороший	Простое решение	Сложное решение
Великолепный	Простое решение	Простое решение

Разница между посредственными и хорошими программистами в том, что хорошие пишут Простые решения для Легких задач. Разница между хорошими и великолепными в том, что великолепные, даже сталкиваясь с Трудными задачами, все равно находят Простое решение.

В какой-то момент задачи становятся настолько трудными, что простых решений уже не существует. Поэтому лучшая мера мастерства программиста — это то, насколько далеко он способен продвинуться по этой шкале, прежде чем его решения начнут усложняться. Чем более трудные задачи он умеет решать простыми средствами, тем он лучше как разработчик.

На это можно взглянуть и иначе: ключевое умение великолепного программиста заключается в том, что он умеет распознавать задачи, которые кажутся Трудными, но на самом деле Простые, если взглянуть на них под правильным углом.

Трудная задача и относительно Сложные решения, которые не работают

Допустим, мы решаем следующую задачу: определить, встречается ли в строке какая-либо перестановка данного набора букв (скажем, «abc»). То есть, получив набор букв («строку перестановки»), нужно выяснить, встречается ли какая-то последовательность этих букв подряд в другой строке («строке поиска»). Для строки перестановки **abc** функция должна вернуть **true**, если строка поиска — **cabbage** или **abacus**, но вернуть **false**, если строка поиска — **scramble** или **brackish**.

Решение такой задачи не совсем очевидно, правда? Первое, что приходит в голову, — сгенерировать все перестановки заданного набора букв, а затем проверить, встречается ли какая-нибудь из них в строке. Рекурсивная генерация перестановок довольно проста: нужно поочередно брать каждый символ из исходной строки и добавлять его к каждой перестановке оставшихся символов. Вот первая попытка такого решения:

```
vector<string> generatePermutations(const string & permute)
{
```

```
    vector<string> permutations;

    if (permute.length() == 1)
    {
        permutations.push_back(permute);
    }
    else
    {
        for (int index = 0; index < permute.length(); ++index)
        {
            string single = permute.substr(index, 1);
            string rest = permute.substr(0, index) +
                            permute.substr(
                               index + 1,
                               permute.length() - index - 1);

            for (string permutation : generatePermutations(rest))
            {
                permutations.push_back(single + permutation);
            }
        }
    }
    return permutations;
}

bool findPermutation(const string & permute, const string & search)
{
    vector<string> permutations = generatePermutations(permute);
    for (const string & permutation : permutations)
    {
        if (search.find(permutation) != string::npos)
            return true;
    }
    return false;
}
```

Логика здесь довольно проста, и на первый взгляд все работает... пока строка для перестановки остается короткой. Но стоит ей чуть увеличиться, и все рушится. Количество перестановок растет факториально относительно длины строки, поэтому функция **`findPermutation`** очень быстро становится непригодной.

Если передать ей четыре символа, как в нашем примере, она работает прекрасно. Но если дать ей дюжину, то программа погрузится в рекурсивную бездну и никогда не вернет результат*.

Первая реакция на такую «взрывную» проблему — заметить, что программа делает лишнюю работу. Если среди символов встречаются повторяющиеся, то и в списке перестановок появятся дубликаты. Может, если их убрать, станет лучше?

```
vector<string> generatePermutations(const string & permute)
{
    vector<string> permutations;
    if (permute.length() == 1)
    {
        permutations.push_back(permute);
    }
    else
    {
        for (int index = 0; index < permute.length(); ++index)
        {
            string single = permute.substr(index, 1);
            string rest = permute.substr(0, index) +
                          permute.substr(
                                index + 1,
                                permute.length() - index - 1);

            for (string permutation : generatePermutations(rest))
            {
                permutations.push_back(single + permutation);
            }
        }
    }

    sort(
        permutations.begin(),
        permutations.end());
```

* Ну, не совсем «никогда»: для четырех букв **generatePermutations** выполняется настолько быстро, что это сложно измерить на моем компьютере. Для восьми букв операция выполняется примерно за сотую долю секунды. А вот для двенадцати букв мне пришлось ждать 42 секунды, прежде чем вентилятор компьютера не заработал на полную мощность в отчаянной попытке спасти что-то внутри от перегорания.

```
    permutations.erase(
        unique(permutations.begin(), permutations.end()),
        permutations.end());

    return permutations;
}
```

Да нет, не особо помогло. Кода добавилось немного, что хорошо, и новый код сам по себе прост, но суть проблемы это не решает. Факториальный рост не «оптимизировать» никакими ухищрениями. Если только все наборы символов для перестановки не окажутся слишком короткими или почти полностью состоящими из повторов, этот код все равно остается непрактичным.

Трудная задача, относительно Сложное решение

Лучше вообще отказаться от идеи генерировать все перестановки. Этот путь заранее обречен.

Вместо этого изменим сам подход к задаче. Будем проверять каждую подстроку строки поиска. Если удастся сопоставить каждому символу строки перестановки символ из этой строки, то мы нашли перестановку:

```
bool findPermutation(const string & permute, const string & search)
{
    int permuteLength = permute.length();
    int searchLength = search.length();

    vector<bool> found(permuteLength, false);

    for (int lastIndex = permuteLength;
         lastIndex < searchLength;
         ++lastIndex)
    {
        bool foundPermutation = true;

        for (int searchIndex = lastIndex - permuteLength;
             searchIndex < lastIndex;
             ++searchIndex)
        {
            bool foundMatch = false;
```

```
            for (int permuteIndex = 0;
                 permuteIndex < permuteLength;
                 ++permuteIndex)
            {
                if (search[searchIndex] == permute[permuteIndex] &&
                    !found[permuteIndex])
                {
                    foundMatch = true;
                    found[permuteIndex] = true;
                    break;
                }
            }

            if (!foundMatch)
            {
                foundPermutation = false;
                break;
            }
        }

        if (foundPermutation)
            return true;

        fill(found.begin(), found.end(), false);
    }

    return false;
}
```

И этот вариант работает, хотя логика вложенных циклов немного запутанна. Остается некоторое волнение по поводу производительности, ведь первое решение провалилось именно из-за нее, — но на практике кубическая сложность (N^3) здесь не критична. Пока строка для перестановок не достигает тысячи символов, производительность не проблема.

Если здесь и есть проблема, то в сложности логики. Это упрощенный пример, адаптированный под формат книги, поэтому сама задача на деле не настолько трудна. Вы догадываетесь, что мы можем найти простое решение для нее, и показанное выше к таким не относится. Такое решение написал бы хороший программист — оно полностью рабочее, но слишком усложненное.

На практике более типичное для хорошего программиста решение выглядело бы как попытка преждевременной оптимизации с целью избавиться от трех вложенных циклов. Например, можно было бы вести подсчет символов и хешировать набор этих счетчиков для приближения алгоритма функции к линейной сложности:

```
#define LARGE_PRIME 104729

bool findPermutation(const string & permute, const string & search)
{
    int permuteCounts[UCHAR_MAX] = {};
    int currentCounts[UCHAR_MAX] = {};

    int permuteHash = 0;
    int currentHash = 0;

    for (unsigned char character : permute)
    {
        ++permuteCounts[character];
        permuteHash += character * (character + LARGE_PRIME);
    }

    int permuteLength = permute.length();
    int searchLength = search.length();

    if (searchLength < permuteLength)
        return false;

    for (int searchIndex = 0; searchIndex < permuteLength;
++searchIndex)
    {
        unsigned char character = search[searchIndex];

        ++currentCounts[character];
        currentHash += character * (character + LARGE_PRIME);
    }

    for (int searchIndex = permuteLength;; ++searchIndex)
    {
        if (currentHash == permuteHash)
```

```
        {
            bool match = true;

            for (char character : permute)
            {
                if (permuteCounts[character] !=
currentCounts[character])
                    match = false;
            }

            if (match)
                return true;
        }

        if (searchIndex >= searchLength)
            break;

        unsigned char removeCharacter = search[searchIndex
-permuteLength];
        unsigned char addCharacter = search[searchIndex];

        --currentCounts[removeCharacter];
        currentHash -= removeCharacter * (removeCharacter + LARGE_
PRIME);

        ++currentCounts[addCharacter];
        currentHash += addCharacter * (addCharacter + LARGE_PRIME);
    }

    return false;
}
```

Опять-таки, код этот рабочий, только излишне Сложный. В некоторых случаях он действительно демонстрирует лучшую производительность по сравнению с предыдущим вариантом, но это несущественно. Производительность предыдущего решения была вполне приемлемой, а понять его значительно проще.

Трудная задача, Простое решение

Можно ли решить задачу проще — так, чтобы код был легко читаем и понятен? Великолепные программисты отличаются от просто хороших как раз умением находить такие решения.

В данном случае с самим алгоритмом все в порядке — мы по-прежнему проверяем каждую подстроку строки поиска, чтобы определить, является ли она перестановкой заданной строки перестановки. Запутан способ реализации этого алгоритма. Проверять совпадения можно проще.

Если упорядочить буквы в строке перестановки, а затем аналогично упорядочивать буквы в каждой проверяемой подстроке, то можно просто сравнивать эти упорядоченные строки:

```
bool findPermutation(const string & permute, const string & search)
{
    int permuteLength = permute.length();

    string sortedPermute = permute;
    sort(sortedPermute.begin(), sortedPermute.end());

    for (int index = permuteLength; index < search.length(); ++index)
    {
        string sortedSubstring = search.substr(
                                    index - permuteLength,
                                    permuteLength);
        sort(sortedSubstring.begin(), sortedSubstring.end());

        if (sortedPermute == sortedSubstring)
            return true;
    }

    return false;
}
```

Основной алгоритм здесь не изменился, но реализация его стала значительно понятнее. Этот Простой вариант, тогда как предыдущий был Сложным. Великолепный программист умеет находить такие простые и ясные решения — и понимает, что простота и ясность почти всегда важнее всего остального. Великолепный программист — это не тот, кто пишет самый сложный код, а тот, кто находит самые простые ответы на самые сложные задачи.

Правило 15

Пропалывайте сорняки

Когда мои дочери были маленькими, у нас дома стояла игровая приставка Nintendo GameCube. Как оказалось, один из побочных эффектов того, что твой отец занимается видеоиграми, состоит в том, что дом заполнен всеми игровыми консолями, какие только существуют. Мои дети не сразу поняли, что далеко не у всех дома так же, как у нас.

Нашей любимой игрой была игра Animal Crossing, в которой мы втроем делили небольшую деревушку, населенную антропоморфными животными. В деревне можно было заниматься чем угодно — искать окаменелости, создавать новую одежду, украшать дом, собирать ракушки, ловить рыбу, дружить с жившими поблизости животными или просто отдыхать, слушая, как милый песик играет на гитаре.

Время от времени нам приходилось выпалывать сорняки. Каждую ночь, независимо от того, играли ли вы накануне или нет, в деревне появлялись новые сорняки. Пропалывать их было просто: подбегаешь, нажимаешь кнопку — и хоп! Сорняка нет. Главное было не запускать их. Сорняки росли всегда, пропалывали их или нет. Сорняки появлялись, даже если вы выходили из игры. Стоило помедлить с прополкой, как они заполоняли всю деревню.

Игры из серии Animal Crossing выходят до сих пор, спустя двадцать лет. Всего в игры этой серии поиграли десятки миллионов человек, и все они переживали одно и то же — возвращаешься в свою аккуратную деревушку после пары недель перерыва, а она полностью заросла. Я даже сейчас, спустя двадцать лет, ощущаю эту боль.

Ваш проект — это та же деревня. Его нужно пропалывать, то есть выдергивать сорняки — мелкие раздражающие элементы, которые неизбежно накапливаются в любой кодовой базе. Работаете ли вы над проектом каждый день или нет, выдергиваете ли сорняки регулярно или нет, новые сорняки появляются всегда. Если их не вырывать, они задушат проект.

Что представляют собой «сорняки» в этом метафорическом смысле? Это мелкие проблемы, которые легко исправить, но так же легко и проигнорировать. Как и в играх Animal Crossing, удаление сорняка — это секундное дело, нажал кнопку, и все. Никаких побочных эффектов, ничего не ломается. Результат только один — одним сорняком меньше.

Вот пример кода с сорняками:

```
// @brief Удаляет из вектора целые дубликаты
//
// @param values Вектор целых для сжатия

template <class T>
void compressVector(
    vector<T> & values,
    bool (* is_equal)(const T &, const T &))
{
    if (values.size() == 0)
        return;

    int iDst = 1;

    for (int iSrc = 1, c = values.size(); iSrc < c; ++iSrc) {
        // Проверка на уникальность значений
        if (!is_equal(values[iDst - 1], values[iSrc]))
        {
            values[iDst++] = values[iSrc];
        }
    }
    values.resize(iDst);
}
```

Здесь несколько проблем с комментариями. Во-первых, заголовок функции не соответствует ее фактической реализации. Похоже, функция изначально предназначалась для удаления дубликатов из вектора целых чисел, а при превращении ее в шаблон комментарий не обновили. Во-вторых, сам комментарий слишком обобщенный: функция удаляет только соседние дубликаты, а не все дубликаты. Если массив не отсортирован, то это совсем разные вещи. И, наконец, во втором комментарии есть опечатка.

Вот исправленный и уточненный вариант комментариев:

```
// @brief Сжимает последовательности одинаковых значений в
// векторе
//
// Для каждой последовательности значений в векторе,
// считающихся равными, оставляет только первое значение в этой
```

```
// последовательности, дубликаты удаляются.
//
// @param values Вектор для сжатия
// @param is_equal Используемая функция сравнения

template <class T>
void compressVector(
    vector<T> & values,
    bool (* is_equal)(const T &, const T &))
{
    if (values.size() == 0)
        return;

    int iDst = 1;
    for (int iSrc = 1, c = values.size(); iSrc < c; ++iSrc) {
        // Проверка на уникальность значений
        if (!is_equal(values[iDst - 1], values[iSrc]))
        {
            values[iDst++] = values[iSrc];
        }
    }
    values.resize(iDst);
}
```

Исправление этих проблем — это и есть «прополка сорняков». Сделать это легко. Комментарии не влияют на остальную часть кода, а сам код становится лучше, потому что более точные комментарии, скорее всего, когда-нибудь уберегут кого-то от ошибки.

Можно сделать еще больше. В коде встречаются проблемы с именами и форматированием. Переменные **i** и **c** не соответствуют стандартным соглашениям, так как в общем проекте используются имена **index** и **count**, а не отдельные символы. Аргумент **is_equal** лучше переименовать в **isEqual** в соответствии с принятым для проекта стилем именования функций. Фигурные скобки расставлены непоследовательно, и, кроме того, проект запрещает размещать несколько аргументов в заголовке **for**. По правилам проекта за комментариями должны следовать пустые строки, чего мы не видим для второго комментария.

Всё это легко исправить:

```
// @brief Сжимает последовательности одинаковых значений в
// векторе
//
// Для каждой последовательности значений в векторе,
// считающихся равными, оставляет только первое значение в этой
// последовательности, дубликаты удаляются.
//
// @param values Вектор для сжатия
// @param is_equal Используемая функция сравнения

template <class T>
void compressVector(
    vector<T> & values,
    bool (* isEqual)(const T &, const T &))
{
    int count = values.size();

    if (count == 0)
        return;

    // Копирует значения, не равные предыдущему элементу

    int destIndex = 1;
    for (int sourceIndex = 1; sourceIndex < count; ++sourceIndex)
    {
        if (!isEqual(values[destIndex - 1], values[sourceIndex]))
        {
            values[destIndex++] = values[sourceIndex];
        }
    }

    values.resize(destIndex);
}
```

Все эти изменения также безопасны для кода, хотя и не в такой степени, как изменение комментариев в первом случае. Здесь все же можно было допустить ошибку — например, напечатать **sourceIndex** вместо **destIndex**. Да, маловероятно, но не невозможно.

Что такое «сорняки»?

Но что вообще считать «сорняком» в коде? Здесь главное — это вопрос безопасности. Если какую-то погрешность можно исправить без всяких последствий, то это сорняк, который стоит вырвать. Исправлять опечатки в комментарии абсолютно безопасно. Также можно безопасно устранять некоторую двусмысленность в комментариях, подобно тому как мы исправляли комментарии в первом случае, — при условии, что вы на самом деле понимаете, что делает функция.

Ошибки в именовании тоже можно исправлять без риска. Здесь сработает простая операция поиска и замены по участку исходного кода, а компилятор, по крайней мере в языках вроде C++, скорее всего, заметит все ваши промахи.

Во втором раунде изменений я не только переименовал переменные, но и немного переставил их. Это было относительно безопасно, хотя и менее надежно по сравнению с другими правками. Это все еще «сорняк», но уже не такой безобидный.

В этом деле, конечно, есть свои градации. Ни одна из сделанных мной правок не затрагивала функциональность, и компилятор сгенерирует более или менее тот же самый код. Мы просто улучшили читаемость и согласованность программы, не изменяя ее поведения.

Можно представить себе более значительные изменения, тоже не влияющие на работу кода, — например, переименование члена класса (что потребует соответствующих правок во многих исходниках) или новый комментарий с описанием того, как этим классом предполагается пользоваться. Пока функциональность кода не меняется, это все еще «сорняки». Изменения, которые не должны влиять на поведение программы при соблюдении некоторой внимательности, — вроде перемещения переменной или ее переименования — по-прежнему относятся к «сорнякам», но требуют чуть большей осторожности.

Но сознательная замена функциональности — это уже не прополка «сорняка», а исправление «бага», и здесь уже действуют другие правила. Сорняки вырываются «на автомате»; ошибки же нет, потому что попытка их исправить может породить новые проблемы. Прополка сорняка по определению не порождает новых проблем.

Но если вырывать сорняки легко и база кода без сорняков ощущается в работе гораздо приятнее… так почему же в большинстве проектов их так много?

Как код зарастает сорняками

С одной стороны, пропалывать сорняки легко, с другой — столь же легко их игнорировать. Мы все очень заняты, и у нас много дел, требующих внимания.

Затраты времени и усилий на прополку, хоть они и невелики, но ощущаются сразу, а польза пусть и проявляется, но ощущается не всегда сразу. Заметив незначительную опечатку, всегда легче просто пройти мимо.

Кроме того, то, что для одного программиста «сорняк», для другого может быть «цветком»*. Допустим, вы не совсем понимаете комментарий и не уверены в том, стоит ли его исправлять. Можно было бы внимательнее изучить код или спросить коллегу, который разбирается в этом вопросе лучше вас, но, как было сказано в предыдущем абзаце, у вас и без того много дел, а исправление случайных комментариев в их список не входит.

Или, допустим, вы заметили проблемы с форматированием в участке кода, написанном новым членом команды. Вы могли бы исправить их сами, но в таком случае новичок бы не усвоил эти правила. Лучше дождаться, когда на его погрешности обратят внимание во время ревизии кода.

Итак, как бы ни было легко пропалывать сорняки, столь же легко их и оставлять по вполне понятным причинам: необходимость сосредоточиться на более важных задачах, неуверенность в том, что это действительно сорняк, и т. п.

Но сорняки плодят новые сорняки. Даже когда у вас имеются четко прописанные правила по именованию и форматированию, но если при этом проект полон сорняков, не соответствующих этим правилам, никто уже не знает, чему верить — соглашениям или самому коду. И я знаю, что происходит в таком случае: люди просто пожимают плечами и копируют вариант, который им привычнее.

Вас смутил какой-то комментарий? Он смутит и следующего, кто его прочитает. Вы удивитесь, насколько часто прополка сорняков — будь то исправление комментария, проверка работы функции с этим комментарием, обсуждение комментария на ревизии кода — приводит к обнаружению «настоящего» бага.

Прополка сорняков — это по определению быстрая работа, и ее не нужно планировать заранее. Если что-то требует заметных затрат времени, то это уже не сорняк.

Прополка сорняков в студии Sucker Punch работает хорошо, потому что у нас есть общее представление о том, что считать сорняками, и мы приняли строгие соглашения на этот счет (см. Правило 12). Мелкие правки по большей части как раз устраняют несоответствие таким соглашениям. Этот

---

* Как говорят садоводы: «Сорняк — это растение не на своем месте». Что напоминает мне о том случае, когда мне захотелось сделать сюрприз для своей жены и прополоть ее огород. Сюрпризом в данном случае оказалось то, что я вырвал всю посаженную ею недавно спаржу. Если моей целью было добиться того, чтобы меня никогда больше не просили пропалывать огород… ну что ж, задача выполнена.

процесс сам по себе укрепляет стандарты — в немалой степени потому, что правки рассматриваются на ревизии кода и люди сообща выносят вердикт, действительно ли это сорняки, которые нужно было удалить. Если вы в роли рецензента считаете, что автор кода тратит время на несущественные детали, то у вас с ним просто разное представление о том, что важно. Нужно это исправить.

В конечном счете ничего сложного здесь нет. Если вы точно уверены, что это сорняк, вырывайте его. Если вы лишь подозреваете, что это сорняк, то стоит потратить немного времени, выяснить, действительно ли это сорняк, и вырвать его*.

Парадоксально, но самые высокие требования при этом предъявляются самым опытным членам команды. Именно они чаще всего замечают сорняки, ведь те, кто составлял правила проекта, лучше других замечают отклонения от этих правил. Скорее всего, правила писали самые старшие специалисты команды. Логично, чтобы они и потратили немного времени на исправление мелких недочетов, не так ли?

Совершенно так! Очистка проекта от сорняков упрощает жизнь всем и делает более заметными по-настоящему важные вещи.

---

* Только убедитесь в том, что это не спаржа.

Правило 16

Двигайтесь от результата к коду, а не от кода к результату

Прошу прощения за небольшое погружение в метафору.

Программирование — это искусство наводить мосты. У вас есть задача, которую нужно решить, и есть некий набор кодовых инструкций и технологий, с которыми нужно работать. Между ними лежит пропасть. Вы строите мост, постепенно расширяя и перестраивая существующий код, комбинируя его части по-новому и решая задачу кусочек за кусочком, пока не достигаете цели.

Иногда пропасть совсем небольшая, и задачу решает уже имеющийся код — нужно лишь вызвать его с нужными параметрами. Это даже едва ли можно назвать программированием, нужно просто воспользоваться уже готовым решением.

Иногда же пропасть огромна и не совсем понятно, как может помочь имеющийся код. Бывает, что даже сама задача не до конца ясна! Особенно часто это случается в сфере разработки видеоигр, когда невозможно заранее сказать, что покажется увлекательным, пока не проверишь работу на практике. У нас в Sucker Punch, к сожалению, нередко бывает так: мы решаем сложную техническую задачу, а потом выясняется, что все было зря, потому что играть в это неинтересно.

Каждая пропасть имеет две стороны. Вы стоите на одной и смотрите на другую. Вопрос только в том, на какой именно стороне вы стоите: на стороне своего существующего кода или на стороне задачи?

Иными словами, если отбросить метафору, вы либо рассматриваете задачу через призму своего кода, либо рассматриваете свой код через призму задачи.

Скорее всего, вы стоите на первой стороне. Вы ведь отлично знаете свой код, тогда как задача может оказаться совершенно новой. Ее могут описывать на языке, совершенно не похожем на тот, которым обычно оперируете вы, рассуждая о технологиях. Мы в Sucker Punch, например, описываем игровую

механику через эмоции, которые она вызывает у игрока, — допустим, говорим о том, что какая-то способность вызывает ощущение «тяжести» или «стабильности». А ведь не совсем понятно, как такие ощущения перевести на язык циклов и структур данных!

Даже если вы работаете в более традиционной сфере, задачи в ней все равно формулируются в специфических терминах этой сферы: в юриспруденции, например, упоминают требования к аудиту данных, а в бизнес-школе говорят про «измеримые показатели».

Пытаться осмыслить задачу через призму уже имеющихся технологий вполне естественно. У нас в Sucker Punch я, возможно, подумаю о том, как ощущение «тяжести» в игре можно реализовать с помощью систем анимации, звука и визуальных эффектов, а также с использованием технологий тактильной отдачи. То есть я размышляю о том, как объединить эти технические элементы, чтобы добиться нужного ощущения.

В более традиционных примерах я бы подумал о том, как адаптировать систему журналирования под требования аудита или как с помощью инструментов интерфейса создать интерфейс, позволяющий отделу продаж отслеживать потенциальных клиентов (что, как выясняется, и имел в виду новый операционный директор, говоря об «измеримых показателях»).

Пример

Допустим, я создаю систему, в которой нужно настраивать множество параметров для разных сред. Некоторые из них просты — например, максимальное число рабочих потоков или путь к файлу журнала. Другие сложнее — например, список подключаемых модулей с их параметрами. В зависимости от среды может понадобиться настроить сотни таких параметров.

Похоже, здесь напрашивается конфигурационный файл. К счастью, у меня уже есть готовый код для работы с JSON, и он кажется мне отличным выбором: типы данных и структура параметров хорошо вписываются в такой формат на основе текста, который легко редактировать и отлаживать; при этом JSON достаточно гибок, чтобы по мере необходимости можно было уверенно добавлять новые параметры. Кажется, решение идеально!

Вот как выглядит код моего интерфейса для работы с JSON:

```
namespace Json
{
    class Value;
    class Stream;
```

```
    struct Object
    {
        unordered_map<string, Value> m_values;
    };

    struct Array
    {
        vector<Value> m_values;
    };

    class Value
    {
    public:
        Value() :
            m_type(Type::Null),
            m_str(),
            m_number(0.0),
            m_object(),
            m_array()
            { ; }

        bool isString() const;
        bool isNumber() const;
        bool isObject() const;
        bool isArray() const;
        bool isTrue() const;
        bool isFalse() const;
        bool isNull() const;

        operator const string & () const;
        operator double () const;
        operator const Object & () const;
        operator const Array & () const;

        void format(int indent) const;
        static bool tryReadValue(Stream * stream, Value * value);
    };
};
```

Использовать JSON с таким интерфейсом просто: нужно распарсить данные с помощью **`Json::Value::tryReadValue`**, чтобы получить объект **`Value`**, затем проверить его тип и воспользоваться соответствующим аксессором. Если вызвать аксессор неподходящего типа — например, попытаться преобразовать массив в объект, — код выдаст **`assert`** и вернет значение по умолчанию*.

В этом упрощенном примере один из настраиваемых параметров — список заблокированных серверов. Вот фрагмент этой части конфигурационного файла JSON:

```
{
    "security" : {
        "blocked_servers" : [
            "www.espn.com",
            "www.theathletic.com",
            "www.xkcd.com",
            "www.penny-arcade.com",
            "www.cad-comic.com",
            "www.brothers-brick.com"
        ]
    }
}
```

Похоже, тот, кто настраивал параметры, решил, что мне на работе не стоит тратить время впустую, ведь это почти весь мой список закладок в Chrome. В остальном все выглядит хорошо: такой JSON легко читать и редактировать.

Реализовать функцию для проверки списка заблокированных серверов несложно, если быть готовым к некоторой непредсказуемости в конфигурационном файле. В конце концов, это просто JSON, редактируемый вручную в текстовом редакторе, поэтому нельзя рассчитывать, что все будет строго в ожидаемом формате. Тот, кто набирает файл, может ошибиться — опечататься в названии параметра, использовать устаревший вариант, указать число вместо строки или одну строку вместо массива.

Я пользуюсь парсером JSON, который сам проверяет корректность структуры данных, так что об этом можно не беспокоиться. К тому же некоторая

---

* Самые глазастые из вас, вероятно, уже заметили, что этот обработчик JSON не полностью соответствует стандарту. Он предполагает уникальные ключи, поскольку используется **`unordered_map`**, тогда как в спецификации JSON дублированные ключи явно разрешены. Но не будем углубляться.

степень непредсказуемости предусмотрена самим дизайном — например, ключи **security** и **blocked_servers** необязательны. Если они отсутствуют, значит, заблокированных серверов нет. Однако код должен быть устойчив и к другим видам непредсказуемости — допустим, если кто-то вставит в список заблокированных серверов число.

Написать надежный код для работы с этим файлом несложно, хотя он получается немного объемным:

```
bool isServerBlocked(string server)
{
    if (!g_config.isObject())
    {
        log("expected object for config");
        return false;
    }

    const Object & configObject = g_config;
    const auto & findSecurity = configObject.m_values.find("security");
    if (findSecurity == configObject.m_values.end())
        return false;

    if (!findSecurity->second.isObject())
    {
        log("expected object for config.security");
        return false;
    }

    const Object & securityObject = findSecurity->second;
    const auto & find = securityObject.m_values.find("blocked_
servers");
    if (find == securityObject.m_values.end())
        return false;

    if (!find->second.isArray())
    {
        log("expected string array for config.security.blocked_
servers");
        return false;

    }
```

```
    const Array & blockedServersArray = find->second;
    for (const Value & value : blockedServersArray.m_values)
    {
        if (!value.isString())
        {
            log("expect string array for config.security.blocked_
servers");
            continue;
        }

        const string & blockedServer = value;
        if (blockedServer == server)
            return true;
    }
    return false;
}
```

Так что же, дело сделано? Я взял существующую библиотеку JSON и быстро адаптировал ее для работы с файлом конфигурации. Код, который я написал для обхода разобранных данных JSON, получился немного громоздким, но его легко было писать и он легок для чтения. Код JSON-библиотеки известен своей надежностью, поэтому логично использовать ее для решения этой задачи.

Я шел от технологии к задаче и без особых усилий получил вполне приемлемое решение. Любой, кто уже пользовался этой JSON-библиотекой, сможет без проблем работать и с этим файлом конфигурации.

Некоторое беспокойство

Впрочем, со временем мои коллеги, скорее всего, станут недовольно морщиться при мысли о том, сколько кода требуется для доступа к данным конфигурации. Этот код несложно писать и читать, но после того как напишешь одно и то же с полдесятка раз — проверка на объект, поиск ключа, обработка отсутствующего ключа, и так по кругу, — хочется все это обобщить.

Я уже сделал шаг в этом направлении, добавив сообщения об ошибках. В них перечисляются ключи, разделенные точками — это безопасно, пока сами ключи не содержат точек. Такое ограничение для имен ключей в моем конфигурационном файле вполне приемлемо, и на его основе можно написать простую функцию, проходящую по вложенным объектам **`Objects`** (при условии вызова функций для разделения и объединения строк):

```
const Value * evaluateKeyPath(const Value & rootValue, string keyPath)
{
    vector<string> keys = splitString(keyPath, ".");

    const Value * currentValue = &rootValue;
    for (unsigned int keyIndex = 0; keyIndex < keys.size(); ++keyIndex)
    {
        if (!currentValue->isObject())
        {
            log(
                "expected %s to be an object",
                joinString(&keys[0], &keys[keyIndex + 1], ".").c_
str());
            return nullptr;
        }

        const Object & object = *currentValue;
        const auto & findKey = object.m_values.find(keys[keyIndex]);
        if (findKey == object.m_values.end())
            return nullptr;

        currentValue = &findKey->second;
    }
    return currentValue;
}
```

Это избавляет **isServerBlocked** от части громоздкости и сокращает код примерно вдвое:

```
bool isServerBlocked(string server)
{
    const Value * value = evaluateKeyPath(
                          g_config,
                          "security.blocked_servers");

    if (!value)
        return false;

    if (!value->isArray())
```

```
    {
        log("expected string array for security.blocked_servers");
        return false;
    }

    const Array & blockedServersArray = *value;
    for (const Value & value : blockedServersArray.m_values)
    {
        if (!value.isString())
        {
            log("expected string array for security.blocked_servers");
            continue;
        }

        const string & blockedServer = value;
        if (blockedServer == server)
            return true;
    }

    return false;
}
```

Можно было бы еще больше упростить код, добавив версию **evaluateKeyPath**, которая проверяет, возвращает ли она именно массив:

```
bool isServerBlocked(string server)
{
    const Array * array = evaluateKeyPathToArray(
                              g_config,
                                 "security.blocked_servers");

    if (!array)
        return false;

    for (const Value & value : array->m_values)
    {
        if (!value.isString())
        {
            log("expected string array for security.blocked_servers");
            continue;
```

```
        }

        const string & blockedServer = value;
        if (blockedServer == server)
            return true;
    }

    return false;
}
```

И это реальный прогресс — получившаяся версия **`isServerBlocked`** вдвое короче первой. При сотнях параметров это серьезная разница — чистая арифметика.

Но несмотря на улучшения, остается ощущение, что мы все еще пишем слишком много шаблонного и рутинного кода. Что идет не так?

Выбор стороны моста

Во всех этих примерах — и в исходном решении воспользоваться библиотекой JSON, и в последующих улучшениях — я отталкивался от технологии и двигался вперед от нее. В нашей команде имеется библиотека JSON, с которой все хорошо знакомы, и я попытался сообразить, как применить ее к задаче с конфигурационным файлом. Получив работающий фрагмент кода, я постепенно улучшал его.

Если вспомнить метафору с мостом, я стою на стороне технологии и смотрю через пропасть на проблему, которую нужно решить. Рассматривать проблему через призму предполагаемого решения — распространенный подход.

Но вот в чем проблема такого подхода. Сквозь призму JSON конфигурационный файл начинает приобретать очертания JSON. Любой список похожих элементов воспринимается как JSON-массив. Для каждого параметра нужно придумать имя, ведь очевидно, что это ключ в объекте JSON. Родственные параметры группируются в объекты, потому что это естественный способ организовать большой JSON-файл. А если один из параметров представляет собой перечислимое значение, указывающее, выполнять ли код локально или удаленно, вы начинаете воспринимать его как строку — ведь это один из типов JSON.

Но ведь такой формат заранее жестко не определен. Выбрав другой формат, вы бы думали о задаче иначе. У нас в Sucker Punch, например, один из конфигурационных файлов хранится в бинарном формате, а не в виде текста. Мы не задумывались о какой-либо иерархии параметров

конфигурации, потому что используемая нами технология сериализации этого не предполагала. Мы записываем целые и вещественные числа напрямую, а не приводим всё к типу **float**, как это делает JSON, потому что это естественнее. Одним словом, выбранная технология сильно влияет на восприятие задачи.

А что если избавиться от этой зависимости и рассмотреть задачу конфигурационного файла без привязки к способу ее решения? Как бы я рассуждал, если бы стоял на другой стороне пропасти, то есть на стороне самой проблемы, и двигался бы не от технологии к задаче, а наоборот, от задачи к решению?

Двигаемся в обратном направлении

Взглянуть на задачу можно и по-иному. Если бы чтение и запись конфигурационного файла происходили «по волшебству», как удобнее всего было бы реализовать функцию **isServerBlocked**?

Очевидно, проще всего было бы задать глобальную структуру, содержащую все параметры конфигурации*. Тогда список заблокированных серверов можно было бы представить как множество строк внутри этой структуры. Что-то вроде следующего:

```
struct Config
{
    set<string> m_blockedServers;
};
const Config g_config;

bool isServerBlocked(string serverURL)
{
    return (g_config.m_blockedServers.count(serverURL) > 0);
}
```

Хм. Это куда проще, чем даже самая минимальная версия **isServerBlocked**, построенная поверх нашей JSON-библиотеки. Такой вариант легче писать и читать, и он оставляет гораздо меньше места для оши-

* Если идея глобального объекта вас пугает, прошу прощения. Этот конкретный пример — опции конфигурации, которые читаются при запуске и больше не меняются, а сотни проверок этих опций разбросаны по всему проекту — отлично демонстрирует, почему глобальные объекты бывают полезными.

бок. К тому же он и работает быстрее, хотя в данном случае это, пожалуй, не имеет значения.

Вообще-то дело здесь не совсем в простоте. Если как следует проанализировать мое решение с конфигурационным файлом на JSON, то можно осознать и другие проблемы с ним. Например:

- Мой JSON-код выдает предупреждения, если какая-то часть файла *config.json* не имеет ожидаемой структуры — например, если значение ключа имеет неверный тип или отсутствует обязательный ключ. Но неподдерживаемые и неизвестные параметры просто игнорируются. Пока файл остается корректным с точки зрения JSON, он может содержать любые «псевдопараметры», которые нигде не используются. Это плохо, потому что будет трудно обнаружить типичные ошибки вроде опечаток в именах параметров.
- Команда узнает о проблемах с параметрами только в момент использования параметров. Например, функция **`isServerBlocked`** записывает обнаруженные ошибки в лог, но только тогда, когда ее вызывают для проверки блокировки какого-то сервера. Если ее не вызовут, то ошибки в соответствующем разделе конфигурации останутся незамеченными. А если ее вызывают часто, лог быстро засорится повторяющимися сообщениями об одной и той же проблеме.
- Рано или поздно нам все равно придется составлять документацию для конфигурационного файла. И тогда придется начинать с нуля. Структура конфигурации определяется использующими эту конфигурацию разрозненными кусками кода, и эти куски кода разбросаны по всему проекту. Чтобы понять, какие параметры вообще допустимы, придется устраивать настоящее расследование.

Все эти проблемы звучат знакомо, и неслучайно, потому что мы незаметно для себя затронули давний спор между сторонниками раннего связывания и сторонниками позднего связывания. Любому программисту, скорее всего, довелось воспользоваться обоими подходами — позднее связывание характерно для языков вроде Python, Lua или JavaScript, а раннее — для языков C или Java.

Если максимально упростить, то при позднем связывании ошибки проявляются на поздней стадии, а при раннем связывании как минимум часть ошибок обнаруживается заранее. В языках с ранним связыванием часть багов (хотя далеко не все) выявляется на этапе компиляции; при позднем связывании все ошибки проявляются только уже во время выполнения.

Решение, построенное мной на основе нашей библиотеки JSON, работает по принципу позднего связывания. Любые проблемы с ключом **`security`**.

blocked_servers обнаруживаются только после вызова **isServerBlocked**. В отличие от этого, решение на основе глобальной структуры **Config** работает по принципу раннего связывания. Инициализируя эту структуру Config — например, загружая ее из конфигурационного файла, — я устраняю все обнаруженные проблемы, что значительно упрощает реализацию **isServerBlocked**.

Но, пожалуй, я на самом деле ничего не улучшил, а только переместил проблему. Да, эта версия **isServerBlocked** гораздо проще, но не потому ли, что я не стал писать код парсинга и валидации, который все равно где-то должен существовать? Писать парсер для сотен опций в нашем конфигурационном файле — явно еще то удовольствие.

Ничто не мешает мне объединить два подхода — воспользоваться нашей библиотекой JSON для чтения конфигурационного файла, а для доступа к параметрам кода воспользоваться структурой **Config**. Нужно лишь написать функцию, которая распакует считанные парсером JSON данные в структуру **Config**. При правильном наборе вспомогательных функций это несложно:

```
void unpackStringArray(
    const Value & value,
    const char * keyPath,
    set<string> * strings)
{
    const Array * array = evaluateKeyPathToArray(value, keyPath);
    if (array)
    {
        for (const Value & valueString : array->m_values)
        {
            if (!valueString.isString())
            {
                log("expected %s to be an array of strings", keyPath);
            }
            strings->emplace(valueString);
        }
    }
}

void unpackConfig(const Value & value, Config * config)
{
    unpackStringArray(
```

```
        value,
        "security.blocked_servers",
        &config->m_blockedServers);
}
```

В конфигурационном файле сотни опций, но большинство из них достаточно простые — простые типы или списки простых типов, доступ к которым осуществляется через простое иерархическое пространство имен. Почти всё можно обработать с помощью десятка функций вроде «*unpack*». Если придется обрабатывать список структурированных данных — в терминах JSON это массив **`Array`** объектов **`Objects`** — код придется писать, но он не будет сложным.

Такое структурирование решает некоторые из упомянутых ранее проблем. Ошибки в конфигурационном файле обнаруживаются на ранней стадии, потому что все они фиксируются при распаковке файла. Журнал не засоряется множественными сообщениями об одной и той же проблеме. Если в файле есть обязательные опции, можно переписать код с их учетом, и если чего-то не хватит, функция **`unpackConfig`** сообщит об ошибке и завершит работу при запуске.

Но я решил не все проблемы. Ранее я говорил о необходимости документации формата конфигурационного файла, и здесь прогресса нет. Также не выявляются попытки задать неподдерживаемые опции.

В текущей реализации поддерживаемая структура конфигурационного файла определяется распаковывающим его кодом, так что, пожалуй, можно было бы вывести структуру на основе вызовов распаковки. Так как распаковывается весь файл, то, например, можно считать не поддерживаемым всё в JSON-файле, что не было распаковано. Если отслеживать, какие элементы JSON были успешно распакованы, то любые оставшиеся можно помечать как неподдерживаемые опции.

Кроме того, поскольку я распаковываю всё, мне известны имя и тип каждой опции в файле. На основе имен и типов можно построить минимальную документацию с перечнем поддерживаемых опций и их типов. Минимальная документация лучше отсутствующей, и у нее есть одно большое преимущество — она надежна и точна, потому что создается непосредственно на основе кода!

В принципе, можно реализовать оба подхода, но это не так уж просто. Я написал код для выявления неподдерживаемых опций и вывода соответствующего сообщения — не слишком объемный, но слишком длинный для этой главы. Кроме того, мне казалось, что я по-прежнему двигаюсь вперед от уже построенного решения, а не от проблемной области.

А теперь нечто совершенно иное

Вот вам смелая идея: если проблема заключается в том, что на основании кода трудно вывести структуру конфигурационного файла, то почему бы не обратить задачу? Сначала определим структуру, а затем на ее основе напишем код.

Небольшое предупреждение — пример этот довольно длинный! Мне не хотелось что-то скрывать, показывая, как может выглядеть работа, когда двигаешься, отталкиваясь от результата. Можно сказать, что с учетом функциональности код в этом примере получился даже на удивление компактным. И в отличие от многих примеров в этой книге, им можно пользоваться в готовом виде. Так что потерпите несколько страниц!

Для данного простого примера структуру конфигурационного файла можно определить с помощью глобальной структуры для управления всеми опциями конфигурации:

```
struct Config
{
    Config() :
        m_security()
        { ; }

    struct Security
    {
        Security() :
            m_blockedServers()
            { ; }

        set<string> m_blockedServers;
    };
    Security m_security;
};

Config g_config;

StructType<Config::Security> g_securityType(
    Field<Config::Security>(
        "blocked_servers",
        new SetType<string>(new StringType),
```

```
        &Config::Security::m_blockedServers));

StructType<Config> g_configType(
    Field<Config>("security", &g_securityType, &Config::m_security));
```

Замысел должен быть очевиден, хотя код и выглядит как довольно замысловатая акробатика с шаблонами C++. Каждый объект из JSON-файла описывается глобальной переменной с использованием шаблона **StructType**. Здесь объект «security» («безопасность») описан через **g_securityType**; опция конфигурации «server.blockedServers» описана как часть **g_securityType**. Весь конфигурационный файл описан через **g_configType**.

Эти дескрипторы определяют преобразование JSON-объектов в структуры C++. Для этого нужно знать четыре вещи: ключи и типы JSON для полей объектов, а также типы C++ и указатели на члены для соответствующих структур C++. Выполнять такого рода «метапрограммирование» в C++ немного сложно, но осуществимо.

Сложность здесь в том, чтобы правильно манипулировать информацией о типах C++ для обеспечения типовой безопасности. Для этого я определил шаблонный класс, связывающий тип C++ с соответствующим типом JSON:

```
struct UnsafeType
{
protected:
    template <typename T> friend struct StructType;
    virtual bool tryUnpack(const Value & value, void * data) const = 0;
};

template <class T>
struct SafeType : public UnsafeType
{
    virtual bool tryUnpack(const Value & value, T * data) const = 0;

protected:
    virtual bool tryUnpack(const Value & value, void * data) const
override
        { return tryUnpack(value, static_cast<T *>(data)); }
};
```

Абстрактная структура **SafeType** обеспечивает безопасную по типу распаковку для конкретного типа C++; строки гарантированно распаковываются

в строковые переменные, целые числа — в целочисленные и т. д. В основном я буду использовать **SafeType** для распаковки данных. Когда дело дойдет до структур, код станет немного проще благодаря небезопасному интерфейсу **UnsafeType**, но при этом останется безопасным по типу благодаря хитростям с шаблоном.

Вот определения **SafeType** для некоторых типов C++:

```
struct BoolType : public SafeType<bool>
{
    bool tryUnpack(const Value & value, void * data) const override;
};

struct IntegerType : public SafeType<int>
{
    bool tryUnpack(const Value & value, void * data) const override;
};

struct DoubleType : public SafeType<double>
{
    bool tryUnpack(const Value & value, void * data) const override;
};

struct StringType : public SafeType<string>
{
    bool tryUnpack(const Value & value, void * data) const override;
};

bool BoolType::tryUnpack(const Value & value, bool * data) const
{
    if (value.isTrue())
    {
        *data = true;
        return true;
    }
    else if (value.isFalse())
    {
        *data = false;
        return true;
    }
    else
```

```
    {
        log("expected true or false");
        return false;
    }
}

bool IntegerType::tryUnpack(const Value & value, int * data) const
{
    if (!value.isNumber())
    {
        log("expected number");
        return false;
    }

    double number = value;
    if (number != int(number))
    {
        log("expected integer");
        return false;
    }

    *data = int(number);
    return true;
}

bool DoubleType::tryUnpack(const Value & value, double * data) const
{
    if (!value.isNumber())
    {
        log("expected number");
        return false;
    }

    *data = value;
    return true;
}

bool StringType::tryUnpack(const Value & value, string * data) const
{
    if (!value.isString())
```

```
    {
        log("expected string");
        return false;
    }

    *data = static_cast<const string &>(value);
    return true;
}
```

Это тот же тип кода, что и в предыдущих примерах с нашим **Config**. Здесь он оформлен в виде класса, предназначенного для работы с конкретным типом, но суть та же — проверять типы значений JSON и преобразовывать их в соответствующие значения C++. Так можно обрабатывать типичные случаи — например, значения конфигурации, которые заведомо должны быть целочисленными. В JSON все числовые значения представлены в формате числа с плавающей точкой (вещественного числа), поэтому код **IntegerType** проверяет, действительно ли вещественное значение на самом деле целочисленное.

Переходя к более сложным случаям, рассмотрим списки строк, вроде списка заблокированных серверов из примера с конфигурацией. Для представления этого списка я воспользовался множеством C++ (контейнер **set**), поэтому нужно создать **SetType**:

```
template <class T>
struct SetType : public SafeType<set<T>>
{
    SetType(SafeType<T> * elementType);
    bool tryUnpack(const Value & value, set<T> * data) const override;

    SafeType<T> * m_elementType;
};

template <class T>
SetType<T>::SetType(SafeType<T> * elementType) :
    m_elementType(elementType)
{ }

template <class T>
bool SetType<T>::tryUnpack(const Value & value, set<T> * data) const
{
```

```
    if (!value.isArray())
    {
        log("expected array");
        return false;
    }

    const Array & array = value;
    for (const Value & arrayValue : array.m_values)
    {
        T t;
        if (!m_elementType->tryUnpack(arrayValue, &t))
            return false;
        data->emplace(t);
    }

    return true;
}
```

Так как опций в конфигурационном файле сотни, то, вероятно, будут и списки. В качестве примера для читателей я оставил **VectorType**, который оборачивает контейнер **vector** — он почти идентичен **SetType**. Единственное отличие в том, что он вызывает метод **push_back()** контейнера **vector**, тогда как **SetType** использует метод **emplace()** контейнера **set**.

Последнее, что нужно сделать, — это сопоставить объекты JSON со структурами C++, точнее, прописать сопоставление пар ключ/значение JSON с членами структур или объектов C++. Для использования в **StructType** я определил безопасную по типу структуру **Field**:

```
template <class S>
struct Field
{
    template <class T>
    Field(const char * name, SafeType<T> * type, T S::* member);

    const char * m_name;
    const UnsafeType * m_type;
    int S::* m_member;
};

template <class S>
```

```
template <class T>
Field<S>::Field(const char * name, SafeType<T> * type, T S::* member) :
    m_name(name),
    m_type(type),
    m_member(reinterpret_cast<int S::*>(member))
    { ; }
```

Безопасная типизация в структуре **Field** обрабатывается немного иначе. Безопасность типов обеспечивает конструктор. Я требую, чтобы **SafeType** и указатель на член имели совпадающие типы. Так можно безопасно пользоваться небезопасными по типам **UnsafeType** и указателями на члены структуры **Field**, потому что их типы действительно совпадают.

Структура **StructType** прописана на удивление просто. С учетом всего шаблонного хаоса я воспользовался в конструкторе вариативным шаблоном (шаблоном, принимающим переменное число аргументов). Раз уж взялся, надо делать на совесть:

```
template <class T>
struct StructType : public SafeType<T>
{
    template <class... TT>
    StructType(TT... fields);

    bool tryUnpack(const Value & value, T * data) const override;
protected:

    vector<Field<T>> m_fields;
};

template <class T>
template <class... TT>
StructType<T>::StructType(TT... fields) :
    m_fields()
{
    m_fields.insert(m_fields.end(), { fields... });
}
```

Метод **tryUnpack** довольно прост — он перебирает поля объекта JSON, сопоставляя каждое из них с полем из нашего **StructType**. Построенный

таким образом цикл позволяет легко сообщать о неопознанных опциях в конфигурационном файле, что решает одну из оставшихся проблем:

```
template <class T>
bool StructType<T>::tryUnpack(const Value & value, T * data) const
{
    if (!value.isObject())
    {
        log("expected object");
        return false;
    }

    const Object & object = value;
    for (const auto & objectValue : object.m_values)
    {
        const Field<T> * match = nullptr;
        for (const Field<T> & field : m_fields)
        {
            if (field.m_name == objectValue.first)
            {
                match = &field;
                break;
            }
        }

        if (!match)
        {
            log("unrecognized option %s", objectValue.first.c_str());
            return false;
        }

        int T::* member = match->m_member;
        int * fieldData = &(data->*member);

        if (!match->m_type->tryUnpack(objectValue.second, fieldData))
            return false;
    }

    return true;
}
```

Получив **Value** из библиотеки JSON, я просто вызываю функцию **tryUnpack** для объекта **StructType**, созданного для конфигурационного файла:

```
bool tryStartup()
{
    FILE * file;
    if (fopen_s(&file, "config.json", "r"))
        return false;

    Stream stream(file);
    Value value;
    if (!Value::tryReadValue(&stream, &value))
        return false;

    if (!g_configType.tryUnpack(value, &g_config))
        return false;

    return true;
}
```

На этом этапе все выглядит хорошо! Код выполняет надежный, безопасный по типу парсинг конфигурационного файла, и при этом не пришлось много писать — вся логика типов и распаковки занимает менее 300 строк на C++. Для книги это огромный пример, но в большинстве реальных проектов он был бы лишь крошечным фрагментом. Для сотен опций конфигурации время и усилия быстро окупаются. Конечно, документация схемы конфигурационного файла по-прежнему не составляется сама по себе, но это легко исправить — нужно лишь добавить к каждому объекту **Field** строку описания, а затем для генерации документации прописать простой рекурсивный обход иерархии типов.

Движение вперед и движение назад

В этой главе были описаны два подхода к парсингу (разбору) конфигурационного файла. В первом случае мы отталкивались от формата файла конфигурации, решив, что все опции можно легко представить в формате JSON. У нас уже имелся готовый парсер JSON, и мы просто пошли дальше. Подход сработал — файл конфигурации было легко разбирать, и извлекать из него опции также было достаточно просто.

Во втором случае мы посмотрели на задачу с другой перспективы: с точки зрения программиста, который должен реализовать сотни поддерживаемых опций, а не программиста, который должен разбирать файл конфигурации. В первом случае я выбрал решение, удобное для реализации, но неудобное для использования. Во втором случае мы отталкивались от предполагаемого решения, а не от имеющихся технологий, и в результате получили более простое и лучшее решение*.

* Если вам интересно что-то подобное, советую рассмотреть решения, связанные с генерацией кода, вроде буферов протокола или «магии отражения» в C++ с использованием clReflect.

Правило 17

Большие задачи порой решать проще

«Выбирайте самый скучный способ решения каждой задачи. Если вам пришел в голову какой-то захватывающий способ, то, скорее всего, это плохая идея».

Если бы вы так переформулировали большинство советов из этой книги, я бы не стал возражать. Многие из этих правил и вправду «портят все удовольствие». Они описывают какой-нибудь интересный или изящный способ решения задачи и тут же предупреждают, что применять этот способ не стоит. И действительно, самый простой и скучный подход почти всегда оказывается лучше. Но только почти всегда!

Иногда бывают моменты, когда тучи величественно расходятся и сквозь них пробивается луч солнца, освещающий тебя и твою клавиатуру*. И в этот короткий, восхитительный миг ты осознаёшь, что проще и легче решить более общую версию той конкретной задачи, над которой ты работаешь.

Наслаждайтесь такими моментами, ведь они случаются нечасто. И если уж они пришли, будьте готовы воспользоваться ими. Напишите простой код, решите общую задачу и погрейтесь в лучах успеха.

Поспешные выводы

Вот вам пример. В одной из первых игр студии Sucker Punch игроки должны были управлять енотом-грабителем Слаем Купером, который прыгал по уровням. Слай очень ловкий; он умеет подпрыгивать, приземляться на крошечные выступы или ходить по тонкому канату, натянутому между зданиями. Управление простое: по кнопке X персонаж прыгает, с помощью стика задается направление полета (физически невозможное, но правдоподобное), по кнопке *O* персонаж приземляется на шпиль или на канат.

---

* Простите мне метафору. Я прожил в Сиэтле 35 лет, я заслужил право на метафоры с облаками.

Конечно, для всего этого потребовался код*. Самым трудным был выбор места, куда Слай должен приземлиться по нажатию кнопки *О*. Игрок-то знает, куда хочет попасть, а вот игре нужно как-то угадать его намерение. Если она угадывает, все выглядит естественно, почти как по волшебству; если нет, игрок сильно расстраивается.

Угадать, где именно игрок хочет приземлиться, гораздо сложнее, чем кажется. Представим, что игрок направляет Слая к канату и рядом больше нет никаких других возможных точек для приземления. Программа должна определить, на какую именно точку на канате пытается опуститься игрок. Какой для этого выбрать алгоритм?

Простые решения здесь работают плохо. Если выбрать точку на канате, ближайшую к текущему положению Слая, то получится, что при прыжках вдоль каната его будет оттягивать назад, к тому месту, где он был, когда игрок нажал кнопку. Вместо этого можно продолжить текущую траекторию Слая и выбрать точку на канате, ближайшую к этой траектории. Это решение уже получше, но и оно во многих случаях оказывалось неудачным. К тому же реализовать его было непросто, потому что канат представлял собой не прямую линию, а кубическую кривую.

Так мы приступили к созданию серии все более безумных прототипов, в которых испытывались новые, все более изощренные эвристические методы (или «костыли»), в попытках заставить Слая правильно приземляться на канат. На это мы потратили несколько недель. Болезненных, мучительных недель**.

И вот однажды, если выражаться метафорически, разошлись тучи, и клавиатуру осветил луч солнца — ту клавиатуру, за которой я мучился, испытывая очередной безнадежный прототип. Меня вдруг осенило, что проблема заключается в узости мышления. Нужно думать шире.

До этого я (безуспешно) пытался найти аналитическое решение, ту самую волшебную формулу, которая по входным данным (положение и скорость Слая, команды игрока и геометрия каната) сразу выдавала бы идеальную точку приземления. Но теперь я понял: мне вовсе не нужно вычислять наилучшее место посадки напрямую. Мне нужна функция, оценивающая качество каждой точки на канате, а затем просто выбирающая лучшую из этих точек.

---

* Алгоритм этот защищен патентом США № 7 147 560. Увы, в то время были в моде патенты на программное обеспечение. Но не тревожьтесь; патент истек 12 декабря 2023 года, и после этого вы сможете совершенно свободно создавать собственный платформер про ловкого енота, пользуясь описанным здесь алгоритмом.

** Автор книги лично прошел через все мучения, связанные с написанием этого кода.

Точнее, не совсем. На самом деле мне не нужно было вычислять значение функции для каждой отдельной точки на канате, нужно было лишь найти самую подходящую точку. Нужно было минимизировать функцию стоимости (измеряющую «приемлемость») на некотором диапазоне значений (параметр, задающий положение точки на канате).

Иными словами, передо мной стояла задача оптимизации. Определить точку с наименьшей стоимостью можно было, решив более общую задачу: найти локальный минимум функции, сопоставляющей значения вещественного типа (float) с другими значениями этого типа, а затем написать функцию стоимости приземления Слая в конкретной точке. Если функция стоимости будет соответствовать предположению игрока, игра выберет правильное место на канате.

Алгоритм оптимизации методом «золотого сечения» довольно надежен и несложен для реализации*:

```
float optimizeViaGoldenSection(
    const ObjectiveFunction & objectiveFunction,
    float initialGuess,
    float step,
    float tolerance)
{
    // Отслеживаем пару диапазон + значение нужной функции

    struct Sample
    {
        Sample(float x, const ObjectiveFunction & objectiveFunction) :
            m_x(x),
            m_y(objectiveFunction.evaluate(x))
            { ; }
        Sample(
            const Sample & a,
            const Sample & b,
            float r,
            const ObjectiveFunction & objectiveFunction) :
```

* Алгоритм действительно несложен, и в нем можно разобраться по приведенному ниже коду, не особо перенапрягая мозги. Но если у вас есть дела поважнее, объяснение можно найти в «Википедии». Впоследствии оптимизатор на основе «золотого сечения» помог нам решить множество задач, и всего мы вызывали его из кодовой базы Sucker Punch несколько десятков раз.

```
        m_x(a.m_x + (b.m_x - a.m_x) * r),
        m_y(objectiveFunction.evaluate(m_x))
        { ; }

    float m_x;
    float m_y;
};

// Получаем начальную тройку выборок вокруг исходного приближения

Sample a(initialGuess - step, objectiveFunction);
Sample mid(initialGuess, objectiveFunction);
Sample b(initialGuess + step, objectiveFunction);

// Убедимся, что сторона «a» дает меньшее значение диапазона, чем
// сторона «b». Если не повезло с исходным диапазоном,
// содержащим минимум, будем двигаться в сторону «a»,
// пока не найдем подходящий.

if (a.m_y > b.m_y)
{
    swap(a, b);
}

// Находим точку, где значение функции в «mid» меньше,
// чем в «a» и «b». Это гарантирует, что где-то между
//«a» и «b» есть локальный минимум.

while (a.m_y < mid.m_y)
{
    b = mid;
    mid = a;
    a = Sample(b, mid, 2.61034f, objectiveFunction);
}

// Продолжаем, пока диапазон не станет достаточно узким

while (abs(a.m_x - b.m_x) > tolerance)
{
    // Убеждаемся, что сторона «a» охватывает больший диапазон,
```

```
        // чем «b»,чтобы золотое сечение отнималось с большей
        // стороны.

        if (abs(mid.m_x - a.m_x) < abs(mid.m_x - b.m_x))
            swap(a, b);

        // Проверяем точку между текущим «mid» (лучшим приближением)
        // и выборкой «a». Если значение в новой точке лучше, она
        // становится новой «серединой» («mid»), а старая становится
        // новой крайней точкой «b». Иначе новая выборка становится
        // новой точкой «a».

        Sample test(mid, a, 0.381966f, objectiveFunction);

        if (test.m_y < mid.m_y)
        {
            b = mid;
            mid = test;
        }
        else
        {
            a = test;
        }
    }

    // Возвращаем значение в диапазоне, соответствующее лучшему
    // найденному результату

    return mid.m_x;
}
```

Доведя реализацию универсального алгоритма «золотого сечения» до оптимальной формы, я смог реализовать функцию стоимости для приземления на трос. Тросы моделировались как кривые Безье, один из многих способов представления кубической кривой. Для некоторого упрощения функция стоимости вычисляется как произведение времени, необходимого, чтобы достичь точки приземления (то есть более предпочтительными считаются точки, на которые игрок может приземлиться раньше), на максимальное ускорение, необходимое для приземления в этой точке. Нужный для вычислений

контекст (например, текущая позиция и скорость Слая) — это часть объекта, реализующего функцию стоимости:

```
struct BezierCostFunction : public ObjectiveFunction
{
    BezierCostFunction(
        const Bezier & bezier,
        const Point & currentPosition,
        const Vector & currentVelocity,
        float gravity) :
        m_bezier(bezier),
        m_currentPosition(currentPosition),
        m_currentVelocity(currentVelocity),
        m_gravity(gravity)
    {
    }

    float evaluate(float u) const override;

    Bezier m_bezier;
    Point m_currentPosition;
    Vector m_currentVelocity;
    float m_gravity;
};
```

Метод **evaluate** не очень сложный:

```
float BezierCostFunction::evaluate(float u) const
{
    // Получаем точку на кривой

    Point point = m_bezier.evaluate(u);

    // Вычисляем, сколько времени потребуется,
    // чтобы спуститься до высоты этой точки

    QuadraticSolution result;
    result = solveQuadratic(
              0.5f * m_gravity,
```

```
                m_currentVelocity.m_z,
                m_currentPosition.m_z - point.m_z);

    float t = result.m_solutions[1];

    // Предполагаем, что вся горизонтальная скорость перед
    // приземлением обнуляется

    Vector finalVelocity =
        {
            0.0f,
            0.0f,
            m_currentVelocity.m_z + t * m_gravity
        };

    // Получаем мгновенные и конечные ускорения... так как
    // следуем за кубической кривой, одно из них будет
    // максимальным ускорением

    Vector a0 = (6.0f / (t * t)) * (point - m_currentPosition) +
                -4.0f / t * m_currentVelocity +
                -2.0f / t * finalVelocity;

    Vector a1 = (-6.0f / (t * t)) * (point - m_currentPosition) +
                 2.0f / t * m_currentVelocity +
                 4.0f / t * finalVelocity;

    // Игнорируем ускорение по Z, так как это сила тяжести

    a0.m_z = 0.0f;
    a1.m_z = 0.0f;

    // Вычисляем функцию стоимости

    return t * max(a0.getLength(), a1.getLength());
}
```

Запуск оптимизатора «золотого сечения» на этой функции давал точку приземления, которая ощущалась относительно естественной. Потребовалась некоторая настройка, и в реализации **evaluate** были опущены крайние

случаи*, но даже первые результаты были лучше многих предыдущих прототипов, которые заменила эта реализация. После настройки все работало предсказуемо и естественно; на практике игроки подстраивались под функцию стоимости, а игра почти всегда угадывала их намерения**.

Конечно, почти всегда лучше решать конкретную и понятную вам задачу, вместо того чтобы пытаться решить общую задачу, которую вы не понимаете. Согласно Правилу 4, обобщать следует только после трех частных примеров. Но эта истина не абсолютна и не универсальна, и иногда решение общей задачи оказывается проще решения конкретной. В данном случае выбрать точку приземления Слая на канат было труднее, по крайней мере, аналитически; решить же общую задачу было относительно легко.

Поиски ясного пути

Приведу еще один пример из истории Sucker Punch, на этот раз более недавней. В игре Ghost of Tsushima игрок может совершать резкий рывок вперед — обычно для того, чтобы увернуться от атаки. Направление выбирается с помощью стика на контроллере, затем нажимается кнопка для рывка. Хоп — и наш герой-самурай Дзин Сакай уклоняется от опасности в выбранную сторону. По крайней мере, это так ощущается игроками.

Но Дзин не всегда делает рывок точно в выбранном игроком направлении. Когда герой сражается с врагами, например в лесу, было бы неприятно, если бы он врезался в дерево, даже если это именно то самое направление, которое выбрал игрок. Никому не понравится играть за неуклюжего самурая! Код выбирает направление, в котором нет деревьев, но при этом на максимально близком расстоянии от направления, выбранного игроком. При этом игрок ничего не замечает, он просто видит, как герой ловко уклонился, и все довольны.

В движке Sucker Punch имеется код для прокладывания пути через игровую среду — грубо говоря, код ищет маршрут до дальней границы поиска с обходом зон, через которые игрок не может пройти. Сильно упрощая, можно сказать, что код проверяет отрезки линий от исходной точки до конца

---

* Крайние случаи не настолько уж сложны, и их легко обрабатывать, добавив штрафы к функции стоимости. Если функция `evaluate` пытается вычислить точку за пределами кривой, добавляется штраф. Тщательная настройка штрафа помогает направлять оптимизатор обратно в допустимый диапазон кривой. Аналогично добавляется отдельный штраф, если Слай подпрыгнул недостаточно высоко, чтобы достичь точки.

** Если вы попытаетесь использовать для решения задач оптимизатор наподобие этого, то учтите, что он найдет минимум, но, возможно, не тот, который вам нужен. Вам нужно будет построить функцию стоимости столь же тщательно, как и сформулировать исходное предположение.

области поиска, выявляет препятствия, которые нужно обойти, а затем по возможности выравнивает полученный маршрут. Чтобы совсем было проще: это A* по выявленному графу*.

Для начальной реализации изящных рывков Дзина разумно было воспользоваться именно этим кодом, но для релиза этого было недостаточно. Дзин грациозно огибал деревья, но неуклюже врезался в других персонажей, врагов или союзников. Очевидным решением было добавить проверки доступности пути с учетом персонажей, но существующий код не умел это делать! Он работал с фиксированными препятствиями окружения, такими как деревья и здания, но ничего не знал про временные препятствия — например, про мчавшегося навстречу главному герою врага.

Добавить поддержку персонажей показалось не слишком сложной задачей, и я именно этим и занялся. Вспомним, что на базовом уровне код построения маршрута проверяет, не пересекается ли маршрут, то есть выбранный путь, с какими-либо препятствиями. Если пересечение есть, код прокладывает новый маршрут с обходом препятствий. При расширении кода для персонажей нужно было учесть два момента: пересечение пути с персонажами и построение обходного пути вокруг персонажей. Эти задачи не слишком сложны, хотя пришлось писать новый код, потому что персонажи были представлены в виде окружностей. К тому же добавилась еще одна задача — учитывать пересечение областей, выделенных под персонажей, с препятствиями в окружении, а также пересечение областей самих персонажей. Выбранный маршрут мог теоретически закончиться в любой из этих точек пересечений.

Ну что ж, подход работал, но для этого нужно было написать много кода для множества новых случаев пересечений. Уже и без того сложный код еще более усложнялся — и все только ради обработки отдельного сценария рывка игрового персонажа. Затем выяснилось, что простые круговые области, которые выделялись для персонажей, для движущихся персонажей работают плохо, и я перешел к «капсулообразным» областям, отчего стало еще хуже**. Дополнительная сложность казалась неоправданной.

---

* Алгоритм A* не очень сложен, но объяснять его здесь было бы долго. Под выявленным графом я имею в виду то, что теоретическое пространство поиска состоит из связей между любой парой точек в мире. По мере движения мы определяем, какие пары точек доступны друг для друга, проверяя отрезок между ними на пересечение со всеми имеющимися в мире препятствиями.

** Круг — это множество точек на фиксированном расстоянии от центра. «Капсулообразная» (lozenge) форма — это множество точек на фиксированном расстоянии от отрезка. Такую форму имеют скрепки и беговые дорожки, то есть две половины окружностей, соединенные параллельными линиями.

А далее наступил пресловутый момент озарения (вспоминаем метафору с тучами и лучом света). Я добавлял код не в том месте! Ведь код для построения пути уже поддерживал произвольную геометрию окружения. Деревья — это круговые препятствия, так же как и стоящий на месте персонаж. Если просто считать людей временными деревьями, код построения маршрута сможет обходить их без всякой добавленной мной дополнительной сложности.

Едва я переключился на новую схему, передо мной открылся ясный путь. В своей основе представление деревьев, зданий, заборов и всех остальных фиксированных частей окружения — это просто очень большая сетка, где каждая ячейка отмечена как проходимая или непроходимая. Временные препятствия можно добавлять через простой интерфейс, и для этого достаточно лишь проверки, свободна ли конкретная ячейка сетки:

```
struct GridPoint
{
    int m_x;
    int m_y;
};

struct PathExtension
{
    virtual bool isCellClear(const GridPoint & gridPoint) const = 0;
};
```

Этот интерфейс расширения можно передавать в базовые вызовы кода построения пути, например при проверке отрезка на препятствия или при поиске наилучшего маршрута в заданном направлении. Вот как выглядели исходные вызовы:

```
class PathManager
{
public:
    float clipEdge(
        const Point & start,
        const Point & end) const;
    vector<Point> findPath(
        const Point & startPoint,
        float heading,
        float idealDistance) const;
};
```

А вот новая версия:

```
class PathManager
{
public:
    float clipEdge(
        const Point & start,
        const Point & end,
        const PathExtension * pathExtension = nullptr) const;
    vector<Point> findPath(
        const Point & startPoint,
        float heading,
        float idealDistance,
        const PathExtension * pathExtension = nullptr) const;
};
```

Внутренние детали **clipEdge** и **findPath** практически не изменились. Там, где раньше проверялась большая сетка маршрутов, я добавил дополнительную проверку через интерфейс **PathExtension**. Всего меньше дюжины строк кода для этих функций и пары других похожих функций.

Здесь я не стану углубляться в подробности реализации интерфейса **PathExtension**, но она была примерно такой же простой:

```
struct AvoidLozenges : public PathExtension
{
    struct Lozenge
    {
        Point m_points[2];
        float m_radius;
    };

    bool isCellClear(const GridPoint & gridPoint) const override
    {
        Point point = getPointFromGridPoint(gridPoint);

        for (const Lozenge & lozenge : m_lozenges)
        {
            float distance = getDistanceToLineSegment(
                                point,
```

```
                                lozenge.m_points[0],
                                lozenge.m_points[1]);

            if (distance < lozenge.m_radius)
                return false;
        }

        return true;
    }
    vector<Lozenge> m_lozenges;
};
```

Вот и все. Пара десятков строк простого кода вместо тысячи строк сложного, когда я пытался решить задачу напрямую.

Очевидно, что новый интерфейс **`PathExtension`** получился гораздо более общим — он обрабатывал любые добавляемые в сетку временные препятствия, без ограничений по форме, размеру или способу представления. Это шаг вперед по сравнению с первой попыткой, когда поддерживались только круги и «капсуловидные» объекты.

Но дело здесь не в дополнительном обобщении! Главное не то, что решение стало более общим, а то, что оно вышло простым и более легким в реализации, чем специфическое, менее универсальное решение. Даже сейчас, когда я пишу эти строки, в кодовой базе Sucker Punch имеется только одна реализация **`PathExtension`** — та самая, что была показана выше. Мы не воспользовались преимуществом экстра-обобщения, это совершенно нормально.

Как разглядеть возможность

В большинстве случаев конкретное решение в реализации проще, чем общее. Решайте понятную вам задачу. Не пытайтесь решить более общую задачу, пока не наберется достаточное количество примеров, и пока не станет ясно, что общую задачу все же стоит решить*.

Но изредка возникают ситуации вроде описанных в этом Правиле, когда общая задача проще и когда она решается легче конкретной. Приведенные выше два примера разделяет почти восемнадцать (!) лет, и почти все остальные сотни задач, которыми мы занимались между ними, решались просто и прямо.

---

* «Достаточное количество» здесь означает «по меньшей мере три».

Но, несмотря на редкость подобных случаев, они показательны и важны. Эти два решения для студии Sucker Punch можно назвать настоящими прорывами. Иногда «прорыв» подразумевает новую парадигму, как в первом случае, когда локальный оптимизатор помог нам решать множество игровых задач; иногда это одноразовое решение сложной задачи как во втором случае. И оба они помогли улучшить продукты и добиться успеха.

Остается открытым вопрос: какие признаки говорят о том, что более общее решение будет более простым? Как понять, что в коде можно реализовать именно такое простое и общее решение?

Проработав в Sucker Punch почти четверть века, я пришел к одному выводу. Во всех случаях, когда общее решение оказывалось простым, нужно было радикально изменить точку зрения. Общее решение подразумевало совершенно иной способ осмысления задачи, и коренным образом упростить решение помогала новая перспектива.

В примерах этого Правила речь шла о смене технической точки зрения — о переходе от аналитического подхода к оптимизационному в первом случае и о добавлении новой функции в совершенно другой части кода во втором примере.

Иногда же речь идет совсем не о технической точке зрения. Так, например, можно вдруг осознать, что ваш код ориентируется не на тех пользователей. Часто мы находили более простые решения, предназначая какой-то функционал кода, например, не для программистов, как раньше, а для членов производственной команды («продакшен»), и наоборот.

Рано или поздно наступает момент озарения, когда понимаешь, что все это время размышлял о задаче неправильно. В такие моменты велика вероятность того, что тучи разойдутся, сквозь них пробьется луч солнца и тебя охватит бесконечная радость от того, что у тебя получилось найти простое и общее решение для какой-то одной конкретной и сложной задачи.

Правило 18

Пусть код сам рассказывает свою историю

В этой книге много внимания уделяется читаемости кода, будь то за счет скрывающих поведение кода тщательно созданных абстракций, понятных и отражающих суть имен, или выбора самого простого из работающих решений. Когда код легко читается, все остальное становится проще, ведь основную часть времени мы проводим не за написанием, а за чтением кода и за его отладкой. Во время отладки понять, что пошло не так, гораздо легче, если можно быстро разобраться в коде и понять, что в нем вообще происходит.

Особенно это верно для командной работы над большим проектом, но применимо и к индивидуальной разработке. Если вы работаете над чем-то покрупнее простого эксперимента, если на ваш личный проект уходят недели, месяцы или даже годы, то рано или поздно вам придется разбирать собственный код, который вы написали когда-то давно. Мысли, которые тогда крутились у вас в голове, давно исчезли; остался один лишь код. В такой момент вы оказываетесь практически на положении своего коллеги по проекту: вам приходится читать некий фрагмент кода и разбираться в его работе (или, по крайней мере, пытаться это делать).

Можно взглянуть на такую ситуацию и так: будущие вы — это другой человек.

Если ваш проект живет дольше пары дней, будьте готовы к тому, что однажды вы посмотрите на свой код как на чужой. Сделайте будущему себе одолжение — напишите код так, чтобы его было легко читать.

Представьте, что вам нужно объяснить кому-то фрагмент своего кода. Если в вашей команде приняты ревизии кода (а они должны быть приняты, см. Правило 6), то вы, вероятно, уже привыкли к таким объяснениям. Вы перечисляете задачи, которые решает этот код, объясняете структуру проекта, рассказываете, почему вы приняли те или иные решения. Вы отмечаете возможные подводные камни (надеюсь, их будет немного) и нерешенные вопросы, показываете, как взаимодействуют между собой разные части кода, и, возможно, описываете ход выполнения отдельных написанных вами функций.

Говоря вкратце, вы рассказываете историю своего кода. И чем лучше у вас это получается, тем быстрее и лучше аудитория поймет, как работает этот код. В идеале код сам должен рассказывать свою историю без ваших пояснений.

Конечно, это высокая планка, особенно в случаях, когда код неизбежно должен получиться хоть немного сложным, чтобы выполнить поставленную задачу. Тем не менее именно к этому идеалу стоит всегда стремиться — к такому коду, который «сам себя объясняет».

Не нужно говорить неправду

Так что же имеется в виду под выражением «код сам объясняет себя»? Я уже упоминал о том, как важно выбирать подходящие имена (см. Правило 3) и как нужно стараться делать код максимально простым и очевидным (см. Правило 1). В конце концов, простую историю всегда легче понять, чем запутанную!

До сей поры я почти не касался темы форматирования и комментариев — примеры кода в этой книге приводятся в основном без комментариев. Это не какая-то принципиальная позиция, просто здесь я могу объяснять смысл примеров прямо в тексте, поэтому логичнее показывать компактные фрагменты кода без лишних пометок. В реальной же жизни комментарии часто очень сильно выручают и помогают понять код.

Но это не значит, что все комментарии полезны. Некоторые только вредят. Одни говорят неправду с самого начала; другие когда-то были верными, но потом код изменился, и реальность поменялась.

Вот пример:

```
void postToStagingServer(string url, Blob * payload)
{
    // Всегда возвращает корректный дескриптор с
    // помощью Connect::Retry

    ConnectionHandle handle = connectToStagingServer(
                              url,
                              Connect::Retry |
Connect::InternalServer);
    // Отправка данных
    postBlob(handle, payload);
}
```

Код вроде бы простой и однозначный, но это не так. Да, когда его писали, флаг **Connect::Retry** действительно гарантировал успешное соединение. Но предположим, что в нашем воображаемом случае команда решила, что бесконечные повторные попытки в ожидании успешного подключения — это плохая стратегия в любом случае. Поведение **Connect::Retry** изменилось, но этот фрагмент кода, зависящий от прежнего сценария, не обновили.

Теперь в функции **PostToStagingServer** имеется ошибка, проявляющаяся нечасто, и **ConnectToStagingServer** работает почти всегда. Мне жаль бедного программиста, которому придется заниматься отладкой этого кода, особенно если функция **PostBlob** позволяет передавать пустой дескриптор, который **ConnectToStagingServer** возвращает при ошибках. Программист читает код, видит комментарий, верит ему и идет дальше, не выявляя настоящей проблемы.

Если бы не этот комментарий, ошибку обнаружили бы гораздо раньше, потому что возможность сбоя здесь очевидна. Из-за подобных ситуаций некоторые разработчики считают любые комментарии вредными, утверждая, что вред от устаревших комментариев превышает пользу от правильных.

Вспомним, что код, который не выполняется, не работает (см. Правило 8). Если какая-то часть программы используется редко, она со временем перестанет работать, и вы об этом не узнаете, потому что она не выполняется.

В каком-то смысле комментарии — это код, который не выполняется никогда. В лучшем случае под «выполнением» можно подразумевать те моменты, когда их кто-то читает и сверяет с реальной логикой программы. Такое происходит нечасто, и, как правило, комментарии читают поверхностно, поэтому неудивительно, что комментарии «перестают работать», когда поведение кода постепенно расходится с их содержанием.

Самый простой способ избавиться от этой проблемы — заменить комментарии проверкой **assert**. Не нужно писать в комментарии, что один из аргументов функции не должен быть равен **null**, — лучше явно проверить это через **assert**. В данном случае не нужно писать, что **ConnectToStagingServer** всегда возвращает корректный дескриптор; лучше добавить проверку посредством **assert**. Так вы рассказываете ту же историю, но гораздо эффективнее.

Убедитесь, что в истории вообще есть смысл

Иногда комментарии не то чтобы неправдивы, а просто бесполезны. Все мы видели подобные ничего не говорящие комментарии, часто вставляемые в код из-за формальных требований проекта, когда для всех функций

требуют обязательной документации. Вот пример кода, оформленного в соответствии с внутренними стандартами, требующими документировать каждую функцию с помощью Doxygen*. Комментарий здесь соблюдает правило, но на самом деле не передает никакой полезной информации:

```
/**
* @brief Отправка данных на staging-сервер
*
* Попытка отправить указанный payload на staging-сервер по заданному
адресу, * возврат @c true, если отправка успешна, и @c false в случае
* ошибки.
*
* @param url URL сервера
* @param payload Данные для отправки
* @returns true при успехе
*/
bool tryPostToStagingServer(string url, Blob * payload);
```

В этом комментарии нет ошибок, просто он не очень полезен. И без него можно предположить, что префикс «try» («попытка») в начале имени функции в соответствии с проектным соглашением об именованиях и указывает на функцию, возвращающую **true** при успешном выполнении. Если это так, то вся информация, содержащаяся в комментарии, уже прямо подразумевается объявлением функции. Комментарий ничего нового не добавляет; он лишь повторяет то, что уже очевидно из имени функции.

Когда комментарии, как в этом примере, просто сопровождают объявление функции а не ее определение, и когда они должны быть фиксированного формата, то они быстро увеличивают общий объем кода и начинают занимать слишком много места по сравнению с собственно объявлениями функции. Это и есть цена такого стиля комментирования — комментарии затрудняют поиск реального кода. Ситуацию усугубляют любые «странности»

---

* Doxygen — это широко используемый инструмент, извлекающий из исходного текста комментарии с причудливым оформлением и генерирующий на их основе документацию проекта. Идея заключается в том, что документация внутри самого кода с большей вероятностью остается актуальной. Это верно, хотя, по моему опыту, только в ограниченных случаях и смотря что подразумевать под «с большей вероятностью». Прошу прощения за форматирование у тех, кто не знаком с C++. Мне хотелось привести реальный пример, а все эти инструменты для генерации документации специально используют странное форматирование, благодаря которому инструмент понимает, какой текст нужно обрабатывать.

в формате комментария — например, символы `@c` в комментарии вовсе не улучшают его читаемость*.

Потенциальная польза в таких комментариях есть, потому что смысл Doxygen не только в комментировании, но и в составлении генерируемой на их основе документации. Такая документация сейчас не столь полезна, как раньше, потому что современные редакторы лучше умеют делать гиперссылки внутри проекта, но при продуманном использовании она все еще бывает полезной.

Впрочем, на практике до этого редко доходит. Программисты спешат и пишут комментарии впопыхах, лишь бы выполнить формальные требования и побыстрее перейти к следующей задаче. Оттого и появляются корректные, но совершенно неинформативные комментарии, вроде показанного в этом примере. На основе неинформативных описаний отдельных функций и типов не создается никаких полезных списков или справочников по коду, и из составленной таким образом документации по сравнению с простым чтением самого кода не узнаешь ничего нового.

Рассказываем хорошие истории

Так что же делает комментарий хорошим?

Самый очевидный ответ — хороший комментарий сообщает читателю что-то о коде, что не очевидно из самого кода. Комментарий, который просто повторяет очевидное, как в предыдущем примере, корректен, но бесполезен. Хороший комментарий помогает понять код и почему он написан именно так, он дает рекомендации по использованию функции или отмечает участок логики, который может потребовать доработки.

Хорошие комментарии выполняют еще одну важную задачу — они структурируют код. Они показывают, как разные фрагменты кода связаны между собой, и отделяют друг от друга участки кода, представляющие отдельные мысли. В этом смысле комментарии можно уподобить пробелам и знакам препинания в тексте. Да, можно прочесть текст и без пробелов и знаков препинания… но ведь гораздо легче читать текст, где они есть, верно? Пробелы разделяют слова. Знаки препинания разделяют предложения и части сложных предложений. Абзацы разделяют отдельные мысли.

То же самое справедливо и для кода: пробелы и комментарии в коде выполняют функции пробелов и знаков препинания в обычном тексте. Ниже приве-

---

* Они отмечают термины, которые должны передаваться моноширинным шрифтом наподобие Courier. Отсюда и «c».

ден пример. (Я здесь немного переусердствовал, придумав сверхкомпактные имена переменных и расставив минимальное количество пробелов.)

```
bool findPermutation(const string & p, const string & s)
{
    int pl = p.length(), sl = s.length();
    if (sl < pl) return false;
    int pcs[CHAR_MAX] = {}, scs[CHAR_MAX] = {};
    for (unsigned char c : p)
    { ++pcs[c]; }
    int si = 0;
    for (; si < pl; ++si)
    { ++scs[static_cast<unsigned char>(s[si])]; }
    for (;; ++si)
    {
    for (int pi = 0;; ++pi)
    {
    if (pi >= pl) return true;
    unsigned char c = p[pi];
    if (pcs[c] != scs[c]) break;
    }
    if (si >= sl) break;
    --scs[static_cast<unsigned char>(s[si - pl])];
    ++scs[static_cast<unsigned char>(s[si])];
    }
    return false;
}
```

В принципе понять, что делает эта функция, вполне можно — имя у нее понятное («permutations» — «перестановки»), так что отправная точка есть. Но если задать переменным более осмысленные имена, отделить смысловые блоки друг от друга пробелами и снабдить их комментариями, читать ее станет гораздо легче:

```
// Проверяет, встречается ли в строке поиска какая-либо
// перестановка заданной строки permute

bool tryFindPermutation(const string & permute, const string & search)
{
    // Если строка поиска короче, чем строка permute,
```

```
// перестановка невозможна. Для упрощения дальнейшей логики
// выходим сразу.

int permuteLength = permute.length();
int searchLength = search.length();
if (searchLength < permuteLength)
    return false;

// Подсчитываем, сколько раз встречается каждая буква
// в строке permute. Этот счетчик будем сравнивать
// с динамическим счетчиком по строке поиска.

int permuteCounts[UCHAR_MAX] = {};
for (unsigned char c : permute)
{
    ++permuteCounts[c];
}

// Делаем тот же подсчет для первого возможного совпадения
// в строке поиска.

int searchCounts[UCHAR_MAX] = {};
int searchIndex = 0;

for (; searchIndex < permuteLength; ++searchIndex)
{
    unsigned char c = search[searchIndex];
    ++searchCounts[c];
}

// Перебираем все возможные подстроки в строке поиска

for (;; ++searchIndex)
{
    // Проверяем, совпадают ли текущие подсчеты
    // со строкой permute

    for (int permuteIndex = 0;; ++permuteIndex)
    {
            // Если после проверки всех символов в строке
```

```
            // permute несовпадений не найдено, мы нашли
            // перестановку. Возвращаем true.

        if (permuteIndex >= permuteLength)
            return true;

        // Если количество данного символа в строке permute
          // не совпадает с количеством этого символа
          // в подстроке поиска, текущая подстрока не является
          // перестановкой. Переходим к следующей.

        unsigned char c = permute[permuteIndex];
        if (permuteCounts[c] != searchCounts[c])
            break;
    }

    // Прерываем цикл, если в строке поиска проверены все
     // возможные подстроки

    if (searchIndex >= searchLength)
        break;

    // Обновляем подсчеты символов для следующего
     // возможного совпадения

    unsigned char drop = search[searchIndex - permuteLength];
    unsigned char add = search[searchIndex];

    --searchCounts[drop];
    ++searchCounts[add];
}

// Если дошли до этого места, значит, все подстроки проверены,
// но ни одной перестановки не найдено, так как они
 // должны возвращаться сразу же после нахождения.

return false;
}
```

Теперь разбираться в коде стало гораздо легче, не правда ли? Чтобы понять, что происходит в предыдущем примере, нам пришлось бы изрядно потрудиться. А этот пример без труда читается сверху вниз, и сразу же становится понятно, что именно делает этот код и почему.

Дополнительные отступы визуально разделяют логические части, точно так же, как пробелы разделяют слова в предложении. Вложенные блоки группируют связанные между собой мысли. Удачно подобранные имена переменных помогают понять их предназначение — это самый первый шаг к пониманию кода и первая, самая важная форма документации. Комментарии задают контекст и не просто сообщают, «что делает код», а объясняют, «почему» он это делает.

Если вы привыкли объяснять код или слушать объяснения других, то при чтении этого кода вы ощутите знакомые чувства. Чтение хорошо написанных комментариев сродни чтению увлекательного рассказа.

Можно сказать и иначе: хороший код похож на песню. В песне есть музыка и слова; эти ее части выполняют разные, но взаимодополняющие функции. В хорошем коде все так же: сам код и комментарии играют разные, но связанные между собой роли. Строки кода отвечают за работу программы; удачные имена и аккуратное форматирование делают понятным назначение каждой строки. Комментарии задают контекст и объясняют, как различные фрагменты кода связаны между собой.

Современные редакторы, подсвечивающие код разными цветами, обычно выделяют комментарии отдельным цветом. Наш мозг хорошо различает цвета, и такое выделение помогает легко сосредоточиваться либо на коде, либо на комментариях, без потери целостного восприятия. Это все равно что читать запись песни, когда ноты записаны на линейках нотного стана, а слова рядом с ними примерно на одном уровне. Так по желанию можно сосредоточиваться либо на мелодии, либо на тексте.

Не забывайте о том, что хорошие комментарии не дублируют код, а дополняют его, и все у вас будет отлично. Комментарии оживляют сухую механику кода, превращают ее в связную историю, и благодаря им код становится гораздо понятнее. Если только по одним лишь вашим комментариям, не глядя на код, можно догадаться о том, что этот код делает, значит вы выполнили свою работу блестяще.

Правило 19

Откройте для себя параллельную переработку

Занимаясь своими повседневными делами, вы как программист лишь ненадолго отвлекаетесь от основной кодовой базы. Вы анализируете какую-то проблему, выделяете нужные файлы, исправляете ошибку, тестируете исправления, проверяете их, а затем отправляете куски кода в основную ветку. Иногда весь цикл проходится за день, хотя при обязательном тестировании это довольно быстро; часто же рабочие файлы «извлекаются» из базы на несколько дней.

Но рано или поздно вы сталкиваетесь с задачей, при выполнении которой эта простая модель перестает работать. Допустим, вы трудитесь в паре с другим программистом. При одиночной работе промежуточные версии проекта хранятся только на вашем компьютере, но при совместной работе это невозможно. Вам обоим нужно поддерживать общую версию текущей разработки.

Стандартный метод заключается в том, чтобы создать в системе управления исходным кодом новую ветку для отдельной части кода, над которой будут совместно работать несколько человек. Поначалу она идентична основной ветке, но она быстро начинает расходиться с основной. Время от времени вы с коллегой будете интегрировать изменения из основной ветки, чтобы не отставать от общей команды, и при этом устранять конфликты, возникающие в результате этих изменений. Под конец работы вы осуществляете окончательную интеграцию с основной веткой, проводите финальное тестирование, просматриваете изменения и фиксируете их (выполняете коммит).

Помехи на пути

Такой метод работает — потому он и стал стандартным решением, — но у него имеются свои проблемы.

Интегрировать изменения из основной ветки бывает непросто. Остальные члены команды не видят, чем вы с коллегой занимаетесь в своей ветке, поэтому их работа легко может что-то сломать в вашей текущей версии. Иногда это просто раздражает, как в случаях, когда кто-то добавляет новый вызов в систему, которую вы переделываете прямо сейчас. Иногда вам приходится выполнять дополнительную работу — как в случае, когда кто-то исправляет ошибку в старой версии, так что вам приходится повторять это исправление в своей. А бывает по-настоящему больно — когда, например, кто-то решает переделать исходный файл и уничтожает все ваши «диффы», на которые вы полагались.

Если вносить изменения «на ходу», то есть в систему текущей версии, как это обычно и бывает, легко забыть, как эта система работала изначально. Поначалу вы проверяете функционал в самой системе, но с каждой новой или измененной строкой система меняется, и вы уже не помните, какое ее поведение было изначальным, а какое добавилось уже в ходе работы. Существуют способы обойти эту проблему — например, хранить полную копию исходного кода для консультаций или постоянно обращаться к «диффам» оригинала, — но это неудобно и затратно.

В большом коллективе трудно уследить за изменениями, вносимыми всеми остальными участниками. Обычно бывает так, что задача вашей «малой команды» затрагивает лишь несколько исходных файлов, но обращения к этим файлам могут быть разбросаны по десяткам других файлов. Любое изменение в этих десятках файлов потенциально грозит конфликтом при слиянии. Если в большом коллективе много людей постоянно отправляют («коммитят») изменения в основную ветку, то малая команда, работающая в своей ветке, может просто захлебнуться в потоке этих изменений, пытаясь их интегрировать.

В запутанной сети веток системы контроля версий легко потеряться. Ветки предоставляют соблазнительную гибкость, которой легко увлечься. Описанный выше простой случай, когда от основной отходит одна дополнительная ветка, а чуть позже они сливаются, отслеживать достаточно легко. С усложнением ситуации, когда создаются временные ветки для тестирования новых подходов к проблеме, ветки для личных резервных копий, несколько основных веток для управления стадиями релиза, — запутаться во всех этих ветках можно очень быстро.

Мы в студии Sucker Punch на своем опыте испытали все прелести такого стандартного метода «делай ветки и изменяй». В некоторых больших проектах результаты оказывались весьма болезненными, и поэтому мы попробовали альтернативы. Особенно хорошо зарекомендовал себя способ, который можно назвать моделью «дублируй и переключай».

Стройте параллельную систему

Идея такова: вместо того чтобы менять систему на месте, строится параллельная система. Новая система в процессе работы проверяется, но активируется с помощью отдельного переключателя (runtime switch) только для работающей с ней малой команды. Остальная часть команды продолжает пользоваться старой системой, не затрагивая связей нового кода. Когда новая система готова к использованию, мы активируем ее для всех. Как только все подтверждают, что новая система у них успешно работает, старая удаляется из проекта.

Здесь отлично подходит афоризм Кента Бека:

Каждый раз, как вы хотите что-то изменить, подумайте, как сделать это легко (что бывает трудно), а потом измените это легко.

Что касается небольших проектов, следовать такой рекомендации довольно просто. Техника параллельной системы позволяет воплотить ту же идею в масштабных проектах, в которых подготовительная работа затрагивает множество участков кода от разных разработчиков. Вся трудная работа подготавливает их к моменту переключения, который по сравнению с этой работой оказывается легким.

Конкретный пример

Рассмотрим реальный пример. Описание контекста займет несколько страниц, это нужно для объяснения идеи параллельной системы.

В студии Sucker Punch мы во многих местах вместо стандартного аллокатора кучи используем аллокатор памяти по принципу стека. Основная идея стекового аллокатора заключается в том, чтобы упростить распределение памяти за счет отказа от освобождения выделенных блоков, по крайней мере отдельных. При стандартном распределении памяти каждый выделенный блок впоследствии нужно освобождать. Стековый же аллокатор работает, скорее, как переменные в стеке вызовов: любой выделенный внутри функции блок при выходе из нее освобождается автоматически. Такое выделение памяти проще, потому что не приходится заботиться об освобождении блоков. Кроме того, оно работает очень быстро, что критически важно для многих сценариев в играх.

Области видимости определяются с помощью объекта «контекста». Все стековые выделения памяти привязываются к текущему контексту. Когда контекст выходит из области видимости, все связанные с ним блоки памяти

освобождаются. Эти блоки выделяются последовательно, и потому освобождать их просто, как и выделять. Все сводится к манипуляции указателями. Вот код аллокатора:

```
class StackAlloc
{
    friend class StackContext;

public:

    static void * alloc(int byteCount);
    template <class T>
    static T * alloc(int count)
    { return static_cast<T *>(alloc(sizeof(T) * count)); }

protected:

    struct Index
    {
        int m_chunkIndex;
        int m_byteIndex;
    };

    static Index s_index;
    static vector<char *> s_chunks;
};

StackAlloc::Index StackAlloc::s_index;
vector<char *> StackAlloc::s_chunks;

const int c_chunkSize = 1024 * 1024;

void * StackAlloc::alloc(int byteCount)
{
    assert(byteCount <= c_chunkSize);
    while (true)
    {
        int chunkIndex = s_index.m_chunkIndex;
        int byteIndex = s_index.m_byteIndex;
```

```
        if (chunkIndex >= s_chunks.size())
        {
            s_chunks.push_back(new char[c_chunkSize]);
        }

        if (s_index.m_byteIndex + byteCount <= c_chunkSize)
        {
            s_index.m_byteIndex += byteCount;
            return &s_chunks[chunkIndex][byteIndex];
        }

        s_index = { chunkIndex + 1, 0 };
    }
}
```

Стековый аллокатор ведет список «чанков» (фрагментов) памяти, из которых могут выделяться блоки. Если запрашиваемый блок может разместиться в последнем чанке с уже выделенными блоками, он добавляется туда сразу же после последнего размещенного блока. Если же места не хватает, блок размещается в начале следующего чанка; при необходимости создается новый чанк.

Принцип работы объекта контекста еще проще. Он всего лишь запоминает позицию, по которой будет выделяться следующий блок памяти:

```
class StackContext
{
public:

    StackContext()
        : m_index(StackAlloc::s_index)
        { ; }
    ~StackContext()
        { StackAlloc::s_index = m_index; }

protected:

    StackAlloc::Index m_index;
};
```

Такая модель выделения памяти имеет несколько преимуществ. Она гораздо быстрее обычного аллокатора кучи, потому что выделение памяти сводится к простым арифметическим операциям с указателями, а освобождение контекста практически ничего не стоит*. Еще важнее то, что она обеспечивает отличную локальность данных, так как выделенные друг за другом блоки располагаются в памяти по порядку. А поскольку блоки освобождаются автоматически, то нет риска, что кто-то забудет их освободить.

Разумеется, у модели стекового выделения памяти хватает и недостатков**, но особенно хорошо она подходит для двух основных сценариев. Во-первых, часто бывает нужно выделить небольшой участок памяти для внутренней логики функции, и стековое выделение для этого подходит идеально. Во-вторых, выделение памяти с помощью **`StackAlloc`** работает прекрасно, когда функция возвращает данные переменного размера.

Стековое выделение на практике

Изначально реализация стекового распределения памяти у нас в Sucker Punch выглядела примерно так, как в приведенном выше коде. Со временем мы в основном перешли на стековые векторы. Небольшой поиск по кодовой базе показывает, что вызовов «чистого» стекового распределения осталось лишь несколько сотен, тогда как векторы на основе стека используются около пяти тысяч раз.

Ниже приведена упрощенная версия класса вектора со стековым выделением памяти с именами методов, совпадающими с методами стандартного вектора C++:

```
template <class ELEMENT>
class StackVector
{
public:

    StackVector();
    ~StackVector();
```

---

* Да, Правило 5 говорит о том, что не нужно оптимизировать без особой необходимости. Но поверьте, быстрое динамическое распределение памяти в играх играет очень важную роль, что подтверждается огромным объемом данных.

** Самый главный заключается в том, что блоки существуют, пока существует контекст. Если нужно хранить какие-то данные дольше, чем существует контекст, то ничего не получится.

```
    void reserve(int capacity);
    int size() const;
    void push_back(const ELEMENT & element);
    void pop_back();
    ELEMENT & back();
    ELEMENT & operator [](int index);

protected:

    int m_count;
    int m_capacity;
    ELEMENT * m_elements;
};
```

Создать вектор легко, так как изначально он не содержит элементов. Уничтожить его почти так же просто. Освобождать память не нужно, поэтому достаточно лишь вызвать деструктор для каждого элемента вектора:

```
template <class ELEMENT>
StackVector<ELEMENT>::StackVector() :
    m_count(0),
    m_capacity(0),
    m_elements(nullptr)
{
}

template <class ELEMENT>
StackVector<ELEMENT>::~StackVector()
{
    for (int index = 0; index < m_count; ++index)
    {
        m_elements[index].~ELEMENT();
    }
}
```

Базовые операции с вектором тривиальны. Заметьте, что если вектор нужно изменить по размеру, старую память освобождать не требуется. Элементы просто нужно копировать в новое хранилище, и на этом всё:

```
template <class ELEMENT>
void StackVector<ELEMENT>::reserve(int capacity)
{
    if (capacity <= m_capacity)
        return;

    ELEMENT * newElements = StackAlloc::alloc<ELEMENT>(capacity);

    for (int index = 0; index < m_count; ++index)
    {
        newElements[index] = move(m_elements[index]);
    }

    m_capacity = capacity;
    m_elements = newElements;
}

template <class ELEMENT>
int StackVector<ELEMENT>::size() const
{
    return m_count;
}

template <class ELEMENT>
void StackVector<ELEMENT>::push_back(const ELEMENT & element)
{
    if (m_count >= m_capacity)
    {
        reserve(max(8, m_capacity * 2));
    }

    new (&m_elements[m_count++]) ELEMENT(element);
}

template <class ELEMENT>
void StackVector<ELEMENT>::pop_back()
{
    m_elements[--m_count].~ELEMENT();
}
```

```
template <class ELEMENT>
ELEMENT & StackVector<ELEMENT>::back()
{
    return m_elements[m_count - 1];
}

template <class ELEMENT>
ELEMENT & StackVector<ELEMENT>::operator [](int index)
{
    return m_elements[index];
}
```

Простой пример использования:

```
void getPrimeFactors(
    int number,
    StackVector<int> * factors)
{
    for (int factor = 2; factor * factor <= number; )
    {
        if (number % factor == 0)
        {
            factors->push_back(factor);
            number /= factor;
        }
        else
        {
            ++factor;
        }
    }

    factors->push_back(number);
}
```

Пока всё вроде бы просто! Перед нами обычный вектор с улучшенной производительностью в некоторых четко определенных случаях, и его концепцию довольно просто понять. Вот почему стековые векторы так широко используются в нашей кодовой базе.

Туча на горизонте

Но тут подспудно возникает одна настораживающая проблема. Стековые векторы используются в двух основных сценариях: выделение временной памяти для функции (рутины) и возврат значения из функции, а они плохо сочетаются между собой. Сейчас объясню.

Представьте, что вам нужно написать для видеоигры функцию, возвращающую всех врагов в радиусе пяти метров от игрока. Допустим также, что у вас уже есть хорошая заготовка: функция, возвращающая всех ближайших персонажей, независимо от их отношения к игроку, вместе с их позициями, и всё это в стековом векторе. Казалось бы, можно вызвать ее, получить ближайших персонажей, а затем отфильтровать врагов.

И вот вы собираетесь написать нечто вроде следующего:

```
void findNearbyEnemies(
    float maxDistance,
    StackVector<Character *> * enemies)
{
    StackContext context;
    StackVector<CharacterData> datas;
    findNearbyCharacters(maxDistance, &datas);

    for (const CharacterData & data : datas)
    {
        if (data.m_character->isEnemy())
        {
            enemies->push_back(data.m_character);
        }
    }
}
```

Но работать этот код не будет — по крайней мере, с описанным в предыдущем разделе простым стековым аллокатором.

Проблема в том, что здесь перепутались два стековых контекста. Вы создаете **`StackContext`** и **`StackVector`** для относящихся к персонажам данных, возвращаемых функцией **`findNearbyCharacters`**, что отлично работает. Но при вызове **`enemies->push_back`** во второй части функции память выделяется из локально созданного стекового контекста, а не из стекового контекста,

связанного с массивом **enemies**. Массив **enemies**, скорее всего, определен в вызывающей функции, совсем в другом стековом контексте.

И это плохо! При использовании возвращаемого массива в вызывающей функции результаты будут непредсказуемыми. В реальном коде у нас в Sucker Punch происходит следующее: когда класс стекового вектора пытается выделить память с несоответствующим стековым контекстом, он для выявления именно такого рода ошибки запускает проверку **assert**. Существуют кое-какие способы, позволяющие устранить путаницу со стековыми контекстами, но здесь я показывать их код не буду, потому что, честно говоря, выглядит он не очень.

Делаем стековые контексты немного умнее

Исправить описанную проблему можно достаточно просто. Определенный ранее стековый аллокатор — глобальный, как и стандартный аллокатор кучи; при выделении блока он ассоциируется с самым последним созданным стековым контекстом. Именно здесь и заключается корень проблемы с перепутанными стековыми контекстами, которую мы пытаемся исправить. Все было бы гораздо проще, если бы можно было ассоциировать блок с конкретным стековым контекстом.

Сделать это нетрудно. Самый простой способ — перенести метод **alloc** в объект **StackContext**. При выделении памяти из текущего контекста будут выделяться байты из общего стека. В редком случае при выделении памяти не из текущего контекста память будет выделяться по дополнительному сценарию. При аккуратной реализации это можно сделать без потери всех преимуществ стекового аллокатора.

Сначала рассмотрим код переделанного класса **StackContext**:

```
class StackContext
{
public:

    StackContext();
    ~StackContext();

    void * alloc(int byteCount);

    template <class T>
    T * alloc(int count)
       { return static_cast<T *>(alloc(sizeof(T) * count)); }
```

```
    static StackContext * currentContext();

protected:

    struct Index
    {
        int m_chunkIndex;
        int m_byteIndex;
    };

    static char * ensureChunk();
    static void recoverChunk(char * chunk);

    struct Sequence
    {
        Sequence() :
            m_index({ 0, 0 }), m_chunks()
            { ; }

        void * alloc(int byteCount);

        Index m_index;
        vector<char *> m_chunks;
    };

    Index m_initialIndex;
    Sequence m_extraSequence;

    static const int c_chunkSize = 1024 * 1024;

    static Sequence s_mainSequence;
    static vector<char *> s_unusedChunks;
    static vector<StackContext *> s_contexts;
};
```

По мере необходимости создаются новые функции для выделения больших блоков памяти, которые потом, когда они больше не нужны, можно использовать повторно:

```
char * StackContext::ensureChunk()
{
    char * chunk = nullptr;

    if (!s_unusedChunks.empty())
    {
        chunk = s_unusedChunks.back();
        s_unusedChunks.pop_back();
    }
    else
    {
        chunk = new char[c_chunkSize];
    }

    return chunk;
}

void StackContext::recoverChunk(char * chunk)
{
    s_unusedChunks.push_back(chunk);
}
```

Код для выделения нового блока в последнем чанке перенесен в новый объект **Sequence**:

```
void * StackContext::Sequence::alloc(int byteCount)
{
    assert(byteCount <= c_chunkSize);
    while (true)
    {
        int chunkIndex = m_index.m_chunkIndex;
        int byteIndex = m_index.m_byteIndex;

        if (chunkIndex >= m_chunks.size())
        {
            m_chunks.push_back(new char[c_chunkSize]);
        }

        if (m_index.m_byteIndex + byteCount <= c_chunkSize)
        {
```

```
            m_index.m_byteIndex += byteCount;
            return &m_chunks[chunkIndex][byteIndex];
        }

        m_index = { chunkIndex + 1, 0 };
    }
}
```

Оставшиеся методы стекового контекста просты и отслеживают текущий набор вложенных стековых контекстов. Когда память выделяется из верхнего, текущего контекста (типичный случай), она берется из глобальной последовательности. При выделении из любого другого контекста (особый случай) используется последовательность, принадлежащая этому стековому контексту:

```
StackContext::StackContext() :
    m_initialIndex(s_mainSequence.m_index),
    m_extraSequence()
{
    s_contexts.push_back(this);
}

StackContext::~StackContext()
{
    assert(s_contexts.back() == this);

    for (char * chunk : m_extraSequence.m_chunks)
    {
        recoverChunk(chunk);
    }

    s_mainSequence.m_index = m_initialIndex;
    s_contexts.pop_back();
}

void * StackContext::alloc(int byteCount)
{
    return (s_contexts.back() == this) ?
                s_mainSequence.alloc(byteCount) :
                m_extraSequence.alloc(byteCount);
}
```

При обычном выделении дополнительная последовательность в стековом контексте практически не используется, поэтому новая функциональность почти не вредит производительности. Память по-прежнему выделяется быстро и просто.

Новый код стекового контекста требует немного изменить класс **StackVector**, который теперь должен указывать, из какого стекового контекста выделять память. Далее показаны только изменения, все остальное осталось прежним:

```
template <class ELEMENT>
class StackVector
{
public:

    StackVector(StackContext * context);

protected:

    StackContext * m_context;
    int m_count;
    int m_capacity;
    ELEMENT * m_elements;
};

template <class ELEMENT>
StackVector<ELEMENT>::StackVector(StackContext * context) :
    m_context(context),
    m_count(0),
    m_capacity(0),
    m_elements(nullptr)
{
}

template <class ELEMENT>
void StackVector<ELEMENT>::reserve(int capacity)
{
    if (capacity <= m_capacity)
        return;
```

```
    ELEMENT * newElements = m_context->alloc<ELEMENT>(capacity);

    for (int index = 0; index < m_count; ++index)
    {
        newElements[index] = move(m_elements[index]);
    }

    m_capacity = capacity;
    m_elements = newElements;
}
```

Новая реализация исправила ситуацию. Мы сохранили преимущества стекового распределения памяти — очень быстрое выделение и освобождение, а также хорошую локальность данных — и устранили источник потенциальной проблемы из-за конфликта локального временного хранилища с возвращаемыми данными переменного размера.

Миграция со старых стековых контекстов на новые

Теперь пора вернуться к основной идее главы, которую вы, вероятно, уже позабыли из-за всего этого потока исходного кода. Напомню: у нас имеется старая версия стекового распределения и новая версия, и они не совсем одинаковы.

Как же перейти от точки А к точке Б? Новый стековый аллокатор и новый стековый массив концептуально такие же, как и их прежние версии, но интерфейсы их немного изменились. В кодовой базе существуют тысячи мест, где используется старая модель **`StackContext`** и **`StackVector`**, не полностью соответствующая новому интерфейсу, поэтому просто так заменить старую реализацию на новую нельзя. Небольшие изменения нужно внести в весь существующий код.

При переходе на новую реализацию следует соблюдать крайнюю осторожность, чтобы не добавить новых проблем. В этих тысячах местах, где используется старая модель, могут скрываться свои баги. Другие члены команды могут быть совершенно уверены в том, что старая модель работает идеально, и при этом выделять память из неправильного стекового контекста. При переходе на новую модель, поддерживающую фактически выделение памяти вне порядка стека, их фрагменты кода могут «поломаться».

Простой способ решения этих проблем — параллельные реализации с флагом, переключающим их во время выполнения.

Во-первых, зададим двум классам разные имена, благодаря чему они смогут сосуществовать в одной кодовой базе. В C++ классы **StackAlloc** и **StackContext** можно обернуть в два пространства имен, например, **OldStack** и **NewStack**, и тогда классы будут называться по типу **NewStack::StackContext**. (Можно было бы и просто переименовать классы в **NewStackAlloc** и **OldStackAlloc**.)

Во-вторых, создадим новые классы-адаптеры **StackAlloc** и **StackContext**. Эти адаптеры будут делегировать вызовы либо старой, либо новой версии **StackAlloc** и **StackContext** в зависимости от нового глобального флага. Адаптеры поддерживают объединенный интерфейс, включающий небольшие различия между старой и новой реализацией.

Это довольно просто:

```
bool g_useNewStackAlloc = false;

class StackAlloc
{
public:
    static void * alloc(int byteCount);
};

void * StackAlloc::alloc(int byteCount)
{
    return (g_useNewStackAlloc) ?
       NewStack::StackContext::currentContext()->alloc(byteCount):
       OldStack::StackAlloc::alloc(byteCount);
}
```

Адаптер **StackAlloc** просто проверяет флаг времени выполнения и вызывает нужный аллокатор. Адаптер **StackContext** еще проще — так как у старого **StackContext** не было метода **alloc**, код для его вызова не пишется. Любой новый код, вызывающий метод **alloc** через адаптер **StackContext**, будет автоматически пользоваться новым **StackContext**:

```
class StackContext
{
public:

    StackContext() :
        m_oldContext(),
```

```
        m_newContext()
        { s_contexts.push_back(this); }
    ~StackContext()
    { s_contexts.pop_back(); }

    void * alloc(int byteCount);

    static StackContext * currentContext()
    { return (s_contexts.empty()) ? nullptr : s_contexts.back(); }

protected:

    OldStack::StackContext m_oldContext;
    NewStack::StackContext m_newContext;

    static vector<StackContext *> s_contexts;
};

vector<StackContext *> StackContext::s_contexts;

void * StackContext::alloc(int byteCount)
{
    return m_newContext.alloc(byteCount);
}
```

На этом этапе ваша цель — минимизировать вмешательство в старый код. Пока глобальный флаг равен **false**, код выполняется почти точно так же, как до изменений. Старые **StackContext** создаются как и прежде, и **StackAlloc** работает как и раньше, поэтому при тестировании не должно возникнуть серьезных проблем.

На этом этапе при условии успешного тестирования можно зафиксировать изменения (сделать «коммит»). Обновлять **StackVector** пока не нужно, так как он будет работать «как есть». Он выделяет стековую память точно так же, как и любой другой пользователь стековой памяти, а переключаться между старым и новым стековыми аллокаторами можно с помощью флага во время выполнения.

Возможность зафиксировать частично выполненную работу в основной ветке — большое преимущество техники параллельной разработки. В данном случае оно не играет такой уж большой роли, так что можно пропустить промежуточные изменения и объединить последующие шаги в один большой

«коммит». Но в более реалистичном проекте переход к новому решению через серию частичных изменений значительно облегчает весь процесс.

Подготовка к миграции класса StackVector

Следующим шагом нужно решить, как подготовить для миграции класс **StackVector**. Очевидный вариант — следовать той же модели, которую мы применили для **StackContext**, когда новый класс-обертка **StackVector** содержит в себе и старый, и новый классы **StackVector**, а переключение между ними осуществляется с помощью глобального флага. Как следствие, можно написать методы делегирования вроде такого:

```
template <class ELEMENT>
size_t StackVector<ELEMENT>::size() const
{
    if (g_useNewStackAlloc)
        return m_oldArray.size();
    else
        return m_newArray.size();
}
```

В качестве временной меры это не так уж плохо. Расписывать функции делегирования немного утомительно, но, по крайней мере, любой, кто наткнется на этот код во время перехода на новую систему, поймет, что происходит.

Другой вариант — вставлять переключатель там, где вызывается стековый аллокатор. В данном случае этот вариант подходит идеально, ведь класс **StackVector** выделяет стековую память ровно в одном месте. Если этот код сможет работать как с глобальным стеком (как в оригинальном коде), так и с указанным стековым контекстом (как в коде, на который мы переходим), все будет в порядке:

```
template <class ELEMENT>
class StackVector
{
public:

    StackVector();
    StackVector(StackContext * context);

    void reserve(int capacity);
```

```
protected:

    bool m_isExplicitContext;
    StackContext * m_context;
    int m_count;
    int m_capacity;
    ELEMENT * m_elements;
};

template <class ELEMENT>
StackVector<ELEMENT>::StackVector()
: m_isExplicitContext(false),
   m_context(StackContext::currentContext()),
   m_count(0),
   m_capacity(0),
   m_elements(nullptr)
{
}

template <class ELEMENT>
StackVector<ELEMENT>::StackVector(StackContext * context)
: m_isExplicitContext(true),
   m_context(context),
   m_count(0),
   m_capacity(0),
   m_elements(nullptr)
{
}

template <class ELEMENT>
void StackVector<ELEMENT>::reserve(int capacity)
{
    if (capacity <= m_capacity)
        return;

    assert(
        m_isExplicitContext ||
        m_context == StackContext::currentContext());
```

```
    ELEMENT * newElements = (m_isExplicitContext) ?
        m_context->allocNew<ELEMENT>(capacity) :
        m_context->alloc<ELEMENT>(capacity);
    for (int index = 0; index < m_count; ++index)
    {
        newElements[index] = move(m_elements[index]);
    }
    m_elements = newElements;
}
```

Два конструктора **StackVector** позволяют определять, какой из способов выделения памяти уместен в том или ином случае. Весь уже существующий код будет использовать первый конструктор, по крайней мере на начальном этапе, потому что он принимает те же аргументы, что и исходная версия класса. В новом коде вы будете ориентироваться на второй конструктор, но этот код пока не написан. Для первого конструктора **isExplicitContext** не устанавливается, и метод **reserve** работает точно так же, как и раньше.

Опять-таки, на этой стадии можно безопасно зафиксировать изменения (отправить «коммит»). Если глобальный флаг не установлен, все существующие вызовы **StackVector** будут вызывать старый способ выделения памяти. Любой код с установленным глобальным флагом будет включать новый способ, как и код, создающий стековый массив с помощью нового конструктора с явным контекстом.

Собственно миграция

Теперь можно приступать к миграции!

В Sucker Punch этот процесс проходил бы поэтапно. Сначала новый глобальный флаг, переключающий систему на новый стековый аллокатор, устанавливает небольшое число «первых пингвинов»*.

Если они не обнаруживают проблем, мы привлекаем более широкую группу. Когда все и дальше идет гладко, мы фиксируем изменения, установив для глобального флага значение **true**, и новой системой пользуются уже все. Если на каком-то этапе возникает проблема, мы просто переключаем всех

* Если вы не знакомы с этим выражением, вот вам объяснение. Пингвины гнездятся на суше, но охотятся в море. Это значит, что им приходится прыгать со льдин в океан, в котором может скрываться голодный морской леопард. Пингвины обычно собираются толпой у кромки воды и ждут, пока какой-нибудь пингвин не наберется смелости и не нырнет в воду — точнее, когда кого-нибудь из них не столкнут, так как понятия чести и доблести им не знакомы. Если этот пингвин остается жив, за ним следуют остальные. Отсюда и выражение «первый пингвин».

обратно на старую систему, после чего занимаемся диагностикой и исправлением ошибки.

После того как новой системой начинают пользоваться все и она демонстрирует безопасность, можно приступать к удалению адаптеров. Новый класс `StackContext` заменяет старый без проблем. Можно убрать и небольшой кусок кода, добавленный в класс `StackVector`, так как теперь все проходит через новый аллокатор.

Далее нужно определиться с тем, требовать ли передачи контекста каждому `StackVector`. Это компромисс между удобством автоматического определения верхнего контекста стека и ошибками, возникающими, когда стековый контекст случайно удаляется или перемещается. Если мы договоримся требовать контекст, то внедрять это правило можно поэтапно — необязательно сразу обновлять все пять тысяч (!) фрагментов, в которых создаются стековые массивы.

Может показаться странным, что мы вообще рассматриваем вариант с конвертацией пяти тысяч строк кода, когда имеющаяся по умолчанию разумная стратегия позволяет обойтись без изменений… но мы ориентируемся на долгосрочную перспективу. Если обязательное предоставление контекста повысит эффективность работы и позволит избежать целого класса ошибок, то усилия стоят того.

К тому же конвертировать код не так уж и сложно. Небольшого фрагмента на Python будет достаточно, чтобы найти все объекты класса `StackVector`, выяснить, какой для них подразумевался стековый контекст (что нетрудно, ведь это почти всегда последний определенный `StackContext`), обновить конструктор и зафиксировать измененный файл. Основная трудность здесь не в обновлении кода, а в тестировании изменений.

Компилятор выловит почти все проблемы, но в данном случае я бы рекурсивно применил стратегию этой главы. Я бы добавил специальный конструктор для всех мест, где автоматически был выявлен контекст, а затем бы во время выполнения проверял соответствие переданного конструктора верхнему контексту стека. Убедившись в том, что контексты не изменились, я бы переключился на обычный конструктор. При наличии хороших тестов, охватывающих почти весь код, это не так уж сложно.

Далее остается только код, выделяющий стековую память напрямую. Здесь у нас два варианта: продолжать поддерживать глобальное выделение стека с неявным использованием текущего стекового контекста или конвертировать несколько сотен строк кода, чтобы вызывать `alloc` через переменную `StackContext` напрямую. В данном случае я конвертировал бы всё и получил бы новую надежную модель выделения памяти.

После конвертации последних участков с прямым выделением памяти задачу можно считать выполненной. Все остатки старого стекового аллокатора исчезли, и мы сделали это постепенно, поэтапно, выполняя небольшие и безопасные шаги с периодическими фиксациями кода («коммитами»). При возникновении проблем на этом пути можно, следуя той же стратегии, вернуться к старой версии, не прерывая работы всей команды.

Уместность параллельной переработки

Стратегия параллельной переработки при подходящих обстоятельствах действует отлично, но это никоим образом не панацея. Временами вам все равно придется исправлять ошибки в двух местах. Вашу работу могут неосознанно нарушить программисты, работающие со старой версией системы. Конечно, со временем такие случаи будут все реже и они будут все менее разрушительными по сравнению с той ситуацией, когда вам бы выделили отдельную ветку в системе контроля версий, но они все же возможны.

Параллельная переработка также в какой-то степени увеличивает нагрузку на программистов. Наличие трех отдельных имен для одной и той же концепции — для исходного класса, переработанного класса и класса-адаптера — подразумевает некоторые умственные усилия, да и объем кода в целом увеличится, потому что, скорее всего, для некоторых частей исходной версии придется создавать несколько копий.

Бывает и так, что переработанная система настолько сильно отличается от исходной версии, что параллельная переработка теряет смысл. При невозможности определить слой адаптера, который динамически переключался бы между старой и новой версиями, описанная техника неприменима.

Тем не менее во многих случаях модель параллельной переработки предлагает управляемый, постепенный и безопасный способ внесения изменений в кодовую базу. У нас в Sucker Punch мы не пользуемся этой моделью во всех без исключения случаях изменений, но это наша основная стратегия для масштабной переработки кода.

Правило 20

Не забывайте про математику

В этой книге не так уж много математики. В некоторых Правилах, конечно, встречаются числа (как, например, в Правиле 4 «Обобщайте после трех примеров», и в Правиле 11 «Станет ли от этого вдвойне лучше?»), но Правила, скорее, представляют собой концепции, а не какие-то уравнения.

Казалось бы, в программировании должно быть больше математики. В конце концов, компьютеры — это же просто машины для обработки чисел. В них все сводится к числам для обработки: слова — это представленные числами последовательности символов; растровые изображения — это пиксели, представленные цветами, которые, в свою очередь, также представлены числами; музыка — это описания волн, представленных сериями чисел. Можно было бы ожидать, что числа неизбежно просочатся наружу и что программисту рано или поздно придется решать какие-то сложные уравнения. Но такое случается далеко не так уж часто.

Чаще программистам приходится принимать решения без каких-то четких и определенных параметров. Например, что лучше — написать большой комментарий к сложной логике или попытаться сделать код попроще? Назвать функцию `getPriority` («установить приоритет») или `calculatePriority` («вычислить приоритет»)?* На каком этапе стоит переходить на новую версию какой-либо системы?

Легко прийти к мысли, что такие вообще все решения, а не только большинство. Но некоторые из них все же сводятся к простой математике, и вы должны уметь их распознавать. В противном случае, забыв про простые расчеты, потом об этом можно сильно пожалеть и осознать, что вы смогли бы сэкономить кучу времени, не забудь вы про математику. Иногда бывает так, что несколько вычислений помогают понять, что выбранный вами метод не сработает и что вообще не стоило браться за него. Так что лучше посчитать всё заранее.

---

* Неудивительно, что у нас в Sucker Punch приняты соглашения и по этому поводу: «get» в названии означает, что в коде не производится вычислений (или их мало), а «calculate» или нечто подобное означает, что для получения значения нужно производить расчеты. Так с самого начала легче понимать, что делает функция.

Автоматизировать или не автоматизировать

Рассмотрим типичную ситуацию. Допустим, вы выполняете какой-то процесс вручную и задумываетесь, стоит ли его автоматизировать.

Да, это как раз задача из области математики! Если вы потратите меньше времени на написание кода, то автоматизация будет оправданна. Если нет, то она не будет оправданна.

Звучит вроде бы очевидно, и так и есть, но очень часто в таких случаях люди вообще не производят никаких расчетов.

Судя по моему опыту, о расчетах гораздо чаще забывают. Типичен обратный пример: программиста настолько раздражает какой-то ручной процесс, что он сразу же решает автоматизировать его и две недели разрабатывает какой-то макрос, чтобы потом поздравить себя и запускать этот макрос пару раз в неделю на пятнадцать секунд.

Возможно, он с удовольствием провел эти две недели, но задача не была оправданна, и выполненные заранее расчеты об этом сказали бы. В конце концов, мы же стали программистами, потому что нам нравится программировать. У нас по умолчанию склонность решать проблемы с помощью программирования, но программирование решает далеко не каждую проблему.

Решение об автоматизации процесса — это задача оптимизации: вы оптимизируете рабочий процесс, а не выполнение программы. При этом следует выполнять те же шаги, что и для любой оптимизации, включая неизбежное измерение процесса до его оптимизации.

Рассмотрим конкретный пример автоматизации. Представьте, что вы пишете книгу про программирование. Все примеры программ вы набираете и редактируете в Visual Studio, но саму книгу пишете в Word. В исходниках программ встречаются отступы, которых не должно быть в книге*. Вы делаете это вручную, в ходе довольно простого и быстрого процесса:

1. Выделить блок кода в редакторе.
2. Убрать отступы.
3. Скопировать его в буфер обмена.
4. Отменить удаление отступов.
5. Переключиться в Word.
6. Создать абзац с нужным стилем.
7. Вставить пример кода в этот абзац.

Стоит ли автоматизировать этот процесс? Сэкономите ли вы время в целом, автоматизировав эту операцию? Пора выполнить расчеты.

---

* Для описания этого примера, скажем так, мне не пришлось слишком напрягаться.

В этих расчетах есть два аспекта: цена и выгода. Цена — это какой объем работы придется выполнить для реализации автоматизации. Выгода — это количество сэкономленного после автоматизации времени.

В этом конкретном случае какие-то шаги придется выполнять даже после автоматизации. Вам все равно придется переходить в редактор кода для выделения блока кода и все равно придется переходить в Word, чтобы вставить блок. При выполнении расчетов можно проигнорировать неизменяемые шаги, потому что вас интересует только разница во времени до и после автоматизации. Все остальное, что можно автоматизировать, будет занимать фактически нуль времени.

Выполнять расчеты без чисел нельзя. При любой возможности используйте реальные числа, а не примерные оценки. Это значит, что нужно измерять все, что поддается измерениям, и в данном случае нужно измерить, сколько времени уходит на ручной процесс. Допустим, на описанные выше шаги вы тратите по 6 секунд*. Вы рассматриваете уже написанные главы, и в них примерно по 8 примеров кода в каждой. Согласно договору, вам нужно написать примерно 20 глав, и будем считать, что в каждой из них будет 8 примеров кода. Вы также заметили, что многие примеры кода приходится исправлять, то есть копировать и вставлять их по несколько раз. По вашим примерным оценкам, каждый пример вставляется 3 раза — это единственная неточная прикидка во всех расчетах. Этого уже достаточно для расчета выгоды:

6 секунд (на одну операцию копирования) ×
8 (примеров кода на главу) ×
20 (глав) ×
3 (правки каждого примера) = 48 минут

Итак, такова наша выгода. Переходим к цене.

Сколько времени уйдет на автоматизацию процесса? Автоматизировать что-то в программе Visual Studio нелегко, по крайней мере, в стандартной ее конфигурации, но Word поддается автоматизации на удивление хорошо. Если вам уже приходилось писать в Word макросы, и особенно если вы писали код, работающий с буфером обмена, то вы уже неплохо подготовлены. Помимо манипуляций с буфером, нужно будет добавить кое-какие операции с текстом.

И эти операции не кажутся такими уж трудными. Передать содержимое буфера обмена в массив строк, по одной строке. Определить минимальный уровень отступа для каждой строки и пересобрать массив с удалением этого

* Именно столько они заняли у меня.

отступа. Вероятно, придется подумать и о том, как на процесс влияют пустые строки, а также учесть, что пробелы и табуляции в редакторе кода выглядят почти одинаково, но в Word заметно по-разному. После очистки строк нужно снова собрать их в текстовый блок и вставить его в документ. Получившийся макрос нужно привязать к горячей клавише.

Допустим, по вашим оценкам, для корректного результата получается около часа работы*. Час — это больше 48 минут, так что расчеты подсказывают, что автоматизировать процесс не стоит. Но числа уж очень похожи; возможно, вы допустили какие-то неточности. Может, в среднем каждый пример приходится редактировать четыре раза, а не три. Тогда математическая выгода будет уже положительной. В таком случае расчеты говорят, что заниматься автоматизацией стоит. Да и вообще ручные действия раздражают, даже если отнимают всего по шесть секунд.

Но погодите, подумаем еще раз. Что более вероятно — чуть более пессимистичные, то есть заниженные, ожидания в оценке выгоды или чуть более оптимистичные ожидания в оценке времени на написание кода? Вы, как программист, наверняка уже знаете ответ. Гораздо вероятнее, что вы недооценили время, которое придется потратить на разработку кода.

Если оценка цены и выгоды оптимизации выглядит как ничья, то браться за автоматизацию не нужно.

Ищите жесткие ограничения

Если в вашей задаче или в предполагаемом решении имеются какие-то жесткие ограничения, их стоит учитывать с самого начала проектирования.

Одно из удобств в разработке игр для консолей заключается в том, что консоли задают множество жестких ограничений. Например, у игровых консолей фиксированный объем памяти. На диске Blu-ray размещается фиксированное количеств байтов. Размер UDP-пакета фиксирован. Каждый кадр — это шестидесятая доля секунды, и точка.

В нашей команде мы иногда сами придумываем жесткие ограничения, уточняя технический процесс проектирования. Возьмем, к примеру, доступную пропускную способность сети. Для разных пользователей она различна и бывает непредсказуемой, но у нас имеются достаточно точные данные по измеренной пропускной способности у пользователей по всему миру. Мы можем ввести искусственное жесткое ограничение на сетевую пропускную способность, применимое почти ко всем нашим пользовате-

* На мой взгляд, по очень оптимистичным оценкам.

лям. При соблюдении этого ограничения наша игра будет хорошо работать почти у всех*.

Казалось бы, как можно вообще воспринимать положительно какие-то ограничения? Почему жесткие лимиты — это повод для оптимизма?

Возьмем для примера жесткое ограничение по памяти для консолей. На первый взгляд, это плохо — разве не упростила бы разработку виртуальная память? Да, разумеется, но ценой превращения жесткого ограничения в более мягкое. При заполнении доступной физической памяти виртуальная память начнет выгружать страницы на диск, и время их обработки замедлится, а для видеоигр это проблема. Пока память занята загрузкой и выгрузкой, она не может адекватно рендерить кадры с требуемой частотой шестьдесят кадров в секунду. Лучше заранее принять ограничение памяти как жесткое и отталкиваться от него.

Поэтому для упрощения процесса разработки мы сразу же задаем жесткие ограничения, а любые мягкие переопределяем как жесткие. Это полезно для всех сотрудников, но, как выяснилось у нас в Sucker Punch, особенно это важно для тех, кто не является программистом. Воспринимать жесткие границы проще, и они превращают часть разработки в элементарную математику, которая дается легко**.

Для примера рассмотрим разработку сетевого протокола. Соединение для кооперативной игры строится по фиксированной базовой схеме одноранговой сети («peer-to-peer», то есть сети равноправных узлов), и каждая машина связана с другими напрямую. Каждая машина заведует некоторым поднабором персонажей в игре и пересылает другим машинам данные о состоянии этих персонажей. При этом нужно соблюдать следующие жесткие ограничения: входная пропускная способность 1 Мбит/с и исходящая способность 256 кбит/с. В таких пределах хорошая производительность будет почти у всех. Отдельный сеанс связи поддерживает подключение четырех игроков.

Вы решаете, что для таких условий подойдет дизайн, при котором каждая машина рассылает UDP-пакеты с данными о позиции и ориентации каждого персонажа под ее контролем, а также с информацией, которая позволит воссоздать текущую анимацию этих персонажей. Этих данных достаточно, чтобы расположить каждого персонажа и придать ему нужную позу на всех

---

* Но не совсем у всех. Летом на станции Мак-Мердо в Антарктиде тысяча человек делят между собой 25 Мбит/с. Это меньше нашего жесткого ограничения, уж извините, ученые. И, возможно, кому-то из вас стоит меньше смотреть Netflix, тут тоже извините. Но продолжайте работу в том же духе.

** По крайней мере, дается легко программистам, и им же легко объяснять все расчеты тем, кто не хочет вникать в математику.

других машинах. Если какие-то пакеты и теряются, это не так уж и страшно, потому что данные о каждом персонаже отправляются 60 раз в секунду.

И снова все сводится к математике! Вас жестко ограничивает пропускная способность, поэтому нужно вычислить, сколько данных ваш проект будет отправлять каждую секунду. Следовательно, нужно будет измерить все, что подлежит измерению, а если измерить нельзя, то оценить примерно.

Самый простой вариант — передавать данные в их изначальном виде. Позиция персонажа — это вектор из трех 32-битных вещественных значений (чисел с плавающей точкой), а ориентацию можно выразить как направление компаса, также с вещественным значением. Остается передать данные для восстановления анимации на удаленной машине.

Измерения можно произвести в однопользовательской версии игры. Так оказывается, что в среднем поза каждого персонажа определяется смешением шести анимаций. Значит, нужно отправлять общее число анимаций и данные для восстановления этих анимаций. Различать их между собой можно по уникальным 8-байтовым идентификаторам. Также нужно передавать положение на временной шкале анимации, а это 4-байтовое число с плавающей точкой, и коэффициент смешения двух анимаций — также число с плавающей точкой.

Итак, математика для одного персонажа понятна, по крайней мере для самой простой версии. Все вещественные значения занимают по 4 байта, а для счетчиков по умолчанию используются 4-байтовые целые числа.

Позиция занимает 12 байт, плюс 4 байта на ориентацию, плюс 4 байта для счетчика анимаций. Каждая анимация — это 8-байтовый идентификатор, плюс два 4-байтовых вещественных числа для позиции на временной шкале и коэффициента смешивания. Итого: $12 + 4 + 4 + 6 \times (8 + 4 + 4) = 116$ байт на персонажа. Вроде бы не так уж много.

Но расчеты на этом не закончены. Данные о каждом персонаже передаются каждый кадр, то есть чтобы получить число байтов в секунду, получившееся число нужно умножить на 60.

В одноранговой сети каждая машина каждый кадр отправляет три копии данных своих персонажей — по одному для каждого другого узла; от них же она получает данные персонажей, контролируемых соответствующими машинами. В худшем случае контролировать всех персонажей будет одна машина, тогда ей придется отправлять три копии данных всех персонажей и не получать ни одной.

Здесь нужно учесть еще одну фиксированную величину — количество персонажей. Допустим, ваши гейм-дизайнеры решили, что их должно быть 30. Теперь данных для расчетов достаточно:

30 (кадров в секунду) ×
3 (копии для других машин) ×
30 (персонажей) ×
116 (байт на персонажа) ×
8 (бит в байте) = 2,5 Мбит/с

Ого! В десять раз больше допустимой пропускной способности. Математика показала, что простой способ не работает. И самое опасное здесь в том, что при тестировании системы во внутренней сети студии со скоростью передачи данных в 1 Гб/с дизайн будет работать нормально, и нагрузку на систему, возможно, даже никто не заметит. Проблема всплывет, только когда пользователи запустят игру на своих устройствах в реальных условиях.

Придется этот простой дизайн перерабатывать.

Сжимать данные о персонажах можно разными способами, так что начнем с этого. Возможно, для координат хватит и 16 битов, притом что многопользовательские карты невелики, а направление можно передавать с помощью 8 битов. Если создать таблицу всех передаваемых по сети анимаций, то для их идентификации будет достаточно 10 бит, а каждая анимация могла бы формировать свое сетевое состояние, которое лучше подвергается сжатию, чем «сырые» показания весов смешивания и значения времени. Предположим, что, применив разнообразные трюки сжатия, мы снизили количество байт на персонажа со 116 до 16.

Но расчеты по-прежнему не сходятся:

30 (кадров в секунду) ×
3 (копии для других машин) ×
30 (персонажей) ×
16 (байт на персонажа) ×
8 (бит в байте) = 345 Кбит/с

Уже гораздо ближе, но все еще выше жесткого лимита. Придется что-то менять — возможно, получится убедить команду дизайнеров в том, что будет достаточно 24 персонажей. Что касается технической стороны, то можно передавать данные персонажей не каждый кадр, а через кадр. Любой из этих вариантов позволит уложиться в установленные границы*.

---

* Вообще-то самый простой вариант — повременить с выпуском игры на пару лет в надежде на увеличение скорости интернет-соединения у ваших пользователей. И это решение на удивление часто оказывается окончательным способом решения проблем с производительностью.

Самое главное здесь в том, что все расчеты делаются до начала реализации. Если расчеты покажут, что предполагаемый дизайн не сработает, то будет гораздо легче перейти на новый до того, как будет написан код. Но с готовым многопользовательским контентом убеждать гейм-дизайнеров в необходимости сокращения количества персонажей до 24 будет куда сложнее!

Важно понимать, что цель таких расчетов — отбросить варианты, которые точно не будут работать, а не подтверждать работоспособность решения. Описанная выше простая схема сетевого соединения может провалиться по множеству других причин, но вы хотя бы будете знать, что она провалилась не по причине того, что кто-то не смог проделать элементарные математические операции.

Когда математика меняется

Вернемся к первому примеру, в котором нужно было выбирать между ручным и автоматизированным процессами копирования и вставки примеров кода из Visual Studio в Word. Основное внимание в нем уделялось оформлению отступов, и расчеты показали, что автоматизация себя не окупает.

Представим теперь, что мы не сразу осознали все условия задачи. Оказывается, недостаточно просто правильно оформить отступы. Всю табуляцию в примерах кода тоже нужно преобразовать в пробелы, потому что так принято в издательстве.

Можем ли мы ориентироваться на прежние расчеты? Конечно нет, потому что мы измеряли процесс, не соответствующий обновленным требованиям. Нужно добавить дополнительные шаги — например, найти в Visual Studio плагин, который преобразует табуляцию в пробелы*, запустить этот плагин для выделенного фрагмента, после копирования добавить еще один шаг для возврата оригинала к исходному виду, а потом измерить весь процесс заново.

Корректировка влияет на обе части уравнения. Два дополнительных шага — преобразование табуляций в пробелы и дополнительный возврат к оригинальному оформлению — замедляют ручной процесс. Допустим, теперь он занимает десять секунд вместо шести, так что выгода от автоматизации увеличивается.

Но модификация повлияла и на цену ручного процесса, потому что вы теперь тратите время на поиск и установку нужного расширения, а затем на

---

* Потому что для вас совершенно неприемлема альтернатива — использование в коде пробелов вместо табуляции. У всех нас есть свои причуды.

эксперименты, чтобы понять, как это все работает. Например, вы узнаете, что процесс сильно зависит от того, как новый плагин взаимодействует со стеком отмены. Будет справедливо добавить это в расчетах к цене, потому что время, потраченное на возню с расширением, вы могли бы потратить на автоматизацию.

Как следствие, после перерасчета баланс изменится. Сначала для ручного процесса:

10 секунд (на одну операцию копирования) ×
8 (примеров кода на главу) ×
20 (глав) ×
3 (правки каждого примера) = 80 минут

С добавлением 45 минут на поиск подходящего плагина, преобразующего табуляцию в пробелы, включая установку и эксперименты, а также с добавлением затрат на написание кода для табуляции, время на автоматизацию увеличится до 90 минут:

80 минут + 45 минут (ручной процесс) > 90 минут (автоматизированный процесс).

Теперь математика показывает, что автоматизацию стоит делать. Конечно, можно продолжать править примеры и вручную, но этот процесс будет медленнее и потребует больше времени на доработку. Проще сразу автоматизировать его.

Обратно от математики к словам

Если вы примете к сведению советы этой главы, то вам будет легче выявлять задачи, требующие некоторых математических расчетов. Важные ориентиры — это количественные ограничения и решения, допускающие измерения. Наличие и того и другого говорит о том, что лучше произвести расчеты и отбросить решения, которые никогда не сработают.

Но будьте внимательны, среди количественно определяемых задач могут скрываться и задачи, требующие качественного подхода! Взять для примера задачу автоматизации процесса — ее не всегда можно решить с помощью простых расчетов.

Ваша главная цель при автоматизации задачи — сократить общее время на ее выполнение… но ведь это может быть не единственная цель. Например, при ручном процессе чаще возникают ошибки. В таком случае можно было

бы рассчитать вероятность ошибок и время, необходимое для их исправлений, но сделать это точно сложно.

Или возьмем для примера ручную задачу, которую следует выполнять ежедневно, но которая настолько раздражает, что ее выполняют только раз в неделю. Тут бессмысленно учитывать только время на ее исполнение. Если после автоматизации она будет выполняться ежедневно, то затраты на ее автоматизацию оправдают себя, даже если сэкономленного времени будет немного.

В качестве одной из «мягких» целей было бы разумно учитывать также психологический комфорт членов команды. Ручная задача может занимать немного времени, но если она постоянно раздражает и если ее можно относительно легко автоматизировать, то ее стоит автоматизировать, даже если оценки выгоды и цены почти совпадают. Не бойтесь иногда потратить день на то, чтобы облегчить жизнь всем, особенно когда расчеты дают близкие результаты.

С другой стороны, если вы плохо понимаете задачу, беритесь за ее автоматизацию с крайней осторожностью, даже если цифры и выглядят убедительно! В предыдущих примерах я описывал автоматизацию одной из моих личных задач, и я знал все ее тонкости и нюансы. Если бы копировал и вставлял примеры кода в эту книгу кто-то другой, то ситуация показалась бы мне не такой прозрачной. Я не был бы уверен в том, что выбрал правильный подход к автоматизации, и тем более не был бы уверен в верности математических расчетов.

Впрочем, доверять цифрам в целом стоит. Если проверить обоснованность какого-то варианта решения задачи можно с помощью несложной и быстрой математической формулы — делайте расчеты.

Правило 21

Иногда нужно просто забивать гвозди

Программирование по своей сути — это творческое и интеллектуально сложное занятие. Во многом именно поэтому я его люблю, и я уверен, что вы сказали бы то же самое. Каждая задача отличается от предыдущих и требует хотя бы немного сообразительности, хотя, как видно по описанным в этой книге Правилам, сообразительности не должно быть слишком много!

Но далеко не у каждой задачи бывает изящное решение. Даже в самом интересном и увлекательном занятии бывает своя рутина; некоторыми задачами трудно восхищаться, и их никто не хочет выполнять. Легко соблазниться чем-то более интересным и отложить рутину на потом, втайне надеясь, что ее выполнят другие члены команды.

Думаю, что после такого вступления вы не удивитесь морали этой главы, а именно моему совету не избегать рутины. Неинтересная задача никуда не денется. Нет никаких волшебных эльфов, которые магическим образом сделают все за вас, пока вы спите. А наполовину выполненные задачи — это яд, медленно отравляющий ваш проект.

Здесь главное вовремя замечать тревожные признаки. Вы люди умные*; достаточно умные в том, чтобы придумывать всякие объяснения тому, почему необязательно прямо сейчас заниматься неинтересными делами. Особенно когда в списке задач полно куда более увлекательных пунктов.

В каком-то смысле осознание того, какие именно задачи вы склонны игнорировать, — это важный элемент самопознания. Ваш список может не совпадать с моим или со списком ваших коллег — что для одного программиста рутина, то для другого, если так можно выразиться, прогулка в парке. Когда вы понимаете, каких задач избегаете, вам проще расположить их по приоритету, которого они заслуживают.

И все же эта глава была бы пустой без примеров! Найти их было несложно — я взял как те задачи, что неприятны лично мне, так и те, которых на моих глазах избегали другие.

---

* И в самом деле, вы добрались до этого Правила, что я считаю признаком ума и сообразительности.

Новый аргумент

Представьте, что у вас есть такая функция:

```
vector<Character *> findNearbyCharacters(
    const Point & point,
    float maxDistance);
```

Функция возвращает всех персонажей внутри некоторой сферы. По всему вашему коду разбросаны десятки вызовов этой функции, и вы обнаружили несколько мест, где она работает не совсем правильно. В этих местах нужно исключить из поиска некоторых персонажей, и вы решили добавить новый аргумент для обработки этого исключения:

```
vector<Character *> findNearbyCharacters(
    const Point & point,
    float maxDistance,
    vector<Character *> excludeCharacters);
```

Теперь перед вами стоит выбор — обновить все места, то есть добавить новый аргумент, где вызывался старый вариант функции, или указать новый аргумент по умолчанию. Например, так:

```
vector<Character *> findNearbyCharacters(
    const Point & point,
    float maxDistance,
    vector<Character *> excludeCharacters = vector<Character *>())
{
    return vector<Character *>();
}
```

Или же можно избежать лишней работы, создав две перегруженные версии функции?

```
vector<Character *> findNearbyCharacters(
    const Point & point,
    float maxDistance);
vector<Character *> findNearbyCharacters(
    const Point & point,
    float maxDistance,
```

```
    vector<Character *> excludeCharacters);
```

Благодаря перегрузке и указанию аргументов по умолчанию можно не обновлять существующие вызовы **`findNearbyCharacters`**. Это же хорошо, верно? Можно переходить к другим задачам из списка.

Да, можно поступить и так, но необязательно. Проходя по местам вызова старого варианта функции, вы не просто конвертируете их; вы также анализируете поведение кода в них. Велика вероятность, что в некоторых случаях какие-то персонажи уже исключаются из поиска, что соответствует новому аргументу. Эти случаи стоит переделать с использованием нового аргумента.

Представьте, что вскоре после этого вам понадобилась более тонкая фильтрация. Допустим, теперь нужно находить не всех персонажей, а только представляющих угрозу ближайших активных врагов. Вы решаете добавить простой интерфейс фильтрации:

```
struct CharacterFilter
{
    virtual bool isCharacterAllowed(Character * character) const = 0;
};

vector<Character *> findNearbyCharacters(
    const Point & point,
    float maxDistance,
    CharacterFilter * filter);
```

Вот пример фильтра, который выделяет активных врагов, но пропускает союзников и выведенных из строя персонажей:

```
struct ThreatFilter : public CharacterFilter
{
    ThreatFilter(const Character * character)
        : m_character(character)
        {;}

    bool isCharacterAllowed(Character * character) const override
    {
        return !character->isAlliedWith(m_character) &&
               !character->isIncapacitated();
    }
```

```
    const Character * m_character;
};
```

Теперь перед вами снова встает выбор. Добавить еще одну перегруженную версию **findNearbyCharacters**? Или две новые перегруженные версии: одну — с фильтром и списком исключаемых персонажей, а другую только с фильтром? Тогда общее количество перегруженных вариантов увеличится до трех или четырех. Синхронизировать три-четыре версии функции уже сложнее. И где расставлять точки останова? Ситуация начинает выходить из-под контроля.

Пожалуй, будет лучше обрабатывать исключаемых персонажей через фильтр. Реализовать **CharacterFilter** с проверкой персонажей по списку — задача тривиальная. Так не придется писать дополнительные варианты функции, и, скорее всего, вы также найдете несколько случаев вызова **findNearbyCharacters**, в которых будет проще указать фильтр:

```
struct ExcludeFilter : public CharacterFilter
{
    ExcludeFilter(const vector<const Character *> & characters)
        : m_characters(characters)
        {; }

    bool isCharacterAllowed(Character * character) const override
    {
        return m_characters.end() == find(
                                        m_characters.begin(),
                                        m_characters.end(),
                                        character);
    }

    vector<const Character *> m_characters;
};
```

Переход на фильтры повсюду подразумевает определенный объем работы. Функция **findNearbyCharacters** вызывается в десятках мест; все эти вызовы нужно просмотреть, и хотя бы часть из них придется перевести на новую модель с фильтрами. Работа кажется скучной и однообразной. Столкнувшись с таким объемом скучной работы, хочется оставить три перегруженные версии и конвертировать только то, что необходимо.

И это было бы ошибкой или, в лучшем случае, разумным решением, но принятым по неправильным причинам. Вы выбираете краткосрочную выгоду — не просматривать и не обновлять существующий код — за счет долгосрочного проигрыша и получаете более запутанную и менее точную модель поиска ближайших персонажей.

Большинство программистов, как правило, выбирают краткосрочную выгоду, а позже об этом жалеют. Если вы понимаете, какое решение в данном случае было бы правильным, но не хотите выполнять его только из-за большого объема работы, просто возьмите волю в кулак и сделайте то, что нужно сделать.

Ошибки не приходят по одной

Вот вам еще пример. Допустим, вы увидели баг, когда один фрагмент кода неправильно вызывает другой фрагмент. В целом ситуация понятна: во втором фрагменте неудачно выбраны имена, и к тому же он не содержит никакой документации:

```
void squashAdjacentDups(
    vector<Unit> & units,
    unsigned int (* hash)(const Unit &));
```

В принципе идея понятна. По всей видимости, функция удаляет («выдавливает», «squashes out») соседние («adjacent») дубликаты («duplicates») с помощью передаваемой хеш-функции. И почти именно это она и делает:

```
void squashAdjacentDups(
    vector<Unit> & units,
    int (* hash)(const Unit &))
{
    int nextIndex = 1;

    for (int index = 1; index < units.size(); ++index)
    {
        if (hash(units[index]) != hash(units[nextIndex - 1]))
        {
            units[nextIndex++] = units[index];
        }
    }
```

```
    while (units.size() > nextIndex)
    {
        units.pop_back();
    }
}
```

Проблема в том, что, судя по тому, как написан код, функция **squashAdjacentDups** ожидает от переданной хеш-функции полностью уникальное значение. Но хеш-функция так не работает. Для двух эквивалентных объектов она действительно вернет одинаковое хеш-значение, но может вернуть то же самое значение и для объекта, который не эквивалентен другому. Поэтому после сравнения хешей всегда нужно проверять эквивалентность объектов, чего функция не делает.

Замеченный нами баг связан как раз с этой особенностью: передает функцию хеширования, а не функцию, возвращающую уникальный идентификатор:

```
struct Unit
{
    int m_id;
    string m_firstName;
    string m_lastName;
    string m_userName;
};

unsigned int hashUnit(const Unit & unit)
{
    return combineHashes(
                hashString(unit.m_firstName),
                hashString(unit.m_lastName),
                hashString(unit.m_userName));
}
```

Работает это почти во всех случаях, поэтому раньше баг и не проявлялся... но стоит двум соседним **Unit** передать одинаковый хеш, как все ломается.

Так что достаточно просто исправить баг и идти дальше? Нет, потому что рутина пока не выполнена.

Во-первых, нужно переименовать аргумент **hash**. Это имя не передает истину, что грозит новыми проблемами. Либо напишем настоящую хеш-

функцию с отдельной проверкой на эквивалентность значений, либо переименуем аргумент, чтобы его имя отражало его фактическое назначение.

Во-вторых, нужно просмотреть все остальные места, где вызывается **`squashAdjacentDups`**. Есть большая вероятность, что по крайней мере в одном из них проявится точно такой же баг, который вы только что исправили. На самом деле, весьма возможно, что все вызовы **`squashAdjacentDups`** содержат эту ошибку. Вы уже проделали работу по диагностике достаточно тонкого бага — воспользуйтесь полученными сведениями, чтобы найти другие вхождения подобной ошибки в коде.

Переименовать аргумент недолго, и с этим, конечно, все согласятся. А вот просмотр всех остальных вызовов **`squashAdjacentDups`** кажется тяжелой и нудной работой. Но она окупится — может, и не для вас лично, но для кого-то из вашей команды. Как всегда: немного боли ради долгосрочной выгоды. Не торопитесь, постарайтесь как следует проанализировать все вызовы и исправить возможные проблемы.

Соблазн автоматизации

Сталкиваясь с какой-либо рутиной, программисты реагируют предсказуемо: они хотят ее автоматизировать.

Автоматизация может принимать разные формы. Можно, например, быстро набросать в редакторе кода какое-то регулярное выражение для поиска всех вызовов **`findNearbyCharacters`** и замены старого аргумента на новый. Или, что лучше, написать небольшой фрагмент кода на Python — так будет проще обрабатывать исключения. Более того, если вам уже приходилось добавлять аргументы, то, пожалуй, стоит написать на Python универсальный инструмент для добавления аргументов. Отличная идея для проекта; приступаем к настоящему делу!

Поверьте, я вас понимаю. Конструировать идеальное регулярное выражение, которое должно выявить абсолютно все варианты текущей скучной задачи, или обдумывать с той же целью скрипт на Python, куда как интереснее, чем вручную править одно и то же. Но это не самое разумное решение. Скорее всего, на все ручные правки вы потратите меньше времени, чем на автоматизацию. Сделайте расчеты, как говорит Правило 20.

Да, иногда можно ускорить выполнение задачи с помощью регулярных выражений, особенно простых и не требующих большой возни. Если с помощью простого выражения можно проделать 80% работы, то почему бы и нет? Оставшиеся 20% можно доделать вручную.

Но имейте в виду: пусть подобные задачки и кажутся повторяющимися, они редко бывают настолько однообразными, чтобы их можно было легко

автоматизировать. Для их выполнения по-прежнему нужно шевелить мозгами, пусть даже речь идет о такой с виду простой задаче, как разбиение вызова функции на несколько строк или переделка комментария в соответствии с новой сигнатурой функции.

Проблема размера файлов

Код со временем развивается и увеличивается в объеме. Как бы ни было полезно и весело удалять лишний код, в ваш проект все равно будет добавляться больше строк, чем удаляться. Рано или поздно наступит такой момент, что длина ваших исходников будет доставлять некоторые неудобства.

Файлы могут разрастаться из-за принятых в вашей команде правил и соглашений. У нас в **Sucker Punch** исходный код конкретного класса принято хранить в одном исходном файле и одном заголовочном файле. И, несмотря на все наши усилия, в итоге у нас оказывается несколько «свалок» вроде класса главного персонажа. Это удобная точка в иерархии классов для добавления функциональности, поэтому в такие файлы-«свалки» добавляют множество разных возможностей, то есть много кода. Я только что проверил, и оказалось, что файл реализации нашего класса главного персонажа содержит 19 000 строк. Ого.

Проблема ли это? Да, по крайней мере небольшая, но проблема. С файлом такого размера труднее работать. Пролистывая его, так ничего просто не найдешь, поэтому приходится пользоваться функцией поиска редактора. Да и компилируется такой файл дольше других, что затрудняет сборку всего проекта. Когда разные фрагменты кода разделяют тысячи строк, труднее понять, как они связаны между собой.

Так почему же мы до сих пор не исправили эту ситуацию? Потому что для уменьшения файла нужно будет проделать много рутинной работы: копировать и вставлять код в новые файлы, заниматься рефакторингом и переносом блоков поведения в отдельные классы (согласно нашему соглашению «один исходный файл на класс»), пересматривать заголовочные файлы для старого и нового файлов, а после переноса устранять все «мертвые» ссылки. Такая работа вообще не доставляет никакого удовольствия, а мы и без того все заняты своими делами. В итоге мы коллективно пришли к решению «не раскапывать могилу» и оставить всё как есть, хотя всем нам было бы гораздо приятнее работать с более короткой версией файла.

Стоит сказать, что Sucker Punch — слаженный коллектив, и каждый сотрудник ежедневно доказывает свою приверженность порядку, поддерживая чистоту и работоспособность кодовой базы. Что касается первых двух примеров, то каждый из нас выбрал бы более тяжелый путь — стал бы обновлять

код под новые аргументы или искать ошибки, похожие на ошибки из второго примера. Но у нас все еще есть исходный файл на 19 000 строк, за который нам всем слегка неловко.

Поймите, что приниматься за скучную рутину сложно даже для дисциплинированной команды. Первым шагом следует признать, что вы избегаете какой-то задачи только потому, что вам не хочется ее выполнять. Вторым шагом нужно как следует подумать и оценить долгосрочную выгоду от ее выполнения; вполне может оказаться так, что особой пользы от выполнения неприятной задачи не будет, так что вы можете и дальше с чистой совестью ее игнорировать. Но если в долгосрочной перспективе она окупится, пусть даже в ближайшей перспективе будет откровенно тяжело, то пора переходить к третьему шагу — забивать гвозди.

Коротких путей не бывает

Представьте, что у вас есть большая деревянная доска с сотней торчащих из нее гвоздей. Из-за них эту доску невозможно использовать ни для чего другого. Можно просто игнорировать гвозди. Можно надеяться, что их за вас забьет кто-нибудь другой. Можно потратить кучу времени, придумывая машинку для удаления гвоздей, которая, как вы надеетесь, заработает (но, возможно, и не заработает).

А можно просто взять в руки молоток и приступить к нудной, но очень нужной работе. Иногда гвозди нужно просто забивать.

Заключение

Настраивайте правила под себя

Изложенные в этой книге Правила — квинтэссенция практических уроков, которые мы усвоили, работая в Sucker Punch на протяжении четверти века существования нашей студии. Они отражают наш личный опыт. Они имеют отношение к тому, что мы считаем важным для себя, — нашу культуру программирования. А эта культура, в свою очередь, отражает конкретные ограничения и особенности разработки тех видеоигр, которые делает Sucker Punch.

Вы прочитали уже немало Правил. Наверняка какие-то из них сразу же показались вам полезными для вашей собственной работы; к другим же вы пока относитесь с некоторым сомнением. И это неудивительно! Если ваша работа радикально отличается от нашей, то некоторые наши Правила действительно вам не подходят.

Так что же делает наши игры такими особенными и как это влияет на правила?

- Во-первых, срок разработки наших проектов довольно большой. На создание нашей последней игры, Ghost of Tsushima, ушло около шести лет. И мы начинали не с нуля; очень много фрагментов кода представляли собой развитие кода предыдущих игр студии Sucker Punch (или же были напрямую скопированы из них). Мы вынуждены уделять первостепенное внимание качеству кода: с очень большой вероятностью то, что мы пишем сегодня, должно будет работать и через десять лет.
- У нас большая команда программистов — около тридцати штатных разработчиков. Вам же, в зависимости от вашей ситуации, такое количество может показаться как крошечным, так и огромным. Лично я считаю «маленькой» такую команду, в которой один человек знает все подробности кода. По этому критерию команда Sucker Punch давно перестала быть маленькой. Сейчас никто из нас не знает абсолютно всех деталей кода, и каждому приходится время от времени разбираться в незнакомых ему фрагментах. Если код не будет легким для чтения и понятным, у нас возникнут серьезные проблемы.

- Для видеоигр критически важен параметр производительности — куда больше, чем для большинства программ. На некоторых сайтах нашу производительность измеряют с точностью до миллисекунды! Но это не значит, что весь наш код должен работать быстро. Как и в любом проекте, какие-то фрагменты кода в этом отношении чувствительнее других. Какие-то действительно должны выполняться быстро, но почти все должны быстро разрабатываться.
- Мы выпускаем игры редко. Это не относится ко всем играм, так как мобильные проекты обновляются постоянно, но мы занимаемся другой категорией игр. Благодаря этому нам легче осуществлять крупные изменения в коде. Кроме того, нам не нужно слишком сильно волноваться о качестве; код должен оставаться надежным и корректным, иначе будут недовольны 80% сотрудников Sucker Punch, не связанных с программированием, но вносимые нами изменения сказываются на пользовательском опыте далеко не сразу. Мы можем позволить себе временные баги, если так разработка игр будет идти быстрее.
- Каждую игру мы начинаем с чистого листа. Да, новые игры в какой-то степени строятся на основе предыдущих, но мы не связаны с ними жесткими ограничениями. Нам не нужно поддерживать обратную совместимость, и нам легче принимать решения о масштабных изменениях.
- Наш процесс разработки игр носит итеративный характер. Успех приходит в результате множества проб и поиска рабочих вариантов, а не от того, что мы сначала проектируем на бумаге, а затем реализуем замысел. Частично рабочие идеи дорабатываются и переосмысливаются; не работающие идеи сразу же отсеиваются. Наш приоритет — быстрая разработка и итерация, но при этом мы помним, что прошедший испытания код, скорее всего, не останется навсегда. Это непростое сочетание.

Все эти особенности напрямую влияют на Правила. Например, то, что мы выпускаем игры редко, сильно меняет отношение к крупным изменениям кода. Если бы у нас были еженедельные релизы, то все пришлось бы делать иначе.

Действуйте по своему усмотрению

Вам может показаться, что некоторые правила подталкивают вас к противоречивым решениям. Допустим, в вашей команде принято, чтобы к защищенному состоянию объекта обращались функции **get** и **set**, но вы понимаете,

что вам придется прописывать функцию **set**, которая, насколько вы знаете, никогда не будет вызываться. Это конфликт между соблюдением командных соглашений (Правило 12) и удалением неиспользуемого кода (Правило 8). В таких случаях полагайтесь на свое профессиональное чувство меры; я бы придерживался командных соглашений для простой функции **set**, но это лишь мой выбор.

Вам также придется самим решать, какие Правила соответствуют специфике вашей работы. Если ваша работа довольно далека от того, чем мы занимаемся в Sucker Punch, то некоторые из наших Правил могут плохо вписываться в вашу ситуацию. В таком случае не следуйте Правилам слепо. Правила — это не догма, а лишь полезные ориентиры.

Но велик шанс и того, что какое-то Правило верно и для вас, даже если вам поначалу трудно с ним согласиться. Сейчас я соглашаюсь с немалым количеством того, что отверг бы лет десять-пятнадцать назад. Взять для примера Правило 10 «Локализируйте сложность». В первые годы своей работы в Sucker Punch я проектировал и создавал множество систем, представлявших собой по сути клубок взаимодействующих объектов. И лишь годы (и множество неудачных реализаций) спустя я понял фундаментальную суть своей ошибки. Правило 10 родилось из неудач и из наших более поздних успехов, когда мы научились локализировать сложность.

Обсуждайте Правила между собой

Эта книга ни в коем случае не претендует на полноту, и описанные в ней Правила — это лишь некоторые рекомендации. Рассматривайте их как основу для размышлений, а не как окончательное руководство к действию. Постарайтесь разработать свой собственный набор Правил.

Очевидно, эти Правила будут работать лучше, если вы с вашими коллегами придерживаетесь единой системы взглядов. Если каждый из вас будет выбирать для себя отдельные правила самостоятельно, то ожидайте хаоса и конфликтов. Вряд ли вам захочется работать в коллективе, где каждый слепо настаивает на своем.

Вот вам идея: устройте некоторое подобие «книжного клуба». Пусть все участники команды читают по одному-другому Правилу, а затем обсуждают их применимость к вашим проектам. Подумайте, как можно было бы изменить то или иное не очень удачное, по вашему мнению, Правило, чтобы оно лучше подходило к вашей ситуации. Или же полностью откажитесь от него, если такой вариант кажется вам самым разумным.

Если ваш коллектив похож на большинство технических команд, то вряд ли вы так часто обсуждаете между собой философию программирования.

А если и обсуждаете, то обычно в контексте конкретной технической проблемы, требующей решения, и тогда философские термины неизбежно смешиваются с техническими. Для реального прогресса этого недостаточно. Лучше разделить эти темы, и шанс прийти к правильному решению будет выше.

Вы будете гораздо эффективнее работать, придерживаясь каких-то общих, принятых для всего коллектива соглашений о коде. Быстрее всего добиться такой согласованности можно благодаря обсуждениям. Описанные в этой книге Правила послужат вам хорошей отправной точкой для таких обсуждений. Они придадут вашим рассуждениям структуру и послужат основой для выработки общего подхода к написанию кода. И такие обсуждения общей философии программирования стоят потраченного на них времени — впоследствии они окупятся многократно.

Завершение

Вот и все! Правил больше не будет!

Я писал эту книгу с удовольствием и надеюсь, вам было интересно ее читать.

Любыми впечатлениями или комментариями можно поделиться на сайте *The Rules of Programming** — там есть нужные ссылки. Обещаю, ваши сообщения не отправятся прямиком в dev/null. На сайте также можно найти использованные в книге примеры исходного кода.

* https://oreil.ly/jTEGo

Приложение А

Чтение C++ для программистов на Python

Примеры в этой книге представлены на языке C++, на котором я программирую чаще всего и которым владею лучше всего. Впрочем, я также написал изрядное количество кода и на языке Python — это второй по популярности язык программирования в Sucker Punch. На данный момент в нашей кодовой базе около 2,8 миллиона строк на C++ и около 600 000 строк на Python.

Если вы программируете на Python, вам необязательно специально изучать язык C++, чтобы читать примеры из этой книги. В основе своей код есть код: цикл — это цикл, переменные — это переменные, а функции — это функции. Да, между этими языками имеются некоторые поверхностные различия, но базовые идеи из показанных в этой книге примеров на языке C++ без особого труда можно напрямую перевести на Python, даже если перевод этот очевиден не сразу!

В этом приложении как раз и объясняются принципы такого перевода. Прочитав его, вы не научитесь писать на C++ — для этого потребовалась бы целая книга, но понимать код вам будет гораздо легче.

Типы

Лучший способ показать прямолинейность восприятия кода на языке C++ для программиста на Python — это разобрать некоторые примеры. Вот простая функция, вычисляющая сумму массива чисел. Сначала на Python:

```
def calculateSum(numbers):

    sum = 0

    for number in numbers:
        sum += number

    return sum
```

А теперь на C++:

```
int calculateSum(const vector<int> & numbers)
{
    int sum = 0;
    for (int number : numbers)
        sum += number;

    return sum;
}
```

В принципе, код тот же самый, верно? Да, в версии на C++ имеются некоторые «добавки», но переменные и логика те же.

Вы, как программист на Python, в примерах этой книги можете практически игнорировать фигурные скобки и точки с запятой. Фигурные скобки и точки с запятой в C++ выделяют блоки кода подобно отступам в Python. Впрочем, в этих примерах на C++ я также использую отступы, потому что они облегчают чтение; фигурные скобки и точки с запятой не несут в этом случае большой ценности*.

Самая непонятная часть C++ для программиста на Python в классическом его стиле — это указание типов переменных, то есть синтаксис вроде **`int`** и **`const vector<int>&`**. Такие аннотации (объявления) типов сообщают компилятору C++, какого вида значения следует ожидать для помеченных переменных или аргументов; в данном случае это целое число и список целых чисел. Компилятору нужно знать типы до того, как он сможет скомпилировать код.

В Python, конечно же, тоже есть типы, пусть даже язык и не заставляет вас задумываться о них. К какому типу принадлежит переменная в Python, обычно выясняется во время выполнения кода, а не во время компиляции. Узнать фактический тип того или иного выражения всегда можно с помощью вызова `isinstance()`.

Явное объявление типов в C++ имеет свои преимущества, самое важное из которых — более легкое обнаружение багов на раннем этапе, но из-за этого приходится писать чуть больше кода. Python же позволяет пропустить некоторые шаги, обязательные для C++, отчего писать небольшие фрагменты кода на Python становится проще.

---

* Поэтому Гвидо от них и избавился.

То, что свои преимущества имеются у обоих методов, понятно по нововведениям в этих языках: последние версии C++ позволяют во многих случаях пропускать аннотации типов, а последние версии Python, в свою очередь, позволяют их добавлять. Теперь и на Python можно писать код, более похожий на C++:

```
def calculateSum(numbers: Iterable[int]) -> int:

   sum:int = 0

   for number in numbers:
        sum += number

    return sum
```

А на C++ писать код, более похожий на Python:

```
auto calculateSum(const vector<int> & numbers)
{
    auto sum = 0;
    for (auto number : numbers)
        sum += number;

    return sum;
}
```

В этой книге примеры ориентируются на «классический» C++ с явно объявляемыми типами. Такова наша политика в Sucker Punch (мы считаем, что это делает код легче для чтения), поэтому я продолжил такую практику и здесь.

Форматирование и комментарии

Иногда бывает так, что общая структура в C++ и Python одинакова, но синтаксис, используемый для ее выражения, различается сильнее, чем в первом примере. Вот функция, которая встречается в Правиле 1 и выполняет объединение двух массивов, один с помощью тасования типа «риффл»*. Сначала на Python:

* Подобно тому, как перемешивают колоду карт. Или фишки для покера, если вы пытаетесь выглядеть как профессиональный игрок в покер.

```
# Тасование типа «риффл» для двух списков: объединяет их в один
# случайным образом, с выбором элементов то из одного,
# то из другого списка, пока оба списка не будут исчерпаны.
def riffleShuffle(leftValues, rightValues):
    leftIndex = 0
    rightIndex = 0

    shuffledValues = []

    while leftIndex < len(leftValues) or \
          rightIndex < len(rightValues):

        if rightIndex >= len(rightValues):
            nextValue = leftValues[leftIndex]
            leftIndex += 1
        elif leftIndex >= len(leftValues):
            nextValue = rightValues[rightIndex]
            rightIndex += 1
        elif random.randrange(0, 2) == 0:
            nextValue = leftValues[leftIndex]
            leftIndex += 1
        else:
            nextValue = rightValues[rightIndex]
            rightIndex += 1

        shuffledValues.append(nextValue)

    return shuffledValues
```

Алгоритм прост: очередное значение для добавления в список перемешанных значений выбирается то из одного, то из другого списка, пока оба списка не будут исчерпаны.

На C++ тот же код выглядит так:

```
// Тасование типа «риффл» для двух списков: объединяет их в один
// случайным образом, с выбором элементов то из одного,
// то из другого списка, пока оба списка не будут исчерпаны.

vector<int> riffleShuffle(
```

```
    const vector<int> & leftValues,
    const vector<int> & rightValues)
{
    int leftIndex = 0;
    int rightIndex = 0;

    vector<int> shuffledValues;

    while (leftIndex < leftValues.size() ||
           rightIndex < rightValues.size())
    {
        int nextValue = 0;

        if (rightIndex >= rightValues.size())
        {
            nextValue = leftValues[leftIndex++];
        }
        else if (leftIndex >= leftValues.size())
        {
            nextValue = rightValues[rightIndex++];
        }
        else if (rand() % 2 == 0)
        {
            nextValue = leftValues[leftIndex++];
        }
        else
        {
            nextValue = rightValues[rightIndex++];
        }
        shuffledValues.push_back(nextValue);
    }
    return shuffledValues;
}
```

Опять же, структура кода идентична. Оба языка обладают схожими базовыми возможностями, и поэтому легко увидеть, как соотносятся друг с другом различные части программы, однако при этом можно заметить множество мелких синтаксических различий.

Комментарии

Рассмотрим для начала комментарии. В C++ предусмотрено несколько способов комментирования кода, но в примерах этой книги используется двойная косая черта («прямой слеш»):

```
// Тасование типа «риффл» для двух списков: объединяет их в один
// случайным образом, с выбором элементов то из одного,
// то из другого списка, пока оба списка не будут исчерпаны.
```

Сравните с комментариями в Python, которые начинаются с символа #:

```
# Тасование типа «риффл» для двух списков: объединяет их в один
# случайным образом, с выбором элементов то из одного,
# то из другого списка, пока оба списка не будут исчерпаны.
```

Отступы и переносы строк

Также различаются правила переноса строк. В C++ пробелы, табуляции и переводы строк взаимозаменяемы, поэтому для разделения условия цикла **`while`** на две строки не требуется специального синтаксиса. Достаточно просто заменить пробел на новую строку:

```
while (leftIndex < leftValues.size() ||
       rightIndex < rightValues.size())
```

В Python переносы строк играют особую роль, поэтому запись условия в две строки требует явного символа продолжения строки — обратной косой черты («обратный слеш»):

```
while leftIndex < len(leftValues) or \
      rightIndex < len(rightValues):
```

Отступы в C++ также допускают более свободную форму, к чему программисту на Python, возможно, потребуется привыкнуть. Формально блоки кода разделяются фигурными скобками и точками с запятой, а сами по себе отступы не несут смысловой нагрузки. В этом примере два фрагмента выровнены одинаково, но это сделано исключительно для удобства чтения.

Логические операции

Логические операции в C++ представлены символами. В условии цикла C++ используется оператор `||`, тогда как в Python используется гораздо более простой для восприятия **`or`**. Точно так же в C++ используются **`&&`** и **`!`**, тогда как в Python — **`and`** и **`not`**.

Иногда соответствия в двух языках бывают не такими прямыми. Функция **`rand`** в C++ возвращает случайное целое число; в этом примере для случайного выбора вектора-источника производится проверка случайного числа на четность или нечетность. Символ **`%`** здесь означает операцию определения остатка от деления целых чисел; в данном случае получается либо **`0`**, если число четное, либо **`1`**, если число нечетное:

```
else if (rand() % 2 == 0)
```

В Python для того же самого используется функция **`randrange`** из модуля **`random`**:

```
elif random.randrange(0, 2) == 0:
```

Списки

То, что в Python называется «списками» (lists), в C++ называется «векторами» (vectors). Несмотря на разницу в названиях, списки в Python и векторы в C++ работают практически одинаково. Для добавления элемента в конец вектора в C++ используется довольно изящное имя метода **`push_back`** («толкнуть назад»):

```
shuffledValues.push_back(nextValue);
```

Метод **`append`** («добавить») в Python гораздо понятнее:

```
shuffledValues.append(nextValue)
```

Для получения длины вектора в C++ используется метод **`size`**, а не вызов глобальной функции **`len`**, как в Python.

Операторы инкремента

И наконец, если рассматривать наш пример (как удивительно много различий можно найти в таком коротком фрагменте), в C++ предусмотрено синтаксическое сокращение для увеличения или уменьшения переменной (операций «инкремента» и «декремента»). Следующее выражение извлекает из **leftValues** значение с индексом **leftIndex**, увеличивает **leftIndex** на единицу, а затем помещает извлеченное значение в **nextValue**:

```
nextValue = leftValues[leftIndex++];
```

Та же логика на Python занимает две строки:

```
nextValue = leftValues[leftIndex]
leftIndex += 1
```

Итак, в целом можно заметить немало мелких различий, но в данном примере на C++ для всего есть очень близкие аналоги в Python.

Классы

И C++, и Python поддерживают классы, и синтаксис для их объявления не так уж сильно различается. Реализация классов в этих языках отличается заметно, но для примеров в этой книге это неважно. Вам будет легче понимать примеры, если вы будете воспринимать класс C++ как класс Python, для которого все атрибуты экземпляра устанавливаются в методе **__init__**.

Вот класс на Python, реализующий концепцию трехмерного вектора:

```
class Vector:

    _vectorCount = 0

    def __init__(self):
        self.x = 0
        self.y = 0
        self.z = 0
        self._length = 0
        Vector._vectorCount = Vector._vectorCount + 1
```

```
    def __del__(self):
        Vector._vectorCount = Vector._vectorCount - 1

    def set(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        self._calculateLength()

    def getLength(self):
        return self._length

    def getVectorCount():
        return Vector._vectorCount

    def _calculateLength(self):
        self._length = math.sqrt(self.x ** 2 + self.y ** 2 + self.z **
2)
```

Для любого класса 3D-вектора будут отслеживаться три координаты: *x*, *y* и *z*. Этот конкретный класс также кеширует длину вектора и подсчитывает количество существующих в данный момент векторов. Две этих последних возможности, кстати, не имеют особого смысла для данного примера и показаны исключительно для иллюстрации пары синтаксических различий между Python и C++.

Вот эквивалент на C++:

```
class Vector
{
public:
    Vector() :
        m_x(0.0f),
        m_y(0.0f),
        m_z(0.0f),
        m_length(0.0f)
       { ++s_vectorCount; }
    ~Vector()
       { --s_vectorCount; }
```

```
    void set(float x, float y, float z);

    float getLength() const
    {
        return m_length;
    }

    static int getVectorCount()
    {
        return s_vectorCount;
    }

protected:

    void calculateLength();

    float m_x;
    float m_y;
    float m_z;
    float m_length;

    static int s_vectorCount;
};

void Vector::set(float x, float y, float z)
{
    m_x = x;
    m_y = y;
    m_z = z;
    calculateLength();
}

void Vector::calculateLength()
{
    m_length = sqrtf(m_x * m_x + m_y * m_y + m_z * m_z);
}
```

Я здесь непреднамеренно продемонстрировал одно из преимуществ Python — обычно он более лаконичен, чем C++. Версия этого примера на Python занимает примерно в два раза меньше строк, чем версия на C++.

Эти две версии класса содержат одни и те же части, только расположены эти части по-разному. Иногда перестановка очевидна. В Python метод **__init__** вызывается при создании объекта, а **__del__** — при его уничтожении:

```
class Vector:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.z = 0
        self._length = 0
        Vector._vectorCount = Vector._vectorCount + 1

    def __del__(self):
        Vector._vectorCount = Vector._vectorCount - 1
```

В C++ для этих двух методов используется имя самого класса с добавлением тильды (~) перед вторым методом. Эти два метода в C++ называются конструктором и деструктором. Также существует специальный синтаксис для инициализации переменных экземпляра:

```
class Vector
{
public:
    Vector() :
        m_x(0.0f),
        m_y(0.0f),
        m_z(0.0f),
        m_length(0.0f)
    { ++s_vectorCount; }
    ~Vector()
    { --s_vectorCount; }
};
```

Для доступа к переменным класса используется немного другой синтаксис. В Python нужно явно указывать это с помощью ключевого слова **self**:

```
class Vector:
    def getLength(self):
        return self._length
```

В C++ это необязательно. Можно, конечно, использовать **this** (эквивалент **self** в C++), но переменные класса уже находятся в области видимости. Компилятор может неявно искать переменные-члены класса за вас:

```
class Vector
{
    float getLength() const
    {
        return m_length;
    }
};
```

Если вы видите в C++ нечто похожее на ссылку на переменную, которая еще не была определена, скорее всего, это переменная класса.

Видимость

Python и C++ по-разному управляют теми частями внутренней логики класса, к которым пользователи класса не должны иметь доступа. В Python действует следующее соглашение: имя, начинающееся с символа подчеркивания, считается внутренним. В примере на Python так обозначается переменная класса:

```
class Vector:
    _vectorCount = 0
```

А также для метода:

```
class Vector:
    def _calculateLength(self):
        self._length = math.sqrt(self.x ** 2 + self.y ** 2 + self.z ** 2)
```

C++ решает ту же задачу с помощью синтаксиса, а не соглашений. Ключевое слово **public** в начале объявления класса **Vector** помечает последующие члены как видимые и доступные для любого кода, в котором имеется экземпляр объекта. Расположенное ниже ключевое слово **protected** скрывает последующие части от любых внешних по отношению к классу частей кода. Как следствие, пользователи класса могут вызывать метод **set**, но не могут вызвать напрямую метод **calculateLength**. Его можно вызывать только изнутри других методов класса **Vector**.

```
Class Vector
{
public:

    void set(float x, float y, float z);

protected:
    void calculateLength();
};
```

Объявления и определения

Рассмотрим теперь, как объявление функции отделяется от ее определения. В Python все методы класса определяются в самом классе:

```
class Vector:
    def set(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        self._calculateLength()

    def getLength(self):
        return self.length
```

Во многих примерах в этой книге то же самое делается и для C++:

```
class Vector
{
public:
    float getLength() const
    {
        return m_length;
    }
};
```

В других же примерах объявления метода отделяются. Сначала в коде прописывается само существование метода с данным именем и сигнатурой типа:

```
class Vector
{
public:
    void set(float x, float y, float z);
};
```

Затем отдельно определяется функция:

```
void Vector::set(float x, float y, float z)
{
    m_x = x;
    m_y = y;
    m_z = z;
    calculateLength();
}
```

Эти две формы обычно компилируются в C++ по-разному; методы, определенные внутри класса, компилируются как встраиваемые, что означает генерацию отдельной копии функции и ее подстановку в каждое место вызова, тогда как для функций с отдельным определением существует единственная копия. Однако для примеров в этой книге это различие не имеет значения, поэтому не стоит заострять на нем внимание.

Последнее, на что стоит обратить внимание в этом примере, — это то, как C++ работает с атрибутами класса. В Python атрибуты класса определяются операторами внутри класса, а атрибуты экземпляра добавляются в методе **`__init__`**:

```
class Vector:
    _vectorCount = 0

    def __init__(self):
        self.x = 0
        self.y = 0
        self.z = 0
        self._length = 0
```

В C++ и атрибуты класса, и атрибуты экземпляра (или «члены» в терминологии C++) добавляются в определении класса. Ключевое слово **`static`**

помечает атрибуты класса; все, что не помечено **static**, будет атрибутом экземпляра:

```
class Vector
{
protected:
    float m_x;
    float m_y;
    float m_z;
    float m_length;

    static int s_vectorCount;
};
```

Перегрузка функций

В C++ есть и другие особенности, не имеющие прямых аналогов в Python. Разумеется, верно и обратное — в Python имеется много особенностей, отсутствующих в C++, но это здесь не важно, поскольку главное для нас сейчас — это уметь читать код на C++.

Среди того, что может сбить с толку программистов на Python, можно назвать наличие двух функций с одинаковым именем. Вот пример на C++:

```
int min(int a, int b)
{
    return (a <= b) ? a : b;
}

int min(int a, int b, int c)
{
    return min(a, min(b, c));
}

void example()
{
    printf("%d %d\n", min(5, 8), min(13, 21, 34));
}
```

Казалось бы, не совсем понятно, как это работает. Какая именно функция будет выбрана при попытке ее вызова? Компилятор C++ принимает это

решение на основе аргументов. Если переданы два целых числа, вызывается первая версия; если три целых числа — вторая. В данном случае функция **example** выведет 5 и 13.

Заодно в этой функции я указал еще одну полезную синтаксическую деталь C++, а именно оператор вопросительного знака, позволяющий выбирать между двумя значениями на основе определенного выражения:

```
return (a <= b) ? a : b;
```

Эквивалент его в Python выглядит не менее своеобразно:

```
return a if a <= b else b
```

Если уж на то пошло, то больше меня сбивает с толку синтаксис Python. Так что если вы хотите, чтобы программисты на C++ тоже почувствовали себя немного не в своей тарелке, обязательно добавляйте в свой код много тернарных выражений.

Шаблоны

Другая концепция C++, не имеющая аналога в Python — это шаблоны. Если говорить упрощенно, то шаблоны C++ — это способ написать один блок кода, который работает с несколькими типами. Для каждого набора типов в шаблоне компилятор генерирует новый код.

В первом примере в этом приложении вычислялась сумма целочисленных значений массива. Если бы вам понадобилось вычислить сумму массива с вещественными значениями, то для этого пришлось бы написать новый код:

```
float calculateSum(const vector<float> & numbers)
{
    float sum = 0;

    for (float number : numbers)
        sum += number;

    return sum;
}
```

Либо можно было бы написать одну-единственную версию **calculateSum** с шаблоном и позволить сделать всю остальную работу за вас компилятору:

```
template <class T>
T calculateSum(const vector<T> & numbers)
{
    T sum = T(0);

    for (T number : numbers)
        sum += number;

    return sum;
}
```

Версия **calculateSum** с шаблоном будет работать для любого типа, который реализует оператор **+=** и поддерживает значение 0. Это верно как для целых чисел, так и для вещественных, но также легко можно записать шаблоны и для других типов. Например, можно было бы реализовать оператор **+=** и поддержку значения **0** для класса **Vector** из примеров в предыдущих разделах и получить возможность суммировать массив векторов.

В Python все это не нужно. Написанный на Python код для суммирования значений списка прекрасно работает для любого типа, поддерживающего сложение и значение **0**:

```
def calculateSum(numbers):

    sum = 0

    for number in numbers:
        sum = sum + number

    return sum
```

Иными словами, в тех случаях, когда вы видите в примерах на C++ синтаксис шаблонов, он обычно отмечает некоторый фрагмент кода, который в Python прекрасно работал бы без всяких шаблонов.

Указатели и ссылки

Есть еще один аспект, который Python в основном скрывает, а C++ заставляет программистов о нем задумываться, и это передача аргументов по значению или передача аргументов по ссылке. В Python такие простые типы данных,

как числа или строки, передаются по значению, то есть при присваивании их переменной или передаче в функцию создается их новая копия. Для таких типов, как списки или объекты, ситуация сложнее. Вот пример:

```
def makeChanges(number, numbers):

    number = 3
    numbers.append(21)
    print(number, numbers)

globalNumber = 0
globalNumbers = [3, 5, 8, 13]

print(globalNumber, globalNumbers)
makeChanges(globalNumber, globalNumbers)
print(globalNumber, globalNumbers)
```

При выполнении этот код выводит:

```
0 [3, 5, 8, 13]
3 [3, 5, 8, 13, 21]
0 [3, 5, 8, 13, 21]
```

Обратите внимание на то, как простое значение изменяется внутри **makeChanges** на **3**, а затем, после возврата из функции, становится снова равным **0**, тогда как список исходное состояние уже не принимает. Это происходит потому, что **0** передавался по значению и при вызове **makeChanges** была создана его новая копия. Когда функция **makeChanges** присваивает переменной **number** значение **3**, она меняет только переданную ей копию.

С другой стороны, Python не создавал копию списка **globalNumbers**, а передал оригинальный список. И **numbers**, и **globalNumbers** ссылаются на один и тот же список. Значение **21** добавляется в этот самый список. При выводе **numbers** до возврата из функции и при вызове **globalNumbers** после возврата отображается один и тот же список с добавленным значением **21**.

В C++ передача осуществляется в более явном виде. Все переменные и аргументы явно передаются либо по значению, либо по ссылке. Эквивалент приведенного выше кода на Python на C++ выглядит так:

```
void makeChanges(int number, vector<int> & numbers)
{
```

```
    number = 3;
    numbers.push_back(21);
    cout << number << " " << numbers << "\n";
}
```

Амперсанд перед **numbers** играет важную роль и сообщает компилятору, что этот аргумент передается по ссылке, а не по значению, поэтому при вызове **makeChanges** компилятор не будет создавать его копию. Но перед **number** амперсанд не указан, поэтому компилятор создает копию **number**.

Если амперсанды поменять местами, то компилятор создаст копию **numbers**, но не будет создавать копию **number**:

```
void makeChanges(int & number, vector<int> numbers)
{
    number = 3;
    numbers.push_back(21);
    cout << number << " " << numbers << "\n";
}
```

Эта версия выдает другой результат. Теперь именно простое значение меняется навсегда, а список возвращается к своему исходному значению:

```
0 [3 5 8 13]
3 [3 5 8 13 21]
3 [3 5 8 13]
```

В примерах этой книги часто используются ссылки — так сокращаются затраты памяти и ресурсов. В большинстве случаев то, что не должно меняться, несмотря на передачу по ссылке, помечается ключевым словом **const** («константа»). Такой прием (хотя он до сих пор и не объяснялся) можно заметить уже в первом примере на языке C++ в этом приложении:

```
int calculateSum(const vector<int> & numbers)
```

В этом случае использовании копия **numbers** не создается, но и компилятор не позволяет вносить в **numbers** изменения. На практике такое решение похоже на передачу по значению, но с гораздо меньшими затратами.

И еще одно замечание напоследок: в нескольких случаях в примерах используется еще одно (да, никто не говорил, что будет легко) синтаксическое решение C++ для передачи по ссылке, а именно указатели. Указатели

и ссылки — это почти одно и то же, и различие между ними не так уж важно, если вы просто стараетесь понять, что делает код. Ключевые различия заключаются в следующем:

- Указатели обозначаются с помощью знака * вместо &.
- Для доступа к членам указатели используют -> вместо . (точки).
- Для преобразования указателя в ссылку используют *; для преобразования ссылки в указатель используют &.

Вот пример кода с указателями:

```
void example(int number, vector<int> * numbers)
{
    number = 3;
    numbers->push_back(21);
    cout << number << " " << *numbers << "\n";
}

void callExample()
{
    int number = 0;
    vector<int> numbers = { 3, 5, 8, 13 };
    cout << number << " " << numbers << "\n";
    example(3, &numbers);
    cout << number << " " << numbers << "\n";
}
```

В примерах этой книги константные ссылки используются в тех случаях, когда по идее аргумент должен был бы передаваться по значению, но фактическая передача по значению была бы слишком ресурсозатратной. Во всех остальных случаях используются указатели.

Приложение Б

Чтение C++ для программистов на JavaScript

Примеры в этой книге представлены на языке C++, на котором я программирую чаще всего и которым владею лучше всего.

Если же вы пишете на JavaScript, не отчаивайтесь, «Правила» все равно будут для вас полезны! Для этого вам вовсе не нужно будет изучать C++. В основе своей код есть код: цикл — это цикл, переменные — это переменные, а функции — это функции. Да, между этими языками имеются некоторые поверхностные различия, но базовые идеи из показанных в этой книге примеров на C++ без особого труда можно напрямую перевести на JavaScript, даже если перевод этот очевиден не сразу!

В этом приложении объясняется, как делать такой перевод — как читать C++ и мысленно превращать его в эквивалент на JavaScript. Прочитав его, вы не научитесь писать на C++ — для этого понадобилась бы целая книга, — но понимать код вам будет гораздо легче.

Типы

Пора привести конкретный пример! Вот простая функция, вычисляющая сумму массива чисел, сначала на JavaScript*:

```
function calculateSum(numbers) {

    let sum = 0;
    for (let number of numbers)
        sum += number;

    return sum;
}
```

* Для примеров мне нужно было выбрать версию JavaScript, хорошо это или плохо, но я выбрал ES6. Прошу прощения, если вы до сих пор работаете с более ранней версией.

А теперь на C++:

```
int calculateSum(const vector<int> & numbers)
{
    int sum = 0;

    for (int number : numbers)
        sum += number;

    return sum;
}
```

Хм... как будто почти один и тот же код. Может, никакое приложение и не понадобится?

Или понадобится? Синтаксис JavaScript во многом был вдохновлен синтаксисом языка C, так что тут мне не придется объяснять, что значат фигурные скобки и точки с запятой, в отличие от приложения, посвященного языку Python. Но и здесь отличий все же достаточно.

Во-первых, в C++ каждая переменная имеет свою область видимости. Представьте, что перед каждой переменной в примерах неявно стоит **`let`** или **`const`** — это автоматически верно для всех переменных в C++.

Если вы пока не пробовали TypeScript — то есть до сих пор работали только с «чистым» JavaScript, — то явное указание типов переменных в примере может вас смутить. Аннотации **`int`** и **`const vector<int>&`** сообщают компилятору C++, какие значения следует ожидать; в данном случае это целое число и список целых чисел. Компилятору нужно знать типы до того, как он сможет скомпилировать код.

В JavaScript, конечно же, тоже есть типы, пусть даже язык и не заставляет вас задумываться о них. Если тип все же важен, можно вызвать **`typeof()`**, но обычно на это не обращают внимания. В JavaScript тип значения чаще всего определяется во время выполнения кода, а не во время компиляции. Хотя так бывает не всегда, браузер старается по максимуму определить типы сам, потому что так код на JavaScript можно будет скомпилировать более эффективно и выполнять его гораздо быстрее.

C++ же сразу ориентируется на «гораздо быстрее», даже если ради этого и приходится писать чуть больше кода. Явное объявление типов позволяет компилятору находить целый класс ошибок еще до выполнения, что представляет собой огромное преимущество.

То, что свои преимущества имеются у обоих методов, понятно по нововведениям в этих языках: последние версии C++ позволяют во многих случаях пропускать аннотации типов, а все более популярное приложение TypeScript для JavaScript позволяет эти аннотации добавлять. Теперь и на JavaScript/TypeScript можно писать код, более похожий на C++:

```
function calculateSum(numbers: int[]): int {

    let sum: int = 0;

    for (let number of numbers)
        sum += number;

    return sum;
}
```

А на C++ писать код, более похожий на JavaScript:

```
auto calculateSum(const vector<int>& numbers)
{
    auto sum = 0;

    for (auto number : numbers)
        sum += number;

    return sum;
}
```

В этой книге примеры ориентируются на «классический» C++ с явно объявленными типами. Такова наша политика в Sucker Punch (мы считаем, что это делает код легче для чтения), поэтому я продолжил такую практику и здесь.

Массивы

Несмотря на внешнее сходство C++ и JavaScript, различий между ними хватит на весьма объемный справочник. Вот простой пример кода, инвертирующего массив значений. Сначала на JavaScript:

```
function reverseList(values) {
    let reversedValues = []

    for (let index = values.length; --index >= 0; ) {
        reversedValues.push(values[index]);
    }

    return reversedValues;
}
```

А теперь на C++:

```
vector<int> reverseList(const vector<int> values)
{
    vector<int> reversedValues;

    for (int index = values.size(); --index >= 0; )
        reversedValues.push_back(values[index]);

    return reversedValues;
}
```

И снова код выглядит очень похожим. Ближайший аналог массива в JavaScript в C++ — это тип **`vector`** («вектор»). Концептуально они совпадают, но в деталях различаются. В JavaScript количество элементов в массиве сообщает свойство **`length`**. В C++ свойств нет, а только поля данных и методы; количество элементов в структуре данных **`vector`** можно получить вызовом метода **`size()`**.

Точно так же в массивы JavaScript и в «вектора» C++ можно добавлять новые элементы. В C++ для этого используется метод **`push_back()`**, а в JavaScript — **`push()`**. Это довольно простое соответствие.

Стоит отметить, что схожий синтаксис при этом скрывает серьезные различия. Так, например, массив **`values`** в JavaScript может быть списком чего угодно, даже смесью значений разных типов вроде **`[1, «hello», true]`**, тогда как «вектор» C++ всегда содержит только целые числа — ни больше ни меньше.

Впрочем, для чтения примеров на C++ это не важно. Пусть JavaScript и допускает гораздо большую гибкость типов, чем C++, но JavaScript вполне комфортно работает и с простыми списками одного типа.

Классы

И C++, и JavaScript поддерживают классы, и синтаксис для их объявления не так уж сильно различается. Реализация классов в этих языках отличается заметно, но для примеров в этой книге это неважно. Вам будет легче понимать примеры, если вы будете воспринимать класс C++ как класс JavaScript, в котором все поля явно объявлены с модификаторами **public** или **private**.

Вот класс на JavaScript, реализующий концепцию трехмерного вектора:

```
class Vector {

    constructor () {
        ++Vector.#vectorCount;
    }

    set(x, y, z) {
        this.#x = x;
        this.#y = y;
        this.#z = z;
        this.#calculateLength();
    }

    getLength () {
        return this.#length;
    }

    static getVectorCount() {
        return Vector.#vectorCount;
    }

    #calculateLength () {
        this.#length = Math.sqrt(
                                this.#x ** 2 +
                                this.#y ** 2 +
                                this.#z ** 2);
    }

    #x = 0;
    #y = 0;
```

```
    #z = 0;
    #length = 0;

    static #vectorCount = 0;
};
```

А вот эквивалентный класс на C++:

```
class Vector
{
public:
    Vector() :
        m_x(0.0f),
        m_y(0.0f),
        m_z(0.0f),
        m_length(0.0f)
        { ++s_vectorCount; }
    ~Vector()
        { --s_vectorCount; }

    void set(float x, float y, float z);

    float getLength() const
    {
        return m_length;
    }

    static int getVectorCount()
    {
        return s_vectorCount;
    }

protected:

    void calculateLength();

    float m_x;
    float m_y;
    float m_z;
```

```
    float m_length;

    static int s_vectorCount;
};

void Vector::set(float x, float y, float z)
{
    m_x = x;
    m_y = y;
    m_z = z;
    calculateLength();
}

void Vector::calculateLength()
{
    m_length = sqrtf(m_x * m_x + m_y * m_y + m_z * m_z);
}
```

Обе версии класса **Vector** содержат одни и те же части, только расположены эти части по-разному. Соответствие довольно прямое. В JavaScript для доступа к полям и методам экземпляра всегда нужно явно использовать ключевое слово **this**, а для статических членов — имя класса:

```
class Vector {

    getLength () {
        return this.#length;
    }

    static getVectorCount() {
        return Vector.#vectorCount;
    }

    #length = 0;
    static #vectorCount = 0;
};
```

В C++ это тоже можно указывать явно: **this->** для обычных членов и **Vector::** для статических. Но в примерах в книге используется неявный доступ, и этого вполне достаточно:

```
class Vector
{
public:
    float getLength() const
    {
        return m_length;
    }

    static int getVectorCount()
    {
        return s_vectorCount;
    }

protected:
    float m_length;
    static int s_vectorCount;
};
```

Эти примеры также показывают, как в обоих языках реализована видимость членов класса. В JavaScript приватные поля помечаются префиксом `#` («хештег»). В C++ для этого используется ключевое слово **`private`**; оно начинает секцию, где все объявленные методы и поля считаются приватными. Во всех примерах этой книги используется похожее ключевое слово **`protected`**.

Перейдем теперь к конструкторам. В JavaScript при создании нового объекта класса **`Vector`** вызывается специальный метод **`constructor`**:

```
class Vector {
    constructor () {
        ++Vector.#vectorCount;
    }
};
```

В C++ для конструктора используется имя самого класса. Такая запись также называется конструктором (что может слегка сбивать с толку):

```
class Vector
{
public:
```

```
    Vector() :
        m_x(0.0f),
        m_y(0.0f),
        m_z(0.0f),
        m_length(0.0f)
        { ++s_vectorCount; }
};
```

В классах C++ также есть важное понятие, которого практически нет в JavaScript, а именно деструктор. Конструктор вызывается при создании объекта, а деструктор — при его уничтожении. Деструктор обозначается именем класса с тильдой (~) в начале:

```
class Vector
{
public:
    ~Vector()
        { --s_vectorCount; }
};
```

В JavaScript нет прямого аналога деструктора. Ближайшим подобием можно назвать регистрацию обратного вызова через объект **`FinalizationRegistry`**, добавленного относительно недавно (и, по мнению многих, еще не до конца продуманного). Но этот механизм работает не так, как деструктор C++. В C++ деструктор вызывается сразу же, как только объект выходит из области видимости. Вот пример, в котором **`Vector`** создается как локальная переменная внутри функции:

```
void functionA()
{
    printf("%d\n", Vector::getVectorCount());
    Vector a;
    printf("%d\n", Vector::getVectorCount());
};

Vector b;
functionA();
printf("%d\n", Vector::getVectorCount());
```

Этот код при выполнении выводит «1, 2, 1», потому что сразу же при завершении **functionA** вызывается деструктор объекта **a**, и счетчик **s_vectorCount** тут же уменьшается. В JavaScript финализирующий вызов срабатывает при сборке мусора, а ее момент зависит от реализации и не гарантируется. Такого же точного поведения, как в примере на C++, в JavaScript получить нельзя.

И это досадно: отсутствие деструкторов исключает ряд полезных приемов. Мы в Sucker Punch часто пользуемся временем жизни объектов для надежного управления ресурсами, и вы увидите это в примерах на C++. Просто помните, что деструктор вызывается сразу, и вам будет легче следить за кодом.

Объявления и определения

Рассмотрим теперь, как объявление функции отделяется от ее определения. В JavaScript все методы класса определяются в самом классе:

```
class Vector {
    set (x, y, z) {
        this.#x = x;
        this.#y = y;
        this.#z = z;
        this.#calculateLength();
    }

    getLength () {
        return this.#length;
    }
}
```

Во многих примерах в этой книге то же самое делается и для C++:

```
class Vector
{
public:
    float getLength() const
    {
        return m_length;
    }
};
```

В других же примерах объявления метода отделяются. Сначала в коде прописывается само существование метода с данным именем и сигнатурой типа:

```
class Vector
{
public:
    void set(float x, float y, float z);
};
```

Затем отдельно определяется функция:

```
void Vector::set(float x, float y, float z)
{
    m_x = x;
    m_y = y;
    m_z = z;
    calculateLength();
}
```

Эти две формы обычно компилируются в C++ по-разному; методы, определенные внутри класса, компилируются как встраиваемые, что означает генерацию отдельной копии функции и ее подстановку в каждое место вызова, тогда как для функций с отдельным определением существует единственная копия. Однако для примеров в этой книге это различие не имеет значения, поэтому не стоит заострять на нем внимание.

Перегрузка функций

В C++ есть и другие особенности, не имеющие прямых аналогов в JavaScript. Разумеется, верно и обратное — в JavaScript имеется много особенностей, отсутствующих в C++, но это здесь неважно, поскольку главное для нас сейчас — это уметь читать код на C++.

Среди того, что может сбить с толку программистов на JavaScript, можно назвать наличие двух функций с одинаковым именем. Вот пример на C++:

```
int min(int a, int b)
{
    return (a <= b) ? a : b;
```

```
}

int min(int a, int b, int c)
{
    return min(a, min(b, c));
}

void example()
{
    printf("%d %d\n", min(5, 8), min(13, 21, 34));
}
```

Казалось бы, не совсем понятно, как это работает. Какая именно функция будет выбрана при попытке ее вызова? Компилятор C++ принимает это решение на основе аргументов. Если переданы два целых числа, вызывается первая версия; если три целых числа — вторая. В данном случае функция **`example`** выведет **`5`** и **`13`**.

В JavaScript вместо этого написали бы одну функцию **`min`**, поддерживающую произвольное количество аргументов. Например, так:

```
function min() {
    if (arguments.length == 0)
        return Infinity;

    let result = arguments[0];
    for (let index = 1; index < arguments.length; ++index) {
        result = Math.min(result, arguments[index]);
    }

    return result;
}
```

Собственно, именно так и работает встроенная функция **`Math.min`**, поэтому писать ее самим не нужно!

Шаблоны

Другая концепция C++, не имеющая аналога в JavaScript, — это шаблоны. Если говорить упрощенно, то шаблоны C++ — это способ написать один блок

кода, который работает с несколькими типами данных. Для каждого набора типов в шаблоне компилятор генерирует новый код.

В первом примере в этом приложении вычислялась сумма целочисленных значений массива. Если бы вам понадобилось вычислить сумму массива с вещественными значениями, то для этого пришлось бы написать новый код:

```
float calculateSum(const vector<float> & numbers)
{
    float sum = 0;

    for (float number : numbers)
        sum += number;

    return sum;
}
```

Либо можно было бы написать одну-единственную версию **calculateSum** с шаблоном и позволить сделать всю остальную работу за вас компилятору:

```
template <class T>
T calculateSum(const vector<T> & numbers)
{
    T sum = T(0);
    for (T number : numbers)
         sum += number;

    return sum;
}
```

Версия **calculateSum** с шаблоном будет работать для любого типа, который реализует оператор **+=** и поддерживает значение **0**. Это верно как для целых чисел, так и для вещественных, но также легко можно записать шаблоны и для других типов. Например, можно было бы реализовать оператор **+=** и поддержку значения **0** для класса **Vector** из примеров в предыдущих разделах, и тогда можно было бы суммировать массив векторов.

В JavaScript все это не нужно. Написанный на JavaScript код для суммирования значений списка прекрасно работает для любого типа, поддерживающего сложение и значение **0**:

```
function calculateSum(numbers) {
    let sum = 0;

    for (let number of numbers)
        sum += number;

    return sum;
}
```

Поэтому, когда в примерах на C++ вы увидите синтаксис шаблонов, то он обычно отмечает некоторый фрагмент кода, который в JavaScript прекрасно работал бы без всяких шаблонов.

Указатели и ссылки

Есть еще один аспект, который JavaScript в основном скрывает, а C++ заставляет программистов о нем задумываться, и это передача аргументов по значению или передача аргументов по ссылке. В JavaScript такие простые типы данных, как числа или строки, передаются по значению, то есть при присваивании их переменной или передаче в функцию создается их новая копия. Для таких типов, как списки или объекты, ситуация сложнее. Вот пример:

```
function makeChanges (number, numbers) {
    number = 3;
    numbers.push(21);
    console.log(number, numbers);
}

let number = 0;
let numbers = [3, 5, 8, 13];

console.log(number, numbers);
makeChanges(number, numbers);
console.log(number, numbers);
```

При выполнении этот код выводит:

```
0 (4) [3, 5, 8, 13]
3 (5) [3, 5, 8, 13, 21]
0 (5) [3, 5, 8, 13, 21]
```

Обратите внимание на то, как простое значение изменяется внутри **makeChanges** на 3, а затем, после возврата из функции, становится снова равным **0**, тогда как список исходное состояние уже не принимает. Это происходит потому, что **0** передавался по значению и при вызове **makeChanges** была создана его новая копия. Когда функция **makeChanges** присваивает переменной **number** значение **3**, она меняет только переданную ей копию.

С другой стороны JavaScript не создавал копию списка **globalNumbers**, а передал оригинальный список. И **numbers**, и **globalNumbers** ссылаются на один и тот же список. Значение **21** добавляется в этот самый список. При выводе **numbers** до возврата из функции и при вызове **globalNumbers** после возврата отображается один и тот же список с добавленным значением **21**.

В C++ передача осуществляется в более явном виде. Все переменные и аргументы явно передаются либо по значению, либо по ссылке. Эквивалент приведенного выше кода на JavaScript на C++ выглядит так:

```
void makeChanges(int number, vector<int> & numbers)
{
    number = 3;
    numbers.push_back(21);
    cout << number << " " << numbers << "\n";
}
```

Амперсанд (знак &) перед **numbers** играет важную роль и сообщает компилятору, что этот аргумент передается по ссылке, а не по значению, поэтому при вызове **makeChanges** компилятор не будет создавать его копию. Но перед **number** амперсанд не указан, поэтому компилятор создает копию **number**.

Если амперсанды поменять местами, то компилятор создаст копию **numbers**, но не будет создавать копию **number**:

```
void makeChanges(int & number, vector<int> numbers)
{
    number = 3;
    numbers.push_back(21);
    cout << number << " " << numbers << "\n";
}
```

Эта версия выдает другой результат. Теперь именно простое значение меняется навсегда, а список возвращается к своему исходному значению:

```
0 [3 5 8 13]
3 [3 5 8 13 21]
3 [3 5 8 13]
```

В примерах этой книги часто используются ссылки — так сокращаются затраты памяти и ресурсов. В большинстве случаев то, что не должно меняться, несмотря на передачу по ссылке, помечается ключевым словом **const** («константа»). Такой прием (хотя он до сих пор и не объяснялся) можно заметить уже в первом примере на языке C++ в этом приложении:

```
int calculateSum(const vector<int> & numbers)
```

В этом случае использования копия **numbers** не создается, но и компилятор не позволяет вносить в **numbers** изменения. На практике такое решение похоже на передачу по значению, но с гораздо меньшими затратами.

И еще одно замечание напоследок: в нескольких случаях в примерах используется еще одно (да, никто не говорил, что будет легко) синтаксическое решение C++ для передачи по ссылке, а именно указатели. Указатели и ссылки — это почти одно и то же, и различие между ними не так уж важно, если вы просто стараетесь понять, что делает код. Ключевые различия заключаются в следующем:

- Указатели обозначаются с помощью знака * вместо **&**.
- Для доступа к членам указатели используют **->** вместо **.** (точки).
- Для преобразования указателя в ссылку используют *; для преобразования ссылки в указатель используют **&**.

Вот пример кода с указателями:

```
void example(int number, vector<int> * numbers)
{
    number = 3;
    numbers->push_back(21);
    cout << number << " " << *numbers << "\n";
}

void callExample()
{
    int number = 0;
    vector<int> numbers = { 3, 5, 8, 13 };
    cout << number << " " << numbers << "\n";
```

```
    example(3, &numbers);
    cout << number << " " << numbers << "\n";
}
```

В примерах этой книги константные ссылки используются в тех случаях, когда, по идее, аргумент должен был бы передаваться по значению, но фактическая передача по значению была бы слишком ресурсозатратной. Во всех остальных случаях используются указатели.

Указатель

A

Animal Crossing, 222
Area, структура (struct), 72
auditAll, функция, 41
AvoidLozenges, структура (struct), 262
AwarenessEvents, интерфейс, 159

B

BezierCostFunction, структура (struct), 257
BitVector, класс, 148

C

C++, 70. См. JavaScript и работа с C++; Python и работа с C++
  трудность в написании обобщающего кода, 70
  трюк с цепочкой вызовов, 115
  шаблоны для типобезопасных необязательных аргументов, 116
  языковые соглашения в команде, 181
Character, класс, 41
Character, структура (struct), цепочка отладки, 197
chooseRandomValue, функция, 81, 84
Config, структура (struct), 238
convertCoordinateSystem, функция, 107
cosf, функция, 154
countStepPattern, функция, 19
Customer, конструкция (struct), 155

D

DamageArbiter, функция, 202
Doxygen, 268

E

evaluateComplexPolynomial, функция, 51
evaluateKeyPaht, функция, 234

F

Field , структура (struct), 247
findAllies, функция, 126, 130
findNearbyEnemies, функция, 126
findRecentPurchasers, функция, 155
FixedVector, класс, 53
free, функция, 206

G

getDamageReaction, функция, 202
Ghost of Tsushima, 11, 87, 175, 259
GridPoint, решение для поиска пути в обход препятствий, 261

I

inFamous, серия игр, 11, 126
IntegerType, 246
isServerBlocked, функция, 233
isVisibleAlly, конструкция (struct), 127

J

JavaScript в сравнении с C++, 340
  классы, 344
  массивы, 342
  объявления и определения, 349
  перегрузка функций, 350
  типы, 340
  указатели и ссылки, 353
  шаблоны, 351
JSON, преобразование объектов в структуры C++, 243
JSON, пример кода подхода к проблеме, 243

M

malloc, функция, 133, 206
marckCharacterPosition, функция, 111
Minecraft, 104

O

ObjectID, структура (struct), 43
optimizeViaGoldenSection, функция, 254
Ordinal, конструкция (struct), 22

P

Params, структура (struct), 111
PathManater, класс, 261
Person, структура (struct), 124
printf, функция, 105
processVector, функция в сравнении с функцией sort, 147
Python в сравнении с C++
  указатели и ссылки, 336
Python, в сравнении с C++, 320
  видимость, 331
  и оптимизация производительности, 91
  классы, 327
  объявления и определения, 332
  перегрузка функций, 334
  типы, 320
  форматирование и комментарии, 322
  шаблоны, 335

R

reduceFunction, функция, 34
Result, конструкция (struct), 185
RigidBodySimulator, конструкция (struct), 42

S

SafeType, структура (struct), 243
SetStatus, метод, 161
SetType, структура (struct), 243
ShowAuthorRoyalties, функция, 105
shuffleOnce, функция, 25
SignQuery, конструкция (struct), 63, 66, 73
sinf, функция, 154
StackAlloc_class, 290
StackAlloc, класс, 277
StackContext, класс, 277, 284, 295
StackVector, класс, 279, 288, 292
Standard Template Library (STL), соглашения библиотеки, 55
STATUS, класс в игре со стелсом, 167
strcpy, функция, 133
StructType, шаблон, 243, 248
sumVector, функция, 34

T

TDD (test-driven development), 15. См. разработка через тестирование
try, как префикс функций, 185

V

VectorType, 247
vector, класс, пример непоследовательности, 53

W

Weather, класс, 163

Y

YAGNI («вам это не понадобится»), 65

A

A*, алгоритм, 260
абстракция, лучшие практические советы для создания читаемого кода, 142
автоматизация
  воспроизведения состояния, приведшего к проблеме, 205
  математическое тестирование, 298, 304
  рутинных задач, 313
автоматизированное тестирование, за и против, 32
автоматический запуск теста при «коммитах», 33
Адамс, Дуглас, 122
алгоритм оптимизации методом «золотого сечения», 254
аллокатор кучи памяти, 276, 279
анализ больших фрагментов кода, 136
антисоциальное поведение пингвинов, 294
аргументы
  объединение связанных аргументов в один, 108
  сбой из-за несоответствия аргументов, 106
  типобезопасные необязательные аргументы, 116

аудиокодек со сжатием
и автоматизированное тестирование, 33

Б

баги. См. отладка
безопасность
и решение небольших проблем, 226
Бек, Кент, 276
бинарный поиск как метод оптимизации проблем с выделением памяти, 89
битовые «трюки», 17

В

вариативный шаблон, 248
«Вдвое лучше» как практическое правило, 175
венгерская нотация, 60
видеоигры, 11, 316
видимость, 331
в Python и C++, 331
внутренний аудит внутреннего состояния , 40
время исполнения как фактор задержки при тестировании кода, 204
время на написание кода как мера простоты, 18
выделение памяти, 89, 276, 279
вызовы функций, производительность и алгоритм, 83
вызывающая сторона, важность недоверия, 41

Г

глобальные объекты, 238, 242
границы между файлами, важность соблюдения соглашений, 189

Д

движение от решения задачи. См. результат, движение от задачи к результату
дееспособность или недееспособность, статус объекта, 161
динамическое программирование, 21
долговременная и кратковременная память, 137
долговременная память
и кратковременная, 146, 152

Е

единые правила именования для команды, 55, 189

Ж

жесткие ограничения в процессе разработки, 300

З

зависимости, 31
избавление от зависимостей в процессе отладки, 31
зависимость, 42
задачи
движение от задачи к решению и наоборот, 229
и раздражающие элементы в коде, 222
отладка. См. отладка
решение по принципу простоты, 19
решение с помощью JSON, 230
соглашения о решении задач, 182
типы задач, 206
фокус на общем перед частным, 252
запрещенный тип ревизии, 101
запутанность, 42, 45
запутанный код, 28

И

измерение данных перед оптимизацией, 85
инициализация, функция контроля над вызовами, 42
интерфейс расширения PathExtension, 261
исполняемый журнал событий, 205
история, которую рассказывает код, 266
исходные файлы, 226, 314

К

капсулообразные (lozenge) области, 260
квадратичная сложность, 169
классы
в JavaScript и C++, 344
в Python и C++, 327

классы-адаптеры, в параллельной переработке кода, 290
классы-контейнеры, 55
Кнут, Дональд, 79
когнитивная нагрузка
  в результате вложенных вызовов, 143
  в результате смешения соглашений, 55
  и общепринятые знания, 148
  и чтение кода, 137
код без состояния, тестирование, 34, 133, 200
код прокладывания пути через игровую среду, 259
количество новых идей и простота кода, 19
командное написание кода
  одинаковое мышление в команде, 188
  поддержка словаря, 55, 189
  ревизия как средство укрепления командного духа, 103
  соглашения о решении задач, 182
  соглашения о форматировании кода, 180
  языковые соглашения, 181
комментарии
  бесполезные, 267
  в Python и в C++, 322
  использование assert вместо комментариев, 267
  как средство сворачивания кода, 141
  как средство сообщения неочевидного в коде, 269
  правдивые истории, 266
  решение небольших проблем с комментариями, 223
  сравнение с пробелами и знаками пунктуации, 269
компилятор как средство предотвращения ошибок, 108
константы, представление в коде, 189
конструктор и деструктор, модель push-pop, 118
коплексный тип данных, 52
кратковременная память
  ее роль, и почему не получается читать код, 138
  и долговременная, 137, 146, 152
краткость кода и простота, 19
кривые Безье, 256

Л

Легкая задача, 206, 222
Легкая задача, Простое решение, 207
Легкая задача, Сложное решение, 208
логические операции
  в Python и C++, 326
локализация и проблема printf, 106, 109

М

макрос assert, 47, 183, 267
маскировка и сиротский код, 128
массивы, 342
  class StackVector, 288, 292
  в JavaScript и C++, 342
  в решении Легких задач, 207
  объединение для того, чтобы избежать несоответствия типов, 108
математика, 297
  жесткие ограничения, 300
  за и против автоматизации, 298
  учет изменений, 304
мемоизация для повторного использования промежуточных значений, 20
мертвый код. См. сиротский код
многоэтапные конструкции, соглашения о порядке вызова, 114
модель push-pop в сценарии с неуязвимостью, 118
модель веток для контроля за переработкой кода, 274
модель враждебности персонажа как пример простого кода, 124
модульные тесты, 201
молотки
  и «мешок шурупов», 78
  как инструмент для забивания гвоздей, 315

Н

настройка параметров, пример конфигурационного файла, 230
неопределенный результат и плохой дизайн интерфейса, 45
неправильное использование возможностей программы, 104
непрерывное автоматизированное тестирование, 32

неуязвимость персонажа, согласованное управление состоянием, 117

О

области видимости, определение в контексте объектов, 276
обобщение
  YAGNI, 65
  адаптация после непредвиденных случаев, 71
  и предел архитектуры, 171
  как при обобщении можно потерять контекст, 77
  когда на него не нужно тратить силы, 68
  после трех конкретных примеров, 62, 263
  редкие случаи, когда оно предпочтительнее, 263
  склонность начинающих программистов, 11
  смена перспективы при обобщении, 264
  типичных задач, 127
обработка ошибок, 182
обработка ошибок в старомодном стиле C, 183
обработка ошибок с помощью исключений, 186
объектно ориентированный дизайн и локализация сложности, 158
объекты
  глобальные, 238, 242
  модульное тестирование кода без состояния, 133
  обработка ошибок внутри объектов, 184
  определение областей видимости в объектах, 276
  преобразование JSON в структуры C++, 243
  со встроенной возможностью ошибки, 110
объекты без состояния, сложность с модульным тестированием, 133
объявления и определения
  в JavaScript и C++, 349
  в Python и C++, 332
объяснения, проблема для сложного кода, 19
операторы инкремента, 327
оптимизация, 79
  автоматизация как рабочий процесс, 298
  алгоритм оптимизации методом золотого сечения, 254
  когда лучше не оптимизировать, 83
  когда оптимизация бывает чрезмерной, 91
  процесс из пяти шагов, 84
  процесс из пяти шагов на практике, 86
  фактор производительности, 82, 87, 93
отладка, 31, 191
  автоматизация воспроизведения состояния, приведшего к проблеме, 205
  автоматизация или ручная отладка, 313
  аудит состояния, от которого нельзя избавиться, 40
  выявление проблем на ранних стадиях, 32
  диагностика, 192
  жизненный цикл бага, 191
  заразность багов, 31
  и автоматизированное тестирование, 32
  и ревизия кода, 98, 99
  и языки с ранним или поздним связыванием, 239
  когда неизбежна задержка проявления ошибки, 204
  когда состояние неизбежно, 201
  код, помогающий отладке, 110
  минимизация состояния, 196
  не доверять вызывающей стороне, 41
  отрисовка схематичных границ областей, 110
  пошаговое выполнение кода в отладчике для его изучения, 147
  сиротский код, 123
  сложного кода, 213
  сорняки или баги, 227
  тестирование кода без состояния, 34, 133, 200
  устранение симптомов вместо причин, 194
отладка кода, пример для вычисления последовательности Фибоначчи, 196

отладка параметров отрисовки, 110
отступы в Python, 325
ошибки использования, определение на ранних стадиях, 110
ошибки порядка вызовов, как их сделать невозможными, 114

П

память
  выделение памяти компьютером, 89, 276, 279
  долговременная и кратковременная, 146
  кратковременная и долговременная память, 137, 138, 152
  общие знания и новые концепции, 148
  ограничения памяти и ее цена, 135, 146
параллельная система переработки кода, 276
параллельные массивы, как предотвратить несовпадение длины, 108
параллельные системы для переработки кода, 274
  более умные стековые контексты, 284
  конкретный пример, 276
  конфликт разных версий, 283
  миграция со старых стековых контекстов, 289
  подготовка к миграции класса StackVector, 292
  по сравнению с обычным ветвлением, 275
  стековое выделение памяти, 279
  уместность переработки, 296
Паркер, Дороти, 15
«первые пингвины», переключение флагов при параллельной переработке кода, 294
переменные
  важность осознанных и не слишком коротких имен, 50
  венгерская нотация, 60
  удачный выбор имен, улучшающий читаемость кода, 273
переносы строк
  в Python и в C++, 325
пограничные случаи ошибок, 188
подсчет битов без использования циклов, 17
понятность кода как мера простоты, 18
попытки «подстраховаться на будущее», 172
постепенное развитие или обновление дизайна, 173
предел архитектуры, 171
  игнорирование, мелкие правки или рефакторинг, 172
  когда преимущества переделки трудно измерить, 177
  оценка оправданности переделки, 175
  переделка как способ заодно решить все мелкие проблемы, 177
  постепенная эволюция или постоянное изобретение заново, 173
приближение алгоритма функции к линейной сложности, 219
пример задачи поиска перестановки набора букв, 214
пример простого алгоритма, тасование карт, 25
пример с приземлением персонажа на канат, 252
примеры использования
  и устранение сценариев сбоев, 104
  код, отталкивающийся от реальных сценариев, 67
примеры использования кода
  три реальных примера перед обобщением, 68
принцип наименьшего удивления, 115
программисты, типы, 213
производительность и оптимизация, 81, 82, 87, 93
прополка «сорняков» в коде, 222
простота, 16
  измерение, 18
  и локализация сложности, 170
  и решение об оптимизации, 80, 83, 95
  и сокрытие подробностей кода, 155
  как первый шаг оптимизации, 86, 87, 94
  Легкая задача, Простое решение, 207
  решение проблем по принципу простоты, 19
  сравнение простого и запутанного кода, 28

Трудная задача, Простое решение, 214, 221
прототипирование при оптимизации, 85
профилирование, в повседневном программировании, 94, 95
процессорное время и его измерения, 84

Р

рабочая память. См. кратковременная память
размер файлов, проблема, 314
разработка через тестирование, 15, 32, 48
распределенное состояние и сложность в играх со стелсом, 158
распределенное состояние, пример из игры со стелсом, 158
ревизия кода, 97
  важность совместного процесса в реальном времени, 98, 99
  в команде Sucker Punch, 98
  запрещенный тип ревизии, 101
  истинная ценность, 101
  как социальный процесс, 102
  как способ обмена знаниями, 100
  как средство соблюдения общих соглашений, 189
  слабая польза в нахождении неиспользуемого кода, 130
ревизия кода, просмотр кода старшими или младшими программистами, 100
результат
  движение от задачи к результату, 229
  движение от результата к коду, а не наоборот, 229
  раннее связывание и позднее связывание, 238
рефакторинг системы, 172, 177. См. также параллельная система переработки кода
решето Эратосфена, 148
ручное исправление бага с хешированным значением, 311

С

сворачиваемый код, 135
  локализация сложности, 170
  объединение аргументов, 108
  ограничения памяти и ее цена, 135, 146
  улучшение читаемости, 138
  цена абстракции и наилучшее ее использование, 144
сглаживание анимации, зависимость от состояния, 204
сетевой протокол, пример дизайна кода с жесткими ограничениями, 301
сиротский код, 123
  выявление виновников, 131
  добавление маскировки, 128
  обобщение типичных задач, 127
  последствия, 128
  пределы тестирования, 133
система переключателей (параллельная система) для переработки кода, 276
Слай Купер, игровой персонаж, 11, 252
слишком короткие имена переменных, 50
слова
  обратно от математики к словам, 305
сложность, 153
  и добавление новых эффектов, 163
  изоляция, 153
  и «недееспособный» статус игрока, 161
  и пример сложности поиска пути, 259
  и распределенное состояние, 158
  объединение нескольких условий в одном методе, 166
  объекты со «встроенной» сложностью, 110
  программирование как постоянная борьба со сложностью, 30
  сокрытие подробностей, 155
  цена сложности, 213
сложные решения задач, 207
смешение разных стилей именования, 52
согласованность
  в коде команды, 180, 189
  в наименованиях, 52
соглашения, 181
  о регулярном исправлении мелких проблем, 227
  о решении задач, 182
  о форматировании кода, 180
  реализация многоэтапных конструкторов, 114
  ревизия, как средство соблюдения, 189
  языковые, 181
соглашения об именования
  поддержание согласованности, 59

соглашения об именованиях, 50
  важность согласованности в команде, 189
  глобальное исправление несоответствий имен, 226
  и читаемость кода, 57
  слишком короткие имена переменных, 50
  смена имен при передаче параметров в функцию, 57
  смешение разных стилей, 52
  читаемость кода, 142, 273
  чужие соглашения и соглашения компании, 55
сокрытие подробностей кода ради простоты, 155
сопоставление пар ключ/значение JSON с членами структур или объектов C++, 247
сортировка персонажей, примеры тестирования с состоянием и без состояния, 36
состояние
  аудит состояния, от которого нельзя избавиться, 40
  в сглаживании анимации, 204
  когда состояние неизбежно, 201
  минимизация состояния при отладке, 196
  распределенное состояние и сложность, 158
  согласованное управление состоянием, 117
  сортировка персонажей, 36
социальный аспект ревизии кода, 100
списки
  в Python и C++, 326
ссылки
  JavaScript в сравнении с C++, 353
  Python в сравнении с C++, 336
старшие программисты, просмотр кода младших, 100
стек
  более умный контекст, 284
  выделение памяти, 276, 279
  конфликт версий, пример, 283
  миграция со старых контекстов на новые, 289
  подготовка к миграции класса StackVector, 292
стиль Numerical Recipes, 50
стиль написания кода
  важность последовательности при работе в команде, 180
  важность создания своего стиля, 316
структура
  код на основе структуры результата, 242
структура как отправная точка создания кода, 242
субъективные аспекты тестирования, 32
суффиксы, соглашения для уточнения имен переменных, 61

T

тасование карт, простой алгоритм «риффл», 25
тестирование
  автоматизация и математика, 298, 304
  автоматизированное, за и против, 32
  кода без состояния, 34, 133, 200
  сиротский код и пределы тестирования, 133
  упрощение благодаря простоте кода, 48
  чистых функций, 34, 197, 201
типобезопасность, 243
типы
  в JavaScript и в C++, 340
  в Python и в C++, 320
типы кода, 206
  Легкая задача, Простое решение, 207
  Легкая задача, Сложные решения, 208
  типы программистов, 213
  Трудная задача, относительно Сложное решение, 217
  Трудная задача, Простое решение, 207, 221
  Трудная задача, Сложное решение, 207
  цена сложности, 213
три отдельные функции как вариант решения, 68
Трудная задача, 206, 214
Трудная задача, Простое решение, 207, 221
Трудная задача, Сложное решение, 217

У

удаление из кода раздражающих элементов, 222
указатели C++
  для пишущих код на JavaScript, 353
  для пишущих код на Python, 336
устранение сценариев сбоев, 104
  как сделать ошибки порядка вызовов невозможными, 114
  как сделать проявление ошибки невозможным, 122
  компилятор как средство недопущения ошибок, 108
  обнаружение ошибок на ранних стадиях, 110
  объекты со встроенной сложностью, 110
  сбои из-за рикошета, 106
  согласованное управление состоянием, 117
  функции, подталкивающие к сбоям из-за своего дизайна, 105

Ф

файловая система с журналированием, создание с нуля, 206
философия написания кода Sucker Punch, 11, 14, 30, 98, 189, 316
форматирование кода, 189, 322
функции
  в которых легко ошибиться, 105
  перегрузка, 334, 350
  пример трех функций перед обобщением, 68
  проблемы с переименованием при вызовах, 57
  простота отладки/тестирования чистых функций, 201
  простота тестирования/отладки чистых функций, 34
  разграничение функций и сворачиваемый код, 142
  с префиксом try, 184
функции с префиксом try, стандарт соглашения, 185
функция
  простота отладки/тестирования чистых функций, 197
  функция инициализация для контроля над вызовами, 42
функция аудита, 40, 199
функция менеджера осведомленности, 159

Ц

цветная разметка кода для повышения его читаемости, 273
целочисленные значения произвольной точности, 22
цепочка методов, 116

Ч

чистые функции, простота отладки/тестирования, 34, 197, 201
читаемость
  влияние когнитивной нагрузки, 137
  и сила абстракции, 142
  пробелы и комментарии как пунктуация, 269
  роль кратковременной и долговременной памяти, 138, 151
  сворачиваемый код, 135
  соглашения об именовании, 59, 141, 273
  цветная разметка кода, 273
чужие соглашения, 55

Ш

шаблоны
  JavaScript в сравнении с C++, 351
  Python в сравнении с C++, 335
  вариативный шаблон, 248
  вместо цепочек методов, 116
Шекспир, Уильям, 49

Э

эволюция кода, 123
Эйнштейн, Альберт, 16

Я

языки с ранним и поздним связыванием, 238
языковые конструкции, последовательное использование в коде коллектива, 189

Об авторе

Крис Циммерман в 1997 году стал соучредителем студии видеоигр Sucker Punch Productions и более 20 лет руководил командой разработчиков, выпускавшей успешные игры, включая три игры серии Sly Cooper и пять игр серии inFamous; кульминацией их работы стал кандидат на звание «Игры 2020 года», игра Ghost of Tsushima. Наряду с проектированием и написанием кода (например, для системы ближнего боя в Ghost of Tsushima) он занимался общей организацией труда и управлением в коллективе примерно из 20 разработчиков. До работы в студии Sucker Punch Крис около десяти лет работал в компании Microsoft, но проекты там казались ему гораздо менее интересными. Образование он получал в Принстонском университете, закончив его в 1988 году, и, как следствие, у него в гардеробе гораздо больше одежды оранжевого цвета, чем у вас.

Оформление книги

Иллюстрацию на обложке создала Сьюзан Томпсон. Шрифты обложки — Guardian Sans и Gilroy Semibold. Шрифты текста — Scala Pro, Benton Sans и Minion Pro; заголовки набраны шрифтом Benton Sans; шрифт кода — Ubuntu Mono.

O'REILLY. КНИГИ ПО ПРОГРАММИРОВАНИЮ

Майкл Хаузенблас

КАК ОСВОИТЬ СОВРЕМЕННЫЙ LINUX

Полный справочник: от новичка до профессионала

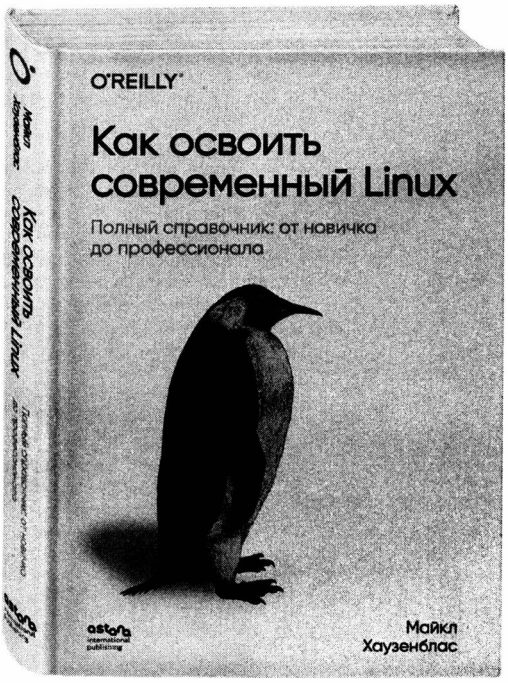
Структурированный подход к использованию Linux в разработке и рабочих процессах.

Книга будет полезна разработчикам, архитекторам программного обеспечения и SRE-инженерам.

Что вы узнаете:

- как использовать Linux не только для администрирования, но и как современную рабочую среду;
- о критически важных компонентах Linux: ядре, терминальном мультиплексоре, оболочках и сценариях командной оболочки;
- о принципах контроля доступа и роли файловых систем;
- об управлении зависимостями приложений и о контейнерах;
- о работе с сетевым стеком и инструментами Linux, включая DNS;
- о современных средствах мониторинга операционных систем;
- о принципах межпроцессного взаимодействия и некоторых аспектах безопасности.

Майкл ХАУЗЕНБЛАС — специалист по наблюдаемости и облачным технологиям с опытом в сфере разработки данных, представитель Cloud Native в CNCF. Работает в AWS, бывшей Red Hat, занимается стартапами и прикладными исследованиями.



O'REILLY®

Барлық құқықтар қорғалған. Бұл кітапты басып шығарушының рұқсатынсыз онлайн немесе кез келген басқа жолмен сканерлеу, жүктеп алу немесе заңсыз тарату заң бойынша жазаланады / Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме с помощью каких-либо электронных или механических средств, включая изготовление фотокопий, аудиозапись, репродукцию или любой иной способ, или систем поиска и хранения информации без письменного разрешения издателя.

12+

Ағылшын тілінен аударма / Перевод с английского
«Астана иностранная пресса»

O'REILLY. КНИГИ ПО ПРОГРАММИРОВАНИЮ

Крис Циммерман

ПРАВИЛА ПРОГРАММИРОВАНИЯ
КАК ПИСАТЬ КАЧЕСТВЕННЫЙ КОД

Бас редактор / Главный редактор
Ысқақбай Рабиға

Басуға 25.06.2026 қол қойылды / Подписано в печать 25.06.2026.
Пішіні 70×100 $^{1}/_{16}$. Қағазы офсеттік. Офсеттік басылыс /
Формат 70×100 $^{1}/_{16}$. Бумага офсетная. Печать офсетная.
Шартты баспа табағы 29,81. Таралымы 1000 дана. Тапсырыс № 4660 /
Усл. печ. л. 29,81. Тираж 1000 экз. Заказ № 4660.

Тауар белгісі / Товарный знак: «Astana International Publishing»
Сапасы женінде мына мекемеге хабарласыңыз /
С претензиями по качеству обращаться: «Астана иностранная пресса» ЖШС
ҚР, Алматы қ., Аль-Фараби даңғ., 19/1, «Нұрлы-Тау» БО, 3Б блогы, 12 офис.
ТОО «Астана иностранная пресса»
РК, г. Алматы, пр. Аль-Фараби, 19/1, БЦ, «Нурлы-Тау», блок 3Б, помещение 12.

Телефон: +7 (771) 276-41-09 E-mail: info@astanapublishing.kz
Кетерме сауда белімі / Отдел оптовых продаж
Алматы қ., Аль-Фараби даңғ., 19/1, «Нұрлы-Тау» БО, 3Б блогы, 12 офис.
г. Алматы, пр. Аль-Фараби, 19/1, БЦ «Нурлы-Тау», блок 3Б, помещение 12.
E-mail: info@astanapublishing.kz

Тауар барлық сапа жене қауіпсіздік стандарттарына сай /
Товар соответствует всем стандартам качества и безопасности
Сертификация қарастырылмаған / Сертификация не предусмотрена
Сақтау мерзімі шектелмеген / Срок годности не ограничен
Өндіруші ел: Қазақстан / Страна-изготовитель: Казахстан
Қазақстан Республикасындағы дистрибьютор: «РДЦ-Алматы»ЖШС /
Дистрибьютор в Республике Казахстан: ТОО «РДЦ-Алматы»
Алматы қ., Домбровский кешесі, 3«а», литер Б, офис 1. Тел.: 8 (727) 251-59-90/91/92
г. Алматы, ул. Домбровского, 3«а», литер Б, офис 1. Тел.: 8 (727) 251-59-90/91/92

Қазастан Республикасындағы Дистрибьютор интернет-дүкені /
Интернет-магазин дистрибьютора в РК: www.book24.kz

«ИПК» ААҚ Ульяновск баспа үйі» баспасында басылды
432980, Ульяновск облысы, Ульяновск, Гончаров кешесі, 14.
Отпечатано в типографии АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ».
432980, Россия, г. Ульяновск, ул. Гончарова, 14.

ISBN 978-601-12-7677-1

9 786011 276771

astana
international
publishing

O'REILLY®

«**"Правила программирования"** сочетают в себе отличное руководство для новичков и уроки, которые могут пригодиться даже экспертам. Циммерман делает процесс увлекательным, доказывая, что он может быть одновременно интересным и поучительным».

Марк Черни, ведущий системный архитектор PlayStation 4 и 5

21 ПРОСТОЕ ПРАВИЛО, чтобы писать качественный код

«Правила программирования» — это сборник простых и полезных принципов, которые помогут начинающим программистам писать оптимальный код, а опытным — взглянуть на привычное под новым углом.

Что внутри:

- **простота и ясность кода;**
- **структурированность, понятность и оптимизация;**
- **борьба с ошибками.**

И продуктивно, и увлекательно:

- **делай максимально просто, но не проще некуда;**
- **дай коду рассказать свою историю;**
- **усвой первый урок оптимизации: не оптимизируй;**
- **локализируй сложности;**
- **иди от результата к коду, а не наоборот;**
- **устраняй сбои;**
- **помни: код, который не выполняется, не работает;**
- **придумай хорошее название — это лучшая документация.**

Крис Циммерман делится основными принципами программирования в виде запоминающихся афоризмов и подкрепляет их примерами реального кода.

Крис ЦИММЕРМАН — соучредитель и почетный глава студии разработки видеоигр Sucker Punch, создателей всеми любимых игр Sly Cooper, inFamous и Ghost of Tsushima. Изучал компьютерные науки в Принстонском университете, имеет научную степень. 9 лет работал в Microsoft.



ISBN 978-601-12-7677-1



astana
international
publishing