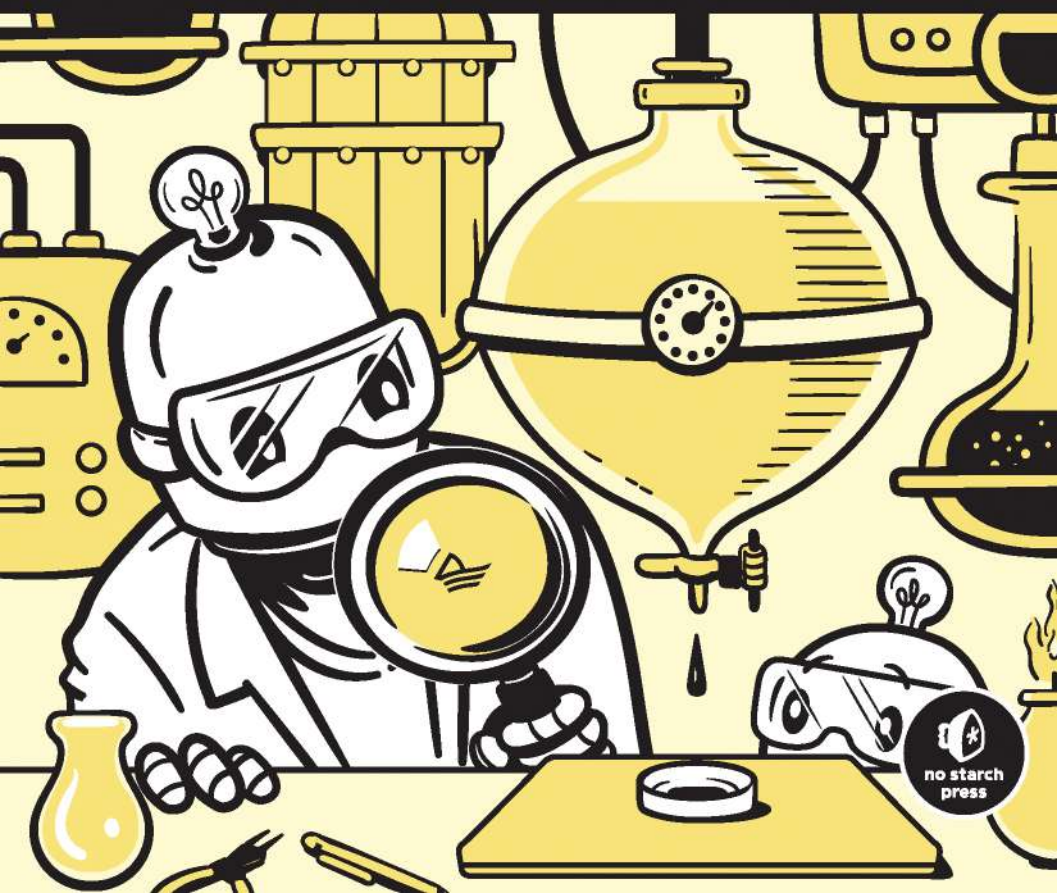
ИСКУССТВО ЧИСТОГО КОДА

КАК ИЗБАВИТЬСЯ ОТ СЛОЖНОСТИ И УПРОСТИТЬ ЖИЗНЬ

КРИСТИАН МАЙЕР



THE ART OF CLEAN CODE

Best Practices to Eliminate Complexity and Simplify Your Life

by Christian Mayer

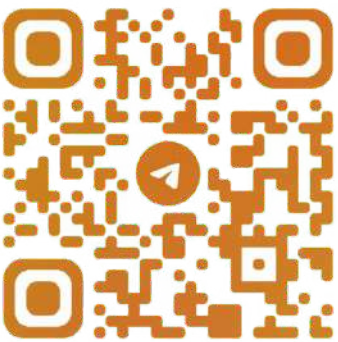
no starch press

San Francisco

ИСКУССТВО ЧИСТОГО КОДА

КАК ИЗБАВИТЬСЯ ОТ СЛОЖНОСТИ И УПРОСТИТЬ ЖИЗНЬ

КРИСТИАН МАЙЕР



Краткое содержание

Оглавление

Об авторе

Кристиан Майер (Christian Mayer) — основатель популярного сайта Finxter.com, посвященного языку Python. Благодаря этой образовательной платформе более пяти миллионов человек в год обучаются программированию. Кристиан имеет степень PhD в области computer science, он опубликовал ряд книг, включая «Python One-Liners» (No Starch Press, 2020)[1], «Leaving the Rat Race with Python» («Выйти из бешеной гонки с помощью Python») (2021) и серию «Coffee Break Python» («Перерыв на кофе с Python»).

О научном редакторе

Ноа Спан (Noah Spahn) имеет богатый опыт в области разработки ПО. Он получил степень магистра в области программной инженерии в Университете штата Калифорния в Фуллертоне. Сегодня Ноа работает в группе компьютерной безопасности Калифорнийского университета в Санта-Барбаре (University of California, Santa Barbara; UCSB), где ранее преподавал Python в группе междисциплинарного сотрудничества. Также он читал лекции по концепциям языков программирования для студентов старших курсов Вестмонтского колледжа. Ноа всегда рад помочь тем, кто заинтересован в обучении.

Предисловие

Я помню, с каким волнением изучал первые строчки кода на Python; мне казалось, что я попал в совершенно новую сказочную вселенную. Со временем я научился работать с переменными, списками и словарями Python. Затем разобрался в том, как писать функции, и с энтузиазмом взялся за разработку более сложного кода. Однако вскоре я понял, что умение писать код не делает меня искусным программистом, — будто я просто выучил несколько фокусов, но еще был очень далек от того, чтобы стать настоящим волшебником в программировании.

Хотя я и выполнял порученную работу, мой код был ужасным: повторяющимся и трудночитаемым. Когда Крис рассказал мне об этой книге, я подумал: *«Жаль, что у меня ее не было, когда я только начинал писать код»*. Существует много литературы, обучающей техническим аспектам программирования, но такие книги, как «Искусство чистого кода», встречаются редко. В ней рассказывается, как усовершенствовать свои навыки с помощью девяти принципов написания кода. А высокое мастерство повышает чистоту кода, сосредоточенность, эффективность использования времени и качество результатов.

Глава 1 «Сложность — враг продуктивности» очень пригодилась бы мне при изучении Python и визуализации данных: я сразу бы понял, что могу создавать мощные дашборды с помощью меньшего количества кода, который к тому же легко читать. В самом начале, по мере освоения новых функций и операций Python, мне просто хотелось использовать все эти чудесные приемы для создания потрясающих визуализаций данных. Но затем я научился реализовывать

их с помощью более чистого кода, вместо того чтобы тупо применять новые фишки, и отладка моего кода стала гораздо проще и быстрее.

Состояние потока и философия Unix, о которых говорится в главах 6 и 7, — это еще два принципа, о которых я хотел бы знать намного раньше. Как правило, многозадачность считается полезным навыком в нашей культуре. Я часто гордился своей способностью писать код, одновременно отвечая на звонки и письма. Мне потребовалось время, чтобы понять, насколько полезно умение полностью исключить отвлекающие факторы и сфокусироваться только на коде. Несколько месяцев спустя я начал выделять время в своем расписании исключительно на разработку. В итоге я не только написал более качественный код с меньшим количеством ошибок, но и получил гораздо больше удовольствия от процесса.

Следуя принципам, описанным в этой книге, вы сократите путь к тому, чтобы стать искусным программистом. На самом деле у меня была возможность воочию убедиться в преимуществах их применения: код, который пишет Крис, — чистый, его работа востребована, и он преуспевает. Мне повезло поработать с Крисом и увидеть, как он реализует подходы, представленные в этой книге.

Умение хорошо писать код требует любознательности и нуждается в практике. Однако есть разница между хорошим кодером и хорошим программистом — эта книга поможет вам стать хорошим программистом: более сфокусированным, продуктивным и эффективным.

Адам Шредер (Adam Schroeder),
комьюнити-менеджер в Plotly,
соавтор книги «Python Dash»
(No Starch Press, 2022)

Благодарности

Создание книги по программированию основывается на идеях и вкладе множества людей. Вместо того чтобы пытаться перечислить их всех, я последую собственному совету: *лучше меньше, да лучше*.

Прежде всего, я хочу поблагодарить вас. Я написал эту книгу, чтобы помочь вам повысить мастерство написания кода и научиться решать практические задачи в реальном мире. Я благодарен вам за то, что вы доверили мне свое драгоценное время. Моя главная цель — сделать эту книгу полезной, поделившись советами и стратегиями, которые помогут вам сэкономить время и снизить эмоциональную нагрузку на протяжении всей вашей карьеры программиста.

Самым большим источником мотивации для меня стали активные члены сообщества Finxter. Каждый день я получаю ободряющие сообщения от студентов Finxter, и это вдохновляет меня на создание контента. Я надеюсь, что по мере чтения этой книги вы присоединитесь к нашему сообществу Finxter[1] и я смогу от всей души поприветствовать вас. Мне очень приятно будет видеть вас здесь!

Я глубоко признателен команде издательства No Starch Press за то, что процесс написания книги стал для меня таким интересным опытом. Хочу поблагодарить моего редактора Лиз Чедвик (Liz Chadwick); именно благодаря ее выдающемуся руководству книга достигла того уровня ясности, которого я не добился бы самостоятельно. Катрина

[1] Вы можете присоединиться к нашей бесплатной Email-академии Python (Python email academy) по ссылке https://blog.finxter.com/subscribe/. У нас есть шпаргалки!

Тейлор (Katrina Taylor) вела книгу от черновика до публикации с редким талантом менеджера и глубоким пониманием текста. Спасибо, Катрина, что сделала книгу реальностью! Мой научный редактор, Ноа Спан, применил свой недюжинный профессионализм, чтобы «отладить» мою писанину. Отдельное спасибо основателю No Starch Press Биллу Поллоку (Bill Pollock) за то, что он позволил мне внести свой небольшой вклад в его миссию по обучению и развлечению разработчиков кода, выпустив еще одну книгу наряду с «Python One-Liners» и «Python Dash». Билл — вдохновляющий и востребованный лидер в области программирования, но он все равно находит время на такие мелочи, как ответы на мои сообщения и вопросы во время праздников, на выходных и даже ночью!

Я бесконечно благодарен моей прекрасной жене Анне, поддерживающей меня в моих начинаниях, моей очаровательной дочери Амалии, полной фантастических историй и идей, и моему любознательному сыну Габриэлю, который делает всех вокруг счастливее.

А теперь не приступить ли нам к делу?

Введение

Когда-то давным-давно родители Билла Гейтса пригласили легендарного инвестора Уоррена Баффета в свой дом погостить. В интервью CNBC Уоррен Баффет вспоминает, как, пользуясь случаем, отец Билла попросил сына и самого Уоррена изложить на бумаге секреты своего успеха. Сейчас я расскажу вам, что же они написали.

К тому времени гениальный программист Билл Гейтс всего пару раз встречался с известным инвестором Баффетом, но они быстро стали друзьями, поскольку оба возглавляли успешные корпорации с миллиардными оборотами. Молодой Билл с его быстрорастущей Microsoft, крупнейшей компанией-разработчиком программного обеспечения, в те годы был на пороге завершения своей главной миссии: *компьютер на каждый стол*. А Уоррен Баффет уже прославился как один из самых успешных в мире гениев бизнеса. Известно, что Уоррен превратил свою мажоритарную компанию Berkshire

Hathaway из разорившегося производителя текстиля в международного тяжеловеса в таких разноплановых сферах, как страхование, транспорт и энергетика.

Итак, что же эти две легенды бизнеса считали секретом своего успеха? Как гласит история, независимо друг от друга Билл и Уоррен написали одно и то же слово: *«фокус»*.

ПРИМЕЧАНИЕ

Вы можете посмотреть, как Уоррен Баффет рассказывает об этом в интервью телеканалу CNBC, в YouTube-видео под названием «One word that accounted for Bill Gates' and my success: Focus» — Warren Buffett»[1].

Хотя этот секрет успеха звучит достаточно просто, вы можете задаться следующими вопросами. Применим ли он и к моей карьере разработчика кода? Что означает «фокус» на практике — писать код по ночам с энергетиками и пиццей или, возможно, придерживаться белковой диеты и вставать на рассвете? Каковы незаметные на первый взгляд последствия ведения такой сфокусированной жизни? И, что самое важное, есть ли действенные советы, как воспользоваться этим абстрактным принципом для повышения продуктивности?

Задача книги — ответить на эти вопросы, чтобы помочь вам вести более сфокусированную жизнь программиста и стать эффективнее в повседневной работе. Я покажу вам, как повысить свою продуктивность за счет написания чистого, лаконичного и сфокусированного кода, более легкого для чтения, создания и совместной работы с другими программистами. Как я продемонстрирую в последующих главах, принцип фокусировки действует на всех этапах разработки ПО. Вы узнаете, как писать чистый код, создавать функции, направленные на качественное выполнение только одной задачи, разрабатывать быстрые и адаптивные приложения,

[1] «Одно слово, объясняющее успех Билла Гейтса и мой успех: “фокус”. — Уоррен Баффет». — *Примеч. пер.*

проектировать пользовательские интерфейсы, ориентированные на эстетику и юзабилити (то есть удобство использования), а также как планировать развитие приложения, используя минимально жизнеспособный продукт. Я даже покажу вам, что достижение состояния чистой сосредоточенности значительно повысит вашу концентрацию и принесет вам больше воодушевления и радости от выполнения поставленных задач. Как вы поймете далее, идея книги в том, чтобы сфокусироваться на цели всеми возможными способами. В следующих главах я покажу вам, как именно это сделать.

Любому серьезному программисту необходимо постоянно совершенствоваться в концентрации внимания (фокусе) и повышении продуктивности. Как правило, чем важнее работа, тем выше вознаграждение. Однако простым увеличением количества этот вопрос не решается. Парадокс в следующем: *вы думаете, что, если писать больше кода, создавать больше тестов, читать больше книг, больше учиться, больше думать, больше общаться и встречаться с бóльшим числом людей, вы добьетесь большего*. Но вы не сможете достичь *большего*, не делая при этом чего-либо в *меньшем объеме*. Время ограничено — у вас всего 24 часа в сутках и 7 дней в неделе, как и у меня, и у всех остальных. Существует неизбежное математическое условие: если в ограниченном пространстве что-то одно увеличивается, то что-то другое должно уменьшаться, освобождая место. Читая больше книг, вы встречаете меньше людей. Общаясь с бóльшим количеством людей, вы пишете меньше кода. Если вы разрабатываете больше кода, то у вас меньше времени остается на тех, кого вы любите. Невозможно избежать фундаментального компромисса: при ограниченных ресурсах нельзя добиться большего, ничем не поступившись.

Вместо того чтобы просто делать больше, я предлагаю другое решение: уменьшить сложность. Это позволит вам работать меньше, получая при этом более весомые результаты. Продуманный минимализм — «Святой Грааль» личной продуктивности, и, как вы увидите далее, он работает. Вы можете создавать бóльшую ценность,

используя меньше ресурсов, правильно программируя и используя всегда актуальные принципы, описанные в данной книге.

Создавая больше ценности, вы можете рассчитывать на рост вознаграждения. Билл Гейтс однажды сказал, что «отличный токарь получает в несколько раз больше, чем средний, но великий программист сто́ит в 10 000 раз больше, чем обычный».

Одна из причин этого заключается в том, что крутой специалист обеспечивает высокую доходность компании, поскольку оптимальный подход к программированию может заменить тысячи рабочих мест и миллионы часов высокооплачиваемой работы. Например, код для управления беспилотными автомобилями эквивалентен труду миллионов водителей, будучи при этом дешевле, надежнее и (по мнению некоторых) гораздо безопаснее.

Для кого эта книга?

Вы — действующий программист, мечтающий создать более значимый продукт с быстрым кодом и меньшей головной болью?

Вы когда-нибудь зацикливались на поиске багов?

Сложность кода частенько приводит вас в замешательство?

Трудно ли вам решить, что изучать дальше, выбирая из сотен языков программирования: Python, Java, C++, HTML, CSS, JavaScript — и тысяч фреймворков и технологий: приложения Android, фреймворк Bootstrap, библиотеки TensorFlow, NumPy?

Если ваш ответ на любой из этих вопросов «ДА!» (или просто «да»), то вы держите в руках нужную книгу!

Она предназначена для всех программистов, заинтересованных в повышении своей продуктивности — делать больше с меньшими затратами. Она для вас, если вы стремитесь к простоте и свято верите в принцип бритвы Оккама: «Не следует множить сущности без крайней необходимости».

Чему вы научитесь?

Эта книга научит вас применять на практике девять несложных принципов, что на порядок повысит ваш потенциал как программиста. Они упростят вашу жизнь, снизят сложность и сократят усилия и время, затрачиваемые на работу. Я не претендую на новизну этих принципов. Так или иначе они приобрели известность и устоялись, а их эффективность доказана самыми успешными разработчиками кода, инженерами, философами и изобретателями. Именно это и делает их принципами! Однако здесь я буду рассматривать их применительно к программистам, приводя практические примеры и, по возможности, фрагменты кода.

В **главе 1** речь пойдет об основном препятствии на пути повышения продуктивности — сложности. Вы научитесь находить источники сложности как в коде, так и в повседневной жизни и поймете, что сложность всегда вредит вашей продуктивности и конечному результату. Сложность повсюду, и вы должны постоянно быть начеку в борьбе с ней. *Будьте проще!*

В **главе 2** раскрывается глубокое влияние *принципа 80/20* на жизнь программиста. Большинство следствий (80 %) вытекает из малой доли причин (20 %) — в сфере программирования это утверждение проявляется во всем. Вы узнаете, что принцип 80/20 является фрактальным: 20 % от 20 % разработчиков кода получают 80 % от 80 % оплаты. Другими словами, 4 % программистов в мире зарабатывают 64 % всех денег. Поиск путей повышения эффективности и оптимизации работы идет непрерывно!

В **главе 3** вы узнаете о *построении минимально жизнеспособных продуктов*. Цель — ранняя проверка предположений, минимизация расходов и ускорение прохождения цикла создания, разработки и изучения. Идея в том, чтобы узнать, на чем лучше сосредоточить свое внимание и усилия, получив обратную связь на начальных этапах.

В **главе 4** обсуждаются преимущества *написания чистого и простого кода*. Обычно люди интуитивно предполагают, что код должен обеспечить минимальную загрузку центрального процессора (ЦП). Однако прежде всего необходимо, чтобы код было удобно читать человеку. Общее потраченное время и усилия программистов гораздо более ценны, чем ресурсы процессора, а код, который трудно понять, снижает эффективность вашей организации (а также нашего коллективного человеческого разума).

В **главе 5** вы познакомитесь с концептуальной основой оптимизации производительности и подводными камнями слишком раннего ее проведения. Дональд Кнут (Donald Knuth), один из отцов computer science, говорил: *«Преждевременная оптимизация — корень всех зол!»* Если вам необходимо оптимизировать код, используйте принцип 80/20: попробуйте улучшить 20 % функций, выполнение которых занимает 80 % времени. Избавьтесь от узких мест. Игнорируйте остальное. Повторите.

В **главе 6** мы вместе совершим экскурс в (буквально) захватывающий мир *потока* Михая Чиксентмихайи (Mihaly Csikszentmihalyi). Состояние потока — это истинная концентрация, которая многократно увеличивает продуктивность и помогает создать атмосферу глубокого погружения в работу, как говорит профессор информатики Кэл Ньюпорт (Cal Newport), который также привнес некоторые идеи в эту главу.

В **главе 7** изложена философия Unix, заключающаяся в том, чтобы *делать что-то одно* и делать это хорошо. Вместо монолитного (и потенциально более эффективного) ядра с огромным набором функций разработчики Unix предпочли реализовать небольшое ядро со множеством дополнительных вспомогательных функций. Это помогает экосистеме Unix расширяться, оставаясь при этом чистой и (относительно) простой. Мы выясним, как применить эти принципы в работе.

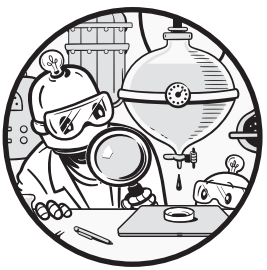
В **главе 8** вы познакомитесь еще с одной очень важной областью компьютерных наук, которая выигрывает от минималистичности мышления: дизайн интерфейса и взаимодействия с пользователем (user experience, UX). Подумайте о различиях между Yahoo Search и Google Search, Blackberry и iPhone, OkCupid и Tinder. Самые успешные технологии часто имеют максимально простой пользовательский интерфейс по той причине, что в дизайне работает принцип *лучше меньше, да лучше*.

В **главе 9** мы вновь обратимся к силе фокуса и рассмотрим, как применять этот принцип в различных областях, чтобы значительно повысить вашу продуктивность и оптимизировать работу ваших программ!

Наконец, мы подведем итоги, дадим практические рекомендации и отпустим вас в сложный мир, снабдив набором надежных инструментов для его упрощения.

1

Сложность — враг продуктивности



В этой главе мы всесторонне рассмотрим важную и крайне мало изученную проблему *сложности*. Что она собой представляет? Где возникает? Как влияет на вашу продуктивность? Сложность — враг рациональной и эффективной компании или личности, поэтому стоит внимательно изучить все причины ее возникновения и формы, которые она может принимать. В этой главе основное внимание уделяется непосредственно проблеме сложности, а в остальных рассмотрены эффективные методы решения проблемы путем перенаправления высвободившихся ресурсов, ранее занятых из-за сложности.

Начнем с краткого обзора этапов, на которых сложность может отпугнуть начинающего программиста:

- Выбор языка программирования.
- Выбор проекта для реализации из тысяч проектов с открытым исходным кодом и миллионов различных задач.

- Выбор библиотек для работы (например, scikit-learn, NumPy или TensorFlow).
- Выбор одной из развивающихся технологий, на освоение которой стоит потратить время: приложения для Alexa, приложения для смартфонов, браузеров, виртуальной реальности, интегрированные приложения Facebook или WeChat.
- Выбор редактора кода: PyCharm, IDLE (Integrated Development and Learning Environment)[1] или Atom.

Неудивительно, что вопрос *«С чего же мне начать?»* — один из самых часто задаваемых новичками, находящимися в замешательстве из-за сложности выбора.

Прямо скажем: выбрать конкретную книгу и прочитать обо всех синтаксических особенностях языка программирования — это не лучший способ начать работу. Многие амбициозные студенты покупают книги по программированию в качестве стимула, а затем добавляют задачу «изучить» в свой список дел — раз уж потратились на книгу, надо ее прочитать, иначе инвестиции будут потеряны. Но, как и многие другие задачи в списке, чтение книг по программированию редко заканчивается успехом.

Лучший способ начать — это выбрать реальный проект (простой, если вы новичок) и довести его до конца. Не читайте книги по программированию или случайные учебные пособия в интернете, пока полностью не выполните задачу. Не прокручивайте бесконечные ленты на StackOverflow. Просто определитесь с проектом и начните писать код, используя свои небольшие навыки и здравый смысл. Одна из моих студенток захотела создать приложение финансового мониторинга, с помощью которого можно отследить данные за прошлые периоды по различным видам распределения активов, то есть получить ответы на вопросы типа «Какой год принес максимальное падение портфеля, состоящего из 50 % акций и 50 % гособлигаций?».

[1] Интегрированная среда разработки и обучения на Python. — *Примеч. пер.*

Вначале она не знала, как подойти к этому проекту, но вскоре узнала о фреймворке Python Dash, где можно создавать веб-приложения на основе данных. Она научилась настраивать сервер и изучила только HTML (HyperText Markup Language — язык разметки гипертекста) и CSS (Cascading Style Sheets — каскадные таблицы стилей), необходимые ей для проекта. Теперь ее приложение живет и помогает тысячам людей найти правильный баланс в распределении активов. Но, что более важно, после этого она присоединилась к команде разработчиков, создавших Python Dash, и даже пишет об этом книгу для No Starch Press. Она сделала все это за один год, и вы тоже сможете. Ничего страшного, если вы пока не знаете, что делаете, — со временем придет понимание. Читайте статьи лишь для того, чтобы добиться прогресса в текущей задаче. В процессе разработки первого проекта возникнет ряд очень важных вопросов, например:

Какой редактор кода следует использовать?

Как установить выбранный для проекта язык программирования?

Как считать ввод из файла?

Как хранить в вашей программе входные данные для последующего использования?

Как происходит обработка входных данных для получения желаемого результата?

Отвечая на эти вопросы, вы постепенно получите необходимый набор навыков и со временем сможете лучше и легче находить ответы на эти вопросы. Вы научитесь решать гораздо более серьезные проблемы и накопите собственную базу паттернов программирования и концептуальных идей. Даже продвинутые разработчики приложений учатся и совершенствуются таким же образом, только проекты у них масштабнее и сложнее.

При обучении на основе проектов вы, скорее всего, обнаружите, что сталкиваетесь со сложностью в таких вопросах, как исправление

багов в постоянно растущих кодовых базах, понимание компонентов кода и их взаимодействия, выбор подходящей функции для следующей реализации и осмысление математических и концептуальных основ кода.

Сложность присутствует везде, на каждом этапе проекта. Невидимая цена, которую мы платим за нее, часто состоит в том, что свежеиспеченные программисты бросают работу, — их проекты уже никогда не увидят свет. Возникает вопрос: как решить проблему сложности?

Нехитрый ответ: *простота*. Стремитесь к ней и фокусируйтесь на каждом этапе программирования. Если вы вынесете из этой книги только одно, пусть это будет следующее: занимайте радикально минималистскую позицию во всех областях, с которыми вы сталкиваетесь при создании кода. Ниже перечислены подходы, которые мы обсудим в этой книге:

- Упорядочьте свой день, делайте меньше дел и сфокусируйтесь на тех задачах, которые действительно имеют значение. Например, вместо того чтобы параллельно начинать 10 новых интересных проектов, тщательно выберите один и сосредоточьте все свои усилия на его завершении. В главе 2 вы более подробно узнаете о принципе 80/20 в программировании.
- В рамках одного программного проекта отбросьте весь ненужный функционал и сосредоточьтесь на минимально жизнеспособном продукте (см. главу 3), выложите его и проверьте свои гипотезы быстро и эффективно.
- По возможности пишите простой и лаконичный код. В главе 4 вы найдете много практических советов, как этого добиться.
- Сократите время и усилия, затрачиваемые на преждевременную оптимизацию, — оптимизация кода без необходимости является одной из основных причин излишней сложности (см. главу 5).

- Уменьшайте количество переключений между задачами, выделяя длительные отрезки времени на программирование, чтобы достичь состояния *потока* — это термин из психологических исследований, описывающий состояние сосредоточенности, которое повышает внимание, концентрацию и продуктивность. Глава 6 посвящена методам достижения такого состояния.
- Следуйте философии Unix, которая гласит, что каждая функция кода должна быть нацелена на выполнение лишь одной задачи («Do One Thing Well»). Подробное руководство по философии Unix с примерами кода на Python см. в главе 7.
- Стремитесь к простоте в дизайне для создания красивых, ясных и сфокусированных пользовательских интерфейсов, которые просты в использовании и интуитивно понятны (см. главу 8).
- Применяйте методы фокусировки при планировании своей карьеры, следующего проекта, своего дня или своей области компетенций (см. главу 9).

Давайте подробнее рассмотрим понятие сложности, чтобы лучше разобраться в одном из главных врагов вашей продуктивности как разработчика кода.

Что такое сложность?

В различных областях термин *«сложность»* имеет разные значения. Иногда ему дается строгое определение, например *вычислительная сложность* компьютерной программы, которая предоставляет средства для анализа кода в зависимости от различных исходных данных[1]. В других случаях он достаточно вольно рассматривается

[1] Если точнее, вычислительная сложность – это функция зависимости объема работы, которая выполняется некоторым алгоритмом, от размера входных данных. — *Примеч. ред.*

как количество или структура взаимодействий между компонентами системы. В данной книге мы будем использовать это понятие в более общем смысле.

Мы определим *сложность* следующим образом:

> **Сложность** — это состоящее из частей целое, которое трудно проанализировать, понять или объяснить.

Сложность характеризует целую систему или объект. Поскольку сложность делает систему труднообъяснимой, она приводит к проблемам и путанице. Реальные системы по сути своей беспорядочны, поэтому сложность можно встретить повсюду: на фондовом рынке, в трендах общественного развития, в формирующихся политических взглядах, в больших компьютерных программах с сотнями тысяч строк кода — например, в операционной системе Windows.

Если вы разработчик, вы особенно предрасположены к чрезмерной сложности. В данной главе мы рассмотрим несколько различных ее форм:

Сложность жизненного цикла проекта.

Сложность ПО и теории алгоритмов.

Сложность в обучении.

Сложность в процессах.

Сложность в социальных сетях.

Сложность в повседневной жизни.

Сложность жизненного цикла проекта

Рассмотрим различные этапы реализации проекта: планирование, определение требований, проектирование, разработка, тестирование и развертывание (рис. 1.1).

Рис. 1.1. Шесть основных этапов разработки ПО согласно официальному стандарту по программной инженерии IEEE[1]

Даже если вы создаете очень небольшой программный продукт, вы, скорее всего, пройдете полный жизненный цикл ПО, и даже неоднократно — в современной разработке предпочтительным считается итеративный подход, при котором каждый этап повторяется несколько раз. Далее мы рассмотрим, как сложность оказывает существенное влияние на каждом этапе цикла.

Планирование

Первая фаза жизненного цикла разработки ПО — это этап планирования, в технической литературе иногда называемый *анализом требований*. Его цель — определить, как будет выглядеть продукт. Результатом успешного планирования является строго определенный набор необходимых функций, которые должны быть поставлены конечному пользователю.

Независимо от того, работаете ли вы над хобби-проектом или отвечаете за управление и организацию сотрудничества между несколькими командами разработчиков, вы должны определить оптимальный

[1] IEEE (Institute of Electrical and Electronics Engineers) — Институт инженеров электротехники и электроники. — *Примеч. ред.*

набор функций вашего ПО. При этом необходимо принять во внимание множество факторов: затраты на создание конкретной функции, риск того, что она не будет успешно реализована, ожидаемая полезность ее для конечного пользователя, результаты исследований службы маркетинга и продаж, удобство сопровождения, масштабируемость, юридические ограничения и многое другое.

Эта фаза имеет решающее значение, ведь она может уберечь вас от огромных затрат усилий в дальнейшем. Порой ошибки в планировании приводят к потере ресурсов на миллионы долларов. С другой стороны, тщательное планирование способствует крупному успеху в бизнесе на годы вперед. Этап планирования — это время, когда вы можете применить новоприобретенные навыки мышления 80/20 (см. главу 2).

Планирование трудно осуществить должным образом именно из-за его сложности. Ее увеличивают несколько факторов, которые нужно учитывать: правильная предварительная оценка рисков, определение стратегического направления развития компании или организации, угадывание реакции клиентов, оценка положительного вклада различных функций-кандидатов и определение юридических последствий той или иной программной функции. В результате вся сложность решения этой многоплановой проблемы подрывает наши силы.

Определение требований

Данный этап состоит в преобразовании результатов планирования в правильно сформулированные требования к ПО. Другими словами, он формализует то, что получилось на предыдущем этапе, чтобы получить подтверждение или отклик от клиентов и конечных пользователей, которые впоследствии будут использовать продукт.

Если вы потратили много времени на планирование и уточнение требований к проекту, но не смогли правильно их донести до исполнителей, это приведет к значительным проблемам и трудностям в дальнейшем. Неверно сформулированное требование, помогающее

проекту, может быть столь же плохим, как и правильно обозначенное, но бесполезное требование. Эффективная коммуникация и точная спецификация имеют решающее значение для того, чтобы избежать двусмысленности и недопонимания. В любой коммуникации между людьми донести свою мысль до собеседника очень сложно из-за «проклятия знания»[1] и других психологических искажений, которые перевешивают значимость личного опыта. Если вы пытаетесь объяснить свои идеи (или требования в нашем случае) другому человеку, будьте осторожны: на этом пути вас всегда поджидает сложность!

Проектирование

Цель этой фазы — сделать черновик архитектуры системы, выбрать модули и компоненты, обеспечивающие заданную функциональность, и разработать пользовательский интерфейс с учетом требований, определенных на предыдущих двух этапах. Золотым стандартом этапа проектирования является создание абсолютно ясной картины того, как будет выглядеть конечный программный продукт и как его построить. Это справедливо для всех методов разработки ПО. Просто при применении Agile-подхода к управлению проектами эти этапы вы пройдете быстрее.

Но дьявол кроется в деталях! Хороший системный разработчик обязан знать о плюсах и минусах огромного количества программных инструментов, применяемых для построения системы. Например, некоторые библиотеки могут быть удобными в использовании, но медленными по скорости выполнения. Создание пользовательских библиотек связано с некоторыми сложностями для программистов, однако в результате может привести к гораздо более высокой скорости выполнения, что повышает юзабилити конечного продукта. На этапе проектирования необходимо зафиксировать эти переменные таким образом, чтобы соотношение выгод и затрат было максимальным.

---

[1] Когнитивное искажение, при котором более информированному человеку очень сложно увидеть проблему с точки зрения людей, знающих меньше. — *Примеч. ред.*

Разработка

Это фаза, на которой многие разработчики кода хотят проводить все свое время. Именно здесь происходит преобразование архитектурного проекта в программный продукт. Ваши идеи превращаются в реальные значимые результаты.

Благодаря правильно проведенной работе на предыдущих этапах многие сложности уже устранены. В идеале разработчики уже знают, какие функции из всех возможных нужно реализовать, как они выглядят и какие инструменты следует использовать для их внедрения. Тем не менее на этапе разработки всегда возникает множество новых трудностей. Неожиданности типа багов во внешних библиотеках, проблемы с производительностью, повреждение данных, а также человеческий фактор замедляют процесс. Создание программного продукта — задача сложная. Небольшая орфографическая ошибка может негативно сказаться на его жизнеспособности.

Тестирование

Поздравляем! Вы реализовали весь необходимый функционал — и программа, похоже, работает. Но это еще не все, ведь вы должны протестировать поведение продукта для различных типов пользовательского ввода и моделей использования. Часто это самый важный этап — настолько, что сегодня многие специалисты выступают за *разработку через тестирование (test-driven development, TDD)*, когда вы даже не приступаете к созданию продукта (на этапе разработки), не написав все тесты. Конечно, с этой точкой зрения можно поспорить, но в целом идея хорошая: потратить время на тестирование продукта, разработав тестовые сценарии, и проверить, обеспечивает ли ПО правильный результат для этих случаев.

Предположим, вы создаете беспилотный автомобиль. Нужно написать *юнит-тесты*, или модульные тесты, и проверить, что каждая небольшая функция (*юнит*) в вашем коде генерирует желаемый результат для заданного ввода. Юнит-тесты обычно выявляют

некоторые некорректные функции, которые ведут себя странно при определенных (экстремальных) входных значениях. Для примера рассмотрим следующую функцию-заглушку (стаб) Python, которая вычисляет среднее значение красного (R), зеленого (G) и синего (B) цветов на изображении; допустим, она используется, чтобы определить, где вы едете — по городу или по лесу:

```
def average_rgb(pixels):
    r = [x[0] for x in pixels]
    g = [x[1] for x in pixels]
    b = [x[2] for x in pixels]
    n = len(r)
    return (sum(r)/n, sum(g)/n, sum(b)/n)
```

Например, следующий список пикселей даст средние значения красного, зеленого и синего цветов 96.0, 64.0 и 11.0 соответственно:

```
print(average_rgb([(0, 0, 0),
                   (256, 128, 0),
                   (32, 64, 33)]))
```

В результате получим:

```
(96.0, 64.0, 11.0)
```

Хотя функция кажется достаточно простой, на практике многое может пойти не так. Что делать, если список кортежей в пикселях поврежден и некоторые кортежи RGB содержат только два элемента вместо трех? Что, если одно значение имеет нецелочисленный тип? А если на выходе должен быть кортеж целых чисел, чтобы избежать ошибки, присущей всем вычислениям с плавающей точкой?

Юнит-тест можно провести для всех этих условий в изолированном режиме, чтобы убедиться, что функция работает.

Вот два простых юнит-теста, один из которых проверяет, работает ли функция для пограничного случая с нулями в качестве входных данных, а другой — возвращает ли функция кортеж целых чисел:

```
def unit_test_avg():
    print('Test average...')
    print(average_rgb([(0, 0, 0)]) == average_rgb([(0, 0, 0),
                                    (0, 0, 0)]))

def unit_test_type():
    print('Test type...')
    for i in range(3):
        print(type(average_rgb([(1, 2, 3), (4, 5, 6)])[i]) == int)

unit_test_avg()
unit_test_type()
```

Результат показывает, что проверка типа данных не удалась и функция не возвращает правильный тип, который должен быть кортежем целых чисел:

```
Test average...
True
Test type...
False
False
False
```

В реальных условиях тестировщики должны написать сотни таких юнит-тестов, чтобы проверить функцию на соответствие всем типам входных данных и определить, выдает ли она ожидаемые результаты. Только когда юнит-тесты покажут, что функция работает корректно, можно переходить к тестированию высокоуровневых функций приложения.

На самом деле, даже если все ваши юнит-тесты успешно пройдены, этап тестирования еще не завершен. Нужно проверить правильность взаимодействия юнитов между собой, поскольку все вместе они создают единое целое. Вы должны провести тесты в реальных условиях, проехав на автомобиле тысячи или даже десятки тысяч миль, чтобы выявить неожиданные модели поведения в странных и непредсказуемых ситуациях. Что, если ваша машина едет по небольшой дороге без дорожных знаков? Что делать, если автомобиль перед вами резко остановится? Что делать, если на перекрестке образуется пробка? Что, если водитель внезапно вырулит на встречную полосу?

Необходимо провести огромное количество тестов; сложность настолько высока, что многие бросают дело на полпути. То, что хорошо выглядит в теории и даже после первой реализации, часто не работает на практике после тестирования ПО на различных уровнях, таких как юнит-тесты или тесты на использование в реальных условиях.

Развертывание

Программное обеспечение уже прошло тщательный этап тестирования. Пришло время заняться его развертыванием! Оно может принимать различные формы. Приложения публикуются на маркетплейсах, пакеты — в репозиториях, а крупные (или мелкие) релизы могут быть выложены в общий доступ. При более итеративном и гибком подходе к разработке ПО вы многократно возвращаетесь к этому этапу несколько раз, осуществляя *непрерывное развертывание*. В зависимости от конкретного проекта на данном этапе вам придется запускать продукт, проводить маркетинговые кампании, общаться с первыми клиентами, исправлять новые баги, которые наверняка обнаружатся пользователями, организовывать развертывание ПО в разных операционных системах, предоставлять техническую поддержку и устранять различные проблемы или сопровождать кодовую базу, постоянно адаптируя и улучшая ее. Этот этап может стать довольно хлопотным, учитывая сложность и взаимозависимость разнообразных программных решений, которые вы разработали и реализовали на предыдущих фазах. В последующих главах будут предложены тактические приемы, которые помогут вам преодолеть этот сумбур.

Сложность в ПО и алгоритмическая теория

Во фрагменте программного обеспечения может быть столько же сложности, сколько и во всем процессе его разработки. В программировании существует множество метрик, измеряющих сложность ПО по формальным признакам.

Для начала мы рассмотрим *алгоритмическую сложность*, которая связана с требованиями различных алгоритмов к ресурсам для их выполнения. Используя понятие алгоритмической сложности, можно сравнивать несколько алгоритмов, решающих одну и ту же задачу. Допустим, вы создали игровое приложение с рейтинговой системой и таблицей рекордов. Вы хотите, чтобы игроки с самым высоким счетом отображались в верхней части списка, а игроки с самыми скромными результатами — внизу. Другими словами, вам нужно *отсортировать* список. Для этого существуют тысячи алгоритмов, и сортировка списка для 1 000 000 игроков требует бóльших вычислительных затрат, чем для 100. Некоторые алгоритмы хорошо масштабируются с увеличением размера входного списка, другие — нет. На самом деле, пока ваше приложение обслуживает несколько сотен человек, не имеет значения, какой алгоритм вы выберете, но по мере роста базы пользователей сложность обработки списка растет нелинейно. Вскоре игрокам придется ждать все дольше и дольше, пока сортируется список. Они начнут жаловаться — и вам понадобятся более продвинутые алгоритмы!

На рис. 1.2 приведен пример алгоритмической сложности для двух гипотетических алгоритмов. По оси *X* указан размер списка, который нужно отсортировать. Ось *Y* показывает время работы алгоритма (в единицах времени). Алгоритм 1 работает намного медленнее, чем алгоритм 2. Фактически его неэффективность становится более очевидной с ростом количества элементов списка для сортировки. При использовании алгоритма 1 скорость работы вашего игрового приложения будет снижаться по мере увеличения числа пользователей.

Давайте узнаем, справедливо ли это для реальных процедур сортировки в Python. На рис. 1.3 сравниваются три популярных алгоритма: пузырьковая сортировка (bubble sort), быстрая сортировка (quicksort) и Timsort. Пузырьковая сортировка имеет самую высокую алгоритмическую сложность. Быстрая сортировка и Timsort имеют одинаковую асимптотическую алгоритмическую сложность. Но алгоритм Timsort все же намного быстрее, поэтому он используется в Python по умолчанию в качестве процедуры сортировки. Время

выполнения алгоритма пузырьковой сортировки стремительно растет с увеличением размера списка.

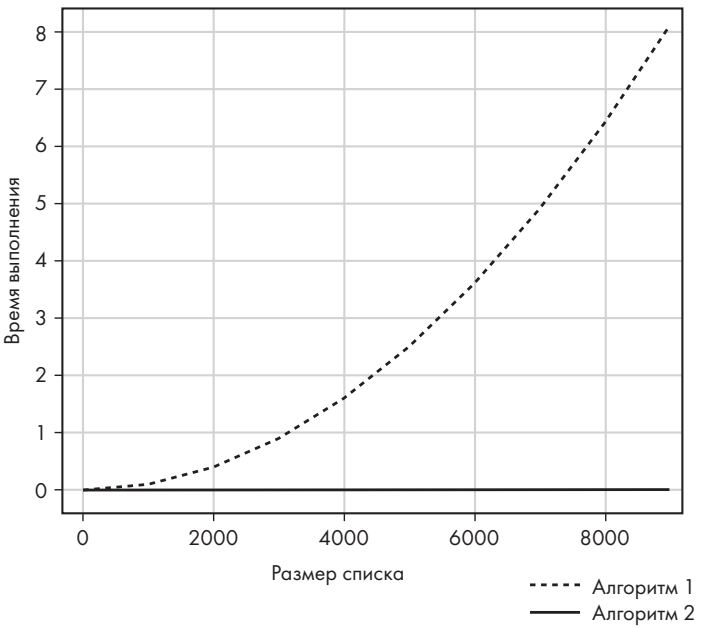


Рис. 1.2. Алгоритмическая сложность двух разных алгоритмов сортировки

На рис. 1.4 этот эксперимент повторяется только для быстрой сортировки и Timsort. И снова наблюдается существенная разница в алгоритмической сложности: Timsort лучше масштабируется и работает быстрее при растущем размере списка. Теперь вы понимаете, почему встроенный в Python алгоритм сортировки не менялся так долго!

В листинге 1.1 показан код на языке Python на случай, если вы захотите воспроизвести эксперимент. Я бы рекомендовал вам выбрать небольшое значение для `n`, потому что на моем компьютере программа до полного завершения выполняется слишком долго.

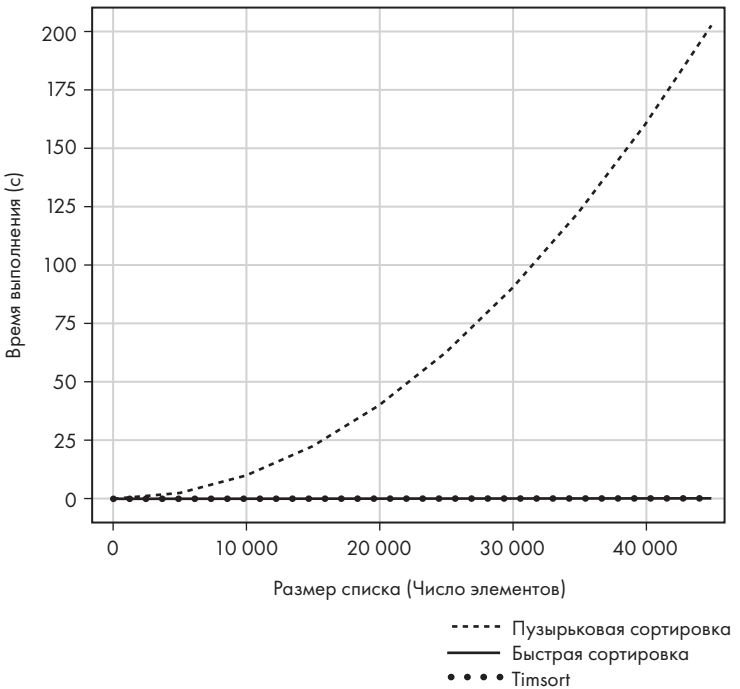


Рис. 1.3. Алгоритмическая сложность для алгоритмов пузырьковой сортировки, быстрой сортировки и Timsort

Листинг 1.1. Измерение времени выполнения для трех наиболее распространенных процедур сортировки

```
import matplotlib.pyplot as plt
import math
import time
import random


def bubblesort(l):
    # Источник: https://blog.finxter.com/daily-python-puzzle-
    # bubble-sort/
    lst = l[:] # Работайте с копией, не изменяйте оригинал
    for passesLeft in range(len(lst)-1, 0, -1):
        for i in range(passesLeft):
           if lst[i] > lst[i + 1]:
```

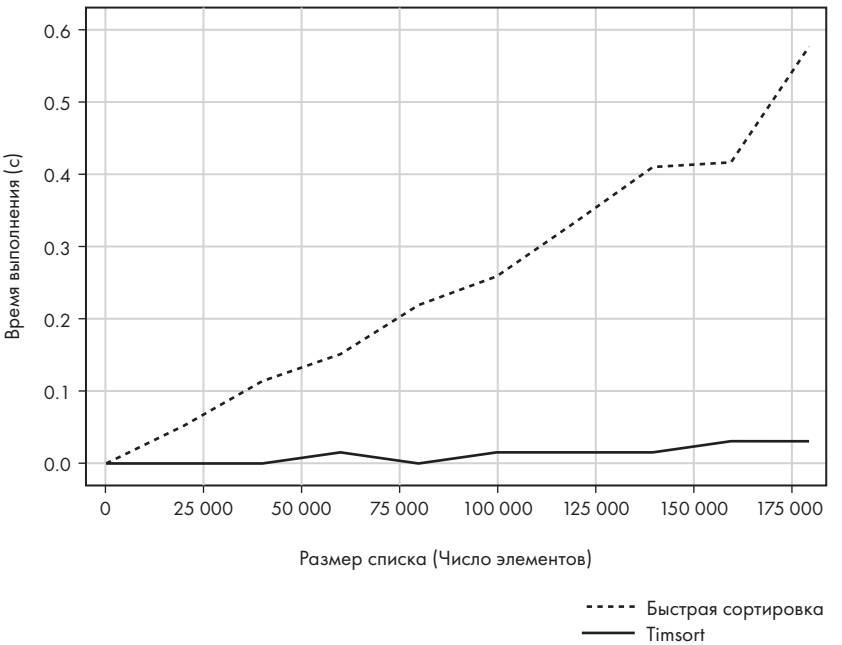


Рис. 1.4. Алгоритмическая сложность для алгоритмов быстрой сортировки и Timsort

```
                lst[i], lst[i + 1] = lst[i + 1], lst[i]
    return lst


def qsort(lst):
    # Пояснение: https://blog.finxter.com/python-one-line-quicksort/
    q = lambda lst: q([x for x in lst[1:] if x <= lst[0]])
                    + [lst[0]]
                    + q([x for x in lst if x > lst[0]]) if lst else []
    return q(lst)


def timsort(l):
      # sorted() использует Timsort внутренне
      return sorted(l)


def create_random_list(n):
```

```
    return random.sample(range(n), n)


n = 50000
xs = list(range(1,n,n//10))
y_bubble = []
y_qsort = []
y_tim = []

for x in xs:

    # Создание списка
    lst = create_random_list(x)

    # Измерение времени для пузырьковой сортировки
    start = time.time()

    bubblesort(lst)
    y_bubble.append(time.time()-start)

    # Измерение времени для быстрой сортировки
    start = time.time()
    qsort(lst)
    y_qsort.append(time.time()-start)

    # Измерение времени для сортировки Timsort
    start = time.time()
    timsort(lst)
    y_tim.append(time.time()-start)


plt.plot(xs, y_bubble, '-x', label='Bubblesort')
plt.plot(xs, y_qsort, '-o', label='Quicksort')
plt.plot(xs, y_tim, '--.', label='Timsort')

plt.grid()
plt.xlabel('List Size (No. Elements)')
plt.ylabel('Runtime (s)')
plt.legend()
plt.savefig('alg_complexity_new.pdf')
plt.savefig('alg_complexity_new.jpg')
plt.show()
```

Проблема алгоритмической сложности — это хорошо изученная сфера. На мой взгляд, оптимизированные алгоритмы, созданные в результате этого исследования, являются наиболее ценным

технологическим достоянием человечества. Они позволяют нам снова и снова решать одни и те же проблемы, снижая ресурсные затраты. Мы действительно «стоим на плечах гигантов»[1].

Помимо алгоритмической сложности можно измерить сложность кода, используя понятие *цикломатической сложности*. Эта метрика, разработанная Томасом Маккейбом (Thomas McCabe) в 1976 году, описывает количество *линейно независимых путей* выполнения кода, то есть число маршрутов в программном коде, которые имеют хотя бы одно ребро, не входящее в другой маршрут. Например, код с оператором `if` дает два линейно независимых пути через ваш код, поэтому он обладает большей цикломатической сложностью, чем плоский код без каких-либо ветвлений, подобных тем, что возникают при использовании `if`. На рис. 1.5 показана цикломатическая сложность

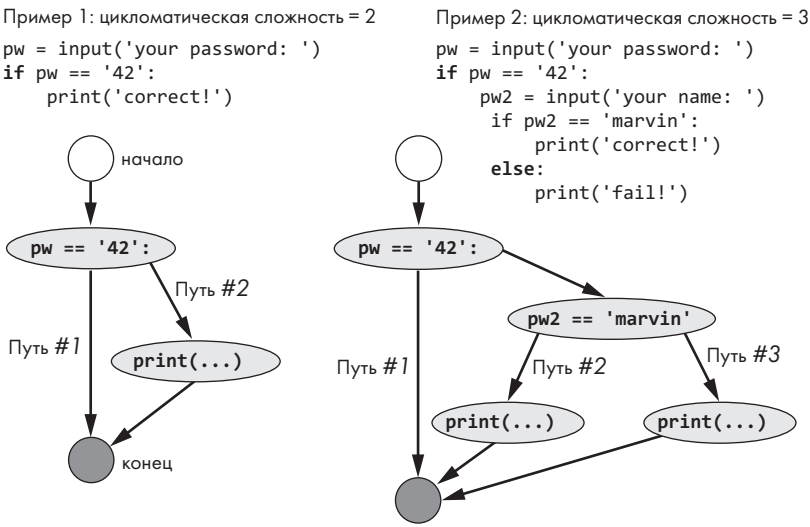


Рис. 1.5. Цикломатическая сложность двух программ на Python

[1] Крылатая фраза, означающая преемственность в познании, науке или искусстве: мы опираемся на достижения и открытия деятелей прошлого. — *Примеч. ред.*

двух программ на Python, которые обрабатывают пользовательский ввод и реагируют соответственно. Первая программа содержит только одну условную ветвь, которую можно считать развилкой. Можно выбрать любую из ветвей, но не обе сразу. Таким образом, цикломатическая сложность равна двум, поскольку существует два линейно независимых пути. Вторая программа содержит две условные ветви, что дает три таких пути и цикломатическую сложность, равную трем. Каждый дополнительный оператор `if` увеличивает цикломатическую сложность.

Цикломатическая сложность является надежной косвенной метрикой для трудноизмеримой *когнитивной сложности*, то есть того, насколько непросто понять данную кодовую базу. Однако цикломатическая сложность игнорирует когнитивную, возникающую, скажем, при использовании нескольких вложенных циклов `for` по сравнению с плоским циклом `for`. Именно поэтому учет других метрик, таких как показатель сложности `NPath`, делает измерение цикломатической сложности более точным. Подводя итог, можно сказать, что сложность кода не только является важным предметом алгоритмической теории, но и имеет прямое отношение ко всем практическим вопросам кодовой реализации, а также к написанию понятного, легко читаемого и надежного кода. Как алгоритмическая теория, так и сложность в программировании тщательно изучались на протяжении десятилетий. Основная цель этих исследований — *снижение вычислительной и невычислительной сложности*, чтобы минимизировать ее пагубное воздействие на производительность и эффективность использования как человеческих, так и машинных ресурсов.

Сложность в обучении

Факты не существуют в вакууме, они взаимосвязаны. Рассмотрим такие два факта:

Уолт Дисней родился в 1901 году.

Луи Армстронг родился в 1901 году.

Если вы введете эти данные в программу, она сможет ответить на вопросы типа *«Когда родился Уолт Дисней?»* или *«Кто родился в 1901 году?»*. Чтобы ответить на второй вопрос, программа должна определить взаимозависимость различных фактов. Можно построить модель этой информации следующим образом:

```
(Уолт Дисней, родился, 1901)
(Луи Армстронг, родился, 1901)
```

Чтобы выявить всех людей, родившихся в 1901 году, можно использовать запрос `(*, родился, 1901)` или любой другой способ установить связь между фактами и сгруппировать их.

В 2012 году Google запустил новую функцию поиска, показывающую инфобоксы на странице результатов. Эти основанные на фактах информационные поля заполняются с помощью структуры данных, называемой *графом знаний*, — массивной базы данных с миллиардами взаимосвязанных фактов для представления сведений в виде сетевой структуры. Вместо того чтобы хранить объективные и независимые факты, эта база содержит информацию о взаимосвязях между ними и другими фрагментами данных. Поисковая система Google использует этот граф, чтобы обогатить результаты поиска знаниями более высокого уровня и формировать ответы в автоматическом режиме.

На рис. 1.6 показан пример. Пусть один из узлов графа знаний посвящен известному ученому в области computer science Алану Тьюрингу. На графе знаний концепт `Алан Тьюринг` связан с различными фрагментами информации: год рождения (`1912`), области исследований (`Computer science`, `Философия`, `Лингвистика`) и научный руководитель (`Алонзо Черч`). Каждый из этих фрагментов связан и с другими фактами (например, сферой научной деятельности Алонзо Черча также была `Computer science`), образуя огромную сеть взаимосвязанных фактов. Вы можете использовать ее для получения новой информации и ответов на запросы пользователей программным путем. Запрос `"область исследований научного руководителя Тьюринга"` приведет к вычисляемому ответу

`"Computer science"`. Хотя это может показаться тривиальным или очевидным, но способность генерировать новые фактоиды[1], подобные этим, привела к прорыву в области поиска информации и релевантности поисковых систем. Вы, вероятно, согласитесь, что гораздо эффективнее учиться на ассоциациях, чем запоминать несвязанные факты.

Вот некоторые кортежи из трех элементов, представленные на графе:

```
("Алан Тьюринг", "имеет научного руководителя", "Алонзо Черч")
("Алан Тьюринг", "область исследования", "Философия")
("Алан Тьюринг", "область исследования", "Лингвистика")
```

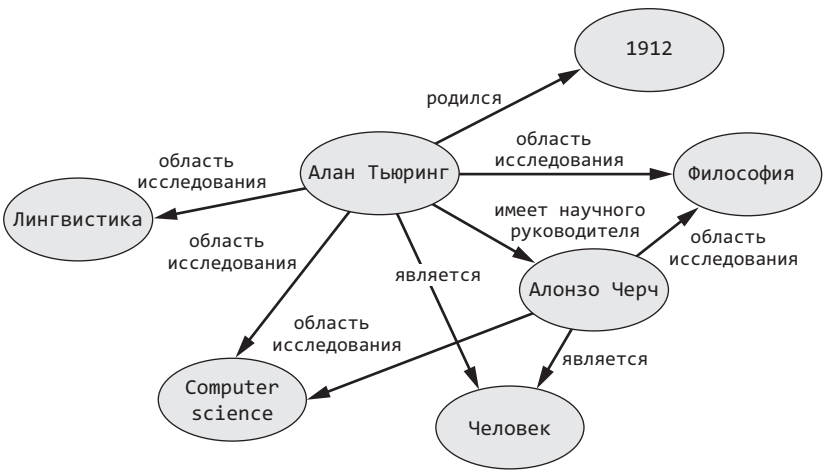


Рис. 1.6. Представление графа знаний

Каждая сфера научной деятельности сконцентрирована в небольшой части графа, каждая из которых, в свою очередь, состоит из множества взаимосвязанных фактоидов. По-настоящему разобраться

[1] В данном контексте — «небольшой элемент информации». — *Примеч. ред.*

в исследуемой области можно, только принимая во внимание все связанные с ней факты. Чтобы полностью понять Алана Тьюринга, вы должны изучить его убеждения, философию и характерные черты его научного руководителя. Чтобы понять Черча, нужно изучить его связь с Тьюрингом. Конечно, в графе содержится слишком много переплетенных между собой зависимостей и фактов, чтобы можно было ожидать полного понимания. Сложность этих взаимосвязей налагает самые фундаментальные ограничения на ваше стремление к познанию. Процесс обучения и сложность — две стороны одной медали: последняя находится на границе приобретенных вами знаний. Чтобы узнать больше, сначала следует понять, как контролировать сложность.

Мы немного отвлеклись на абстрактные рассуждения, так что вернемся к примерам! Допустим, вы хотите запрограммировать торгового бота, который покупает и продает активы согласно набору сложных правил. Перед тем как приступить к проекту, вы могли бы изучить множество полезных вещей: основы программирования, распределенные системы, базы данных, программные интерфейсы приложений (application programming interfaces, API), веб-сервисы, машинное обучение, обработку данных и связанную с ней математику. Также можно узнать о практических инструментах (Python, NumPy, scikit-learn, ccxt, TensorFlow и Flask) и разобраться в торговых стратегиях и философиях фондового рынка. Многие люди подходят к подобным задачам именно с таких позиций, и поэтому им всегда кажется, что браться за проект еще рано. Проблема в том, что чем больше вы знаете, тем менее компетентными вы себя чувствуете. Вы никогда не достигнете достаточного мастерства во всех этих областях, чтобы наконец почувствовать себя готовым. Подавленные сложностью этой затеи, вы уже ощущаете, что хотите все бросить. Сложность вот-вот получит свою следующую жертву — вас.

К счастью, благодаря этой книге вы научитесь различным приемам борьбы со сложностью: фокусировке, стремлению к простоте, уменьшению масштаба, исключению лишнего и минимализму.

Сложность в процессах

Процесс — это последовательность действий, предпринимаемых с целью достижения определенного результата. Его сложность рассчитывается по количеству действий, участников или ветвей. В общем случае чем больше действий (и участников), тем сложнее становится процесс (рис. 1.7).

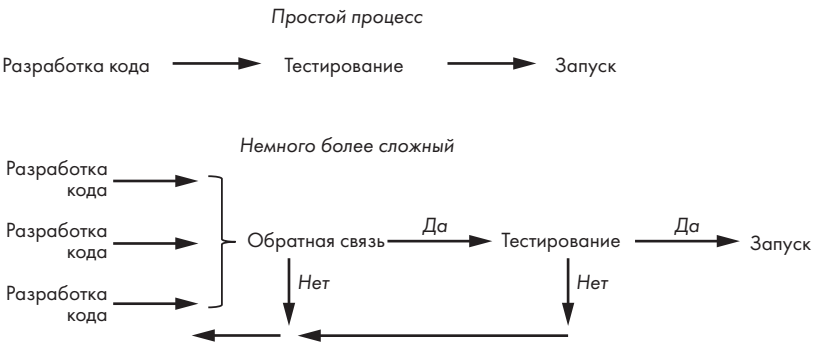


Рис. 1.7. Два примера процессов: разработка одним человеком и разработка в команде

Многие компании-разработчики ПО выбирают модели процессов для различных аспектов бизнеса, пытаясь упростить работу. Вот некоторые примеры:

При создании ПО применяют методологии разработки Agile или Scrum.

Для развития связей с клиентами используют CRM-системы[1] и скрипты продаж.

При создании новых продуктов и бизнес-моделей может использоваться бизнес-модель Canvas.

[1] CRM (customer relationship management) — управление взаимоотношениями с клиентами. — *Примеч. ред.*

Когда в организации накапливается слишком много процессов, сложность начинает затруднять работу системы. Например, до появления Uber процесс поездки из пункта А в пункт Б зачастую включал множество этапов: поиск телефонных номеров компаний-перевозчиков, сравнение тарифов, подготовку различных вариантов оплаты и планирование различных видов транспортировки. Для многих Uber упростил процесс поездки из пункта А в пункт Б, интегрировав весь процесс планирования в несложное мобильное приложение. Радикальное упрощение, реализованное в Uber, сделало поездки более удобными для клиентов и сократило время и затраты на планирование по сравнению с традиционной службой такси.

В чрезмерно сложных структурах гораздо меньше возможностей для внедрения инноваций, потому что они не в состоянии пробиться сквозь сложность. Ресурсы тратятся впустую, поскольку действия в рамках процессов становятся избыточными. Снова и снова менеджеры тратят усилия на создание новых процессов и действий, пытаясь оздоровить страдающий бизнес, и этот порочный круг начинает разрушать бизнес или структуру.

Сложность — враг продуктивности. Все решает минимализм: для оптимизации процесса вам следует полностью избавиться от ненужных шагов и действий. Очень маловероятно, что ваш процесс окажется *слишком упрощенным*.

Сложность в повседневной жизни: «смерть от тысячи порезов»

Цель этой книги — повысить вашу продуктивность в программировании. Однако прогрессу в данной области могут помешать ваши личные повседневные привычки и ежедневная рутина. Вы должны неустанно бороться с постоянными отвлекающими факторами и конкуренцией за ваше драгоценное время. Профессор computer science Кэл Ньюпорт (Cal Newport) рассказывает об этом в своей замечательной книге «Deep Work: Rules for Focused Success in a Distracted

World»[1] (Grand Central Publishing, 2016). Он утверждает, что спрос на работу, требующую глубокого мышления (такую как программирование, научные исследования, медицина и писательская деятельность), *увеличивается*, а время на ее выполнение *сокращается* из-за распространения устройств связи и мультимедийных систем. В случае когда растущий спрос встречает сокращающееся предложение, экономическая теория гласит, что цены будут повышаться. Если вы способны выполнять высокоинтеллектуальную работу, ваша экономическая ценность возрастает. Никогда еще не было лучшего времени для программистов, которые умеют погружаться в работу с головой.

А теперь главное: сегодня практически невозможно это сделать, если жестко не расставить приоритеты. Внешний мир постоянно отвлекает. Коллеги заглядывают к вам в кабинет. Смартфон требует внимания каждые 20 минут. Новые письма приходят десятки раз в день, и каждое из них требует уделить ему часть вашего времени.

Глубокое погружение приводит к отложенному удовольствию; это приятное чувство, когда вы потратили недели на разработку программы и наконец убедились, что она работает. Однако в большинстве случаев вам хочется получить удовлетворение немедленно. Ваше подсознание часто ищет способы уйти от необходимости глубокого погружения. Мелкие поощрения вызывают легкий прилив эндорфинов: проверка сообщений, бессмысленная болтовня, пролистывание Netflix. Призрак отложенного удовольствия становится все менее привлекательным по сравнению со счастливым, красочным и живым миром мгновенного удовлетворения.

Ваши усилия, направленные на то, чтобы оставаться сосредоточенным и продуктивным, могут пасть «смертью от тысячи порезов». Да, вы можете один раз выключить смартфон, усилием воли не проверять соцсети и не включать любимые сериалы, но сможете ли вы делать это постоянно изо дня в день? И здесь решение кроется

[1] Ньюпорт К. «В работу с головой. Паттерны успеха от IT-специалиста». Санкт-Петербург, издательство «Питер».

в применении радикального минимализма по отношению к сути проблемы. *Удалите* приложения социальных сетей, не пытаясь управлять временем, которое вы на них тратите. *Сократите* количество ваших проектов и задач, вместо того чтобы стараться все успеть, работая больше. *Погрузитесь* в один язык программирования и не расходуйте время на переключение между несколькими.

Заключение

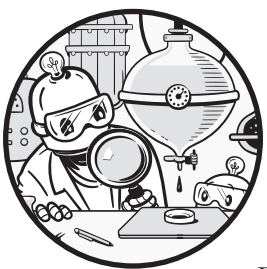
К этому моменту вы уже должны быть глубоко мотивированы необходимостью борьбы со сложностью. Для дальнейшего изучения этой проблемы и того, как можно ее преодолеть, я рекомендую прочитать «Deep Work» Кэла Ньюпорта.

Сложность вредит продуктивности и снижает концентрацию внимания. Если вы не сможете управлять ею на ранней стадии, она быстро поглотит ваш самый ценный ресурс — время. В конце жизни вы не будете оценивать ее смысл по количеству электронных писем, на которые вы ответили, и часов, потраченных на компьютерные игры или головоломки судоку. Научившись управлять сложностью, сохраняя простоту, вы сможете бороться с ней и получите мощное конкурентное преимущество.

В главе 2 вы узнаете о силе принципа 80/20: сосредоточьтесь на жизненно важном и игнорируйте тривиальное.

2

Принцип 80/20



В этой главе вы узнаете о глубинном влиянии *принципа 80/20* на жизнь программиста. У этого термина много наименований, например *закон Парето* — в честь его первооткрывателя Вильфредо Парето (Vilfredo Pareto). Итак, как же работает данное правило и почему это важно? В основе принципа 80/20 лежит идея о том, что большинство следствий (80 %) является результатом малого числа причин (20 %). Он указывает путь к достижению гораздо больших результатов в качестве профессионального разработчика кода: нужно сосредоточить усилия на нескольких важных вещах, игнорируя при этом множество ненужных, едва ли способных сдвинуть дело с мертвой точки.

Основы принципа 80/20

Принцип 80/20 гласит, что большинство следствий вытекает из малого количества причин. Например, львиная доля доходов

зарабатывается меньшинством людей, бóльшая часть изобретений делается небольшим количеством исследователей, подавляющее число книг написано меньшинством авторов и т. д.

Возможно, вы слышали о принципе 80/20 — он повсеместно встречается в литературе по личной эффективности. Причина его популярности двойственна. Во-первых, он позволяет вам быть расслабленным и продуктивным одновременно, пока вы выявляете те вещи, которые имеют значение и составляют те самые 20 % деятельности, приводящие к 80 % результатов, и упорно фокусируетесь на них.

Во-вторых, этот принцип соблюдается в огромном разнообразии ситуаций, что делает его вполне заслуживающим доверия. Трудно даже придумать противоположный случай, когда следствия в равной степени вытекают из причин. Попробуйте найти примеры распределений 50/50, где 50 % факторов дают 50 % результатов! Конечно, распределение не всегда равно 80/20 — конкретные цифры могут меняться на 70/30, 90/10 или даже 95/5, но оно всегда сильно смещено в сторону меньшего количества причин, порождающих большинство следствий.

Давайте опишем принцип Парето с помощью распределения Парето, показанного на рис. 2.1.

Распределение Парето отображает зависимость результатов (ось Y) от причин (ось X). Результатом может быть любой показатель успеха или неудачи, например доход, производительность или количество багов в программном продукте. Причинами могут быть любые объекты, с которыми связаны эти результаты, например сотрудники, предприятия или программный код соответственно. Чтобы получить график зависимости распределения Парето, надо упорядочить причины в соответствии с производимыми ими результатами. Например, человек, зарабатывающий больше всех, идет первым по оси X, затем идет сотрудник со вторым по величине доходом и т. д.

Рассмотрим это на практике.

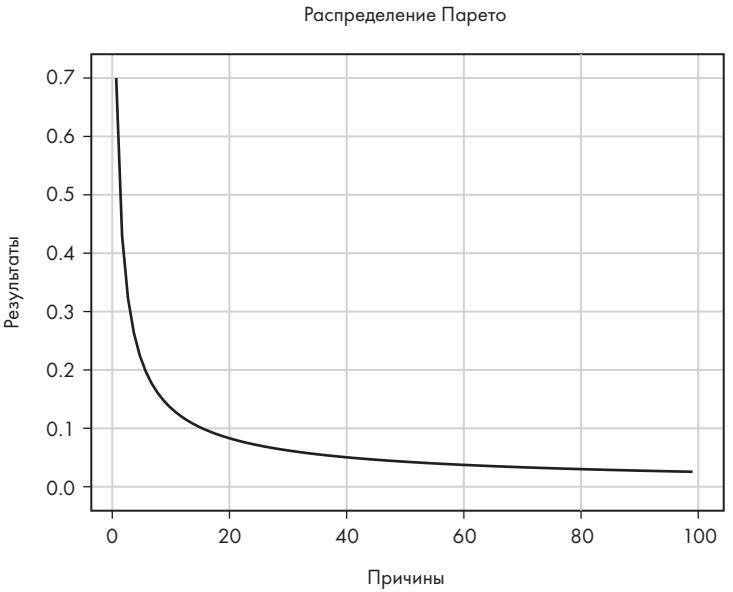


Рис. 2.1. Пример обобщенного распределения Парето

Оптимизация прикладного ПО

На рис. 2.2 представлен принцип Парето в действии на примере гипотетического проекта: меньшая часть кода отвечает за бóльшую часть времени работы программы. По оси *X* отложены функции кода, отсортированные по времени их выполнения, которое показано на оси *Y*. Заштрихованная область, которая составляет бóльшую часть общей площади под кривой, показывает, что подавляющее число функций кода вносит гораздо меньший вклад в общее время выполнения, чем несколько первых. Джозеф Джуран (Joseph Juran), один из первооткрывателей принципа Парето, называет первую часть *жизненно важным меньшинством*, а вторую — *тривиальным большинством*. Потратив много времени на оптимизацию тривиального большинства, мы едва ли уменьшим общее время выполнения программы. Наличие распределений Парето в программных проектах научно подтверждено в статье Луридаса (P. Louridas), Спинеллиса

(D. Spinellis) и Влахоса (V. Vlachos) «Power Laws in Software» («Степенные законы в программном обеспечении») (2008).

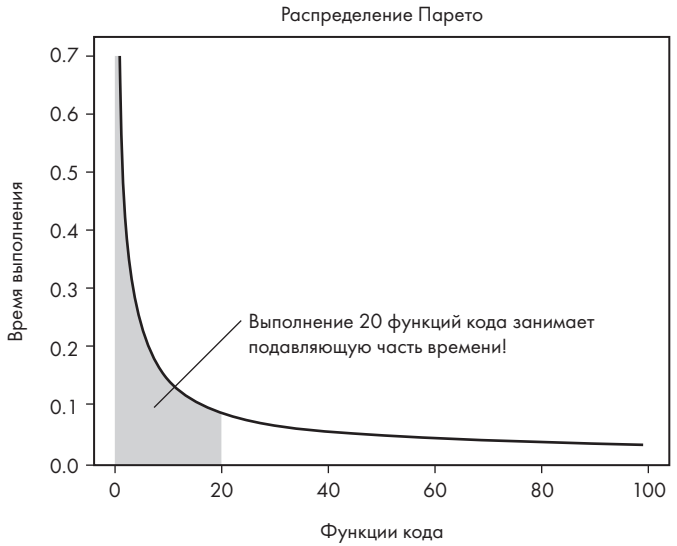


Рис. 2.2. Пример распределения Парето в программировании

Крупные компании, такие как IBM, Microsoft и Apple, используют принцип Парето для создания более быстрых и простых в использовании компьютеров, концентрируя свое внимание на жизненно важном меньшинстве, то есть многократно оптимизируя 20 % кода, который чаще всего выполняется при работе среднестатистического пользователя. Не весь код равнозначен. Меньшая его часть оказывает доминирующее воздействие на клиентский опыт, в то время как бóльшая часть влияет незначительно. Вы можете кликать по иконке проводника несколько раз в день, но вы очень редко меняете права доступа к файлу. Принцип 80/20 подскажет вам, где сосредоточить усилия по оптимизации!

Сам принцип легко понять, но сложнее разобраться, как использовать его в своей жизни.

Продуктивность

Сосредоточившись на жизненно важном меньшинстве, а не тривиальном большинстве, вы можете повысить свою продуктивность в 10, а то и в 100 раз. Не верите? Давайте посмотрим, откуда берутся такие цифры, основываясь на распределении 80/20.

Мы будем использовать традиционные параметры 80/20 (то есть 80 % результатов получены благодаря работе 20 % людей), а затем рассчитаем производительность труда для каждой группы. В некоторых областях (например, в программировании) распределение, вероятно, будет гораздо асимметричнее.

На рис. 2.3 представлена компания из 10 сотрудников, где 2 человека производят 80 % результата, а остальные 8 — только 20 %. Разделив 80 % на 2, мы получим среднюю продуктивность в 40 % на одного самого результативного сотрудника в компании. Разделим оставшиеся 20 % между 8 низкорезультативными работниками и получим среднюю продуктивность 2.5 % на каждого. Вклад этих людей различается в 16 раз!

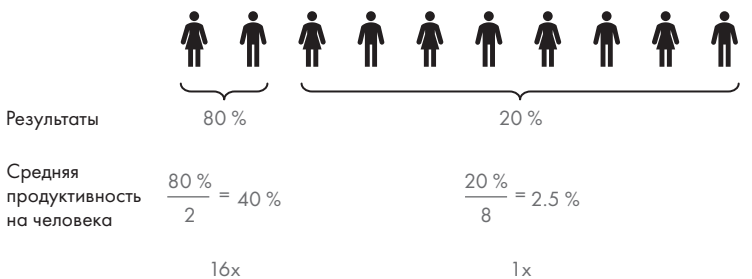


Рис. 2.3. Средняя продуктивность 20 % лучших сотрудников в 16 раз превышает показатель 80 % остальных

Эта 16-кратная разница в средней продуктивности является фактом для миллионов организаций по всему миру. Кроме того, распределение Парето является фрактальным, что означает, что 20 % лучших

из 20 % лучших дают 80 % из 80 % результатов, что объясняет еще бóльшие различия в эффективности отдельных сотрудников в крупных организациях с тысячами людей.

Различия в результативности нельзя объяснить только интеллектом — один человек не может быть в 1000 раз умнее другого. Они обусловлены особенностями поведения конкретного сотрудника или организации. Если бы вы действовали так же, вы получили бы те же результаты. Однако прежде чем изменить свое поведение, вы должны четко осознать, чего хотите добиться: исследования показывают, что эффект сильно меняется в зависимости практически от любых параметров, которые вы только можете себе представить.

Доход. 10 % людей в США получают почти 50 % всего дохода.

Благополучие. Менее 25 % жителей Северной Америки оценивают свое благополучие на 9 или 10 баллов по шкале от 0 до 10, где «наименее благополучная жизнь» — 0, а «наиболее благополучная жизнь» — 10 баллов.

Количество активных пользователей в месяц. Только 2 из 10 крупнейших веб-сайтов, предназначенных для всех групп пользователей, имеют 48 % совокупного трафика, как следует из таблицы 2.1 (по данным сайта https://www.ahrefs.com/).

Продажа книг. Всего 20 % авторов формируют до 97 % продаж книг.

Научная продуктивность. Например, на долю 5.2 % ученых приходится 38 % всех опубликованных статей.

Раздел «Источники» в конце главы содержит ссылки на некоторые статьи, подтверждающие эти данные. Неравномерность результатов — хорошо известное явление в социальных науках, обычно описываемое с помощью статистического показателя под названием *коэффициент Джини*.

Таблица 2.1. Совокупный трафик 10 наиболее посещаемых в США веб-сайтов

#	Домен	Ежемесячный трафик	Совокупный трафик
1	en.wikipedia.org	1 134 008 294	26 %
2	youtube.com	935 537 251	48 %
3	amazon.com	585 497 848	62 %
4	facebook.com	467 339 001	72 %
5	twitter.com	285 460 434	79 %
6	fandom.com	228 808 284	84 %
7	pinterest.com	203 270 264	89 %
8	imdb.com	168 810 268	93 %
9	reddit.com	166 277 100	97 %
10	yelp.com	139 979 616	100 %
4 314 988 360			

Как же стать одним из лучших исполнителей? Или, если сформулировать в более общем виде: как *сместиться влево* по кривой распределения Парето в вашей компании (рис. 2.4)?

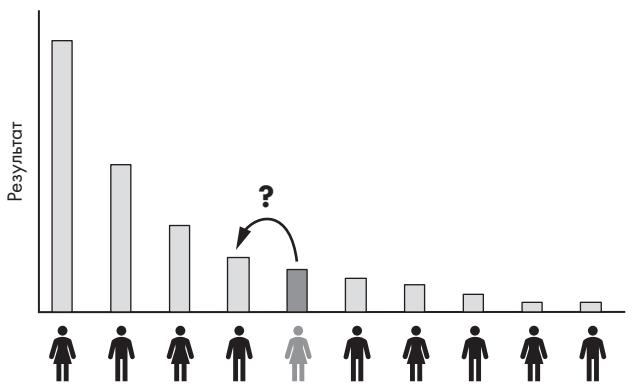


Рис. 2.4. Чтобы дать больший результат, вам нужно сместиться влево по кривой

Метрики успеха

Допустим, вы хотите оптимизировать доход. Как сместиться влево по кривой Парето? Здесь мы должны отложить точную науку, потому что вам нужно найти причины успеха некоторых людей в конкретной отрасли и разработать действенные параметры успеха, которые вы можете контролировать и реализовывать. Мы определим термин *«метрики успеха»* как критерии измерения поведения, которое приводит к большему успеху в вашей сфере. Сложность в том, что наиболее важные метрики в большинстве областей различны. К ним также применим принцип 80/20: некоторые из них оказывают большее влияние на вашу продуктивность в той или иной области, в то время как другие практически не имеют значения.

Например, будучи научным сотрудником со степенью, я вскоре понял, что успех в том, чтобы тебя цитировали другие исследователи. Чем больше ссылок на ваши работы, тем больше у вас авторитета, известности и возможностей. Однако увеличение количества цитирований (индекса цитирования) вряд ли можно назвать реальным показателем успеха, который можно ежедневно оптимизировать. Индекс цитирования — это *запаздывающий индикатор*, поскольку он основан на том, что вы сделали ранее. Проблема заключается в том, что такие индикаторы фиксируют только последствия прошлых действий. Они не подскажут вам, что нужно ежедневно предпринимать для достижения успеха.

Чтобы найти критерий оценки правильности совершаемых действий, было введено понятие *опережающих индикаторов*. Это метрика, которая предсказывает изменение запаздывающего индикатора до того, как событие произойдет. Если вы хорошо поработаете над опережающим индикатором, то запаздывающий, скорее всего, в итоге тоже улучшится. Будучи научным сотрудником, вы получите больше цитирований (запаздывающий индикатор), опубликовав больше отличных научных статей (опережающий индикатор). Это означает, что написание ценных работ является наиболее важным занятием для большинства ученых, в отличие от второстепенных дел, таких как

подготовка презентаций, организация мероприятий, преподавание или употребление кофе. Таким образом, метрикой успеха для научных работников является публикация максимального количества качественных статей, как показано на рис. 2.5.

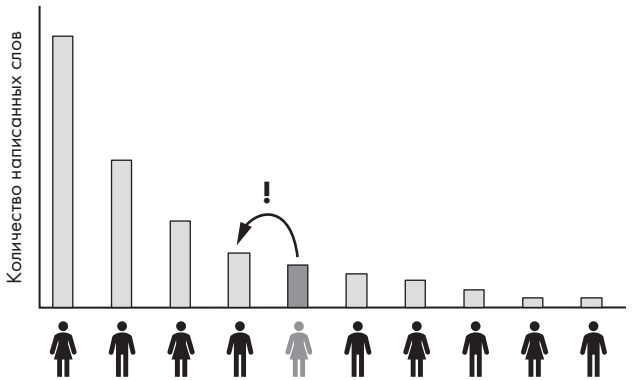


Рис. 2.5. Метрика успеха в научных исследованиях: количество слов, написанных для создания качественной статьи

Чтобы сместиться влево по кривой Парето в научных исследованиях, вы должны ежедневно писать больше слов, раньше опубликовать следующую качественную статью, быстрее получить больше цитирований, расширить свой научный вклад и стать более успешным ученым. Грубо говоря, существует много различных метрик успеха, которые служат косвенным показателем успешности в науке. Например, если упорядочить их по шкале от отстающих к опережающим показателям, можно получить *количество цитирований, количество написанных качественных статей, общее количество слов, написанных за всю жизнь,* и *количество слов, написанных сегодня*.

Подход 80/20 позволяет определить те виды деятельности, на которых следует сфокусироваться. Улучшение большего количества метрик успеха (тех, которые в первую очередь имеют значение для опережающих индикаторов) повысит ваш профессиональный успех; остальное не столь важно. Тратьте меньше времени на все остальные

задачи. Откажитесь «умирать от тысячи порезов». Будьте ленивы во всех делах, кроме одного: *писать больше слов в день*.

Допустим, вы работаете 8 часов в день и делите свой день на 8 занятий по 1 часу на каждое. После выполнения упражнения под названием «Метрика успеха» вы понимаете, что можно пропустить 2 часовых занятия в день и выполнить 4 других задания вдвое быстрее, если снизить свой перфекционизм. Вы сэкономили 4 часа в день, но все равно достигли 80 % результата. Теперь вы можете ежедневно тратить 2 часа на то, чтобы писать больше слов для качественных статей. В течение нескольких месяцев вы направите в печать одну дополнительную работу, и со временем вы подадите больше статей, чем любой из ваших коллег. Вы работаете всего по 6 часов в день, и служебные обязанности в основном выполняете не очень качественно. Но вы проявляете себя там, где это важно: вы публикуете больше научных работ, чем кто-либо другой в вашем окружении. В результате вы скоро войдете в 20 % лучших научных сотрудников. Вы добиваетесь большего меньшими усилиями.

Вместо того чтобы становиться «тем, кто за все берется, но ничего не умеет», вы приобретаете опыт в той области, которая для вас наиболее важна. Вы сосредоточиваетесь на жизненно важном меньшинстве и игнорируете тривиальное большинство. Вы ведете менее напряженную жизнь, но получаете больше отдачи от вложенного труда, усилий, времени и денег.

Фокус и распределение Парето

Вопрос, тесно связанный с темой данной главы, который я хотел бы здесь обсудить, — это *фокус*. Мы будем неоднократно говорить про него в этой книге — например, в главе 9 подробно рассматривается, в чем его сила, — однако принцип 80/20 объясняет, *почему* фокус настолько важен. Перейдем к обсуждению!

Рассмотрим распределение Парето, представленное на рис. 2.6. На графике показан результат в процентах в зависимости от

продвижения к вершине распределения. Алиса — пятый по продуктивности сотрудник в компании. Если она обойдет только одного человека и станет четвертой, то ее результат (зарплата) увеличится на 10 %. Еще один шаг — и ее результат увеличится на *дополнительные* 20 %. В распределении Парето при переходе на одну позицию отдача возрастает экспоненциально, поэтому даже небольшое увеличение производительности может привести к большому росту дохода. Повышение продуктивности приводит к нелинейному росту доходов, счастья и радости от работы. Такое явление иногда называют «победитель получает все» (the winner takes it all).

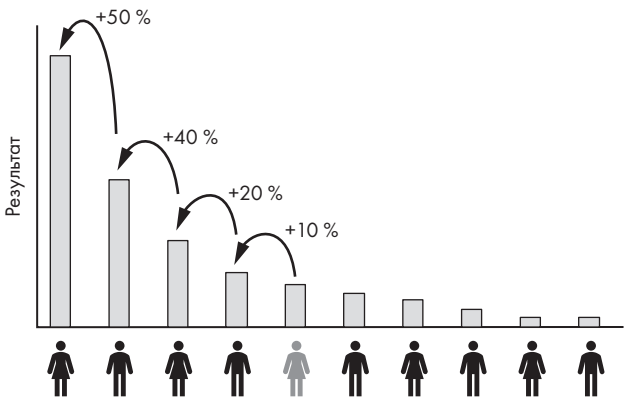


Рис. 2.6. Несоразмерно высокое дополнительное преимущество, получаемое при улучшении позиции в распределении Парето

Вот почему не стоит распылять свое внимание: *если вы не сфокусированы, вы участвуете в большом количестве распределений Парето.* Рассмотрим рис. 2.7: Алиса и Боб могут вкладывать каждый день по три единицы усилий в обучение. Алиса сосредоточена на одной вещи — программировании. Она тратит все три единицы на то, чтобы научиться писать код. Боб распыляет свое внимание на несколько дисциплин: одну единицу времени он тратит на совершенствование навыков шахматиста, одну — на повышение мастерства программирования и еще одну — на продвижение в политике. Он достиг

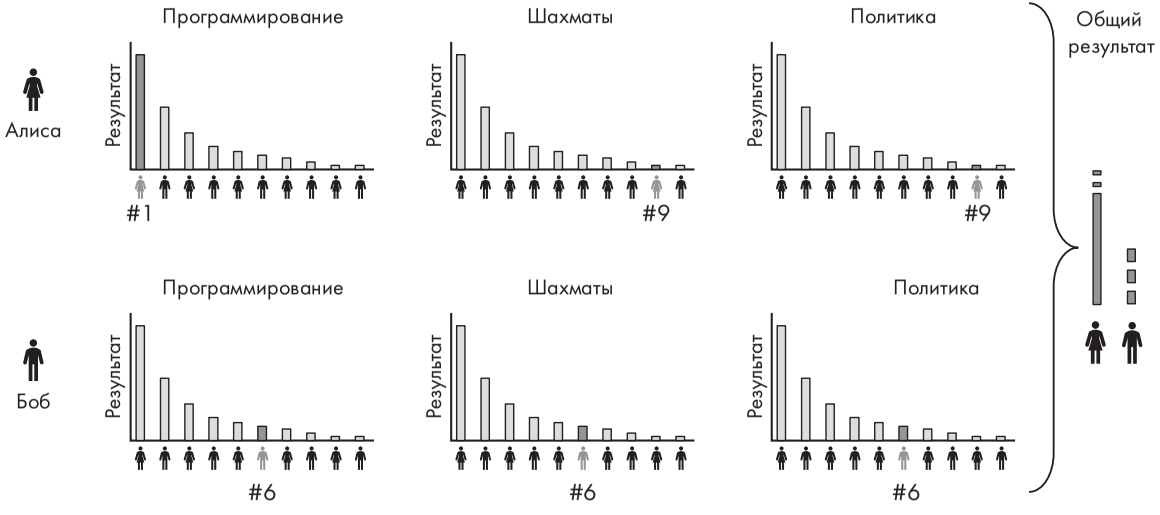


Рис. 2.7. Нелинейность результатов ранжирования — стратегическое объяснение того, в чем сила фокуса

среднего уровня знаний и результатов в каждой из трех областей. Но распределение Парето непропорционально вознаграждает лучших исполнителей, поэтому Алиса получает большее вознаграждение за общий результат.

Непропорциональность вознаграждения сохраняется и внутри каждой области. Например, Боб может потратить свое время на чтение трех общеобразовательных книг (назовем их «Введение в Python», «Введение в C++» и «Введение в Java»), в то время как Алиса прочтет три книги, углубленно изучая машинное обучение на Python (назовем их «Введение в Python», «Введение в машинное обучение на Python» и «Машинное обучение для специалистов»). В результате Алиса сосредоточится на том, чтобы стать экспертом в области машинного обучения, и сможет требовать более высокую зарплату за свой специализированный комплекс навыков.

Значение принципа 80/20 для разработчиков кода

В программировании распределение результатов, как правило, гораздо сильнее смещено в сторону лучших, чем в большинстве других областей. Вместо 80/20 оно нередко больше похоже на 90/10 или 95/5. Билл Гейтс сказал, что *«отличный токарь получает в несколько раз больше, чем средний, но великий программист стóит в 10 000 раз больше, чем обычный»*. Гейтс утверждает, что разница между великим и средним создателем ПО не в 16 раз, а в 10 000! Вот несколько причин, по которым мир программистов склонен к таким экстремальным распределениям Парето:

- Великий программист может справиться с некоторыми проблемами, которые обычный программист просто не в состоянии решить. В отдельных случаях это делает их бесконечно более продуктивными.
- Он способен написать код, работающий в 10 000 раз быстрее, чем код среднего программиста.

- Он пишет код с меньшим количеством багов. Подумайте о том, как влияет один баг в безопасности на репутацию и бренд Microsoft! Более того, каждый дополнительный баг требует затрат времени, энергии и денег на последующие модификации кодовой базы и добавление функций — так работает накопленный эффект багов, наносящий значительный ущерб.
- Он пишет код, который легче расширять, что может повысить производительность тысяч людей, которые будут работать с кодом на более поздних этапах процесса разработки ПО.
- Он мыслит нестандартно и находит креативные решения, позволяющие избежать дорогостоящих этапов разработки и дающие возможность сосредоточить усилия на самом главном.

В реальности всегда действует комбинация этих факторов, поэтому разница в эффективности великого и среднего программиста может быть еще больше.

Итак, для вас ключевой вопрос, наверное, следующий: как стать великим программистом?

Метрика успеха для программиста

К сожалению, установка «стать великим программистом» не является метрикой успеха, которую можно напрямую оптимизировать — эта проблема многогранна. Крутой программист быстро понимает код, разбирается в алгоритмах и структурах данных, знает различные технологии, их сильные и слабые стороны, умеет сотрудничать с людьми, коммуникабелен и креативен, постоянно повышает квалификацию и в совершенстве знает способы организации процесса разработки ПО, а также обладает сотнями других полезных профессиональных навыков и личных качеств. Но вы не сможете стать специалистом во всем этом! Если вы не сосредоточитесь на жизненно важных навыках, вас захлестнет тривиальное большинство. Чтобы стать великим программистом, вы должны сфокусироваться на нескольких особо важных задачах.

Одно из таких важных дел — это написание большего количества строк кода. Чем больше вы напишете, тем лучшим программистом вы станете. Это сильное упрощение многогранной задачи: оптимизируя промежуточную прокси-метрику (написать больше строк кода), вы увеличиваете свои шансы на успех в целевой метрике (стать великим программистом) (рис. 2.8).

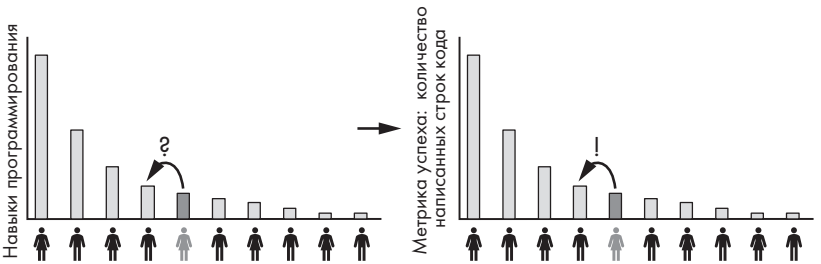


Рис. 2.8. Метрика успеха в программировании — количество написанных строк кода

Написав больше строк кода, вы станете лучше в нем разбираться, будете говорить и вести себя как опытный программист. Вы привлечете в свой круг общения лучших разработчиков и найдете более сложные задачи для реализации, поэтому вам придется писать все больше кода — и вы станете еще компетентнее. Вам будут платить все больше и больше за каждую написанную строчку. Вы или ваша компания сможете передавать на аутсорсинг многие тривиальные задачи.

Вот упражнение в рамках принципа 80/20, которое вы можете выполнять ежедневно: отслеживайте показатель количества строк, которые вы пишете в день, и оптимизируйте его. Сделайте это игрой, чтобы каждый день соответствовать хотя бы своему среднему показателю.

Распределение Парето в реальном мире

Кратко рассмотрим несколько реальных примеров применения распределения Парето.

Вклад в репозиторий TensorFlow на GitHub

Яркий пример распределения Парето — вклад в репозитории на GitHub. Рассмотрим чрезвычайно популярный репозиторий для вычислений в машинном обучении на Python: *TensorFlow*. На рис. 2.9 показаны семь основных контрибьюторов в этот репозиторий на GitHub. В табл. 2.2 те же данные представлены в числовом виде.

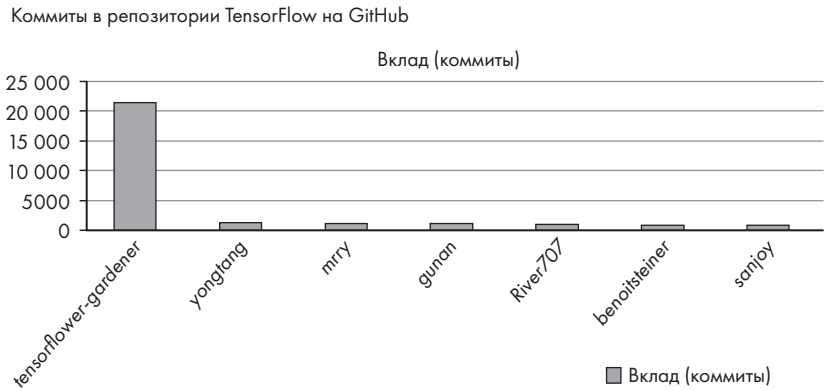


Рис. 2.9. Распределение коммитов в репозитории TensorFlow на GitHub

Таблица 2.2. Количество коммитов TensorFlow и их авторы

Контрибьютор	**Коммиты**
tensorflower-gardener	21 426
yongtang	1251
mrry	1120
gunan	1091
River707	868
benoitsteiner	838
sanjoy	795

Из 93 000 коммитов в этот репозиторий пользователь *tensorflow-gardener* внес более 20 %. Учитывая, что участников тысячи, распределение оказалось гораздо более экстремальным, чем 80/20. Причина в том, что контрибьютор *tensorflow-gardener* — это на самом деле команда программистов Google, которые создали и поддерживают TensorFlow. Тем не менее, даже если их отбросить, оставшиеся отдельные топ-контрибьюторы — очень успешные программисты с впечатляющим послужным списком. Вы можете познакомиться с ними на официальной странице GitHub. Многие из них нашли интересную работу в очень привлекательных компаниях. Добились ли они этого до или после внесения большого количества коммитов в репозиторий с открытым исходным кодом — это чисто теоретический вопрос. С практической точки зрения для достижения успеха вы должны прямо сейчас взять себе в привычку писать как можно больше строк кода каждый день. Ничто не мешает вам стать номером два в репозитории TensorFlow, отправляя туда стоящий код два-три раза в день в течение следующих двух-трех лет. Если вы проявите настойчивость, вы сможете вступить в ряды самых успешных разработчиков кода на земле, просто выбрав одну крутую привычку и следуя ей в течение пары лет!

Доходы программистов

Разумеется, доходы программистов также подчиняются распределению Парето. Из соображений конфиденциальности сложно получить данные о собственных средствах отдельного человека, однако на сайте https://www.networthshare.com/ можно найти данные об открыто заявленном доходе представителей разных профессий, включая программистов. Возможно, данные не совсем точные, но даже они показывают яркое специфическое отклонение от реальных распределений Парето (рис. 2.10).

Довольно много миллионеров в области программного обеспечения оказалось в нашей небольшой выборке из 29 точек данных! Но на самом деле кривая, вероятно, еще больше отличается от стандартной, потому что среди программистов много миллиардеров — например,

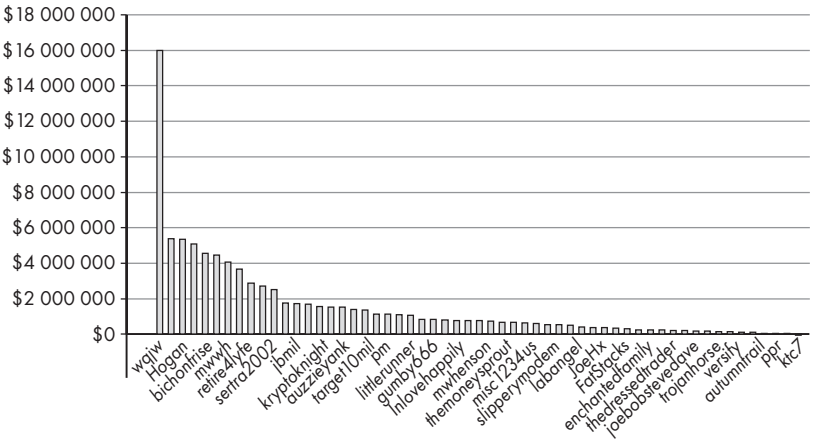


Рис. 2.10. Открыто заявленные доходы 60 программистов

Марк Цукерберг, Билл Гейтс, Илон Маск и Стив Возняк. Каждый из этих гениев-изобретателей сам создавал прототипы своих сервисов, приложив руку к исходному коду. В последнее время мы видим гораздо больше таких богачей-программистов в сфере блокчейна.

Фрилансеры

В сфере фриланс-разработки лидируют два портала, где фрилансеры могут предложить свои услуги, а заказчики — нанять исполнителей: Upwork и Fiverr. Для этих платформ рост количества пользователей и доходов в год выражается двузначными цифрами, и обе они нацелены на то, чтобы кардинально изменить привычную организацию работы талантов со всего мира.

Средний доход разработчика-фрилансера составляет $51 в час. Но это лишь средний показатель — зарплата 10 % лучших специалистов гораздо выше. На более или менее открытых рынках структура доходов напоминает распределение Парето.

Я наблюдал неравномерность распределения доходов на собственном опыте с трех точек зрения: (1) как фрилансер, (2) как заказчик,

нанимающий сотни фрилансеров, и (3) как создатель курсов, предлагающих обучение языку Python для фрилансеров. Большинству студентов не удается достичь даже потенциального среднего заработка, потому что они не могут сохранять свои позиции больше месяца или около того. Те, кто продолжает ежедневно работать над своими фриланс-проектами в течение нескольких месяцев, обычно достигают среднего уровня заработка в $51 в час. Небольшая часть очень амбициозных и преданных своему делу студентов достигают ставки $100 в час и выше.

Но почему одни студенты терпят неудачу, а другие преуспевают? Давайте построим график количества успешных заказов, выполненных разработчиками-фрилансерами на платформе Fiverr со средней оценкой не менее 4 из 5. На рис. 2.11 я сосредоточился на популярной области — машинном обучении. Я собрал данные с сайта Fiverr и отследил количество выполненных заказов для 71 фрилансера в двух самых популярных результатах поиска по категории «Машинное обучение». Для нас неудивительно, что диаграмма похожа на распределение Парето.

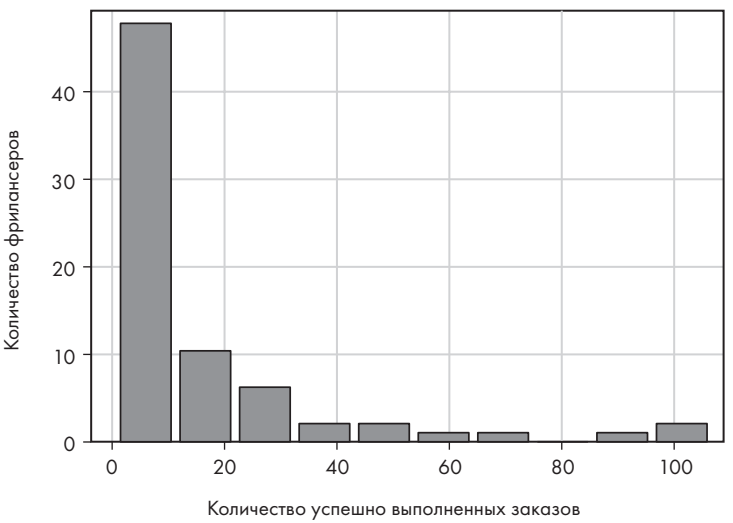


Рис. 2.11. Фрилансеры платформы Fiverr и число завершенных ими заказов

Как преподаватель я работал с тысячами студентов-фрилансеров и поражаюсь тому, что подавляющее большинство из них выполнило менее 10 заказов. Подозреваю, многие из них позже скажут, что фриланс не работает. Для меня это оксюморон — с таким же успехом можно заявить, что «работа не работает» или «бизнес не работает». Эти студенты не справились, так как старались недостаточно долго или усердно. Они думали, что их ждут легкие деньги, а когда поняли, что для достижения высоких результатов придется упорно трудиться, быстро сдались.

Отсутствие настойчивости у фрилансеров на самом деле дает вам прекрасную возможность продвинуться вверх согласно распределению Парето. Простая метрика успеха, которая практически гарантирует, что вы в конечном итоге войдете в 1–3 % лучших фрилансеров, заключается в следующем: *выполняйте больше заказов*. Оставайтесь подольше в игре. Это может сделать каждый. Тот факт, что вы читаете эту книгу, показывает, что у вас есть желание, амбиции и мотивация, чтобы войти в 1–3 % лучших профессиональных разработчиков-фрилансеров. Большинство игроков страдают от недостатка фокуса, и, даже если они квалифицированны, умны и имеют массу полезных знакомств, у них нет шансов конкурировать с сосредоточенным, упорным и ориентированным на принцип Парето программистом.

Фрактальная структура распределения Парето

Распределение Парето является фрактальным. Если вы посмотрите на его часть, увеличив масштаб, вы снова его увидите! Это работает до тех пор, пока данные не будут слишком разрежены; в этом случае фрактальная природа теряется. Одна точка, например, не может считаться распределением Парето. Давайте рассмотрим это свойство с помощью рис. 2.12.

В центре рис. 2.12 находится распределение Парето из рис. 2.1. Я использовал простой скрипт Python в листинге 2.1, чтобы увеличить масштаб этого распределения:

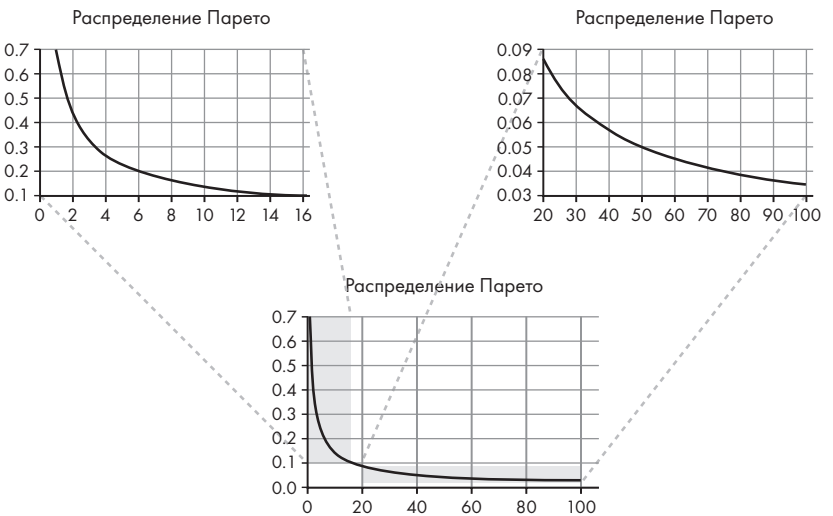


Рис. 2.12. Фрактальная природа распределения Парето

Листинг 2.1. Интерактивный скрипт для увеличения масштаба распределения Парето

```
import numpy as np
import matplotlib.pyplot as plt

alpha = 0.7

x = np.arange(100)
y = alpha * x / x**(alpha+1)

plt.plot(x, y)

plt.grid()
plt.title('Pareto Distribution')
plt.show()
```

Вы можете сами поиграть с этим кодом; просто скопируйте его в оболочку Python и запустите. Сделав это, вы сможете увеличивать масштаб различных областей распределения Парето.

Распределение Парето имеет множество практических применений как в жизни, так и в программировании, и я расскажу вам

о некоторых из них. Однако, по моему опыту, поворотным моментом для вас должно стать то, что вы научитесь *мыслить в парадигме 80/20*; то есть будете постоянно искать способы достичь гораздо большего со значительно меньшими затратами. Обратите внимание, что хотя конкретные числа Парето — 80/20, 70/30 или 90/10 — в вашей жизни зачастую меняются, вы можете извлечь определенную пользу из фрактальной природы распределения продуктивности и результатов. Например, всегда верно не только то, что несколько программистов зарабатывают намного больше других, но и то, что малая часть лучших из топ-группы получает больше остальных внутри нее. Эта закономерность исчезает только тогда, когда данных становится слишком мало. Вот несколько примеров:

- **Доход.** 20 % от 20 % лучших разработчиков кода зарабатывают 80 % от 80 % дохода. Другими словами, 4 % разработчиков зарабатывают 64 % дохода! Это означает, что вы никогда не застрянете в своем текущем финансовом положении, даже если вы уже принадлежите к 20 % лучших программистов. (Эта статья — http://journalarticle.ukm.my/12411/1/29%20Fatimah%20Abdul%20Razak.pdf — лишь одна из многих, демонстрирующих фрактальную природу распределения доходов.)
- **Деятельность.** 20 % от наиболее результативных 20 % от самых эффективных 20 % действий, которые вы совершили на этой неделе, часто отвечают за 80 % от 80 % от 80 % ваших результатов. В рамках такого сценария 0.8 % действий приведут к 51 % результатов. Грубо говоря, если вы работаете 40 часов в неделю, то 20 продуктивных минут могут обеспечить половину результатов вашей рабочей недели! Примером такой 20-минутной деятельности может быть написание скрипта, который автоматизирует некую бизнес-задачу и тем самым сэкономит вам пару часов за каждые несколько недель, которые вы сможете потратить на другие виды деятельности. Если вы программист, то решение пропустить реализацию

ненужной функции может сэкономить вам десятки часов бесполезного труда. Начав мыслить в парадигме 80/20, вы быстро обнаружите множество подобных случаев в своей работе.

- **Продвижение.** Независимо от того, в какой части распределения Парето вы находитесь, вы можете экспоненциально повысить свою результативность, «двигаясь влево», используя привычку добиваться успеха и преимущество фокуса. До тех пор пока оптимум не достигнут, всегда есть место для прогресса и достижения большего с меньшими затратами, даже если речь идет о достаточно продвинутом человеке, компании или экономике.

Действия, которые могут продвинуть вас вверх по кривой Парето, не всегда очевидны, но они никогда не бывают случайными. Многие люди отказываются от поиска метрик успеха в своей области, так как утверждают, что вероятностный характер результатов делает их совершенно случайными. Какой неверный вывод! Знайте, что написание меньшего количества кода в день не сделает вас специалистом в его разработке, а меньшее число ежедневных занятий шахматами не сделает вас гроссмейстером. Здесь важны другие факторы, но это не делает успех игрой случая. Сосредоточившись на метриках успеха в вашей сфере, вы сможете менять вероятности в свою пользу. Если вы мыслите по принципу 80/20, то вы вооружены и в большинстве случаев будете выигрывать.

Практические советы 80/20

Закончим эту главу девятью практическими советами по использованию принципа Парето.

Найдите свои метрики успеха

Сначала определите свою сферу деятельности. Выясните, какие задачи самые успешные профессионалы этой области

выполняют исключительно хорошо и что вы можете ежедневно делать, чтобы приблизиться к топ-20 %. Если вы разработчик, вашей метрикой успеха может быть число написанных строк кода. Если вы писатель — количество слов для следующей книги. Создайте сводную таблицу и ежедневно отслеживайте показатели успеха. Сделайте это игрой, чтобы постоянно придерживаться ее правил и в результате превзойти себя. Установите себе минимальный ежедневный порог и не заканчивайте день, пока его не достигнете.

Определите свои главные цели в жизни

Запишите их. Без четко поставленных долгосрочных целей (например, на 10 лет) вы не сможете придерживаться чего-то одного достаточно долго. Вы видели, что критическая стратегия продвижения по кривой Парето заключается в том, чтобы как можно дольше оставаться в игре, участвуя в меньшем количестве игр.

Постоянно ищите способы достичь тех же целей с меньшими затратами

Как добиться 80 % результатов за 20 % времени? Получится ли исключить прочие виды деятельности, которые занимают 80 % времени, но приносят только 20 % результата? Если нет, можете ли вы отдать их на аутсорсинг? Fiverr и Upwork — это дешевый способ найти талантливых специалистов, а использовать навыки других людей выгодно.

Подумайте о собственных успехах

Что вы сделали такого, что привело к отличным результатам? Каким образом можно делать это чаще?

Проанализируйте свои неудачи

Как делать меньше тех вещей, которые стали причиной неудачи?

Читайте больше книг по вашей тематике

Делая это, вы приобретаете практический опыт, не затрачивая много времени и сил на его реальное получение. Вы учитесь на чужих ошибках. Вы узнаете о новых способах решения задач. Вы приобретаете больше навыков в своей области. Опытный высокообразованный программист может справиться с проблемой в 10–100 раз быстрее, чем новичок. Чтение специализированных книг по вашей тематике, вероятно, и станет одной из тех метрик успеха, которые приведут вас к победе.

Проводите большую часть своего времени, дорабатывая и отлаживая существующие продукты

Делайте это вместо того, чтобы изобретать новое. Это опять же следует из распределения Парето. Если в вашем портфеле только один продукт, вы можете вложить всю свою энергию в его продвижение вверх по кривой Парето, добиваясь экспоненциально растущих положительных результатов для вас и вашей компании. Но если вы постоянно создаете новые продукты, не совершенствуя и не оптимизируя старые, то ваши приложения всегда будут ниже среднего. Никогда не забывайте: крупные достижения находятся в левой части распределения Парето.

Улыбайтесь

Удивительно, насколько просты некоторые следствия. Если вы позитивный человек, многие вещи станут проще. Больше людей будут сотрудничать с вами. Вы получите больше положительных эмоций, радости и поддержки. Улыбка — весьма эффективный инструмент с огромной отдачей и небольшими затратами.

Не делайте того, что понижает вашу эффективность

Это может быть, например, курение, нездоровое питание, недостаток сна, употребление алкоголя и слишком частый просмотр

Netflix. Отказ от того, что тянет вас вниз, — это один из ваших самых важных рычагов влияния. Если вы перестанете делать то, что вам вредит, вы будете полны сил, станете счастливее и успешнее. У вас появится больше времени и денег, чтобы наслаждаться приятными мелочами жизни: отношениями, природой и яркими впечатлениями.

В следующей главе вы познакомитесь с ключевой концепцией, которая поможет вам сосредоточиться на критически важном меньшинстве функций ПО: вы узнаете, как создать минимально жизнеспособный продукт.

Источники

Вот источники, использованные в этой главе; смело изучайте их дополнительно, чтобы найти больше примеров применения принципа Парето!

Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos, «Power Laws in Software», *ACM Transactions on Software Engineering and Methodology* 18, no. 1 (September 2008), https://doi.org/10.1145/1391984.1391986.

Научное доказательство того, что вклад в проекты с открытым исходным кодом подчиняется распределению Парето:

Mathieu Goeminne and Tom Mens, «Evidence for the Pareto Principle in Open Source Software Activity», Conference: CSMR 2011 Workshop on Software Quality and Maintainability (SQM), January 2011, https://www.researchgate.net/publication/228728263_Evidence_for_the_Pareto_principle_in_Open_Source_Software_Activity/.

Источник для построения распределения коммитов в репозитории TensorFlow на GitHub:

https://github.com/tensorflow/tensorflow/graphs/contributors/.

Статья в моем блоге о распределении доходов разработчиков-фрилансеров:

> Christian Mayer, «What's the Hourly Rate of a Python Freelancer?» *Finxter* (blog), https://blog.finxter.com/whats-the-hourly-rate-of-a-python-freelancer/.

Научное доказательство того, что открытые рынки подчиняются принципу Парето:

> William J. Reed, «The Pareto Law of Incomes — an Explanation and an Extension», *Physica A: Statistical Mechanics and its Applications* 319 (March 2003), https://doi.org/10.1016/S0378-4371(02)01507-8.

Статья, демонстрирующая фрактальную природу распределения доходов:

> Fatimah Abdul Razak and Faridatulazna Ahmad Shahabuddin, «Malaysian Household Income Distribution: A Fractal Point of View», *Sains Malaysianna* 47, no. 9 (2018), http://dx.doi.org/10.17576/jsm-2018-4709-29.

Информация о том, как дополнительно подработать в качестве разработчика-фрилансера на Python:

> Christian Mayer, «How to Build Your High-Income Skill Python». Video, https://blog.finxter.com/webinar-freelancer/.

> Python Freelancer resource page, *Finxter* (blog), https://blog.finxter.com/python-freelancing/.

Глубокое погружение в концепцию мышления 80/20:

> Richard Koch, «The 80/20 Principle: The Secret to Achieving More with Less», London: Nicholas Brealey, 1997.

Десять процентов людей в Соединенных Штатах получают почти 50 % общего дохода:

Facundo Alvaredo, Lucas Chancel, Thomas Piketty, Emmanuel Saez, and Gabriel Zucman, *World Inequality Report 2018*, World Inequality Lab, https://wir2018.wid.world/files/download/wir2018-summary-english.pdf.

Менее 25 % жителей Северной Америки оценивают свое благополучие на 9 или 10 баллов по шкале от 0 до 10, где «наименее благополучная жизнь» — 0, а «наиболее благополучная жизнь» — 10:

John Helliwell, Richard Layard, and Jeffrey Sachs, eds., *World Happiness Report 2016, Update* (Vol. 1). New York: Sustainable Development Solutions Network, https://worldhappiness.report/ed/2016/.

Двадцать процентов авторов книг могут обеспечить 97 % продаж:

Xindi Wang, Burcu Yucesoy, Onur Varol, Tina Eliassi-Rad, and Albert-László Barabási, «Success in Books: Predicting Book Sales Before Publication», EPJ Data Sci. 8, no. 31 (October 2019), https://doi.org/10.1140/epjds/s13688-019-0208-6.

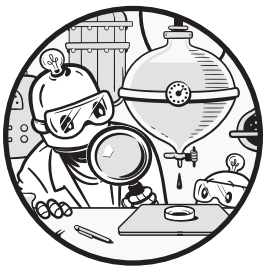
Jordi Prats, «Harry Potter and Pareto's Fat Tail», *Significance* (August 10, 2011), https://www.significancemagazine.com/14-the-statistics-dictionary/105-harry-potter-and-pareto-s-fat-tail/.

На долю 5.2 % ученых приходится 38 % всех опубликованных статей:

Javier Ruiz-Castillo and Rodrigo Costas, «Individual and Field Citation Distributions in 29 Broad Scientific Fields», *Journal of Informetrics* 12, no. 3 (August 2018), https://doi.org/10.1016/j.joi.2018.07.002.

3

Создание минимально жизнеспособного продукта



В этой главе рассматривается известная, но сильно недооцененная идея, популяризированная в книге Эрика Риса (Eric Ries) «The Lean Startup»[1] (Crown Business, 2011). Идея в том, чтобы вначале создать *минимально жизнеспособный продукт (minimum viable product, MVP)*, который представляет собой промежуточную версию конечного продукта без излишнего функционала. Сюда входят только самые необходимые функции: нужно быстро протестировать и подтвердить ваши предположения, не теряя много времени на внедрение того, что пользователям в итоге ни разу не пригодится. В частности, в этой главе вы узнаете, как существенно понизить сложность цикла разработки ПО, сосредоточившись на

[1] Рис Э. «Бизнес с нуля. Метод Lean Startup для быстрого тестирования идей и выбора бизнес-модели».

функциях, которые наверняка необходимы пользователям, ведь вы заведомо убедились в этом еще на этапе создания MVP.

Мы познакомимся с созданием MVP и обсудим все подводные камни при разработке программ без него. Затем мы более подробно остановимся на этой концепции и дадим несколько практических советов о том, как использовать MVP в ваших проектах с целью ускорения процессов.

Проблемный сценарий

Идея создания MVP состоит в том, чтобы бороться с проблемами, возникающими при разработке приложения в скрытом режиме (рис. 3.1). *Скрытый режим* — это когда вы работаете над проектом, не получая обратной связи от потенциальных пользователей до самого его завершения. Допустим, вам пришла в голову замечательная идея программы, которая изменит весь мир: продвинутая поисковая система на основе машинного обучения, специально предназначенная для поиска кода. Вы несколько ночей подряд с энтузиазмом работаете над проектом.

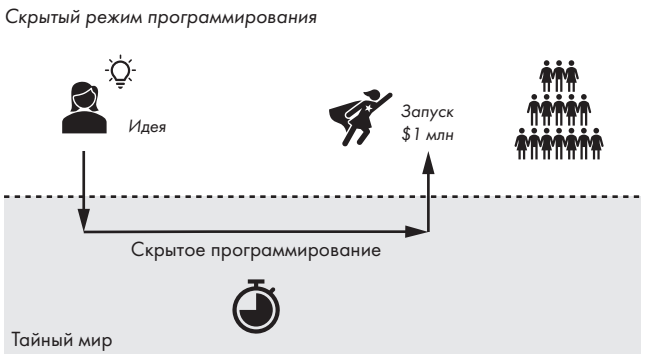


Рис. 3.1. Скрытый режим состоит в том, что приложение держится в секрете, пока не будет выпущена финальная, совершенная версия в надежде на быстрый успех. В большинстве случаев ожидания не оправдываются

Однако на практике программирование приложения на одном дыхании приводит к немедленному успеху очень, очень, очень редко. Вот более вероятный исход скрытого режима.

Вы быстро разрабатываете прототип, но когда пробуете применить свою систему, то обнаруживаете, что многие поисковые запросы в рекомендуемых результатах нерелевантны. При поиске `Quicksort` вы получаете фрагмент кода `MergeSort` с комментарием `# Это не Quicksort`. Что-то здесь не так. Итак, вы продолжаете отлаживать параметры модели, но каждый раз, улучшая ситуацию с одним ключевым словом, вы создаете новые проблемы для других. Результаты поиска вас совершенно не радуют, и вам кажется, что вы никогда не сможете представить миру эту дрянную систему по трем причинам: никто не сочтет ее полезной; первые пользователи создадут негативную рекламу вашему сайту, из-за того что он покажется им непрофессиональным и недоработанным; а если конкуренты увидят плохо реализованную концепцию, они украдут ее и реализуют лучше. Эти мрачные мысли заставляют вас терять уверенность в себе и всякую мотивацию, и ваш прогресс в создании приложения падает до нуля.

На рис. 3.2 показано, что может пойти не так при работе в скрытом режиме программирования.

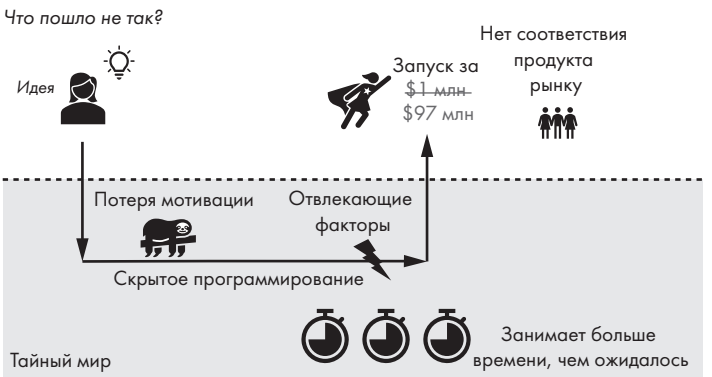


Рис. 3.2. Типичные подводные камни скрытого режима программирования

Здесь я расскажу о шести наиболее распространенных подводных камнях при работе в скрытом режиме.

Потеря мотивации

В скрытом режиме вы остаетесь наедине со своей идеей, и сомнения появляются регулярно. Вы сопротивляетесь им, пока ваш первоначальный энтузиазм по поводу проекта еще не остыл, но чем дольше вы над ним работаете, тем больше растет ваша неуверенность. Возможно, вы наткнетесь на уже существующее похожее программное средство или убедите себя в том, что это вообще невозможно реализовать. Потеря мотивации может полностью погубить ваш проект.

С другой стороны, если вы выпустите «сырую» версию продукта, ободряющие слова от одного из тех, кто впервые ее использовал, могут поддержать вашу мотивацию, а отзывы пользователей — вдохновить вас на улучшение продукта или устранение проблем. В результате вы приобретаете внешнюю мотивацию.

Рассеянность внимания

Когда вы работаете в одиночку в скрытом режиме, трудно игнорировать ежедневные отвлекающие факторы. Вы работаете на основной работе, проводите время с семьей и друзьями, в голову приходят разные идеи. В нынешнее время ваше внимание — это огромная ценность, за которую борются многие устройства и сервисы. Чем дольше вы находитесь в скрытом режиме, тем выше вероятность того, что вас отвлекут, прежде чем вы закончите работу над своим полностью отточенным приложением.

MVP может в этом помочь, сокращая время от идеи до ответной реакции рынка, создавая условия для получения более оперативной обратной связи, которая поможет снова сконцентрировать ваше внимание. И кто знает, быть может, вы найдете несколько энергичных последователей вашего MVP, которые помогут ускорить разработку приложения.

Нарушение сроков

Еще один враг, препятствующий завершению проекта, — неправильное планирование. Допустим, вы предполагаете, что на создание продукта потребуется 60 часов, поэтому изначально планируете работать над ним по 2 часа каждый день в течение месяца. Однако потеря мотивации и отвлекающие факторы приводят к тому, что в среднем вы проводите за работой всего 1 час в день. Дальнейшие задержки могут быть связаны с необходимостью проведения дополнительных исследований, внешними раздражителями, а также с непредвиденными обстоятельствами и багами, которые нужно устранить. Бесконечное число факторов будет увеличивать предполагаемую продолжительность проекта, и лишь немногие из них могут ее уменьшить. К концу первого месяца вы окажетесь слишком далеки от того, на что рассчитывали, и снова попадете в петлю потери мотивации.

Поскольку MVP лишен всех ненужных функций, ошибок в планировании будет меньше, и рабочий процесс станет более предсказуемым. Меньше реализуемых функций — меньше шансов, что возникнет проблема. Кроме того, чем более предсказуем будет ваш проект, тем сильнее вы или люди, инвестирующие в него, будете верить в его успех. Инвесторы и другие заинтересованные лица ценят предсказуемость!

Отсутствие обратной связи

Допустим, вы побороли низкую мотивацию и завершили работу над приложением. Вы наконец выпускаете его на рынок, но ничего не происходит. Только небольшая группа пользователей попробовала его, но и они не в восторге. Наиболее вероятным итогом запуска любого программного продукта является молчание — отсутствие положительных или отрицательных отзывов. Чаще всего причина в том, что он не отвечает запросам пользователей. Почти невозможно с первого раза найти так называемое *соответствие продукта рынку (product-market fit)*. Если в процессе разработки вы не получаете

никакой обратной связи из реального мира, вы отрываетесь от действительности, работая над функционалом, который никогда не будет востребован.

MVP поможет вам гораздо быстрее найти решение проблемы соответствия продукта запросам целевой аудитории, потому что, как вы убедитесь далее, подход, основанный на MVP, позволяет выявить самые насущные потребности пользователей прямо во время разработки. Это увеличивает ваши шансы вовлечь в процесс реальных потребителей и, следовательно, получить от них отклик на ранние версии продукта.

Ложные предположения

Основная причина неудач в скрытом режиме — это ваши собственные догадки. Вы начинаете проект с кучей предположений, например, о том, кто ваши пользователи, чем они зарабатывают на жизнь, с какими проблемами сталкиваются или как часто они будут использовать ваше приложение. Эти допущения часто ошибочны, и без внешнего тестирования вы продолжаете вслепую создавать программные продукты, которые не нужны вашей настоящей целевой аудитории. Если вы не получаете обратной связи или получаете отрицательные отзывы, страдает ваша мотивация.

Когда я создавал свое приложение Finxter.com для изучения языка Python путем решения кодовых головоломок с определенным рейтингом, я предполагал, что большинство пользователей будут студентами факультета computer science, потому что я сам был одним из них (на самом деле большинство пользователей, как оказалось, ими не являются). Я думал, что пользователи придут, как только я выложу приложение (в реальности изначально никто не пришел). Я предполагал, что многие будут делиться своими успехами на Finxter в социальных сетях (но лишь незначительное количество пользователей поделились своими успехами в знании кода). Я считал, что пользователи будут присылать собственные кодовые головоломки (однако из сотен тысяч лишь немногие прислали задачки

с кодом). Я предполагал, что людям нужен причудливый цветной дизайн с картинками (на самом деле всем больше понравилось простое забавное оформление — см. главу 8 о минимализме в дизайне). Все эти догадки привели к конкретным решениям, которые стоили мне десятков, если не сотен, часов на реализацию многих функций, которые были не нужны целевой аудитории. Если бы я знал, я бы сначала проверил эти предположения на MVP, прислушался к отзывам пользователей, сэкономил время и ресурсы и снизил вероятность провала проекта.

Излишняя сложность

Существует еще одна проблема, связанная со скрытым режимом программирования: *излишняя сложность*. Допустим, вы создали программный продукт, который включает в себя четыре функции (рис. 3.3). Вам повезло — рынок его одобрил. Вы потратили много времени на реализацию и теперь принимаете положительные отзывы, подтверждающие правильность вашего выбора. Теперь все новые релизы программного продукта будут обязательно содержать эти четыре функции в дополнение к тем, которые вы добавите в будущем.

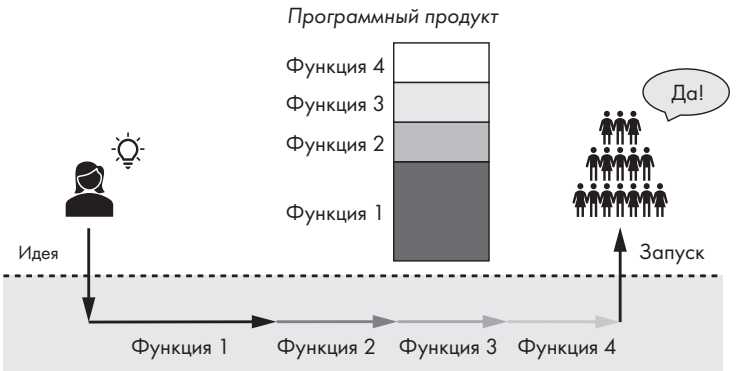


Рис. 3.3. Полезный программный продукт, состоящий из четырех функций

Однако, выпуская целый пакет из четырех функций разом, а не по одной или двум, вы не знаете, принял бы рынок сокращенный набор функций (или хотя бы проявил к нему интерес) (рис. 3.4).

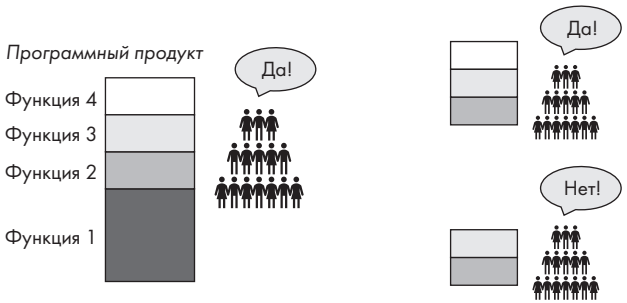


Рис. 3.4. Какие наборы функций были бы приняты рынком?

Функция 1 может быть совершенно неактуальной, даже несмотря на то что на ее реализацию у вас ушло больше всего времени. В то же время функция 4 может оказаться очень полезной и востребованной на рынке. Существует 2^n различных комбинаций пакетов программных продуктов из n функций. Если вы выпускаете их в виде пакета программ, как вы узнаете, какая из них важна, а какая — пустая трата времени?

Затраты на реализацию ненужных функций и так высоки, а выпуск наборов ненужных функций влечет за собой еще и совокупные расходы на их поддержку:

- Большие проекты, перегруженные функциональностью, отнимают больше времени на «загрузку» всего проекта в ваш мозг.
- Каждая функция рискует привнести новые баги.
- Каждая строка кода увеличивает затраты времени на открытие, загрузку и компиляцию проекта.
- Реализация функции n требует проверки всех предыдущих функций: 1, 2, ..., $n - 1$; нужно убедиться, что она не нарушает их работу.

- Каждая дополнительная функция требует новых юнит-тестов, которые должны быть скомпилированы и запущены, прежде чем вы сможете выпустить следующую версию программы.
- Каждая новая функция делает кодовую базу все сложнее для понимания ее программистом, что увеличивает время обучения новых специалистов, приходящих в проект.

Это далеко не полный список, но суть вы поняли. Если каждая функция увеличивает ваши будущие затраты на реализацию на *x* процентов, то сохранение ненужных функций может привести к разнице на порядки в продуктивности написания кода. Вы не можете позволить себе систематически сохранять лишние функции в своих проектах!

Итак, вы спросите: если скрытый режим программирования вряд ли увенчается успехом, то каково же решение?

Создание MVP

Решение простое: создайте серию MVP. Точно сформулируйте предположение (например: *пользователям нравится решать Python-головоломки*) и создайте продукт, который подтвердит лишь эту гипотезу. Удалите все лишнее, создайте MVP для осуществления только этой идеи. Реализуя функции по одной в каждом релизе, вы будете лучше понимать, что из этого принимается рынком и какие из ваших предположений верны. Но любой ценой избегайте сложности. В конце концов, если пользователям не нравится решать головоломки на языке Python, зачем вообще приступать к созданию сайта Finxter.com? После того как вы протестировали свой MVP на реальном рынке и проанализировали, сработал ли он, вы можете создать второй MVP, в который будет добавлена следующая по важности функция. Термин, описывающий стратегию поиска приложения с оптимальной функциональностью при помощи разработки серии MVP, называется *быстрым прототипированием*. Каждый следующий прототип основан на опыте предыдущих релизов, и все они разработаны для того, чтобы дать вам максимум знаний за

минимальное время и с наименьшими усилиями с вашей стороны. Вы *выпускаете продукт «сырым» и довольно часто* только для того, чтобы он в результате *наилучшим образом соответствовал рынку*, то есть определяете потребность в продукте и его соответствие вашей целевой аудитории (даже если поначалу она не очень велика).

Рассмотрим пример с поисковой системой для кода. Сначала вы формулируете гипотезу для проверки: программистам нужен удобный способ поиска кода. Подумайте, какую форму может принять первый MVP вашего приложения. API для командной строки? Бэкенд-сервер, выполняющий поиск в базе данных по всем проектам GitHub с открытым исходным кодом в поисках точных совпадений слов? Первый MVP должен подтвердить основные предположения. Итак, вы решаете, что самый простой способ проверить вашу гипотезу и получить некоторое представление о возможных запросах — это создать пользовательский интерфейс без сложного бэкенд-функционала, который автоматически обрабатывает результаты по запросу. Вы запускаете сайт с полем ввода и обеспечиваете небольшой трафик, делясь своей идеей в группах разработчиков кода и социальных сетях, а также потратив небольшую сумму на рекламу. Интерфейс приложения прост: пользователи вводят код, который хотят найти, и нажимают кнопку поиска. Вы не слишком утруждаете себя оптимизацией результатов выдачи — не в этом смысл вашего первого MVP. Вместо этого вы просто передаете результаты поиска Google после небольшой постобработки. Смысл в том, чтобы собрать первые, скажем, 100 поисковых запросов, которые позволят вам нащупать некоторые общие модели поведения пользователей еще до того, как вы начнете разрабатывать систему.

Вы анализируете данные и обнаруживаете, что 90 % запросов связаны с сообщениями об ошибках; программисты просто копируют и вставляют неправильные строки своего кода в поле поиска. Более того, вы обнаруживаете, что 60 из 90 запросов касаются JavaScript. Вы приходите к выводу, что ваша первоначальная гипотеза подтвердилась: люди действительно ищут куски кода. Однако дополнительно вы получаете ценную информацию, что большинство ищет ошибки

кода, а не, скажем, функции. На основе анализа вы решаете снизить функциональность второго MVP с универсального механизма для поиска кода до системы поиска *ошибок*. Таким образом, вы адаптируете продукт под реальные потребности пользователей и получаете более активную обратную связь от части программистов, чтобы быстро учиться и интегрировать полученные знания в полезный продукт. Со временем вы сможете расширять проект, добавляя другие языки и типы запросов по мере получения все большей информации о рынке. Без первого MVP вы могли бы потратить месяцы на разработку возможностей, которые почти никто не использует, например на реализацию регулярных выражений для поиска произвольных шаблонов в коде за счет функций, которые действительно всем нужны, типа поиска по сообщениям об ошибках.

На рис. 3.5 показан золотой стандарт разработки ПО и создания продукта. Сначала вы находите соответствие продукта рынку путем многократных запусков MVP, до тех пор пока пользователям не понравится ваше предложение. Последовательные релизы MVP со временем вызывают интерес и позволяют вам учесть отзывы пользователей, что мало-помалу корректирует основную идею вашего ПО. Как только вы достигаете полного соответствия продукта рынку, вы начинаете по одной добавлять новые функции. И только в том случае, если функция своим присутствием улучшает ключевые пользовательские показатели, она остается в конечном продукте.

Рассматривая Finxter.com в качестве примера, отмечу, что, если бы я с самого начала следовал правилу MVP, я бы, вероятно, в первую очередь завел простой аккаунт в Instagram, где делился бы кодовыми головоломками и проверил, нравится ли пользователям их решать. Вместо потери года на написание приложения Finxter без такой проверки я мог бы потратить несколько недель или даже месяцев на постинг задачек в социальной сети. Затем, используя информацию, полученную от сообщества, я мог бы создать второй MVP с немного расширенной функциональностью, например разработать специальный веб-сайт, на котором размещены кодовые головоломки и их решения. Такой метод позволил бы мне создать приложение Finxter

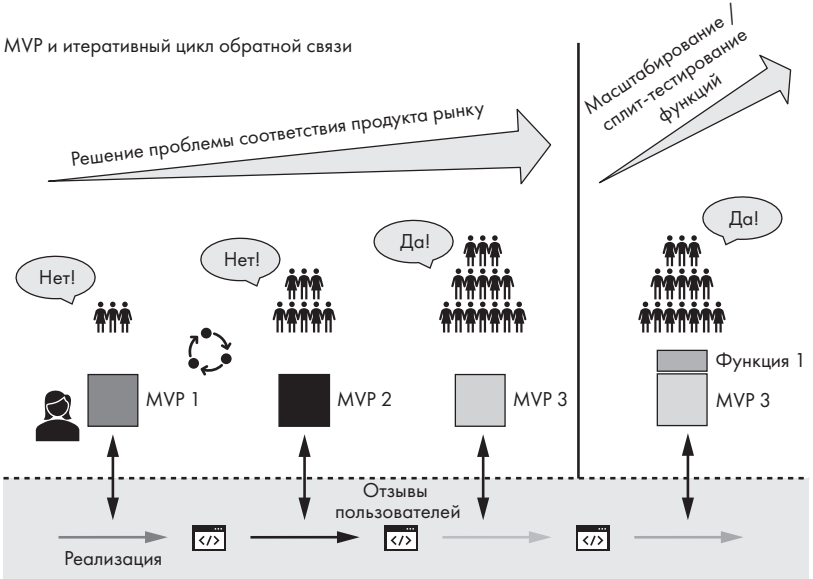


Рис. 3.5. Два этапа разработки ПО включают: (1) решение проблемы соответствия продукта рынку путем итеративного процесса создания MVP, который со временем начинает вызывать интерес; (2) масштабирование путем добавления новых функций и проверки их с помощью тщательно разработанных сплит-тестов

в кратчайшие сроки и с меньшим количеством необязательных функций. Урок создания MVP, лишенного всего ненужного, я усвоил на собственном горьком опыте.

В книге «The Lean Startup» Эрик Рис рассказывает о том, как компания Dropbox, стоимость которой на сегодняшний день составляет более миллиарда долларов, внедрила подход MVP. Dropbox не стала тратить время и силы на воплощение непроверенной и сложной функциональности (синхронизации структуры папок в облаке, которая требовала тесной интеграции с различными операционными системами и тщательной и затратной реализации концепций распределенных систем, таких как синхронизация реплик). Вместо этого основатели компании подтвердили свою гипотезу с помощью

простого видеоролика о продукте (хотя его самого еще не существовало). За одобрением MVP для приложения Dropbox последовало бесчисленное количество итераций по добавлению в основной проект новых полезных функций, упрощающих жизнь пользователей. С тех пор эта концепция была опробована тысячами успешных компаний в области программного обеспечения (и не только).

Если рынок показывает, что пользователям понравилась идея вашего продукта и они оценили ее, значит, вы достигли соответствия продукта рынку с помощью простого, хорошо проработанного MVP. После этого можно итеративно создавать и совершенствовать MVP.

Когда вы используете MVP-подход при разработке ПО, добавляя каждый раз по одной функции, очень важно уметь определять, что оставить, а от чего отказаться. Заключительным этапом процесса создания продукта в рамках MVP-подхода является *сплит-тестирование*: вместо того чтобы выпускать итерации с новыми функциями для всей аудитории, вы предоставляете новый продукт только части пользователей и наблюдаете за скрытой и явной реакцией. Только если вам нравится результат — например, среднее время, проведенное на вашем сайте, увеличивается, — вы сохраняете функцию. В противном случае вы отказываетесь от нее и продолжаете работать с предыдущей итерацией без нее. Это означает, что вам придется пожертвовать временем и силами, потраченными на ее разработку, но это поможет сохранить ваш продукт максимально простым и при этом гибким, динамичным и эффективным. Используя сплит-тесты, вы осуществляете разработку ПО на основе полученных данных.

Четыре кита в создании MVP

При создании своего первого ПО в концепции MVP примите во внимание четыре основных принципа:

Функциональность. Продукт должен предоставлять пользователю четко определенную функцию и хорошо ее выполнять.

Функция не обязана обладать высокой экономической эффективностью. Например, вашим MVP для чат-бота может быть просто общение с пользователями; масштабировать в реальный продукт его никак нельзя, но вы демонстрируете функциональность качественного чата, даже если вы еще не придумали, как реализовать ее экономически выгодным образом.

Дизайн. Продукт должен быть хорошо продуманным и ориентированным на определенную задачу, а его дизайн должен соответствовать тем полезным качествам, которые ваш продукт предлагает целевой аудитории. Одна из распространенных ошибок при создании MVP заключается в том, что вы создаете интерфейс, который недостаточно точно отражает тестируемую функцию MVP. Дизайн может быть простым, но он должен соответствовать функционалу вашего приложения. Вспомните Google Search — разработчики, конечно, не тратили много сил на дизайн при выпуске своей первой версии поисковой системы, но дизайн хорошо подходил для предлагаемого ими продукта: поиск без отвлечения внимания.

Надежность. Тот факт, что ваш продукт минималистичен, еще не означает, что он ненадежен. Обязательно напишите контрольные тесты и тщательно протестируйте все функции в коде. В противном случае отзывы пользователей после релиза MVP будут содержать не только отклик на конкретные возможности приложения, но и негативное впечатление от их некорректной работы. Помните: вы хотите получить максимум полезных данных при минимальных затратах.

Юзабилити. Необходимо, чтобы MVP был прост в использовании. Функциональность должна быть четко определена, и дизайн должен это подчеркивать. Пользователь не обязан тратить много времени на выяснение того, что делать или какие кнопки нажимать. MVP должен достаточно быстро реагировать на внешние запросы, чтобы обеспечивать высокую интерактивность. Часто этого проще добиться с помощью сфокусированного, минималистичного дизайна продукта: любому очевидно, как

использовать страницу с одной кнопкой и одним полем ввода. Опять же ярким примером является первоначальный прототип поисковой системы Google, который изначально был настолько удобен, что прослужил более двух десятилетий.

Многие люди неправильно понимают особенность MVP: они ошибочно полагают, что, поскольку это крайне минималистичная версия продукта, он мало полезен, имеет плохое юзабилити и примитивный дизайн. Однако истинный приверженец минимализма знает, что лаконичность MVP на самом деле является результатом его строгой ориентированности на одну основную функцию, а не небрежности при его разработке. Для Dropbox создать эффективный видеоролик, демонстрирующий намерения, было проще, чем реализовывать сам сервис. Этот MVP показал высококачественный продукт с отличной функциональностью, дизайном, высокой надежностью и юзабилити.

Преимущества MVP-подхода

Разработка программного обеспечения на основе MVP имеет много преимуществ.

- Вы можете проверить свои гипотезы самым дешевым способом.
- Часто вы можете даже не писать код, пока не убедитесь в его необходимости, а когда вы все-таки приступите к нему, то сможете минимизировать объем работы до получения реальной обратной связи.
- Вы тратите гораздо меньше времени на написание кода и поиск багов, зная при этом, что основное время, которое вы тратите, посвящено важным для пользователей функциям.
- Любая новая возможность, которую вы предоставляете пользователям, дает вам мгновенную обратную связь, а постоянный прогресс мотивирует вас и вашу команду выдавать одну функцию за другой. Это значительно минимизирует риски,

которым вы подвергались бы в скрытом режиме программирования.

- Вы сокращаете затраты на сопровождение продукта в будущем, поскольку MVP-подход значительно снижает сложность вашей кодовой базы, следовательно, все последующие функции будут проще в реализации и менее подвержены ошибкам.
- Вы быстрее продвигаетесь вперед, а процесс реализации упрощается на всем протяжении работы над ПО, что поддерживает вас в мотивированном состоянии и ведет к успеху.
- Вы активнее поставляете новые продукты на рынок, быстрее зарабатываете деньги на своем ПО и создаете свой бренд более предсказуемо и последовательно.

Скрытое программирование в сравнении с MVP-подходом

Распространенный аргумент против быстрого прототипирования и *в пользу* скрытого режима заключается в том, что во втором случае ваши идеи защищены. Люди почему-то считают, что их идея настолько особенная и уникальная, что, если они выпустят ее в «сыром» виде (то есть в качестве MVP), ее украдут более крупные и влиятельные компании, которые смогут ее быстрее реализовать. Честно говоря, это заблуждение. Идеи стоят дешево, важно исполнение. Маловероятно, что ваша идея уникальна, скорее всего, она уже была выдвинута кем-то другим. Вместо того чтобы снижать конкуренцию, скрытый режим программирования может даже подтолкнуть других к работе над тем же проектом, поскольку, как и вы, они предполагают, что никто другой до этого еще не додумался. Чтобы идея увенчалась успехом, нужен человек, который воплотит ее в жизнь. Если перенестись на несколько лет вперед, то успешным будет тот, кто предпринял активные и решительные действия, выпускал быстрые и частые релизы, учитывал отзывы реальных пользователей и постепенно улучшал свое программное обеспечение, опираясь на опыт предыдущих версий. Сохранение идеи в секрете просто ограничивает потенциал ее развития.

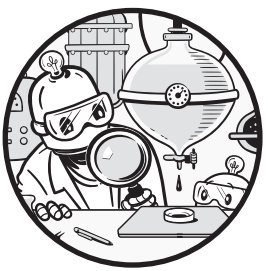
Заключение

Прежде чем писать код, представьте в воображении конечный продукт и подумайте о потребностях вашей аудитории. Работайте над MVP и сделайте его полезным, хорошо продуманным, быстро реагирующим на действия пользователей и удобным. Исключите все функции, кроме тех, которые абсолютно необходимы для достижения цели. Сосредоточьтесь на чем-то одном. А затем быстро и часто выпускайте MVP — улучшайте их со временем, постепенно тестируя и добавляя новые функции. Лучше меньше, да лучше! Тратьте больше времени на обдумывание следующей возможности, которую нужно внедрить, чем на фактическую ее реализацию. Каждое новшество влечет за собой не только прямые, но и косвенные затраты на создание всех последующих функций. Используйте сплит-тестирование для проверки реакции на два варианта продукта одновременно и сразу отбрасывайте функции, которые не улучшают ключевые пользовательские метрики (удержание внимания, время пребывания на странице или активность). Это приведет вас к более целостному подходу к бизнесу — осознанию того, что разработка ПО является лишь одним из этапов всего процесса создания продукта и его стоимости.

В следующей главе вы узнаете, почему необходимо и как правильно писать чистый и простой код, но запомните прямо сейчас: самый верный путь к этому — отказ от написания ненужного кода!

4

Написание чистого и простого кода



Чистый код — это код, который легко читать, понимать и редактировать. Он должен быть минималистичен и лаконичен до той степени, пока эти свойства не мешают легкости его прочтения. Хотя написание чистого кода — это скорее искусство, чем наука, в сфере разработки программного обеспечения выработано несколько принципов, соблюдение которых поможет вам писать более чистый код. В этой главе вы узнаете 17 принципов написания чистого кода, которые значительно повысят вашу продуктивность и помогут справиться с проблемой сложности.

Вы можете задаться вопросом, в чем разница между *чистым* и *простым* кодом. Эти два понятия тесно связаны, поскольку чистый код, как правило, прост, а простой код, как правило, чист. Однако можно встретить и сложный код, который при этом остается чистым. Простота заключается в том, чтобы избежать сложности. Чистый

код идет на шаг дальше и включает в себя управление неизбежной сложностью — например, с помощью эффективного использования комментариев и стандартов.

Зачем писать чистый код?

Из предыдущих глав вы узнали, что сложность — это враг номер один для любого проекта и что простота повышает вашу продуктивность, мотивацию и удобство сопровождения кодовой базы. В этой главе мы продолжим эту мысль и продвинемся еще на один шаг, показав, как нужно писать чистый код.

Такой код будет легче понять как вам самому, так и вашим коллегам-программистам, и поскольку люди с большей вероятностью вносят дополнения в чистый код, то потенциал для сотрудничества увеличится. Следовательно, чистый код может значительно снизить стоимость проекта. Как отмечает Роберт К. Мартин (Robert C. Martin) в своей книге «Clean Code»[1] (Prentice Hall, 2008), разработчики тратят львиную долю своего времени на чтение старого кода для того, чтобы написать новый. Если старый код легко читается, это значительно ускоряет процесс. Мартин говорит, что соотношение времени, затрачиваемого на чтение и написание кода, значительно превышает 10 к 1. Мы постоянно читаем старый код, пытаясь написать новый. Поэтому облегчение чтения упрощает и написание.

Если воспринимать такое соотношение буквально, то эту зависимость можно проиллюстрировать рис. 4.1. Ось *X* соответствует количеству строк кода, написанных в рамках проекта. Ось *Y* показывает время написания одной дополнительной строки. В общем, чем больше кода вы уже написали в одном проекте, тем дольше вы пишете новую строку. Это справедливо как для чистого, так и для грязного кода.

[1] Мартин Р. «Чистый код: создание, анализ и рефакторинг». Санкт-Петербург, издательство «Питер».

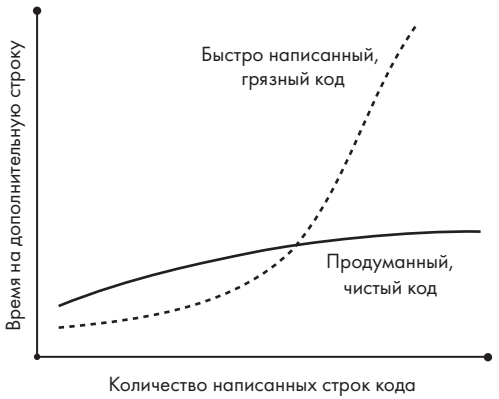


Рис. 4.1. Чистый код улучшает масштабируемость и удобство сопровождения кодовой базы

Допустим, вы написали n строк кода и добавляете $n + 1$ строку. Потенциально это может повлиять на все ранее написанные строки. Например, привести к небольшому снижению производительности, что окажет воздействие на весь проект в целом. В строке может использоваться переменная, определенная в другом месте. Это может внести баг (с вероятностью c), и, чтобы найти его, вам придется выполнить поиск по всему проекту. А значит, ожидаемое время и, следовательно, затраты на одну строку кода составляют $c \times T(n)$ для функции времени T, постоянно возрастающей с увеличением количества строк n. Добавление одной строки может также потребовать написания дополнительных строк кода для обеспечения обратной совместимости.

Более длинный код может вызвать и множество других сложностей, но вы поняли суть: чем больше кода вы пишете, тем больше дополнительная сложность замедляет прогресс.

На рис. 4.1 также продемонстрирована разница между написанием грязного и чистого кода. Грязный код требует меньшего времени в краткосрочной перспективе и может сгодиться для небольших проектов — если бы совсем не было преимуществ в написании такого кода, никто бы этим не занимался! Если вы впихнули всю функциональность в скрипт, состоящий из 100 строк, вам не нужно тратить

много времени на обдумывание и реструктуризацию проекта. Проблемы возникают только по мере разрастания кода: когда ваш не разделенный на блоки кодовый файл увеличится со 100 до 1000 строк, он станет менее эффективным, чем код, написанный в рамках более продуманного подхода, при котором вы логически структурируете код в различных модулях, классах или файлах.

Как показывает опыт, всегда следует писать продуманный и чистый код. Дополнительные затраты на переосмысление, рефакторинг и реструктуризацию многократно окупятся для любого нетривиального проекта. Иногда ставки достаточно высоки. Например, в 1962 году NASA[1] попыталось отправить космический корабль на Венеру, но крошечный баг — пропуск дефиса в исходном коде — привел к запуску команды на самоуничтожение, в результате была потеряна ракета стоимостью на тот момент более $18 млн. Если бы код был чище, инженеры, возможно, заметили бы ошибку еще до запуска.

Независимо от того, занимаетесь ли вы ракетостроением или нет, стратегия тщательного профессионального подхода к программированию поможет вам достигнуть в жизни большего. Простой код также облегчает масштабирование проекта на большее количество программистов и большее число функций, ведь его невысокая сложность отпугнет меньше специалистов.

Итак, не научиться ли нам писать чистый и простой код?

Написание чистого кода: основные принципы

Я учился писать чистый код трудным путем, когда разрабатывал с нуля распределенную систему обработки графов в рамках своей кандидатской диссертации. Если вы хоть раз создавали распределенное

[1] NASA (National Aeronautics and Space Administration) — Национальное управление по аэронавтике и исследованию космического пространства (США). — *Примеч. ред.*

приложение, где два процесса, расположенные на разных компьютерах, взаимодействуют друг с другом посредством сообщений, вы понимаете, что сложность может быстро перейти все границы. Мой код разросся до тысяч строк, и в нем часто появлялись баги. Я не продвигался вперед по нескольку недель подряд; это было очень неприятно. Концепции были убедительны в теории, но почему-то на практике они не работали.

Наконец, приблизительно после месяца ежедневной работы без какого-либо обнадеживающего прогресса я решил радикально упростить кодовую базу. Кроме того, я задействовал библиотеки вместо того, чтобы реализовывать функциональность самостоятельно. Я удалил блоки кода, которые закомментировал для возможного последующего использования, переименовал функции и переменные. Я структурировал код в логические единицы и создал новые классы вместо того, чтобы запихивать все в один класс «Бог». Примерно через неделю мой код стал не только более читабельным и понятным для других, но и более эффективным и с меньшим количеством ошибок. Мое разочарование превратилось в энтузиазм — чистый код спас мой исследовательский проект!

Совершенствование кодовой базы и снижение сложности называется *рефакторингом*. Он должен стать регулярным и важным элементом процесса разработки ПО, если вы хотите писать чистый и простой код. Для создания чистого кода необходимы две вещи: знание лучших способов создания кода с нуля и периодическое возвращение к началу для внесения изменений. Я сформулировал некоторые важные приемы поддержки чистоты кода в 17 принципах. Хотя каждый из них содержит уникальную стратегию написания более чистого кода, некоторые из них пересекаются. Но мне показалось, что объединение смежных принципов может затруднить их понимание и практическое применение. Итак, давайте приступим!

Принцип 1. Представляйте общую картину

Если вы работаете над нетривиальным проектом, то, скорее всего, в итоге у вас получится множество файлов, модулей и библиотек,

работающих вместе в рамках итогового приложения. Ваша *программная архитектура* определяет, как взаимодействуют его элементы. Правильные архитектурные решения способствуют улучшению производительности, легкости сопровождения и юзабилити. Чтобы создать хорошую архитектуру, вам нужно сделать шаг назад и задуматься об общей картине. Решите, какие функции нужны в первую очередь. В главе 3 (про создание MVP) вы узнали, как сфокусироваться на абсолютно необходимых функциях проекта. Если вы сделаете это, вы сэкономите много времени, а код будет гораздо чище с точки зрения структуры. На данный момент предполагается, что вы уже создали свое первое приложение с большим количеством модулей, файлов и классов. Как применить «взгляд сверху», чтобы навести порядок в этом хаосе? Вы сможете почерпнуть несколько идей, как сделать ваш код чище, из ответов на следующие вопросы.

- Нужны ли вам все файлы и модули по отдельности или вы можете объединить некоторые из них и уменьшить взаимозависимость кода?
- Можете ли вы разделить большой и сложный файл на два более простых? Обратите внимание, что обычно существует золотая середина между двумя крайностями: большой, монолитный и совершенно нечитаемый блок кода или множество мелких блоков, которые невозможно мысленно охватить. И то и другое нежелательно, и большинство промежуточных этапов являются лучшими вариантами. Подумайте об этом как о перевернутой U-образной кривой, где максимум представляет собой золотую середину между несколькими большими блоками кода и множеством мелких.
- Можете ли вы обобщить код и превратить его в библиотеку, чтобы упростить основное приложение?
- Можете ли вы использовать существующие библиотеки, чтобы сократить количество строк кода?
- Можете ли вы использовать кэширование, чтобы избежать повторного вычисления одного и того же результата?

- Можете ли вы применить более простые и более подходящие алгоритмы, которые выполняют те же задачи, что и ваши текущие?
- Можете ли вы исключить преждевременные оптимизации, которые не улучшают общую производительность?
- Можете ли вы использовать другой язык программирования, более подходящий для решения поставленной задачи?

Увидеть общую картину — это эффективный способ резко снизить сложность вашего приложения в целом. Иногда эти изменения трудно реализовать на более поздних этапах; также этому может помешать совместная работа над кодом. В частности, размышлять на таком уровне затруднительно, имея дело с приложениями с миллионами строк кода, такими как операционная система Windows. Однако вы просто не можете позволить себе полностью игнорировать эти вопросы, поскольку все мелкие доработки, вместе взятые, не смогут нивелировать негативные последствия от неправильно или небрежно выбранной архитектуры. Обычно, если вы работаете в небольшом стартапе или просто на себя, вы можете быстро принимать смелые архитектурные решения, например о замене алгоритма. Если же вы сотрудник крупной компании, у вас, скорее всего, нет такой возможности. Но чем крупнее приложение, тем выше вероятность того, что вы найдете легкие и лежащие на поверхности решения.

Принцип 2. Встаньте на плечи гигантов

Изобретение велосипеда редко приносит пользу. Программирование — это отрасль с многолетней историей. Лучшие программисты мира оставили нам огромное наследие: базу данных с миллионами отточенных и проверенных алгоритмов и функций. Получить доступ к коллективной мудрости огромного количества специалистов так же просто, как использовать однострочный оператор импорта. Нет причин не использовать эти сверхвозможности в своих проектах.

Применение библиотечного кода наверняка повысит эффективность вашего собственного. Функции, которые использовались тысячами программистов, как правило, гораздо более оптимизированы, чем созданные вами. Кроме того, обращение к библиотеке проще для понимания и занимает меньше места в проекте, чем написанный вами код. Допустим, вам нужен алгоритм кластеризации для визуализации кластеров клиентов. Вы можете «встать на плечи гигантов», импортировав хорошо проверенный алгоритм из внешней библиотеки и использовав свои входные данные. Это гораздо более рационально, чем написание собственного кода — библиотечный алгоритм реализует ту же функциональность с меньшим количеством багов, меньшим объемом кода и большей производительностью. Библиотеки — это один из основных инструментов, с помощью которых профессиональные программисты повышают свою продуктивность в тысячи раз.

В качестве примера библиотечного кода, который экономит наше время, рассмотрим двухстрочный код, импортирующий модуль KMeans из библиотеки scikit-learn Python для поиска центроидов двух кластеров на заданном наборе данных, хранящемся в переменной `X`:

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
```

Реализация алгоритма KMeans самостоятельно заняла бы у вас несколько часов и составила бы более 50 строк, загромождая вашу кодовую базу и осложняя общее выполнение проекта.

Принцип 3. Пишите код для людей, а не для машин

Вы можете подумать, что основная цель фрагмента исходного кода — определить, что и как должен делать компьютер. Это не так. Единственная цель языка программирования, такого как Python, — помочь человеку написать код. Компиляторы выполняют тяжелую работу и переводят ваш высокоуровневый код в низкоуровневый, понятный машине. Да, ваш код в конечном итоге будет выполняться

компьютером. Но он по-прежнему пишется (в основном) людьми, и при современном процессе разработки код, скорее всего, пройдет несколько уровней оценки человеком до того, как будет произведено развертывание ПО. В первую очередь вы пишете код для людей, а не для машин.

Всегда думайте о том, что другие люди будут читать ваш исходный код. Представьте, что вы перешли в новый проект и кто-то другой должен занять ваше место в работе над кодовой базой. Есть много способов облегчить его труд и свести к минимуму его отчаяние. Прежде всего, используйте информативные имена переменных, чтобы читатель легко понимал, для чего предназначена та или иная строка кода. В листинге 4.1 показан пример неудачно выбранных имен переменных.

Листинг 4.1. Код, в котором неудачно выбраны имена переменных

```
xxx = 10000
yyy = 0.1
zzz = 10

for iii in range(zzz):
    print(xxx * (1 + yyy)**iii)
```

Трудно догадаться, что вычисляет этот код. С другой стороны, в листинге 4.2 представлен семантически эквивалентный код, в котором используются информативные имена переменных.

Листинг 4.2. Код, в котором используются информативные имена переменных

```
investments = 10000
yearly_return = 0.1
years = 10

for year in range(years):
    print(investments * (1 + yearly_return)**year)
```

Понять, какой процесс здесь реализован, гораздо проще: имена переменных указывают на то, как рассчитать общую стоимость с первоначальными инвестициями в размере 10 000 и расчетом

сложных процентов за 10 лет, предполагая ежегодную доходность в 10 %.

Мы не будем здесь вдаваться в детали всех способов реализации этого принципа (при рассмотрении следующих принципов некоторые из подходов обсуждаются подробно). Но отметим, что он проявляется и в некоторых других аспектах, которые могут прояснить его суть, таких как отступы, пробелы, комментарии, длины строк и пр. Чистый код радикально оптимизируется для удобства его прочтения человеком. Как утверждает Мартин Фаулер (Martin Fowler), международный эксперт по разработке программного обеспечения и автор популярной книги «Refactoring»[1] (Addison-Wesley, 1999): «Любой дурак может написать код, понятный компьютеру. Хорошие программисты пишут код, понятный людям».

Принцип 4. Соблюдайте правила именования

Вместе с тем опытные программисты часто договариваются о наборе определенных правил присвоения имен функциям, аргументам функций, объектам, методам и переменным; эти правила могут быть как явными, так и неявными. Соблюдение соглашений об именах приносит пользу всем: код становится более понятным, удобным для чтения и менее загроможденным. Если вы нарушите эти правила, читатели вашего кода, скорее всего, решат, что он написан неопытным программистом, и не воспримут его всерьез.

Соглашения могут отличаться для разных языков. Например, Java для именования переменных использует `camelCaseNaming`[2], а Python — `underscore_naming` (underscore — «знак подчеркивания». — *Примеч. ред.*) для переменных и функций. Если вы примените стиль CamelCase в Python, это может ввести читателя в за-

[1] Фаулер М. «Рефакторинг: улучшение проекта существующего кода».

[2] CamelCase («ВерблюжийРегистр») — стиль написания, при котором пробелы опускаются, а каждое слово пишется с заглавной буквы (что напоминает горбы верблюда, отсюда название стиля). — *Примеч. ред.*

блуждение. Вы же не хотите, чтобы ваши нетрадиционные правила присвоения имен отвлекали тех, кто читает ваш код. Нужно, чтобы люди сосредоточились на том, что он делает, а не на том, как он написан. Согласно *принципу наименьшего удивления*, нет никакого смысла изумлять других разработчиков кода выбором нестандартных имен переменных.

Итак, рассмотрим подробнее весь перечень правил именования, которые вам следует учитывать при написании исходного кода.

Выбирайте описательные имена. Допустим, вы создаете функцию для конвертации валют из долларов США (USD) в евро (EUR) в Python. Назовите ее `usd_to_eur(amount)`, а не `f(x)`.

Выбирайте недвусмысленные имена. Вы можете подумать, что имя `dollar_to_euro(amount)` будет хорошим названием для функции конвертации валюты. Конечно, оно лучше, чем `f(x)`, но оно хуже, чем `usd_to_eur(amount)`, потому что вносит ненужную двусмысленность. Что вы имеете в виду: доллары США, Канады или Австралии? Если вы находитесь в Соединенных Штатах, ответ будет для вас очевиден, но австралийский программист может не знать, что код написан в США, и может сделать другой вывод. Сведите к минимуму такие недоразумения!

Используйте произносимые имена. Большинство программистов, читая код, подсознательно произносят его в уме. Если имя переменной труднопроизносимо, его расшифровка отвлекает внимание и требует драгоценных ментальных усилий. Например, имя переменной `cstmr_lst` может быть описательным и недвусмысленным, но оно непроизносимо. Выбор имени `customer_list` стоит того, чтобы занять дополнительное место в вашем коде.

Используйте именованные константы, а не магические числа. В вашем коде может несколько раз встречаться магическое число 0.9 в качестве коэффициента для преобразования суммы в долларах США в сумму в евро. Однако читатель кода (и вы сами в будущем) вынужден будет задуматься над назначением

этого числа. Оно не является самоочевидным. Гораздо лучший способ обращения с магическим числом 0.9 — это сохранить его в переменной, имя которой состоит из прописных букв (обычно так обозначаются константы), например `CONVERSION_RATE = 0.9`, и использовать ее в качестве коэффициента при расчетах с конвертацией валют. Например, вы можете вычислить свой доход в евро как `income_euro = CONVERSION_RATE * income_usd`.

Это лишь несколько правил присвоения имен. Помимо перечисленных кратких советов, лучший способ изучить соглашения об именовании — ознакомиться с искусно написанным кодом экспертов. Для начала поищите в Google соответствующие соглашения (например, «Соглашения об именах в Python»). Также вы можете почитать учебники по программированию, присоединиться к StackOverflow, чтобы задать вопросы коллегам-разработчикам, просмотреть проекты с открытым исходным кодом на GitHub и присоединиться к сообществу амбициозных программистов в блоге Finxter, которые помогают друг другу развивать свои навыки.

Принцип 5. Придерживайтесь стандартов и будьте последовательны

Для каждого языка существует гласный или негласный набор правил о том, как писать чистый код. Если вы действующий программист, то рано или поздно вам придется с ними столкнуться. Однако вы можете ускорить процесс, изначально потратив время на изучение стандартов написания кода на интересующем вас языке программирования.

Например, вы можете получить доступ к официальному руководству по стилю написания кода на языке Python, PEP 8, по этой ссылке: https://www.python.org/dev/peps/pep-0008/. Как и любое руководство по стилям, PEP 8 определяет правильное расположение кода и отступы; способ задания разрыва строки; максимальное количество символов в строке; правильное использование комментариев; оформление документации на собственные функции; соглашения по присвоению

имен классам, переменным и функциям. Например, в листинге 4.3 показан хороший пример из PEP 8 о том, как применять различные стили и соглашения. Вы используете четыре пробела на каждый следующий уровень отступа, соответственно выравниваете аргументы функций, ставите одинарные пробелы при перечислении в списках аргументов значений, разделенных запятыми, и правильно присваиваете имена функциям и переменным, объединяя несколько слов знаком подчеркивания.

Листинг 4.3. Правильное использование отступов, пробелов и имен в Python в соответствии со стандартом PEP 8

```
# Выравнивание по открывающему разделителю
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Добавьте 4 пробела (дополнительный уровень отступа), чтобы отличать
# аргументы от остального текста
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Висячие отступы должны добавлять один уровень
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

В листинге 4.4 показан пример неверного написания кода. Аргументы не выровнены, конструкции, состоящие из нескольких слов, неправильно объединены в имена переменных и функций, списки аргументов не разделены должным образом одним пробелом, а уровни отступов содержат только два или три пробела вместо четырех:

Листинг 4.4. Неправильное использование отступов, интервалов и имен в Python

```
# Аргументы в первой строке запрещены, если не используется
# вертикальное выравнивание
foo = longFunctionName(varone,varTwo,
   var3,varxfour)
```

```
# Требуется дополнительный отступ, поскольку имеющиеся неразличимы
def longfunctionname(
  var1,var2,var3,
  var4):
  print(var_one)
```

Все читатели вашего кода ожидают от вас соблюдения принятых стандартов. Иное приведет лишь к путанице и разочарованию.

Однако чтение руководства по стилям может быть утомительным занятием. В качестве менее скучного способа изучения соглашений и стандартов используйте линтеры[1] и интегрированные среды разработки (integrated development environments, IDE), которые подскажут вам, где и какие ошибки вы допустили. Во время хакатона выходного дня с моей Finxter-командой мы создали инструмент под названием Pythonchecker.com, который в игровой форме поможет вам сделать рефакторинг вашего Python-кода и превратить его из грязного в суперчистый. В этом отношении один из лучших проектов для Python — модуль форматирования *Black* в PyCharm. Подобные инструменты существуют для всех основных языков. Просто погуглите *<Ваш язык> линтер*, чтобы найти лучшие инструменты для вашей среды программирования.

Принцип 6. Используйте комментарии

Как уже говорилось ранее, при написании кода для людей, а не для машин необходимо использовать комментарии, чтобы помочь читателям его понять. Рассмотрим код без комментариев, представленный в листинге 4.5.

Листинг 4.5. Код без комментариев

```
import re

text = '''
    Ha! let me see her: out, alas! She's cold:
    Her blood is settled, and her joints are stiff;
```

[1] Программа, проверяющая код на соответствие стандартам. — *Примеч. ред.*

```
    Life and these lips have long been separated:
    Death lies on her like an untimely frost
    Upon the sweetest flower of all the field.
'''
f_words = re.findall('\\bf\w+\\b', text)
print(f_words)


l_words = re.findall('\\bl\w+\\b', text)
print(l_words)

'''
OUTPUT:
['frost', 'flower', 'field']
['let', 'lips', 'long', 'lies', 'like']

'''
```

Программа в листинге 4.5 анализирует короткий фрагмент текста из шекспировской пьесы «Ромео и Джульетта» с помощью регулярных выражений. Если вы с ними незнакомы, вам, вероятно, будет трудно понять, что делает этот код. Даже осмысленные имена переменных не сильно помогают.

Давайте посмотрим, смогут ли несколько комментариев прояснить ситуацию (листинг 4.6).

Листинг 4.6. Код с комментариями

```
   import re

   text = '''
   Ha! let me see her: out, alas! She's cold:
   Her blood is settled, and her joints are stiff;
   Life and these lips have long been separated:
   Death lies on her like an untimely frost
   Upon the sweetest flower of all the field.
'''

❶  # Найти все слова, начинающиеся с символа 'f'.
   f_words = re.findall('\\bf\w+\\b', text)
   print(f_words)

❷  # Найти все слова, начинающиеся с символа 'l'.
```

```
l_words = re.findall('\\bl\w+\\b', text)
print(l_words)

'''
OUTPUT:
['frost', 'flower', 'field']
['let', 'lips', 'long', 'lies', 'like']
'''
```

Два коротких комментария (❶, ❷) поясняют назначение шаблонов регулярных выражений `'\\bf\w+\\b'` и `'\\bl\w+\\b'`. Я не буду здесь углубляться в понятие «регулярные выражения», но данный пример прекрасно демонстрирует, как комментарии могут помочь вам получить общее представление о чужом коде, не вдаваясь в подробности синтаксических приемов, облегчающих восприятие текста программы.

Вы также можете использовать комментарии, чтобы выделить блоки кода. Например, если у вас есть пять строк кода, которые касаются обновления информации о клиентах в базе данных, добавьте перед блоком короткий комментарий, чтобы пояснить это (листинг 4.7).

Листинг 4.7. Блоки с комментариями дают представление о коде

```
❶ # Обработка следующего заказа
order = get_next_order()
user = order.get_user()
database.update_user(user)
database.update_product(order.get_order())

❷ # Отправка заказа и подтверждение от клиента
logistics.ship(order, user.get_address())
user.send_confirmation()
```

Здесь демонстрируется, как интернет-магазин выполняет заказ клиента за два высокоуровневых этапа: обработка следующего заказа ❶ и его отправка ❷. Комментарии помогают быстро понять назначение кода без необходимости расшифровывать каждый вызов метода.

Также можно использовать комментарии для предупреждения программистов о потенциально нежелательных последствиях. На-

пример, в листинге 4.8 программа предупреждает нас о том, что вызов функции `ship_yacht()` фактически отправит заказчику очень дорогую яхту.

Листинг 4.8. Комментарий в качестве предупреждения

```
##########################################################################

# ВНИМАНИЕ                                                               #

# ВЫПОЛНЕНИЕ ЭТОЙ ФУНКЦИИ ОТПРАВИТ ЯХТУ СТОИМОСТЬЮ $1 569 420!! #

##########################################################################
def ship_yacht(customer):
    database.update(customer.get_address())
    logistics.ship_yacht(customer.get_address())
    logistics.send_confirmation(customer)
```

Вы можете использовать комментарии многими другими полезными способами; они нужны не только для корректного следования стандартам. При написании комментариев держите в голове принцип *«код пишется для людей»* — и все будет в порядке. Читая код опытных программистов, вы со временем успешно и почти автоматически усвоите негласные правила. Поскольку вы являетесь экспертом в написанном вами коде, полезные комментарии позволят посторонним понять ваш образ мышления. Не упустите возможность поделиться своими мыслями с другими людьми!

Принцип 7. Избегайте ненужных комментариев

Тем не менее не все комментарии помогают читателям лучше понять код. В некоторых случаях они даже снижают его читабельность и сбивают с толку. Чтобы писать чистый код, следует не только добавлять полезные комментарии, но и избегать ненужных.

Когда я был научным сотрудником в области computer science, один из моих хорошо подготовленных студентов устроился на работу в Google. Он рассказал мне, что рекрутеры Google раскритиковали

его стиль написания кода, потому что он добавлял слишком много ненужных комментариев. Оценка ваших комментариев — еще один способ, с помощью которого грамотные специалисты определяют ваш уровень (начинающий, средний или опытный). Недочеты в коде, такие как нарушение стиля, лень или небрежность в комментариях, а также написание неидиоматического кода для конкретного языка программирования, называются *запахами кода (code smells)*, то есть признаками, которые указывают на потенциальные проблемы, и опытные разработчики могут почуять их за версту.

Как определить, какие комментарии не следует оставлять? В большинстве случаев комментарий не нужен, если он избыточный. Например, при использовании осмысленных имен переменных код часто становится очевидным и не требует комментариев на уровне строки. Рассмотрим сниппет (фрагмент кода) с осмысленными именами переменных в листинге 4.9.

Листинг 4.9. Фрагмент кода с осмысленными именами переменных

```
investments = 10000
yearly_return = 0.1
years = 10

for year in range(years):
    print(investments * (1 + yearly_return)**year)
```

Уже ясно, что код рассчитывает ваш совокупный доход от инвестиций за 10 лет, предполагая доходность в 10 %. Для наглядности добавим несколько ненужных комментариев (листинг 4.10).

Листинг 4.10. Ненужные комментарии

```
investments = 10000 # Ваши инвестиции, при необходимости можно
                    # изменить
yearly_return = 0.1 # Годовой доход (например, 0.1 --> 10 %)
years = 10 # Количество лет для расчета совокупного дохода

# Перебрать каждый год
for year in range(years):
    # Вывести стоимость ваших инвестиций в текущем году
    print(investments * (1 + yearly_return)**year)
```

Все комментарии в листинге 4.10 избыточны. Некоторые из них оказались бы полезными, если бы имена переменных были менее осмысленными, но объяснение переменной с именем `yearly_return` с комментарием о том, что она представляет собой годовой доход, только перегружает код.

В целом, чтобы решить, нужен ли комментарий, следует руководствоваться здравым смыслом, но есть несколько основных рекомендаций.

Не используйте встроенные комментарии. Их можно полностью избежать, выбирая осмысленные имена переменных.

Не добавляйте очевидные комментарии. В листинге 4.10 комментарий, поясняющий оператор цикла `for`, не нужен. Каждый разработчик знает цикл `for`, и нет никакого смысла в добавлении комментария `# Перебрать каждый год`, учитывая строку `for year in range(years)`.

Не надо комментировать старый код, просто удаляйте его. Мы, программисты, часто держимся за наши любимые сниппеты, даже после того как (неохотно) решили от них избавиться, закомментировав их. Это ухудшает читабельность кода! Всегда удаляйте ненужный код — для собственного спокойствия вы можете использовать инструмент «история версий», например Git. Он сохраняет ранние черновики вашего проекта.

Используйте функцию документирования. Многие языки программирования, такие как Python, имеют встроенную возможность документирования, которая позволяет описать назначение каждой функции, метода и класса в вашем коде. Если каждый из этих элементов выполняет только одну задачу (согласно принципу 10), то обычно достаточно применять документирование вместо комментариев для описания того, что делает ваш код.

Принцип 8. Принцип наименьшего удивления

Принцип наименьшего удивления гласит, что компонент системы должен вести себя так, как ожидает большинство пользователей.

Этот принцип является одним из золотых правил при разработке эффективных приложений и взаимодействия с пользователем. Например, если вы откроете поисковую систему Google, курсор сам разместится в поле ввода поиска, чтобы вы могли сразу начать вводить ключевое слово, как вы и ожидали — никаких сюрпризов.

Чистый код также использует этот принцип разработки. Допустим, вы пишете конвертер валют, который преобразует вводимые данные из долларов США в китайские юани. Вы храните пользовательский ввод в переменной. Какое имя переменной лучше выбрать: `user_input` или `var_x`? Принцип наименьшего удивления ответит на этот вопрос за вас!

Принцип 9. Не повторяйтесь

Не повторяйтесь (Don't repeat yourself, DRY) — это общепризнанный интуитивно понятный принцип, который рекомендует избегать повторяющегося кода. Например, возьмем код Python в листинге 4.11: он пять раз выводит в оболочку одну и ту же строку.

Листинг 4.11. Вывод hello world пять раз

```
print('hello world')
print('hello world')
print('hello world')
print('hello world')
print('hello world')
```

Код, который включает гораздо меньше повторений, представлен в листинге 4.12.

Листинг 4.12. Уменьшение количества повторений, содержащихся в листинге 4.11

```
for i in range(5):
    print('hello world')
```

Код в листинге 4.12 выведет `hello world` пять раз, как и в листинге 4.11, но без дублирования строк.

Функции также могут быть полезным инструментом для сокращения повторений. Допустим, вам нужно преобразовать мили в километры в нескольких местах вашего кода, как в листинге 4.13.

Сначала вы создаете переменную `miles` и конвертируете ее в километры путем умножения на 1.60934. Затем вы конвертируете 20 миль в километры, умножая 20 на 1.60934, и сохраняете результат в переменной `distance`.

Листинг 4.13. Двойное преобразование миль в километры

```
miles = 100
kilometers = miles * 1.60934

distance = 20 * 1.60934

print(kilometers)
print(distance)


'''
OUTPUT:
160.934
32.1868
'''
```

Вы дважды использовали одну и ту же процедуру, умножая значение `miles` на коэффициент 1.60934 для преобразования миль в километры. Принцип DRY предполагает, что лучше один раз написать функцию `miles_to_km(miles)`, как в листинге 4.14, чем несколько раз выполнять одно и то же преобразование в коде.

Листинг 4.14. Использование функции для преобразования миль в километры

```
def miles_to_km(miles):
    return miles * 1.60934


miles = 100
kilometers = miles_to_km(miles)

distance = miles_to_km(20)
```

```
print(kilometers)
print(distance)

'''
OUTPUT:
160.934
32.1868
'''
```

Таким образом код легче поддерживать. Можно, например, подправить функцию, чтобы повысить точность преобразования, и вам придется внести это изменение только в одном месте. В листинге 4.13 вам пришлось бы искать в коде все копии данной команды для внесения такого изменения. Кроме того, применение принципа DRY делает код более понятным для читателя. Не возникает никаких сомнений в значении функции `miles_to_km(20)`, а вот над смыслом произведения `20 * 1.60934`, возможно, придется хорошенько подумать.

Нарушения принципа DRY часто обозначаются аббревиатурой WET, которая расшифровывается так: *нам нравится печатать (we enjoy typing)*, *мы пишем все дважды (write everything twice)* или *тратим чужое время впустую (waste everyone's time)*.

Принцип 10. Принцип единой ответственности

Принцип единой ответственности (single responsibility principle) означает, что каждая функция должна выполнять только одну главную задачу. Лучше использовать много мелких функций, чем одну большую, отвечающую за все задачи сразу. Инкапсуляция функциональности снижает общую сложность кода.

Как правило, каждый класс и каждая функция должны иметь только одну ответственность (обязанность). Роберт К. Мартин, автор этого принципа, понимает *ответственность* как *причину для изменения*. Таким образом, для него золотой стандарт определения класса или функции — ограничить их выполнением одной обязанности. В этом случае только программист, желающий изменить именно эту ответственность, может запросить изменение в определении. Ни один из

специалистов с другими обязанностями даже не подумал бы отправить запрос на изменение этого класса, учитывая, конечно, что код изначально корректен. Например, функция, отвечающая за чтение данных из БД, не должна отвечать еще и за их обработку. В противном случае у нее бы возникло сразу две причины для изменения: при внесении изменений в модель БД либо при смене требований к обработке данных. В такой ситуации несколько программистов могут начать изменять один и тот же класс одновременно. У вашего класса окажется слишком много ответственностей, и он становится путаным и перегруженным.

Рассмотрим небольшой пример кода на Python, который может быть запущен на электронной читалке для моделирования и управления пользовательским опытом чтения (листинг 4.15).

Листинг 4.15. Построение модели класса Book с нарушением принципа единой ответственности

```
❶ class Book:

  ❷ def __init__(self):
        self.title = "Python One-Liners"
        self.publisher = "NoStarch"
        self.author = "Mayer"
        self.current_page = 0

    def get_title(self):
        return self.title

    def get_author(self):
        return self.author

    def get_publisher(self):
        return self.publisher

  ❸ def next_page(self):
        self.current_page += 1
        return self.current_page

  ❹ def print_page(self):
        print(f"... Page Content {self.current_page} ...")
```

```
❺ python_one_liners = Book()

  print(python_one_liners.get_publisher())
  # NoStarch

 python_one_liners.print_page()
  # ... Содержание страницы 0 ...

 python_one_liners.next_page()
 python_one_liners.print_page()
  # ... Содержание страницы 1 ...
```

В листинге 4.15 код определяет класс `Book` ❶ с четырьмя атрибутами: название, автор, издатель и номер текущей страницы. Вы определяете методы чтения для атрибутов ❷, а также некоторую минимальную функциональность для перехода к следующей странице ❸, которая вызывается каждый раз, когда пользователь нажимает кнопку на устройстве для чтения. Функция `print_page()` отвечает за вывод текущей страницы на устройство ❹. Эта функция приведена только в качестве примера, в реальности она была бы гораздо сложнее. Наконец, вы создаете экземпляр `Book` с именем `python_one_liners` ❺ и получаете доступ к его атрибутам путем серии вызовов различных методов и операторов вывода в последней паре строк. Настоящая реализация программы для чтения электронных книг, к примеру, будет вызывать методы `next_page()` и `print_page()` каждый раз, когда пользователь запрашивает новую страницу.

Хотя код выглядит чистым и простым, он нарушает принцип единой ответственности: класс `Book` отвечает как за моделирование данных, таких как содержание книги, так и за вывод текста на устройство. Моделирование и вывод — это две разные функции, но они инкапсулированы в один класс. Налицо несколько причин для изменений. Возможно, вы захотите изменить способ моделирования данных книги: например, использовать базу данных вместо файлового метода ввода/вывода. Но вы также можете решить изменить представление моделируемых данных, скажем, применить иную схему форматирования книги для других типов экранов.

Давайте исправим этот момент в листинге 4.16.

Листинг 4.16. Соблюдение принципа единственной ответственности

```
❶ class Book:

    ❷ def __init__(self):
          self.title = "Python One-Liners"
          self.publisher = "NoStarch"
          self.author = "Mayer"
          self.current_page = 0

      def get_title(self):
          return self.title

      def get_author(self):
          return self.author

      def get_publisher(self):
          return self.publisher

      def get_page(self):
          return self.current_page

      def next_page(self):
          self.current_page += 1


❸ class Printer:

    ❹ def print_page(self, book):
          print(f"... Page Content {book.get_page()} ...")



python_one_liners = Book()
printer = Printer()

printer.print_page(python_one_liners)
 # ... Содержание страницы 0 ...

python_one_liners.next_page()
printer.print_page(python_one_liners)
 # ... Содержание страницы 1 ...
```

Код в листинге 4.16 решает ту же задачу, но удовлетворяет принципу единственной ответственности. Вы создаете класс `Book` ❶ и класс `Printer` ❸. Класс `Book` представляет метаданные книги и номер текущей страницы ❷, а класс `Printer` отвечает за вывод книги на устройство. Вы передаете книгу, для которой вы хотите вывести текущую страницу, в метод `Printer.print_page()` ❹. Таким образом, моделирование данных (каких именно данных?) и представление данных (как данные отображаются для пользователя?) разнесены, и код становится проще поддерживать. Например, если вы хотите изменить модель данных книги, добавив новый атрибут `publishing_year`, вы сделаете это в классе `Book`. А если вы хотите отразить это изменение в представлении данных, предоставив читателям и эту информацию, вы должны осуществить это в классе `Printer`.

Принцип 11. Тестируйте

Разработка через тестирование является неотъемлемой частью современного процесса создания ПО. Каким бы опытным специалистом вы ни были, вы все равно будете допускать ошибки в коде. Чтобы их отследить, необходимо либо периодически проводить тесты, либо изначально создавать код на их основе. Каждая крупная компания-разработчик ПО использует несколько уровней тестирования перед тем, как выпускать конечный продукт на рынок, поскольку гораздо лучше обнаружить ошибки в пределах компании, чем узнать о них от недовольных пользователей.

Хотя нет никаких ограничений на виды тестов, проводимых для улучшения ваших программных приложений, приведем здесь наиболее распространенные из них.

Юнит-тесты (unit tests). При использовании юнит-тестов (называют также «модульные тесты») вы пишете отдельное приложение, чтобы проверить корректность связи между вводом и выводом для различных входных данных каждой функции

в приложении. Юнит-тесты обычно применяются через регулярные промежутки времени, например каждый раз, когда выпускается новая версия ПО. Это снижает вероятность того, что внесенные изменения приведут к внезапному сбою ранее стабильно работавших функций.

Пользовательские приемочные тесты (user acceptance tests). Они позволяют людям из вашей целевой аудитории использовать приложение в контролируемой среде, пока вы наблюдаете за их поведением. Затем вы спрашиваете их, понравилось ли им приложение и как его улучшить. Эти тесты обычно проводятся на заключительном этапе разработки проекта, после всестороннего тестирования внутри компании.

Дымовые тесты (smoke tests). Это грубые тесты, разработанные для того, чтобы попытаться обрушить разрабатываемое приложение еще до того, как создающая его команда передаст его группе тестирования. Другими словами, дымовые тесты часто проводятся разработчиками приложения для контроля качества, прежде чем передать код тестировщикам. После дымового тестирования приложение готово к следующему этапу проверки.

Тесты производительности (performance tests). Они проводятся с целью определить, соответствует ли приложение запросам пользователей по производительности (или, возможно, даже превосходит их), а не для того, чтобы проверить его фактическую функциональность. Например, Netflix, прежде чем выпустить новую функцию, должен протестировать свой сайт на скорость загрузки страниц. Если нововведение слишком замедляет работу фронтенда, Netflix отказывается от этой идеи, заранее избегая негативной реакции пользователей.

Тесты на масштабируемость (scalability tests). Если ваше приложение станет успешным, вам, возможно, придется обрабатывать по 1000 запросов в минуту вместо первоначальных двух. Тест покажет, масштабируется ли ваше приложение настолько,

чтобы с этим справиться. Обратите внимание, что высокопроизводительное приложение не обязательно является масштабируемым, и наоборот. Так, скоростной катер очень эффективен, но не рассчитан на перевозку тысячи людей одновременно!

Тестирование и рефакторинг зачастую способствуют уменьшению сложности кода и снижению количества ошибок. Однако будьте осторожны и не переусердствуйте (см. принцип 14) — тестируйте только реальные сценарии. Например, нет необходимости проверять, может ли приложение Netflix работать с сотней миллиардов потоковых устройств, учитывая, что на планете всего 7 миллиардов потенциальных зрителей.

Принцип 12. Малое — прекрасно

Малый код — это код, который требует относительно небольшого количества строк для выполнения одной конкретной задачи. Вот пример небольшого кода для функции, которая считывает целочисленное значение, введенное пользователем, и проверяет, что оно действительно целое:

```
def read_int(query):
    print(query)
    print('Please type an integer next:')
    try:
        x = int(input())
    except:
        print('Try again - type an integer!')
        return read_int(query)
    return x

print(read_int('Your age?'))
```

Код выполняется до тех пор, пока пользователь не введет целое число. Вот пример выполнения:

```
Your age?
Please type an integer next:
hello
```

```
Try again – type an integer!
Your age?
Please type an integer next:
twenty
Try again – type an integer!
Your age?
Please type an integer next:
20
20
```

Отделив логику чтения целочисленного значения от пользователя, вы можете использовать одну и ту же функцию несколько раз. Но, что более важно, вы разбили код на более мелкие функциональные единицы, которые относительно легко читать и воспринимать.

Вместо этого многие новички (или ленивые программисты среднего уровня) пишут большие, не разделенные на блоки функции кода, или так называемые *«божественные объекты» (God Objects)*, которые делают все и сразу. Такие монолитные блоки кода — это кошмар для поддержки. Во-первых, человеку проще разобраться в одной небольшой функции, чем пытаться отыскать что-то в блоке кода из 10 000 строк. А во-вторых, в крупном блоке потенциально можно допустить гораздо больше ошибок, чем в нескольких мелких функциях и блоках, которые затем интегрируются в существующую кодовую базу.

В начале этой главы на рис. 4.1 было продемонстрировано, что разработка кода становится все более трудоемкой с каждой дополнительной строкой, и чистый код в долгосрочной перспективе пишется намного быстрее, чем грязный. На рис. 4.2 показано, сколько времени уходит на работу с малыми блоками кода, а сколько — на работу с большими монолитными блоками. Для крупных блоков время, необходимое для добавления каждой следующей строки, будет увеличиваться экспоненциально. Однако если вы накладываете несколько небольших функций кода одну на другую, время, затрачиваемое на каждую новую строку, растет практически линейно. Чтобы достичь такого эффекта, вы должны быть уверены, что каждая функция более или менее независима от других. Подробнее об этом вы узнаете в описании следующего принципа, закона Деметры.

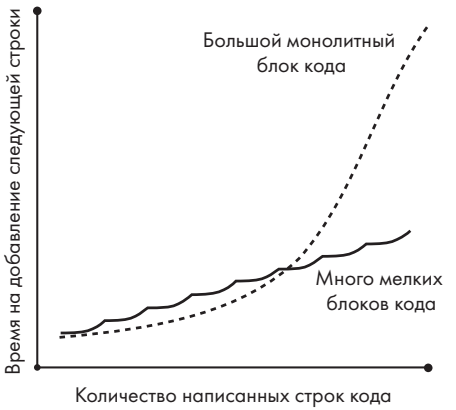


Рис. 4.2. При использовании большого монолитного блока кода время на добавление каждой следующей строки увеличивается экспоненциально. А при использовании множества мелких блоков время растет квазилинейно

Принцип 13. Закон Деметры

Зависимости существуют везде. Когда вы импортируете библиотеку в свой код, он частично зависит от ее функциональности, но также он имеет взаимозависимости и внутри себя. В объектно-ориентированном программировании зависимости могут существовать между двумя функциями, двумя объектами либо между двумя определениями класса.

Чтобы писать чистый код, сведите к минимуму взаимозависимость его элементов, следуя *закону Деметры*, который был предложен в конце 1980-х годов разработчиком программного обеспечения Яном Холландом (Ian Holland). Холланд работал над проектом, названным в честь Деметры, древнегреческой богини земледелия, развития и плодородия. Команда этого проекта продвигала стратегию «выращивания ПО», а не просто его создания. Однако то, что стало впоследствии известно как закон Деметры — практический подход к написанию слабо связанного кода в объектно-ориентированном программировании, — имеет мало общего с этими, скорее, более

метафизическими, идеями. Вот краткая цитата, объясняющая закон Деметры, взятая с сайта проектной группы Demeter Team http://ccs.neu.edu/home/lieber/what-is-demeter.html:

> *Важной концепцией системы Demeter является разделение программного обеспечения как минимум на две части. Первая определяет объекты, а вторая — операции. Целью системы Demeter является поддержание слабой связанности между объектами и операциями, чтобы можно было вносить изменения в один элемент без серьезного влияния на другой. Это значительно сокращает время на поддержку ПО.*

Другими словами, вы должны минимизировать зависимость между объектами кода. Уменьшая ее, вы снижаете сложность кода и, соответственно, повышаете удобство сопровождения. Из этого следует, например, что каждый объект должен вызывать только методы смежных объектов (или собственные), но не может вызывать методы объектов, которые он получает в результате вызова метода соседнего объекта. Для пояснения давайте определим: если объект A вызывает методы объекта B, то A и B считаются *друзьями*. Все просто. Но что, если метод B ссылается на объект C? Теперь объект A может выполнять что-то вроде этого: `B.method_of_B().method_of_C()`. Это называется *цепочкой* вызовов методов — в нашем случае выходит, что вы общаетесь с другом вашего друга. Закон Деметры гласит: *разговаривай только со своими близкими друзьями*, поэтому он запрещает создание цепочек методов такого типа. Поначалу это может показаться довольно запутанным, поэтому давайте подробнее рассмотрим все на практическом примере (рис. 4.3).

На рис. 4.3 показаны два проекта объектно-ориентированного кода, которые рассчитывают стоимость одной чашки (price per cup) кофе для данного человека. Одна из реализаций нарушает закон Деметры, а другая его придерживается. Начнем с отрицательного примера, в котором вы используете цепочку методов в классе `Person`, чтобы поговорить с незнакомцем ❶ (листинг 4.17).

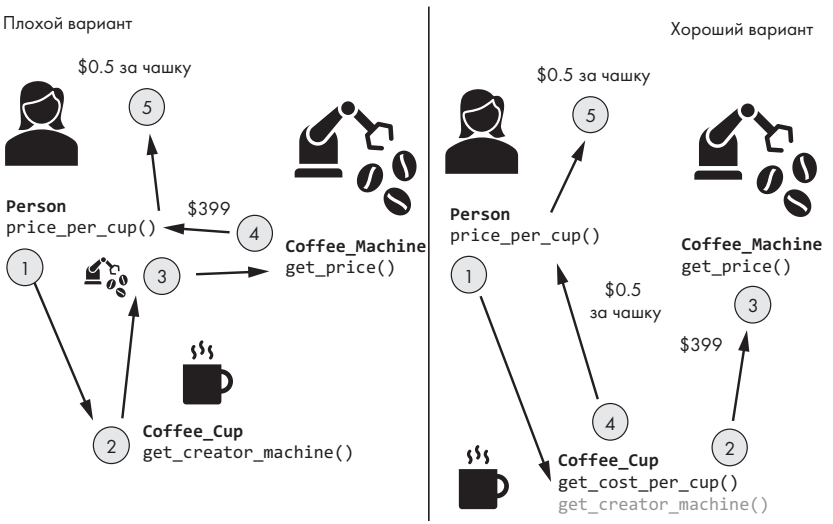


Рис. 4.3. Закон Деметры: разговаривайте только с близкими друзьями, чтобы свести к минимуму взаимозависимости

Листинг 4.17. Код, нарушающий закон Деметры

```
# НАРУШЕНИЕ ЗАКОНА ДЕМЕТРЫ (ПЛОХОЙ ВАРИАНТ)

class Person:
    def __init__(self, coffee_cup):
        self.coffee_cup = coffee_cup

    def price_per_cup(self):
        cups = 798
      ❶ machine_price = self.coffee_cup.get_creator_machine().
                        get_price()
        return machine_price / cups


class Coffee_Machine:
    def __init__(self, price):
        self.price = price

    def get_price(self):
        return self.price
```

```
class Coffee_Cup:
    def __init__(self, machine):
        self.machine = machine

    def get_creator_machine(self):
        return self.machine


m = Coffee_Machine(399)
c = Coffee_Cup(m)
p = Person(c)

print('Price per cup:', p.price_per_cup())
# 0.5
```

Вы создаете метод `price_per_cup()`, который рассчитывает стоимость одной чашки кофе на основе цены кофемашины и количества чашек, приготовленных ею. Объект `Coffee_Cup` собирает информацию о цене кофемашины, влияющей на цену за чашку, и передает ее объекту `Person`, который вызывает метод `price_per_cup()`.

Схема в левой части рис. 4.3 показывает плохую версию реализации этого процесса. Рассмотрим пошаговое пояснение к соответствующему фрагменту кода из листинга 4.17.

1. Метод `price_per_cup()` вызывает метод `Coffee_Cup.get_creator_machine()`, чтобы получить ссылку на объект `Coffee_Machine`, который варит кофе.
2. Метод `get_creator_machine()` возвращает ссылку на объект `Coffee_Machine`, который изготовил содержимое чашки.
3. Метод `price_per_cup()` вызывает метод `Coffee_Machine.get_price()` на объекте `Coffee_Machine`, ссылку на который он только что получил из предыдущего вызова метода с объекта `Coffee_Cup`.
4. Метод `get_price()` возвращает цену кофемашины.
5. Метод `price_per_cup()` вычисляет амортизацию[1] на изготовление одной чашки и использует ее для вычисления цены. Данные возвращаются вызывающему метод объекту.

[1] Амортизация — постепенный перенос стоимости основных средств производства на себестоимость продукции. — *Примеч. ред.*

Это плохая стратегия, потому что класс `Person` зависит от двух объектов: `Coffee_Cup` и `Coffee_Machine` ❶. Программист, ответственный за поддержку этого класса, должен знать определения обоих родительских классов — любое изменение в каждом из них может повлиять и на класс `Person`.

Закон Деметры минимизирует такие зависимости. Вы можете увидеть хороший вариант моделирования той же задачи на рис. 4.3 справа и в листинге 4.18. В этом фрагменте кода класс `Person` не обращается напрямую к классу `Coffee_Machine` — ему даже не нужно знать о его существовании!

Листинг 4.18. Код, который соблюдает закон Деметры, «не разговаривая с незнакомцами»

```
# СОБЛЮДЕНИЕ ЗАКОНА ДЕМЕТРЫ (ХОРОШИЙ ВАРИАНТ)

class Person:
    def __init__(self, coffee_cup):
        self.coffee_cup = coffee_cup

    def price_per_cup(self):
        cups = 798
      ❶ return self.coffee_cup.get_cost_per_cup(cups)


class Coffee_Machine:
    def __init__(self, price):
        self.price = price

    def get_price(self):
        return self.price


class Coffee_Cup:
    def __init__(self, machine):
        self.machine = machine

    def get_creator_machine(self):
        return self.machine

    def get_cost_per_cup(self, cups):
        return self.machine.get_price() / cups


m = Coffee_Machine(399)
```

```
c = Coffee_Cup(m)
p = Person(c)

print('Price per cup:', p.price_per_cup())
# 0.5
```

Давайте пошагово рассмотрим этот фрагмент кода:

- Метод `price_per_cup()` вызывает метод `Coffee_Cup.get_cost_per_cup()` для получения расчетной цены за чашку.
- Метод `get_cost_per_cup()`, прежде чем ответить вызывающему методу, в свою очередь вызывает метод `Coffee_Machine.get_price()`, чтобы получить доступ к цене кофемашины.
- Метод `get_price()` возвращает информацию о цене.
- Метод `get_cost_per_cup()` рассчитывает стоимость за одну чашку и возвращает ее вызывающему методу `price_per_cup()`.
- Метод `price_per_cup()` просто передает это рассчитанное значение вызывающему метод объекту ❶.

Этот подход намного лучше, потому что класс `Person` теперь независим от класса `Coffee_Machine`. Общее число зависимостей уменьшается. Для проекта с сотнями классов такое сокращение значительно снижает общую сложность приложения. Опасность возрастающей сложности больших приложений в том, что количество потенциальных зависимостей повышается нелинейно с ростом числа объектов. Грубо говоря, нелинейная кривая растет быстрее, чем прямая линия. Например, удвоение количества объектов может легко повысить число зависимостей (что равносильно увеличению сложности) в четыре раза. Однако, следуя закону Деметры, можно свести эту тенденцию к минимуму, значительно сократив зависимости. Если каждый объект общается только с k другими объектами и при этом у вас всего n объектов, то общее количество зависимостей ограничено $k*n$, что является линейным соотношением, если k — константа. Таким образом, закон Деметры может математически помочь вам корректно масштабировать приложения!

Принцип 14. Вам это никогда не понадобится

Этот принцип гласит, что вы никогда не должны реализовывать код *в надежде*, что он когда-нибудь вам пригодится, потому что это не так! Пишите код, только если вы на 100 % уверены в его необходимости, — для сегодняшних нужд, а не для завтрашних. Если в будущем вам действительно понадобится код, о важности которого вы раньше только догадывались, ничто не помешает вам реализовать эту функцию позже. Но в данный момент вы не потратите время на написание ненужных строк кода.

Это помогает мыслить исходя из первых принципов. Самый простой и чистый код — это пустой файл. Теперь двигайтесь дальше — что вам *необходимо* в него добавить? В главе 3 вы узнали о MVP: коде, который очищен от лишнего, чтобы сосредоточиться на основной функциональности. Если вы сведете к минимуму количество используемых функций, вы получите более чистый и простой код, чем получили бы с помощью методов рефакторинга или всех других принципов, вместе взятых. Рассмотрите вероятность отказа от функций, которые дают относительно меньший вклад по сравнению с другими. Издержки упущенных возможностей редко измеряются, но часто оказываются значительными. Функция должна быть вам действительно *необходима*, прежде чем вы даже задумаетесь о ее реализации.

Смысл в том, что нужно избегать *чрезмерной инженерии (overengineering)*: создания продукта, который является производительнее и надежнее или содержит больше функций, чем это необходимо. Это добавляет излишнюю сложность, что должно немедленно вас насторожить.

Например, я часто сталкивался с задачами, которые можно было бы решить в течение нескольких минут с помощью примитивного алгоритмического подхода, но, как и многие программисты, я отказывался принимать небольшие ограничения этих способов. Вместо этого я изучал самые современные алгоритмы кластеризации, чтобы увеличить производительность кластеризации на несколько

процентов по сравнению с простым алгоритмом KMeans. Эти длительные оптимизации были невероятно дорогостоящими — мне пришлось потратить 80 % времени, чтобы получить 20-процентное улучшение результата. Это было бы оправдано, если бы мне были *необходимы* эти 20 % и у меня не было другого выхода. Но на самом деле мне совсем не следовало реализовывать эти навороченные алгоритмы кластеризации. Типичный случай чрезмерной инженерии!

Всегда выбирайте в первую очередь легкий путь. Используйте примитивные алгоритмы и простые методы для установления критериев оценки, а затем проанализируйте, какие из новых функций или оптимизаций производительности дадут наилучший результат для всего приложения. Мыслите глобально, а не локально: сосредоточьтесь на общей картине (в соответствии с принципом 1), а не на мелких, требующих времени исправлениях.

Принцип 15. Не используйте слишком много уровней отступов

Большинство языков программирования используют отступы в тексте, чтобы визуализировать иерархическую структуру потенциально вложенных условных блоков, определений функций либо циклов кода. Однако чрезмерное использование отступов ухудшает читабельность кода. В листинге 4.19 показан сниппет со слишком большим количеством уровней отступов, что затрудняет его понимание.

Листинг 4.19. Слишком много уровней отступов для вложенных блоков кода

```
def if_confusion(x, y):
    if x>y:
        if x-5>0:
            x = y
            if y==y+y:
                return "A"
            else:
                return "B"
```

```
        elif x+y>0:
            while x>y:
                x = x-1
            while y>x:
                y = y-1
            if x==y:
                return "E"
        else:
            x = 2 * x
            if x==y:
                return "F"
            else:
                return "G"
    else:
        if x-2>y-4:
            x_old = x
            x = y * y
            y = 2 * x_old
            if (x-4)**2>(y-7)**2:
                return "C"
            else:
                return "D"
        else:
                return "H"

print(if_confusion(2, 8))
```

Если вы теперь попытаетесь угадать, каков результат работы этого сниппета, то обнаружите, что его на самом деле трудно проследить. Функция кода `if_confusion(x, y)` выполняет относительно простые проверки переменных `x` и `y`. Однако в разных уровнях отступов очень легко запутаться. Код совсем нельзя назвать чистым.

В листинге 4.20 показано, как можно написать тот же код более чисто и просто.

Листинг 4.20. Меньше уровней отступов для вложенных блоков кода

```
def if_confusion(x,y):
    if x>y and x>5 and y==0:
        return "A"
    if x>y and x>5:
        return "B"
    if x>y and x+y>0:
```

```
        return "E"
    if x>y and 2*x==y:
        return "F"
    if x>y:
        return "G"
    if x>y-2 and (y*y-4)**2>(2*x-7)**2:
        return "C"
    if x>y-2:
        return "D"
    return "H"
```

В листинге 4.20 мы уменьшили количество отступов и понизили вложенность. Теперь вы можете пробежаться по всем проверкам и увидеть, какая из них первой применяется к вашим аргументам `x` и `y`. Большинству разработчиков гораздо удобнее читать плоский код, чем код с высокой степенью вложенности, даже если это достигнуто за счет избыточного контроля; здесь, например, условие `x>y` проверяется несколько раз.

Принцип 16. Используйте метрики

Используйте метрики качества кода, чтобы отслеживать изменение его сложности с течением времени. Наилучшая, хотя и неофициальная, метрика, известная как количество WTF[1] в минуту, предназначена для измерения неудовлетворенности читателей вашего кода. Результаты будут низкими для чистого и простого кода и высокими для грязного и запутанного.

В качестве косвенного показателя для этого трудно поддающегося количественной оценке стандарта можно использовать официальные метрики, такие как сложность NPath или цикломатическая сложность, рассмотренные в главе 1. Для большинства IDE разнообразные онлайн-инструменты и плагины автоматически вычисляют сложность по мере написания исходного кода. К ним относится CyclomaticComplexity, который можно найти с помощью поиска в разделе плагинов JetBrains по адресу https://plugins.jetbrains.com/.

[1] «What The Fuck!» — понятно без перевода. — *Примеч. пер.*

На мой взгляд, отслеживание текущей сложности менее важно, чем осознание того, что вам нужно избегать ее везде, где только возможно. Однако я настоятельно рекомендую использовать эти инструменты: они помогут вам писать более чистый и простой код. Поверьте, отдача от затраченного вами времени будет феноменальной.

Принцип 17. Правило бойскаутов и рефакторинг

Правило бойскаутов простое: *оставляй лагерь более чистым, чем он был до твоего прихода*. Это отличное правило и в жизни, и при написании кода. Возьмите в привычку чистить каждый фрагмент кода, который вам встречается. Это не только улучшит кодовые базы, в которых вы участвуете, и облегчит вашу жизнь, но и поможет вам обрести острый глаз опытного программиста, способного быстро оценивать исходный код. В качестве бонуса ваша команда начнет работать более продуктивно, а коллеги будут благодарны вам за ценностно-ориентированное отношение. Обратите внимание на то, что это не должно входить в противоречие с правилом, о котором мы говорили ранее, — избегать преждевременной оптимизации (то есть чрезмерной инженерии). Тратить время на очистку кода, чтобы уменьшить сложность, почти всегда целесообразно. Это приносит большие преимущества в виде снижения расходов на обслуживание, количества багов и когнитивных затрат. Короче говоря, чрезмерная инженерия, скорее всего, приведет к *увеличению* сложности, в то время как очистка кода *уменьшит* ее.

Процесс улучшения кода называется *рефакторингом*. Вы можете возразить, что рефакторинг — это общий метод, включающий все принципы, которые мы здесь обсуждали. Будучи хорошим разработчиком, вы с самого начала будете использовать многие принципы чистого кода. Однако даже в этом случае вам все равно придется время от времени заниматься рефакторингом, чтобы устранить все ошибки, которые вы допустили. В частности, вы должны осуществлять его перед выпуском любых новых функций, чтобы поддерживать чистоту кода.

Существует множество методов рефакторинга. Один из них — объяснить свой код коллеге или попросить его просмотреть код, чтобы обнаружить любые неудачные решения, которые вы допустили и сами того не заметили. Например, вы могли создать два класса, `Cars` (легковые автомобили) и `Trucks` (грузовики), потому что думали, что вашему приложению понадобится моделировать оба класса. По мере того как вы объясняете свой код коллеге, вы начинаете понимать, что класс `Trucks` применяется редко, а когда используется, вы прибегаете к методам, которые уже существуют в классе `Cars`. Ваш коллега предлагает создать класс `Vehicle` (транспортное средство), который обрабатывает и легковые, и грузовые автомобили. Это позволяет сразу же избавиться от множества строк кода. Такой подход дает хорошую возможность улучшить проект, поскольку вынуждает вас объяснить свои решения и позволяет окинуть взглядом весь проект «с высоты птичьего полета».

Если вы программист-интроверт, вы можете объяснять свой код резиновой уточке — эта техника известна как *метод утенка (rubber duck debugging)*[1].

Помимо разговора с коллегами (или резиновой уточкой) можно использовать и другие перечисленные здесь принципы создания чистого кода, чтобы время от времени быстро оценивать качество своего кода. В процессе вы, скорее всего, обнаружите нюансы, которые можно быстро исправить, чтобы значительно снизить сложность за счет очистки кодовой базы. Эта неотъемлемая часть процедуры разработки ПО значительно улучшает результат.

---

[1] Метод утенка — психологический метод, при котором решение трудных вопросов делегируется воображаемому помощнику, роль которого может сыграть игрушечный утенок (или любой другой предмет). Суть в том, что, сформулировав вопрос и объяснив «слушателю» проблему, программист может сам найти решение. — *Примеч. ред.*

Заключение

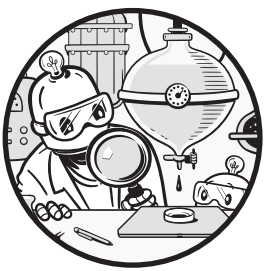
В этой главе вы изучили 17 принципов написания чистого и простого кода. Вы узнали, что чистый код уменьшает сложность и повышает вашу продуктивность, а также способствует масштабируемости и удобству сопровождения проекта. Вы усвоили, что следует использовать библиотеки везде, где только возможно, чтобы избежать перегруженности и повысить качество кода. Вы поняли, что чрезвычайно важно соблюдать стандарты и выбирать осмысленные имена переменных и функций для того, чтобы исключить недовольство со стороны будущих читателей вашего кода. Вы научились разрабатывать функции, направленные на выполнение только одной задачи. Кроме того, вы поняли, что снижение сложности и повышение масштабируемости за счет минимизации зависимостей (согласно закону Деметры) достигается, если избегать прямого или косвенного формирования цепочек методов. Вы научились писать комментарии к коду таким образом, чтобы можно было заглянуть в ваши мысли, а также избегать ненужных или тривиальных комментариев. И, самое главное, вы поняли, что ключ, открывающий дверь к суперсиле чистого кода, в том, чтобы писать код для людей, а не для машин.

Постепенно совершенствуйте навыки написания чистого кода, сотрудничая с крутыми разработчиками, читая их публикации на GitHub и изучая рекомендации по работе с вашим языком программирования. Интегрируйте в свою среду линтер, который динамически проверит ваш код на соответствие лучшим практикам. Время от времени возвращайтесь к указанным принципам чистого кода и проверяйте свой текущий проект на соответствие им.

В следующей главе вы познакомитесь еще с одним врагом эффективного программирования, который идет вразрез с простым написанием чистого кода, — преждевременной оптимизацией. Вы удивитесь, сколько времени и усилий тратят впустую программисты, которые еще не поняли, что *преждевременная оптимизация — это корень всех зол*!

5

Преждевременная оптимизация — корень всех зол



В этой главе вы узнаете, как преждевременная оптимизация может снизить вашу продуктивность. *Преждевременная оптимизация* — это пустая трата ценных ресурсов (времени, усилий, строк кода) на ненужную оптимизацию кода, поскольку вы к тому моменту еще не получили всю необходимую информацию. Это одна из основных проблем плохо написанного кода. Существует несколько типов преждевременной оптимизации; в этой главе мы познакомимся с некоторыми наиболее актуальными. Мы рассмотрим практические примеры такой оптимизации, которые будут полезны при разработке ваших собственных проектов. Завершим главу советами по отладке производительности, предварительно убедившись, что она *не* преждевременна.

Шесть типов преждевременной оптимизации

В оптимизированном коде как таковом нет ничего плохого, но за все всегда приходится платить, будь то дополнительное время программирования или лишние строки. Когда вы оптимизируете сниппеты, вы, как правило, жертвуете простотой ради производительности. В ряде случаев можно одновременно добиться и низкой сложности, и высокой эффективности, например, написав чистый код, но для этого вам придется потратить много времени! Если вы начнете оптимизацию слишком рано, вы рискуете потратить время на код, который, возможно, никогда не будет использоваться на практике или мало повлияет на общее время выполнения программы. Также в этом случае вам придется проводить оптимизацию в условиях недостатка информации о том, когда этот код вызывается и каковы возможные входные значения. Трата таких ценных ресурсов, как время и строки кода, может снизить вашу продуктивность на порядки, поэтому важно уметь грамотно их использовать.

Однако не стоит верить мне на слово. Вот что говорит о преждевременной оптимизации один из самых влиятельных ученых-компьютерщиков всех времен Дональд Кнут (Donald Knuth):

> *Программисты тратят огромное количество времени, размышляя или беспокоясь о скорости работы некритичных частей своих программ, а на самом деле такие попытки повысить эффективность приводят к негативному результату с учетом дальнейшей отладки и поддержки кода. Нас в 97 % случаев не должна сильно заботить низкая эффективность: преждевременная оптимизация — корень всех зол.*[1]

Преждевременная оптимизация принимает различные формы, поэтому для изучения этого вопроса мы рассмотрим шесть наиболее

[1] «Structured Programming with *go to* Statements», *ACM Computing Surveys* 6, no. 1 (1974).

распространенных случаев, с которыми сталкивался я и в которых вы тоже можете погрязнуть, поддавшись искушению преждевременно повысить эффективность приложения, тем самым замедлив процесс разработки.

Оптимизация функций кода

Прежде чем вы узнаете, как часто будут использоваться конкретные функции кода, старайтесь не тратить время на их оптимизацию. Допустим, вы столкнулись с функцией, которую вы просто не можете оставить неоптимизированной. Вы говорите себе, что применять примитивные методы — это плохой стиль программирования и что для решения этой задачи следует использовать более эффективные структуры данных или алгоритмы. Вы погружаетесь в исследование и проводите часы, изучая и настраивая алгоритмы. Но потом выясняется, что в итоговом проекте эта функция выполняется всего несколько раз, и ваша оптимизация не приводит к существенному повышению производительности.

Оптимизация функциональности

Избегайте добавления возможностей, которые не являются строго необходимыми, и не тратьте время на их оптимизацию. Предположим, вы разрабатываете приложение для смартфона, которое переводит текст в азбуку Морзе, отображаемую миганием фонарика. Из главы 3 вы узнали, что лучше всего сначала реализовать MVP, а не создавать отточенный конечный продукт со многими, возможно, ненужными функциями. В данном случае MVP будет простым приложением с одной функцией: перевод текста в азбуку Морзе после ввода фраз через простую форму и нажатия на кнопку. Однако вы считаете, что правило MVP неприменимо к вашему проекту, и решаете добавить несколько дополнительных функций: конвертер текста в звук и ресивер, переводящий световые сигналы в текст. После выпуска приложения вы понимаете, что ваша аудитория никогда не использует эти функции. Преждевременная оптимизация значительно

замедлила цикл разработки вашего продукта и помешала вам быстро учесть отзывы пользователей.

Оптимизация планирования

Если вы проводите преждевременную оптимизацию на этапе планирования, пытаясь найти решения еще не возникших проблем, вы рискуете запоздать с получением полезной обратной связи. Безусловно, не стоит полностью избегать планирования, однако застревание на этом этапе может оказаться не менее затратным. Чтобы выпустить что-то действительно стоящее в реальный мир, вы должны уметь принимать несовершенство. Вам нужны отзывы пользователей и проверка работоспособности от тестировщиков, чтобы понять, на чем следует сосредоточиться. Планирование может помочь вам избежать некоторых подводных камней, но, если вы не начнете действовать, вы никогда не закончите свой проект и останетесь в мире теории, оторванной от жизни.

Оптимизация масштабируемости

Оптимизация масштабируемости приложения, проведенная раньше, чем вы получите реальное представление о целевой аудитории, может стать серьезным отвлекающим фактором и повлечь затраты времени разработчиков и сервера на десятки тысяч долларов. В расчете на миллионы пользователей вы пытаетесь создать распределенную архитектуру; при необходимости она должна динамически добавлять виртуальные машины, чтобы справиться с пиковой нагрузкой. Однако разработка распределенных систем — сложная и подверженная ошибкам задача, на реализацию которой могут уйти месяцы. Многие проекты терпят неудачу; а если вы все-таки добьетесь успеха, который представляется вам в самых смелых мечтах, то у вас будет достаточно времени для масштабирования системы по мере роста спроса на продукт. Хуже всего, что ваша распределенная архитектура может *снизить* масштабируемость приложения из-за увеличения накладных расходов на коммуникацию и согласованность данных.

Масштабируемые распределенные системы имеют свою цену — вы уверены, что хотите ее платить? Не стремитесь масштабировать систему на миллионы пользователей, пока не обслужите первого.

Оптимизация разработки тестов

Преждевременная оптимизация тестов — это один из основных факторов, приводящих к напрасной трате времени разработчика. У процесса разработки, ориентированной на тестирование, есть много ревностных последователей, которые неправильно понимают идею *проведения тестов до реализации функциональности* как необходимость всегда первым делом писать тесты — даже если функция кода экспериментальна или изначально не вполне пригодна для тестирования. Написание экспериментального кода — это проверка концепций и идей, а добавление к нему лишнего уровня тестирования может затормозить продвижение проекта, а кроме того, не укладывается в философию быстрого прототипирования. Предположим, вы строго следуете принципам разработки через тестирование и настаиваете на 100-процентном покрытии тестами. При этом некоторые функции кода, например те, которые обрабатывают произвольный пользовательский текст, не слишком хорошо проходят юнит-тесты из-за непредсказуемости входных данных. Эффективно такие функции может протестировать только человек — в этом случае реальные пользователи и являются единственным значимым тестом. Тем не менее вы все равно проводите преждевременную оптимизацию для идеального прохождения юнит-тестов. Такой подход не имеет смысла: он замедляет цикл разработки, внося при этом ненужную сложность.

Оптимизация объектно-ориентированной картины мира

Объектно-ориентированные подходы зачастую несут с собой излишнюю сложность и преждевременную «концептуальную» оптимизацию. Предположим, вы хотите смоделировать мир своего приложения с помощью сложной иерархии классов. Вы пишете

небольшую игру об автомобильных гонках. Вы создаете иерархию классов, в которой класс `Porsche` наследует от класса `Car`, который в свою очередь наследует от класса `Vehicle`. Ведь каждый Porsche — это автомобиль, а каждый автомобиль — это транспортное средство. Однако многоуровневая иерархия классов приводит к усложнению кодовой базы, и другим программистам будет трудно разобраться, что делает ваш код. Нередко такие типы многоуровневых структур наследования привносят излишнюю сложность. Избегайте их, используя идею MVP: начинайте с самой простой модели и расширяйте ее только при необходимости. Не оптимизируйте свой код под моделирование мира с бóльшим количеством деталей, чем вашему приложению нужно на самом деле.

Преждевременная оптимизация: пример

Теперь, когда у вас есть общее представление о проблемах, к которым может привести преждевременная оптимизация, давайте напишем небольшую программу на Python. Мы посмотрим в реальном времени, как такая оптимизация приводит к излишней сложности кода небольшого приложения для отслеживания транзакций, которое не требует масштабирования до тысяч пользователей.

Алиса, Боб и Карл играют в покер каждую пятницу вечером. После нескольких раундов они решают, что им нужно разработать систему для отслеживания долга участников после каждой ночи игры. Алиса — увлеченный программист, она создает небольшое приложение, которое отслеживает баланс каждого игрока, как показано в листинге 5.1.

Листинг 5.1. Простой скрипт для отслеживания транзакций и балансов

```
transactions = []
balances = {}


❶ def transfer(sender, receiver, amount):
      transactions.append((sender, receiver, amount))
      if not sender in balances:
```

```
            balances[sender] = 0
        if not receiver in balances:
            balances[receiver] = 0
    ❷ balances[sender] -= amount
        balances[receiver] += amount

    def get_balance(user):
        return balances[user]
    def max_transaction():
        return max(transactions, key=lambda x:x[2])


❸ transfer('Alice', 'Bob', 2000)
❹ transfer('Bob', 'Carl', 4000)
❺ transfer('Alice', 'Carl', 2000)

    print('Balance Alice: ' + str(get_balance('Alice')))
    print('Balance Bob: ' + str(get_balance('Bob')))
    print('Balance Carl: ' + str(get_balance('Carl')))

    print('Max Transaction: ' + str(max_transaction()))

❻ transfer('Alice', 'Bob', 1000)
❼ transfer('Carl', 'Alice', 8000)

    print('Balance Alice: ' + str(get_balance('Alice')))
    print('Balance Bob: ' + str(get_balance('Bob')))
    print('Balance Carl: ' + str(get_balance('Carl')))

    print('Max Transaction: ' + str(max_transaction()))
```

В скрипте есть две глобальные переменные — `transactions` и `balances`. Список `transactions` отслеживает транзакции между участниками в течение игрового вечера по мере их совершения. Каждая транзакция представляет собой кортеж из идентификаторов `sender` и `receiver`, а также величины `amount`, которая должна быть переведена от отправителя получателю ❶. Словарь `balances` отслеживает текущий баланс игрока, то есть сопоставляет идентификатор пользователя с количеством имеющихся у него средств на основе транзакций на данный момент ❷.

Функция `transfer(sender, receiver, amount)` создает и сохраняет новую транзакцию в глобальном списке, а также создает новые

балансы для `sender` и `receiver`, если их еще нет, и обновляет балансы в соответствии с заданным значением `amount`. Функция `get_balance(user)` возвращает баланс пользователя, указанного в качестве аргумента, а `max_transaction()` перебирает все транзакции и возвращает ту, которая имеет максимальное значение в третьем элементе кортежа — сумме транзакции.

Изначально все балансы равны нулю. Приложение передает 2000 единиц от Алисы Бобу ❸, 4000 — от Боба Карлу ❹ и 2000 — от Алисы Карлу ❺. В этот момент Алиса должна 4000 (с отрицательным балансом –4000), Боб должен 2000, а Карл имеет 6000 единиц. После вывода максимальной транзакции Алиса переводит 1000 единиц Бобу ❻, а Карл переводит 8000 Алисе ❼. Теперь счета изменились: Алиса имеет 3000, Боб –1000, а Карл –2000 единиц. При этом приложение возвращает следующий результат:

```
Balance Alice: -4000
Balance Bob: -2000
Balance Carl: 6000
Max Transaction: ('Bob', 'Carl', 4000)
Balance Alice: 3000
Balance Bob: -1000
Balance Carl: -2000
Max Transaction: ('Carl', 'Alice', 8000)
```

Но Алиса недовольна этим приложением. Она понимает, что вызов функции `max_transaction()` приводит к повторяющимся вычислениям — поскольку функция вызывается дважды, скрипт два раза пролистывает список `transactions`, чтобы найти транзакцию с максимальной суммой.

Но при выполнении `max_transaction()` во второй раз она частично повторяет те же действия, перебирая все транзакции для нахождения максимума (включая те, для которых он уже известен, то есть первые три транзакции ❸–❺). Алиса справедливо считает, что здесь есть *перспектива для оптимизации* — можно ввести новую переменную `max_transaction`, которая отслеживает максимальную транзакцию на данный момент всякий раз при создании новой транзакции.

В листинге 5.2 показаны три строки кода, добавленные Алисой для реализации этого изменения.

Листинг 5.2. Реализованная оптимизация для сокращения повторяющихся вычислений

```
transactions = []
balances = {}
max_transaction = ('X', 'Y', float('-Inf'))


def transfer(sender, receiver, amount):
...
    if amount > max_transaction[2]:
        max_transaction = (sender, receiver, amount)
```

Переменная `max_transaction` хранит максимальную сумму транзакции среди всех транзакций, наблюдавшихся на данный момент. Таким образом, нет необходимости пересчитывать максимум после каждой игровой ночи. Первоначально вы устанавливаете в качестве максимального значения отрицательную бесконечную величину, чтобы первая реальная транзакция непременно была больше. Каждый раз при добавлении новой транзакции программа сравнивает ее значение с текущим максимумом, и если оно больше, то она сама становится транзакцией с максимальным значением. Без оптимизации при вызове функции `max_transaction()` 1000 раз для списка из 1000 транзакций вам пришлось бы выполнить 1 000 000 сравнений, чтобы найти 1000 максимумов, поскольку перебирался бы список из 1000 элементов 1000 раз (1000 * 1000 = 1 000 000). После оптимизации вам нужно извлекать текущее сохраненное в `max_transaction` значение только один раз при каждом вызове функции. Список состоит из 1000 элементов, и, чтобы определить текущий максимум, потребуется не более 1000 операций. Это приводит к сокращению количества необходимых операций на три порядка.

Многие разработчики кода не могут устоять перед реализацией таких оптимизаций, но при этом сложность накапливается. Например, Алисе вскоре придется фиксировать ряд дополнительных

переменных для отслеживания новой статистики, которая может заинтересовать ее друзей, например `min_transaction`, `avg_transaction`, `median_transaction` и `alice_max_transaction` (для учета ее собственного максимального значения транзакции). Каждая из них повлечет за собой появление в проекте нескольких дополнительных строк кода, увеличивая вероятность появления бага. Например, если Алиса забудет обновить переменную в нужном месте, ей придется потратить драгоценное время на исправление. Хуже того, она может пропустить этот баг, в результате чего данные о балансе в аккаунте Алисы будут испорчены, а ущерб составит несколько сотен долларов. Ее друзья могут даже заподозрить, что Алиса специально написала код в свою пользу! Последнее замечание может показаться несколько ироничным, но в реальных условиях ставки могут быть высоки. Последствия второго порядка от излишней сложности могут быть даже серьезнее, чем более предсказуемые последствия первого порядка.

Всех этих потенциальных проблем можно было бы избежать, если бы Алиса воздержалась от применения *возможной оптимизации*, предварительно хорошенько подумав, а не является ли она преждевременной. Цель приложения — учитывать транзакции одного игрового вечера между тремя друзьями. В реальности будет всего несколько сотен транзакций и дюжина обращений к `max_transaction`, а не тысячи, на которые рассчитан оптимизированный код. Компьютер Алисы мог бы выполнить неоптимизированный код за доли секунды, и ни Боб, ни Карл даже не поняли бы, что код не оптимизирован. Кроме того, неоптимизированный код проще и его легче поддерживать.

Однако предположим, что срабатывает сарафанное радио, и казино, которое полагается на высокую производительность, масштабируемость и долгосрочные истории транзакций, обращается к Алисе с просьбой внедрить такую же систему. В этом случае она по-прежнему может устранить узкое место, связанное с постоянным пересчетом максимума вместо его быстрого отслеживания. Но теперь она точно будет уверена, что дополнительное усложнение кода

действительно принесет пользу. Избегая оптимизации до тех пор, пока этого не потребует приложение, она избавит себя от массы ненужных преждевременных оптимизаций.

Шесть советов по настройке производительности

Пример с Алисой не только детально продемонстрировал нам случай преждевременной оптимизации на практике, но и дал подсказку, как правильно оптимизировать. Важно помнить, что Дональд Кнут не утверждал, что оптимизация *сама по себе* является корнем всех зол. На самом деле настоящая проблема заключается именно в *преждевременности*. Сейчас, когда цитата Кнута стала довольно популярной, многие ошибочно воспринимают ее как аргумент против любой оптимизации. Однако когда оптимизация проводится в нужный момент, она может иметь решающее значение.

Быстрый технологический прогресс последних десятилетий во многом обусловлен оптимизацией: размещение схем на чипах, алгоритмы и юзабилити программного обеспечения постоянно оптимизировались в течение долгого времени. Закон Гордона Мура гласит, что усовершенствования в технологии компьютерных чипов, которые делают вычисления невероятно дешевыми и эффективными, будут продолжаться в геометрической прогрессии еще долгое время. Все нововведения в этой сфере имеют значительный потенциал, и их нельзя считать преждевременными. Оптимизация лежит в основе прогресса, поскольку она приносит пользу многим.

Как правило, ее нужно проводить, только когда у вас есть четкие доказательства (например, измерения с помощью инструментов для оптимизации производительности) того, что данная часть кода или функция действительно является узким местом и что пользователи приложения оценят или даже потребуют повысить производительность. Например, оптимизация скорости запуска Windows

не преждевременна, поскольку она принесет непосредственную выгоду миллионам человек. А вот оптимизация масштабируемости вашего веб-приложения с верхним пределом в 1000 пользователей в месяц (при этом им нужен только статический сайт) является преждевременной. Затраты времени на *разработку* приложения не так высоки, как на его *использование* тысячами людей. Если вы можете потратить один час своего времени, чтобы сэкономить пользователям несколько секунд, это чаще всего победа! Время ваших пользователей более ценно, чем ваше собственное. Вот почему мы в первую очередь используем компьютеры — чтобы потратить меньше ресурсов на начальном этапе и получить гораздо больше впоследствии. Оптимизация не всегда преждевременна. Иногда вы должны провести оптимизацию для того, чтобы создать полезный продукт, — зачем выпускать неоптимизированное ПО, не приносящее никакой ценности? Рассмотрев несколько причин отказа от преждевременной оптимизации, обратимся к шести советам по повышению производительности, которые помогут вам понять, как и когда оптимизировать код.

Сначала измеряйте, потом улучшайте

Измеряйте производительность вашего ПО, чтобы узнать, где можно и нужно его улучшить. Без этого оптимизация невозможна, поскольку вы не можете отследить результат.

В большинстве случаев преждевременность оптимизации состоит в том, что она происходит до измерений, именно это и лежит в основе утверждения о корне всех зол. Оптимизацию следует проводить только после того, как вы сможете измерить параметры производительности вашего неоптимизированного кода — объем памяти или скорость. Это и есть ваш бенчмарк (опорная точка). Бессмысленно пытаться улучшить, например, время выполнения, если вы не знаете его исходное значение. Невозможно определить, действительно ли ваша «оптимизация» увеличивает общее время работы или вовсе не дает никакого измеримого эффекта, если вы не имеете четкого ориентира.

В качестве общей стратегии для измерения производительности начните с написания максимально простого, примитивного кода, который легко читается. Вы можете назвать его *прототипом*, *наивным подходом* или *MVP*. Запишите свои измерения в таблицу. Это ваш первый бенчмарк. Создайте альтернативный код и рассчитайте его производительность по сравнению с исходным. Как только вы убедительно докажете, что ваша оптимизация повышает производительность, новый оптимизированный код станет вашим новым бенчмарком, который все последующие улучшения должны превзойти. Если оптимизация не способна заметно улучшить ваш код, откажитесь от нее.

Таким образом, вы будете отслеживать улучшение кода с течением времени. Кроме того, вы можете документально фиксировать процесс оптимизации, для того чтобы впоследствии обосновать и защитить ее необходимость перед боссом, коллегами или даже перед научным сообществом.

Принцип Парето — всему голова

Принцип 80/20, или *принцип Парето*, рассмотренный в главе 2, также применим и к оптимизации производительности. Поскольку некоторые функции отнимают значительно больше ресурсов (таких как время и объем памяти), чем другие, то фокус внимания на устранении таких узких мест поможет вам эффективно оптимизировать ваш код.

Чтобы проиллюстрировать высокую неравномерность распределения ресурсов между разными процессами, выполняющимися параллельно в операционной системе, обратимся к текущей загрузке моего центрального процессора, приведенной на рис. 5.1.

Если вы построите график в Python, то увидите распределение, подобное распределению Парето (рис. 5.2).

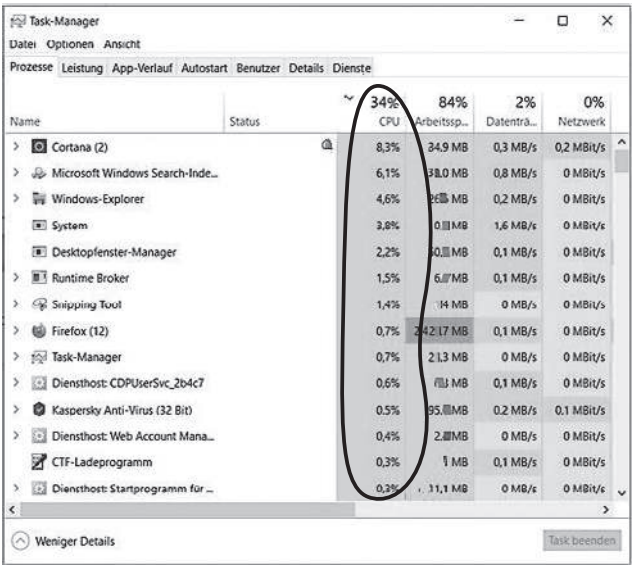


Рис. 5.1. Неравномерное распределение ресурсов ЦП между приложениями на компьютере с ОС Windows

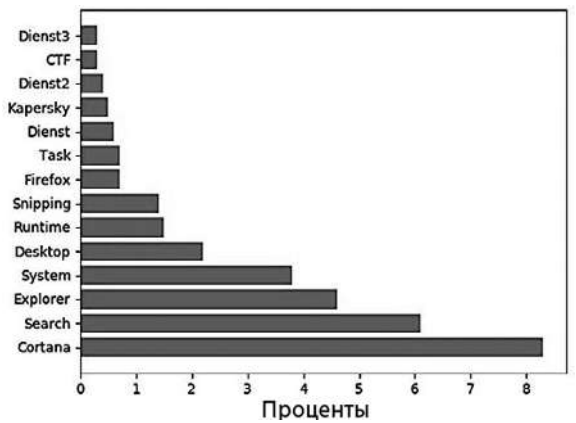


Рис. 5.2. Использование ЦП разными приложениями на компьютере с ОС Windows

Небольшое количество приложений требует значительного процента загрузки ЦП. Если я хочу понизить процент его использования, мне нужно просто закрыть приложения Cortana и Search, и, о чудо, значительная часть нагрузки на ЦП исчезает (рис. 5.3).

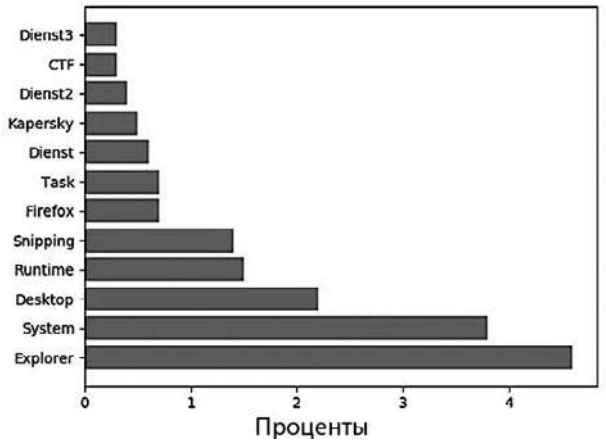


Рис. 5.3. Результаты после «оптимизации» системы Windows путем закрытия ненужных приложений

Удаление двух самых ресурсоемких задач значительно снижает загрузку процессора. При этом новый график на первый взгляд похож на предыдущий: две задачи, на этот раз Explorer и System, по-прежнему занимают намного больше ресурсов, чем остальные. Этот пример демонстрирует важное свойство настройки производительности: ее оптимизация фрактальна. Как только вы устраните одно узкое место, вы обнаружите, что поблизости скрывается другое. Узкие места всегда есть в любой системе, но, если вы будете устранять их по мере появления, вы получите максимальную отдачу от затраченных усилий. В реальном проекте кода вы увидите то же самое распределение, где относительно небольшое количество функций забирает бóльшую часть ресурсов (например, производительности ЦП). Часто вы можете сосредоточить усилия по оптимизации на проблемной функции, требующей максимального

количества ресурсов. Можно переписать ее с помощью более сложных алгоритмов или избежать некоторых вычислений (например, с помощью кэширования промежуточных результатов). Конечно, следующее узкое место появится сразу же после устранения текущего; вот почему нужно постоянно замерять производительность кода, чтобы решить, когда прекращать оптимизацию. Например, нет особого смысла улучшать время отклика веб-приложения с 2 мс до 1 мс, если пользователь все равно не почувствует разницы. Из-за фрактальной природы оптимизаций и принципа Парето (см. главу 2) получение этих небольших преимуществ часто требует больших затрат сил и времени разработчика и при этом дает незначительный выигрыш с точки зрения юзабилити или полезности приложения.

Алгоритмическая оптимизация приносит успех

Допустим, вы решили, что ваш код нуждается в оптимизации, поскольку отзывы пользователей и статистика показывают слишком медленную скорость работы приложения. Вы измерили текущую скорость в секундах или байтах, определили целевой показатель, к которому стремитесь, и нашли узкое место. Следующий шаг — выяснить, как преодолеть эту проблему.

Многие узкие места могут быть устранены путем корректировки *алгоритмов и структур данных*. Например, представьте, что вы разрабатываете финансовое приложение. Вы знаете, что самым проблемным местом является функция `calculate_ROI()`, которая перебирает все комбинации потенциальных транзакций (покупок и продаж) для расчета максимальной прибыли. Поскольку эта функция является узким местом всего приложения, вы хотите найти для нее наилучший алгоритм. После небольшого исследования вы обнаруживаете *алгоритм максимальной прибыли* (maximum profit algorithm) — простую и эффективную альтернативу, которая значительно ускорит ваши вычисления. Такое же исследование можно провести и в отношении структуры данных, создающей узкие места.

Чтобы избавиться от узких мест и оптимизировать производительность, спросите себя:

- Можете ли вы найти более совершенные алгоритмы, которые уже себя зарекомендовали, — например, в книгах, научных работах или даже в Википедии?
- Можете ли вы подстроить существующие алгоритмы под решение вашей конкретной задачи?
- Можете ли вы улучшить структуру данных? Некоторые распространенные простые решения включают использование множеств вместо списков (проверка принадлежности, например, намного быстрее происходит внутри множества, чем внутри списка) или словарей вместо наборов кортежей.

Время, потраченное на изучение этих вопросов, с лихвой окупится как для вашего приложения, так и конкретно для вас. В процессе работы вы станете более продвинутым специалистом в области computer science.

Да здравствует кэш!

После внесения всех необходимых изменений в соответствии с предыдущими советами вы можете перейти к этому легкому и нехитрому трюку для устранения ненужных вычислений: храните результаты ряда выполненных вычислений в кэше. Этот прием удивительно хорошо работает в самых разных приложениях. Перед любым новым вычислением вы сначала проверяете кэш: может быть, вы его уже выполняли. Это похоже на то, как вы подходите к несложным вычислениям в уме — в определенный момент вы на самом деле уже не вычисляете 6×5, а просто полагаетесь на свою память, которая сразу выдает результат. Следовательно, кэширование имеет смысл только в том случае, если промежуточные вычисления одного и того же типа встречаются в вашем приложении несколько раз. К счастью, это справедливо для большинства

реальных приложений. Например, тысячи пользователей могут смотреть одно и то же видео на YouTube в течение одного дня, поэтому кэширование видео рядом с пользователем (а не за тысячи миль в удаленном дата-центре) экономит ограниченные ресурсы пропускной способности сети.

Рассмотрим небольшой пример кода, в котором кэширование приводит к значительному повышению производительности: алгоритм Фибоначчи.

```
def fib(n):
    if n < 2:
        return n
    fib_n = fib(n-1) + fib(n-2)
    return fib_n


print(fib(100))
```

Алгоритм выводит результат многократного сложения последнего и предпоследнего элемента ряда до 100-го элемента последовательности:

```
354224848179261915075
```

Этот алгоритм работает медленно, потому что функции `fib(n-1)` и `fib(n-2)` вычисляют более или менее одинаковые вещи. Например, обе по отдельности вычисляют `(n-3)`-й элемент ряда Фибоначчи, вместо того чтобы повторно использовать результат друг друга. Избыточность нарастает — даже при таком простом вызове функции вычисления занимают слишком много времени.

Одним из способов повышения производительности в данном случае является использование кэша. *Кэширование* позволяет сохранять результаты предыдущих вычислений, и в этом случае значение `fib2(n-3)` рассчитается только один раз, а когда оно понадобится снова, вы сможете мгновенно извлечь ответ из кэша.

В Python мы можем реализовать простой кэш, создав словарь, в котором все входные данные функции (например, в виде строки ввода) ассоциируются с выходными. Затем вы можете вызвать из кэша уже выполненные вычисления.

Вот вариант с применением кэширования для расчета последовательности Фибоначчи на Python:

```
cache = dict()

def fib(n):
    if n in cache:
        return cache[n]
    if n < 2:
        return n
    fib_n = fib(n-1) + fib(n-2)
  ❶ cache[n] = fib_n
    return fib_n


print(fib(100))
# 354224848179261915075
```

Вы сохраняете результат `fib(n-1) + fib(n-2)` в кэше ❶. Если у вас уже есть результат вычисления `n`-го числа Фибоначчи, вы берете его из кэша, а не рассчитываете его снова и снова. На моей машине это увеличивает производительность почти в 2000 раз при вычислении первых 40 чисел Фибоначчи!

Существует две основные стратегии эффективного кэширования:

Выполнять вычисления заранее («офлайн») и хранить результаты в кэше

Это отличная стратегия для веб-приложений: вы можете заполнить большой кэш единожды (или раз в день), а затем предоставлять результаты предварительных вычислений пользователям. Для них ваши расчеты будут казаться молниеносными. Картографические сервисы активно используют этот прием для ускорения вычислений кратчайшего пути.

Выполнять вычисления по ходу («онлайн») и сохранять результаты в кэше

Примером может служить онлайн-проверка биткойн-адресов, которая суммирует все входящие и вычитает все исходящие транзакции для вычисления баланса биткойн-адреса. После этого она может кэшировать промежуточные результаты для конкретного адреса, чтобы избежать избыточного вычисления этих транзакций при повторной проверке тем же пользователем. Эта интерактивная форма является самой простой формой кэширования, при которой вам не нужно заранее решать, какие вычисления выполнять.

В обоих случаях чем больше вычислений вы храните в кэше, тем выше вероятность *попаданий в кэш (cashe hits)*, когда результат может быть получен немедленно. Однако, поскольку обычно объем хранимых в кэше записей ограничен размером памяти, вам потребуется разумная *политика замены кэша (cashe replacement policy)*: кэш конечен и может быстро заполниться. В этом случае он сможет сохранить новое значение, только заменив старое. Распространенной стратегией замены записей является принцип *«первым пришел — первым ушел» (first in, first out; FIFO)*, при котором самая старая запись кэша заменяется новой. Выбор лучшей стратегии зависит от конкретного приложения, но принцип FIFO — хорошее начало.

Лучше меньше, да лучше

Ваша проблема слишком сложна и ее нельзя успешно решить? Упростите ее! Это звучит очевидно, но очень многие разработчики кода — перфекционисты. Они мирятся с невероятной сложностью и вычислительными затратами только для того, чтобы реализовать небольшую функцию, которая, возможно, даже не будет замечена пользователями. Вместо того чтобы оптимизировать, зачастую гораздо полезнее понизить сложность и избавиться от ненужных функций и вычислений. Рассмотрим проблемы, с которыми сталкиваются разработчики поисковых систем, например: что именно

будет идеально соответствовать данному поисковому запросу? Найти оптимальное решение для такой задачи чрезвычайно сложно, ведь требуется произвести поиск по миллиардам веб-сайтов. Однако поисковые системы, такие как Google, не стараются решить эту проблему наилучшим образом; скорее они делают все возможное за то время, которым располагают, используя эвристические методы. Они не проверяют миллиарды сайтов по поисковому запросу пользователя, а фокусируются на паре наиболее вероятных вариантов, используя приближенные эвристические процедуры для оценки авторитетности отдельных сайтов (например, известный алгоритм PageRank). Если ни один из наиболее авторитетных сайтов не отвечает запросу, они обращаются к условно оптимальным сайтам. В большинстве случаев вам тоже стоит использовать эвристические методы, а не самые лучшие алгоритмы. Задайте себе следующие вопросы: что на данный момент является узким местом в моих вычислениях? почему оно возникло? стоит ли вообще пытаться решить эту проблему? может, убрать эту функцию или предложить ее сокращенную версию? Если функцию использует 1 % пользователей, а 100 % сталкиваются с увеличением времени ожидания, возможно, настало время для некоторого минимализма (то есть для удаления функции, которая почти не применяется, да еще и доставляет неудобства людям).

Чтобы упростить код, подумайте, есть ли смысл в одном из следующих действий:

- Полностью устранить текущее узкое место, просто опустив функцию.
- Снизить сложность задачи, заменив ее более простой версией той же задачи.
- Избавиться от одной сложной функции, чтобы добавить десять простых, в соответствии с принципом 80/20.
- Отказаться от одной значимой функции, чтобы иметь возможность реализовать более важную; подумайте о цене вопроса.

Знай меру

Оптимизация производительности может стать одним из самых трудоемких этапов написания кода. Всегда есть куда совершенствоваться, но ваши усилия, направленные на улучшение ситуации, имеют тенденцию нарастать как снежный ком по мере того, как вы исчерпываете все лежащие на поверхности возможности. В какой-то момент это становится просто пустой тратой времени.

Регулярно спрашивайте себя: «Стоит ли продолжать оптимизацию?» Как правило, ответ можно найти, если внимательно изучить потребности пользователей вашего приложения. Какая производительность им нужна? Ощущают ли они разницу между оригинальной и оптимизированной версией? Жалуется ли кто-то из них на низкую производительность? Ответив на эти вопросы, вы получите приблизительную оценку предельно допустимого времени работы приложения. Теперь вы можете начать оптимизировать наиболее проблемные места, пока не достигнете этого предела. Затем остановитесь.

Заключение

В этой главе вы узнали, почему важно избегать преждевременной оптимизации. Она преждевременна, если отнимает больше усилий, чем добавляет пользы. В зависимости от проекта ценность оптимизации можно измерить в терминах времени на разработку, параметров юзабилити, ожидаемой выгоды от приложения или функции либо полезности для узкой группы пользователей. Например, если оптимизация может сэкономить время или деньги тысячам пользователей, она, скорее всего, не является преждевременной, даже если вам придется потратить значительные ресурсы разработчиков на улучшение кодовой базы. Однако если оптимизация не может ощутимо изменить качество жизни пользователей или программистов, то, скорее всего, она преждевременна. Несомненно, существует много более продвинутых моделей разработки ПО, но здравый

смысл и простая осведомленность об опасности преждевременной оптимизации позволяют обойтись без изучения модных книг или научных работ на эту тему. Например, полезный проверенный метод заключается в том, чтобы начать с написания легкочитаемого и чистого кода, не слишком заботясь о производительности, а затем оптимизировать участки с высоким потенциалом — на базе опыта, достоверной информации, полученной с помощью инструментов отслеживания производительности, и реальных результатов пользовательских исследований.

В следующей главе вы узнаете о состоянии потока — лучшем друге программиста.

6

Состояние потока

Состояние потока — это исходный код максимальной производительности человека.

Стивен Котлер (Steven Kotler)

В этой главе вы узнаете о концепции состояния потока и о том, как с его помощью можно повысить продуктивность программирования. Многие работают в офисной среде с постоянными перерывами, совещаниями и другими отвлекающими факторами, которые могут сделать практически невозможным достижение настоящего состояния продуктивного программирования. Чтобы лучше понять, что такое состояние потока и как его достичь на практике, мы рассмотрим в этой главе ряд примеров, но, по существу, *состояние потока* — это состояние чистой концентрации и сосредоточенности — то, что некоторые называют «быть в ударе».

Состояние потока — это не узкоспециализированное понятие, а концепция, которую можно применить к решению задач в любой

области. В данной главе мы выясним, как достичь состояния потока и чем оно полезно.

Что такое состояние потока?

Концепция потока была популяризирована Михаем Чиксентмихайи (Mihaly Csikszentmihalyi), заслуженным профессором психологии и менеджмента Университета Клермонт-Градуэйт и бывшим заведующим кафедрой психологии Чикагского университета. В 1990 году Чиксентмихайи опубликовал революционную книгу о работе всей своей жизни под названием «Flow»[1].

Но что такое состояние потока? Начнем с несколько субъективного описания того, как это ощущается. Позднее мы дадим более конкретное определение этого состояния, основанное на том, что можно измерить — вам как разработчику кода больше понравится второй вариант!

Ощущение потока — это состояние полного погружения в выполняемую задачу: фокус и концентрация. Вы забываете о времени; вы максимально эффективны и внимательны. Вы можете испытывать чувство экстаза, освобождаясь от всех других тягот повседневной жизни. Ясность мысли необыкновенная, вам очевидно, что нужно делать дальше — ваши действия естественным образом перетекают одно в другое. Уверенность в собственной способности завершить следующее задание непоколебима. Успешное завершение само по себе является наградой, и вы наслаждаетесь каждой секундой работы. И ваша производительность, и ваши результаты бьют все рекорды.

Согласно психологическим исследованиям, которые проводились под руководством Чиксентмихайи, состояние потока состоит из шести компонентов:

---

[1] Чиксентмихайи М. «Поток. Психология оптимального переживания».

Внимание. Вы ощущаете глубокое чувство концентрации и полного сосредоточения.

Действие. Вы чувствуете острую потребность в действии; быстро и эффективно продвигаетесь вперед в выполнении текущей задачи — ваше сфокусированное сознание дает дополнительный импульс работе. Каждое новое движение подпитывает следующее, создавая поток успешных действий.

Собственное «я». Вы меньше осознаете себя, отключаете самокритику, сомнения и страхи. Вы меньше думаете о себе (*рефлексия*) и больше о поставленной задаче (*действие*). Вы теряете свое «я» в процессе работы.

Контроль. Даже при том что вы все больше отрешаетесь от себя, вы наслаждаетесь повышенным чувством контроля над ситуацией, что придает вам спокойную уверенность и позволяет мыслить нестандартно и предлагать творческие решения.

Время. Вы теряете способность ощущать течение времени.

Вознаграждение. Вы хотите заниматься только своим делом; вы не ждете внешнего вознаграждения, ведь погружение в работу уже само по себе приносит удовлетворение.

Понятия *«поток»* и *«внимание»* тесно связаны. В диссертации 2013 года, посвященной синдрому дефицита внимания и гиперактивности (СДВГ), Рони Склар (Rony Sklar) указывает на то, что термин *«дефицит внимания»* ошибочно подразумевает, что пациенты, страдающие этим синдромом, не могут сосредоточиться. Другой термин для обозначения состояния потока — *«гиперфокус»*, и огромное количество исследователей в области психологии (например, Кауфман и др.; Kaufmann et al. 2000) доказали, что люди с СДВГ вполне способны к гиперфокусу; они лишь испытывают трудности с поддержанием внимания при выполнении задач, не приносящих внутреннего удовлетворения. Необязательно иметь диагноз СДВГ, чтобы понять: трудно сфокусироваться на том, что вам неинтересно.

Но если вы когда-нибудь полностью погружались в увлекательную игру, создание забавного приложения или просмотр интересного фильма, вы знаете, как легко достичь состояния потока, если вам нравится ваше занятие. В таком состоянии ваш организм выделяет пять нейрохимических веществ — *«гормонов счастья»*, таких как эндорфины, дофамин и серотонин. Это похоже на «кайф» от приема легких наркотиков, но без определенных негативных последствий — даже Чиксентмихайи предупреждал, что состояние потока может вызывать привыкание. Научившись входить в него, вы станете умнее и продуктивнее, — если вам удастся направить свою активность на полезные занятия, такие как программирование.

Теперь вы можете воскликнуть: «Перейдем, наконец, к сути — как мне добиться состояния потока?» Давайте ответим на этот вопрос!

Как достичь состояния потока

Чиксентмихайи выделил три условия для достижения состояния потока: (1) четко сформулированные цели, (2) оперативный механизм обратной связи в вашей среде и (3) наличие баланса между потенциалом и возможностями.

Четко поставленные цели

Если вы пишете код, у вас должна быть четкая цель, на достижение которой направлены все более мелкие задачи. В состоянии потока каждое действие естественным образом ведет к следующему, а оно, в свою очередь, ведет к еще одному и т. д.; поэтому необходима ясная конечная цель. Люди часто достигают состояния потока, играя в компьютерные игры, потому что если вы преуспеваете в мелких действиях — например, перепрыгиваете через движущееся препятствие, — вы в конечном итоге достигаете большой цели — проходите уровень. Чтобы использовать потоковое состояние для повышения продуктивности программирования, вы должны четко понимать цель проекта. Каждая строчка кода приближает вас к успешному

завершению крупного проекта. Отслеживание написанных вами строк — один из способов придать работе над кодом характер игры!

Механизм обратной связи

Механизм обратной связи служит для того, чтобы поощрять правильное поведение и наказывать за неправильное. Специалисты по машинному обучению знают, что для обучения высокоэффективных моделей им необходим отличный механизм обратной связи. Например, можно научить робота ходить, вознаграждая его за каждую секунду, когда он не падает, и указав ему, что нужно оптимизировать свои действия для получения максимального общего вознаграждения. После этого робот будет автоматически корректировать свои действия, чтобы со временем получить самую большую награду. Мы, люди, ведем себя примерно так же, когда учимся чему-то новому. Мы ищем одобрения родителей, учителей, друзей или наставников — даже соседей, которые нам не нравятся, и корректируем свои действия, чтобы добиться максимального признания и минимизировать (общественное) порицание. Так мы приучаемся к совершению определенных действий и воздерживаемся от других. Получение обратной связи жизненно важно при таком способе обучения.

Обратная связь — это непременное условие для достижения состояния потока. Чтобы чаще входить в него во время работы, старайтесь получать больше обратной связи. Еженедельно встречайтесь с партнерами по проекту, чтобы обсудить ваш код и цели проекта, и в дальнейшем учитывайте их мнение. Опубликуйте свой код на Reddit или StackOverflow и попросите оставить отзыв. Публикуйте MVP как можно раньше и чаще, чтобы получать обратную связь от пользователей. Стремление получить отклик на свои программные решения действует безотказно (даже если вы получаете отложенное удовольствие), поскольку это повышает вашу вовлеченность в деятельность, приводящую к получению отзывов. После публикации Finxter, моего приложения для изучения Python, я стал получать нескончаемый поток откликов от пользователей, и меня это зацепило.

Обратная связь заставляла меня возвращаться к работе над кодом и позволяла мне раз за разом входить в состояние потока, улучшая приложение.

Баланс между потенциалом и возможностями

Поток — это активное психическое состояние. Если задача слишком проста, вам станет скучно, и вы быстро потеряете чувство увлеченности. Если она слишком трудна, вы бросите работу раньше срока. Задача должна быть достаточно сложной и при этом посильной.

На рис. 6.1 показан спектр возможных состояний человека; это изображение взято из первоначального исследования Чиксентмихайи.

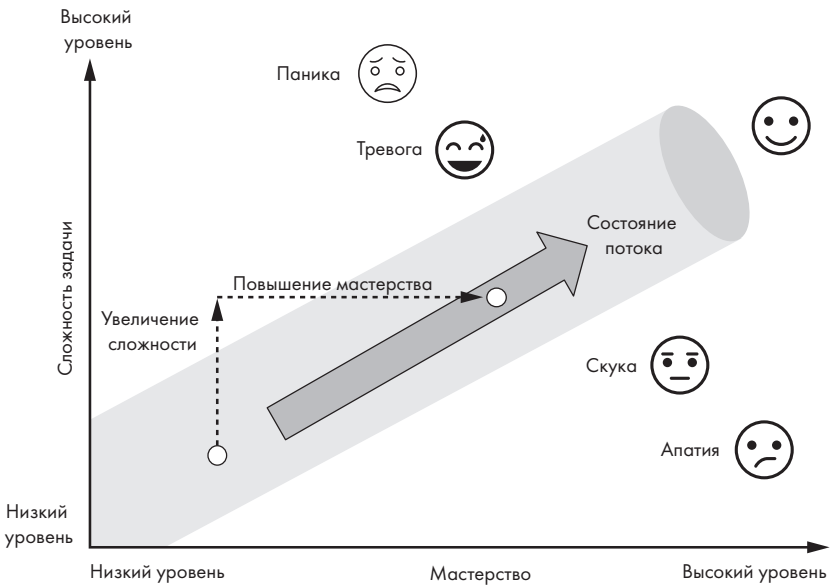


Рис. 6.1. В состоянии потока вы ощущаете, что задача не слишком трудная, но и не слишком легкая, учитывая ваш текущий уровень мастерства

На оси *X* представлен уровень мастерства от низкого к высокому, а по оси *Y* так же показана сложность рассматриваемой задачи. Следовательно, если задача слишком трудна для вашего уровня навыка, вы впадете в панику, а если чересчур легка — в апатию. Но если сложность соответствует вашей квалификации, вероятность достижения состояния потока максимальна.

Хитрость в том, чтобы постоянно искать более сложные задачи, не доходя до уровня тревоги, и соответственно повышать свой уровень мастерства. Такой процесс обучения позволяет увеличивать продуктивность и мастерство, одновременно получая удовольствие от работы.

Советы по достижению состояния потока для программистов

В своей информационной статье 2015 года под названием «Crafting Fun User Experiences: A Method to Facilitate Flow» («Создание увлекательного пользовательского опыта: способ поддержания состояния потока») Оуэн Шаффер (Owen Schaffer) определил семь условий достижения ощущения потока. Нужно: (1) знать, что делать; (2) иметь представление, как это делать; (3) быть в курсе, насколько хорошо вы справляетесь с поставленной задачей; (4) понимать, куда двигаться дальше; (5) искать новые вызовы; (6) работать над своими навыками, чтобы иметь возможность решать более сложные задачи, и (7) уметь ограждать себя от отвлекающих факторов (Human Factors International). Исходя из этих условий и собственных соображений, я отобрал несколько полезных советов и тактических приемов для достижения состояния потока, которые лучше всего подходят разработчикам кода.

Всегда работайте над реальным проектом кода, не тратьте время на бесцельное обучение. Вы будете быстрее усваивать новую информацию, если она реально касается того, что вам небезразлично. Я рекомендую разделить время обучения так: 70 % — это

работа над реальным интересным проектом по вашему выбору и 30 % — это чтение книг и учебников или прохождение обучающих курсов. Из собственного опыта общения и переписки с десятками тысяч разработчиков в сообществе Finxter я понял, что значительная часть студентов, изучающих программирование, имеет опыт зацикливания и застревания в процессе обучения. В этом состоянии люди все время чувствуют себя не готовыми к реальному проекту. Это всегда одна и та же история: такие горе-разработчики навсегда застревают в теории программирования, бесконечно учатся, не имея практики, что заставляет их еще сильнее ощущать ограниченность знаний, — негативная спираль закручивается до состояния полного бессилия. Выход — поставить перед собой четкую цель и довести проект до конца, несмотря ни на что. Это совпадает с одним из трех необходимых условий вхождения в состояние потока.

Работайте над интересными проектами, которые способствуют достижению вашей цели. Состояние потока — это состояние эмоционального подъема, поэтому вы должны быть в восторге от работы. Если вы профессиональный программист, уделите время обдумыванию вашей конечной цели. Определите полезные стороны вашего проекта. Если вы учитесь писать код, вам повезло — вы можете выбрать интересную задачу, которая вас увлечет! Работайте над проектами, которые важны для вас. Вы получите больше удовольствия, повысите вероятность успеха и обретете устойчивость к временным неудачам. Если вы просыпаетесь и не можете дождаться того момента, когда сможете поработать над своим проектом, знайте, что состояние потока уже близко.

Реализуйте свои сильные стороны. Этот совет от Питера Друкера, консультанта по менеджменту, — просто золото. Всегда будет больше областей, где вы слабы, чем тех, в которых вы сильны. В большинстве сфер деятельности ваши навыки ниже среднего. Если вы сосредоточитесь на своих слабостях, вы практически гарантированно потерпите неудачу. Вместо этого сфокусируйтесь

на собственных преимуществах, создайте вокруг них островки компетенций и, по сути, игнорируйте большинство своих слабостей. В чем вы особенно хороши? Каковы ваши конкретные интересы в обширной области computer science? Составьте списки в качестве ответа на эти вопросы. То, что вам действительно нужно сделать для дальнейшего прогресса, — это определить свои сильные стороны, а затем жестко построить свой день вокруг них.

Выделяйте большие отрезки времени на написание кода. Так вы успеете досконально разобраться в стоящих перед вами проблемах и задачах — каждый программист знает, что требуется время, для того чтобы «загрузить» в голову сложный проект кода и втянуться в рабочий ритм. Допустим, Алиса и Боб трудятся над неким проектом. Пройдет 20 минут, пока каждый из них полностью не разберется в требованиях к проекту: просматривая его, погружаясь в суть функций кода и обдумывая общую картину. Алиса уделяет проекту три часа каждые три дня, а Боб — один час каждый день. Кто добьется большего прогресса? Алиса работает над проектом в среднем 53 минуты в день ((3 часа – 20 минут)/3). А Боб — только 40 минут в день, учитывая, что он каждый раз тратит много времени на вхождение в суть дела. Таким образом, при прочих равных условиях Алиса будет работать больше Боба на 13 минут в день. То есть у нее гораздо больше шансов достичь состояния потока, поскольку она может глубже погрузиться в проблему и полностью раствориться в ней.

Устраняйте отвлекающие факторы, пока вы находитесь в состоянии потока. Это кажется очевидным, но как же редко это выполняется! Программисты, которые могут снизить количество отвлекающих факторов: социальных сетей, развлекательных приложений, болтовни с коллегами, — достигают состояния потока гораздо чаще, чем те, кто на это не способен. Чтобы достичь успеха, вы должны сделать то, на что большинство других идти не хочет: убрать отвлекающие факторы. Выключите свой смартфон и закройте вкладки социальных сетей в браузере.

Делайте очевидные и, как вы сами знаете, необходимые вещи вне зависимости от поставленной задачи: высыпайтесь, правильно питайтесь и регулярно занимайтесь спортом. Как разработчику кода вам прекрасно известно выражение *«мусор на входе — мусор на выходе»*: если вы скармливаете системе плохие исходные данные, то получите плохие результаты. Попробуйте приготовить вкусную еду из испорченных продуктов — это практически невозможно! Хорошие входные данные приводят к качественному результату.

Потребляйте только высококачественную информацию, поскольку чем качественнее ваши исходные данные, тем значимее результат. Читайте книги по программированию вместо поверхностных статей в блогах; еще лучше — научные статьи, опубликованные в высокорейтинговых журналах, — это самая качественная информация.

Заключение

Итак, вот несколько самых простых способов, с помощью которых вы можете достичь состояния потока: отводите на работу большие отрезки времени, сосредоточьтесь на одной задаче, придерживайтесь здорового образа жизни и высыпайтесь, ставьте перед собой четкие цели, найдите работу, которая вам нравится, и активно старайтесь войти в состояние потока.

Если вы стремитесь к нему, то в конце концов его достигнете. Если вы будете систематически работать в состоянии потока, ваша продуктивность возрастет на порядок. Это простая, но мощная концепция для программистов и других работников умственного труда. Как говорил Михай Чиксентмихайи:

Лучшие моменты нашей жизни — не те, когда мы пассивны, восприимчивы к внешним сигналам и расслабленны... Лучшие моменты обычно настают тогда, когда наши тело или разум

напряжены до предела в добровольном стремлении достичь чего-то трудновыполнимого и стоящего.

В следующей главе мы вплотную займемся философией Unix, которая состоит в том, что следует *делать что-то одно, но делать это хорошо* — этот принцип оказывается полезным не только при создании масштабируемой операционной системы, но и в обычной жизни!

Источники

Troy Erstling, «The Neurochemistry of Flow States», *Troy Erstling* (blog), https://troyerstling.com/the-neurochemistry-of-flow-states/.

Steven Kotler, «How to Get into the Flow State», filmed at A-Fest Jamaica, February 19, 2019, Mindvalley video, https://youtu.be/XG_hNZ5T4nY/.

F. Massimini, M. Csikszentmihalyi, and M. Carli, «The Monitoring of Optimal Experience: A Tool for Psychiatric Rehabilitation», *Journal of Nervous and Mental Disease* 175, no. 9 (September 1987).

Kevin Rathunde, «Montessori Education and Optimal Experience: A Framework for New Research», *NAMTA Journal* 26, no. 1 (January 2001): 11–43.

Owen Schaffer, «Crafting Fun User Experiences: A Method to Facilitate Flow», Human Factors International white paper (2015), https://humanfactors.com/hfi_new/whitepapers/crafting_fun_ux.asp.

Rony Sklar, «Hyperfocus in Adult ADHD: An EEG Study of the Differences in Cortical Activity in Resting and Arousal States» (MA thesis, University of Johannesburg, 2013), https://hdl.handle.net/10210/8640.

7

«Делай что-то одно, но делай это хорошо» и другие принципы Unix

Такова философия Unix. Пишите программы, которые делают только одну вещь, и делают ее хорошо. Пишите отдельные модули, которые будут работать совместно. Пишите программы для обработки текстовых потоков, потому что это универсальный интерфейс.

Дуглас Макилрой (Douglas McIlroy)

Основная философия операционной системы Unix проста: делай что-то одно, но делай это хорошо. К примеру, это означает, что чаще всего лучше создать функцию или модуль, который надежно и эффективно решает одну задачу, чем пытаться сразу разобраться с рядом проблем. В этой главе вы увидите несколько примеров кода на Python, демонстрирующих этот принцип в действии, и узнаете, что философия Unix применима и к программированию. Затем я расскажу вам о важнейших

принципах, используемых некоторыми самыми выдающимися инженерами компьютерных систем мира при создании современных операционных систем. Если вы программист, вы найдете здесь ценные советы по написанию более совершенного кода для ваших собственных проектов.

Но сначала о главном: что такое Unix и почему это должно нас волновать?

Расцвет Unix

Unix — это философия проектирования, которая вдохновила создателей многих из самых популярных на сегодняшний день операционных систем, включая Linux и macOS. Семейство ОС Unix появилось в конце 1970-х годов, когда компания Bell Systems сделала открытым исходный код этой технологии. С тех пор университеты, частные лица и корпорации разработали множество расширений и новых версий.

Сегодня стандарт торговой марки Unix гарантирует, что операционные системы отвечают определенным требованиям качества. Unix и Unix-подобные ОС оказывают большое влияние на компьютерные технологии. Примерно 7 из 10 веб-серверов работают на Linux, использующей в качестве основы Unix. Большинство суперкомпьютеров сегодня управляются системами на базе Unix. Даже macOS является зарегистрированной Unix-системой.

Линус Торвальдс (Linus Torvalds), Кен Томпсон (Ken Thompson), Брайан Керниган (Brian Kernighan) — список разработчиков и специалистов по поддержке Unix содержит имена из ряда самых влиятельных программистов мира. Логично предположить, что существуют мощные организационные системы, позволяющие людям по всему свету взаимодействовать, создавая масштабную экосистему Unix, которая состоит из миллионов строк кода. И это именно так! Философия, обеспечивающая такой масштаб сотрудничества, — это акроним DOTADIW (серьезно) — или *«Do One Thing And Do It Well»*

(«Делай что-то одно, но делай это хорошо»). О философии Unix написаны целые книги, поэтому здесь мы остановимся только на самых актуальных идеях и используем фрагменты кода на Python для демонстрации некоторых примеров. Насколько мне известно, еще ни в одной книге не проводилось рассмотрение принципов Unix в контексте языка программирования Python. Итак, начнем!

Введение в философию Unix

Основная идея философии Unix заключается в создании простого, ясного, лаконичного модульного кода, который легко расширять и поддерживать. Это может означать многое (подробнее об этом будет рассказано в данной главе), но главное — позволить большому количеству людей совместно работать над кодовой базой, отдавая при этом предпочтение легкости чтения кода, а не его эффективности, и компонуемости вместо монолитного дизайна. При разработке монолитных приложений не используется модульный принцип — это значит, что большие логические куски кода не могут повторно использоваться, выполняться отдельно или отлаживаться без доступа к приложению в целом.

Допустим, вы пишете программу, которая, используя унифицированный указатель ресурса (uniform resource locator, URL), выводит в командной строке HTML-код, взятый с этого URL-адреса. Назовем эту программу `url_to_html()`. Согласно философии Unix, она должна хорошо делать только одну вещь, а именно: брать HTML-код из URL-адреса и выводить его в оболочке (листинг 7.1). Вот и все.

Листинг 7.1. Простая функция кода, которая считывает HTML-код с заданного URL-адреса и возвращает строку

```
import urllib.request


def url_to_html(url):
    html = urllib.request.urlopen(url).read()
    return html
```

Это все, что вам нужно. Не добавляйте дополнительные функции, такие как фильтрация тегов или отладка багов. Например, у вас может возникнуть соблазн добавить код, который будет исправлять распространенные ошибки, совершаемые пользователем. Например, пользователь забудет указать закрывающий тег, как продемонстрировано в примере: тег `<span>` не имеет закрывающего тега `</span>`:

```
<a href='nostarch.com'><span>Python One-Liners</a>
```

Согласно философии Unix, даже если вы видите такие ошибки, вы не должны исправлять их в рамках этой конкретной функции.

Еще одним искушением при создании этой простой HTML-функции является автоматическое исправление форматирования. Например, следующий HTML-код выглядит не очень красиво:

```
<a href='nostarch.com'><span>Python One-Liners</span></a>
```

Для вас может быть предпочтительнее такое форматирование кода:

```
<a href='nostarch.com'>
    <span>
        Python One-Liners
    </span>
</a>
```

Однако ваша функция называется `url_to_html()`, а не `prettify_html()` (приукрасить html). Появление такой функции, как приукрашивание кода, добавило бы вторую функциональность, которая, возможно, не нужна некоторым пользователям этой функции.

Скорее, вам следует создать другую функцию под названием `prettify_html(url)`, единственной задачей которой будет исправление стилистических ошибок в HTML-коде. Эта функция может внутри себя использовать функцию `url_to_html()` для получения HTML-кода перед его дальнейшей обработкой.

Сфокусировав каждую функцию только на одной цели, вы тем самым делаете более удобным сопровождение и даете возможность

расширения вашего кода. Вывод одной программы является вводом для другой. Вы уменьшаете сложность кода, избегаете путаницы в выходных данных и концентрируетесь на реализации одной задачи.

Хотя отдельные подпрограммы могут выглядеть мелкими, даже тривиальными, вы можете комбинировать их для создания более сложных программ, сохраняя при этом простоту их отладки.

15 полезных принципов Unix

Далее мы подробнее рассмотрим 15 принципов Unix, наиболее актуальных на сегодняшний день, и, по возможности, реализуем их на примерах языка Python. Я позаимствовал эти принципы у экспертов по работе с Unix Эрика Реймонда (Eric Raymond) и Майка Ганкарца (Mike Gancarz) и применил их к современному программированию на Python. Заметьте, что многие из них полностью согласуются или частично пересекаются с другими принципами, описанными в данной книге.

1. Пусть каждая функция делает хорошо что-то одно

Всеобъемлющий принцип философии Unix — *делай что-то одно, но делай это хорошо*. Посмотрим, как это выглядит на примере кода. В листинге 7.2 реализована функция `display_html()`, которая берет URL-адрес в виде строки и отображает предварительно обработанный HTML-код, соответствующий этому URL-адресу.

Листинг 7.2. Заставьте каждую функцию или программу выполнять только одну задачу, и делать это хорошо

```
import urllib.request
import re


def url_to_html(url):
    html = urllib.request.urlopen(url).read()
    return html
```

```
def prettify_html(html):
    return re.sub('<\s+', '<', html)

def fix_missing_tags(html):
    if not re.match('<!DOCTYPE html>', html):
        html = '<!DOCTYPE html>\n' + html
    return html

def display_html(url):
    html = url_to_html(url)
    fixed_html = fix_missing_tags(html)
    prettified_html = prettify_html(fixed_html)
    return prettified_html
```

Логика работы этого кода отображена на рис. 7.1.

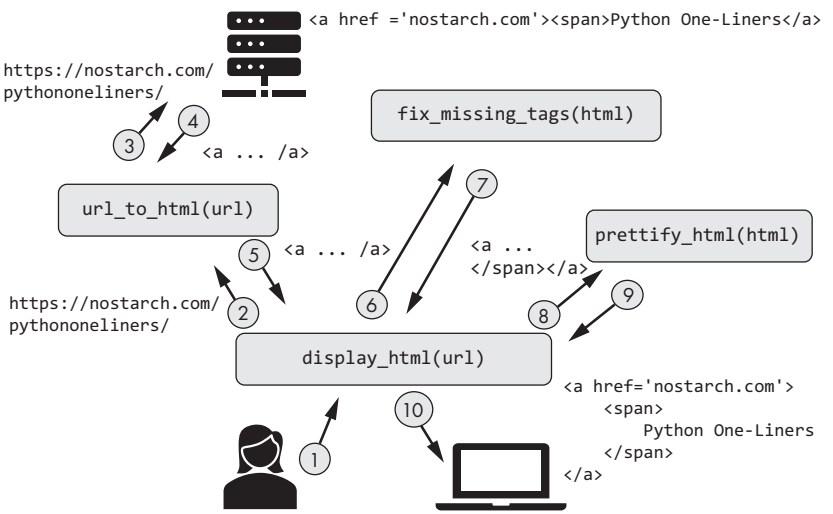


Рис. 7.1. Представление нескольких простых функций, каждая из которых хорошо делает что-то одно, и их совместная работа для выполнения более крупной задачи

Этот код представляет собой пример реализации функции `display_html`, которая выполняет следующие шаги:

1. Получает HTML-код с заданного URL-адреса.
2. Добавляет некоторые отсутствующие теги.
3. Форматирует HTML-код.
4. Возвращает результат программе, вызывающей эту функцию.

Если вы запустите код с URL-адресом, указывающим на не слишком красивый HTML-код `'< a href="https://finxter.com">Solve next Puzzle</a>'`, функция `display_html` исправит плохо отформатированный (и неправильный) HTML путем передачи данных между входами и выходами отдельных небольших функций кода, поскольку каждая из них хорошо выполняет свою конкретную задачу.

Вы можете вывести результат работы основной функции с помощью следующей строки:

```
print(display_html('https://finxter.com'))
```

Этот код выведет HTML-код в вашей оболочке с новым тегом и удаленным лишним пробелом:

```
<!DOCTYPE html>
<a href="https://finxter.com">Solve next Puzzle</a>
```

Представьте всю эту программу как браузер на вашем устройстве. Алиса вызывает функцию `display_html(url)`, та берет URL-адрес и передает его другой функции `url_to_html(url)`, которая забирает HTML-код с заданного URL-адреса. Нет необходимости реализовывать один и тот же функционал дважды. К счастью, разработчик функции `url_to_html()` дал ей минимум функциональности, поэтому вы можете использовать возвращаемый ею на выходе HTML-код непосредственно в качестве входных данных для функции `fix_missing_tags(html)`. На жаргоне Unix это называется *конвейерной передачей (piping)*: выходные данные одной программы передаются в качестве входных другой программе. Возвращаемое значение функции `fix_missing_tags()` — это исправленный HTML-код с закрывающим тегом `</span>`, который отсутствовал в исходном HTML. Затем эти выходные данные передаются в функцию

`prettify_html(html)`, после чего вы получаете исправленный HTML-код с отступами, удобный для просмотра пользователем. Только после этого функция `display_html(url)` возвращает Алисе исправленный HTML-код. Вы видите, что серия небольших функций, связанных вместе и реализованных в виде конвейера, может выполнять довольно большие задачи!

В своем проекте вы можете последовательно реализовать сначала функцию, которая не форматирует HTML, а только добавляет тег `<!DOCTYPE html>`, а затем отдельно функцию, форматирующую HTML, но не добавляющую новый тег. Оставляя код небольшим, вы можете легко создавать новый код на основе существующей функциональности, и в нем не будет излишеств. Модульная структура обеспечивает возможность многократного использования кода, удобство его сопровождения и расширения.

Сравните эту версию с возможной монолитной, не разделенной на блоки реализацией, где функция `display_html(url)` должна выполнять все эти мелкие задачи самостоятельно. Вы не смогли бы отдельно несколько раз использовать такие функциональные возможности, как извлечение HTML-кода по URL-адресу или исправление ошибочного HTML-кода. Если использовать единую функцию кода, которая все делает сама, то это будет выглядеть следующим образом:

```
def display_html(url):
    html = urllib.request.urlopen(url).read()
    if not re.match('<!DOCTYPE html>', html):
        html = '<!DOCTYPE html>\n' + html
    html = re.sub('<\s+', '<', html)
    return html
```

Функция стала более сложной: она решает сразу несколько задач, вместо того чтобы сфокусироваться на одной. Хуже того, если вы реализуете варианты одной и той же функции и не удаляете пробелы после открывающего тега `'<'`, вам придется копировать и вставлять оставшиеся строки кода. Это приведет к избыточности кода и снижению читабельности. Чем больше функциональности вы добавляете, тем хуже!

2. Простое лучше сложного

Простое лучше сложного — это главный принцип всей книги. Вы уже наблюдали его во множестве форм и проявлений — я подчеркиваю это, потому что если вы не предпримете решительных действий в сторону упрощения, вы будете множить сложности. В Python принцип *«простое лучше сложного»* даже вошел в неофициальный свод правил. Если вы откроете оболочку Python и введете `import this`, вы получите знаменитый текст «Дзен Python» («Zen of Python») (листинг 7.3).

Листинг 7.3. Дзен Python

```
> import this
Дзен Python от Тима Петерса (Tim Peters)

Красивое лучше уродливого.
Явное лучше неявного.
Простое лучше сложного.
Сложное лучше запутанного.
Плоское лучше вложенного.
Разреженное лучше плотного.
Читаемость имеет значение.
Особые случаи не настолько особые, чтобы нарушать правила.
При этом практичность важнее безупречности.
Ошибки никогда не должны замалчиваться.
Если они не замалчиваются явно.
Встретив двусмысленность, отбрось искушение угадать.
Должен существовать один (желательно только один) очевидный способ
сделать это.
Хотя этот способ поначалу может быть и неочевиден, если вы не
голландец.
Сейчас лучше, чем никогда.
Хотя никогда зачастую лучше, чем *прямо* сейчас.
Если реализацию сложно объяснить — идея плоха.
Если реализацию легко объяснить — возможно, идея хороша.
Пространства имен — отличная штука! Будем использовать их чаще![1]
```

Поскольку мы уже подробно рассматривали концепцию простоты, я не буду повторяться. Если вы все еще задаетесь вопросом,

[1] Указанная команда выведет текст на английском языке, здесь приведен перевод. — *Примеч. пер.*

почему простое лучше сложного, вернитесь к главе 1, где рассказывалось о негативном влиянии повышенной сложности на продуктивность.

3. Малое — прекрасно

Вместо того чтобы писать крупные блоки кода, создавайте небольшие функции и работайте как архитектор, обеспечивая взаимодействие между ними (см. рис. 7.1). Есть три основные причины, по которым следует поддерживать размер программы небольшим:

Уменьшение сложности

Чем длиннее код, тем он сложнее для понимания. Это когнитивный факт: ваш мозг способен одновременно отслеживать только определенное количество фрагментов информации. Слишком большой объем данных мешает восприятию общей картины. Делая код небольшим и уменьшая число строк в функциях, вы *облегчаете читаемость кода* и снижаете вероятность появления в вашей кодовой базе багов, которые впоследствии могут дорого вам обойтись.

Легкость сопровождения

Разбиение кода на множество мелких функциональных частей облегчает его поддержку. Добавление бóльшего количества простых функций вряд ли даст негативный побочный эффект. И, напротив, в крупном монолитном блоке кода любые вносимые изменения могут легко привести к непредвиденным глобальным последствиям, особенно если над кодом одновременно работают несколько программистов.

Удобство тестирования

Многие современные компании, выпускающие ПО, используют *разработку через тестирование*, что предполагает использование юнит-тестов для проверки каждой функции и модуля

на экстремальных входных данных и сравнения результатов с ожидаемыми. Это позволяет находить и устранять баги. Юнит-тесты гораздо эффективнее и проще реализуются на небольшом коде, где каждая функция сфокусирована на выполнении только одного действия, поэтому вы всегда знаете, каким должен быть результат.

Python сам по себе является лучшим примером этого принципа, чем даже небольшой фрагмент кода на нем. Любой хороший программист использует чужой код для повышения своей продуктивности. Миллионы разработчиков потратили огромное количество времени на оптимизацию кода, который вы можете импортировать в свой код за считаные секунды. Python, как и большинство других языков программирования, предоставляет эту функциональную возможность через библиотеки. Многие из редко используемых библиотек не поставляются по умолчанию с релизом программы и должны быть установлены отдельно. Не предоставляя все библиотеки в качестве встроенных функций, установленная на вашем компьютере среда Python занимает относительно мало места, при этом давая возможность использовать всю мощь внешних библиотек. Кроме того, сами библиотеки относительно невелики — все они фокусируются на ограниченных подмножествах функций. Таким образом, вместо одной монолитной библиотеки, решающей все задачи, у нас есть множество маленьких, каждая из которых отражает небольшую часть общей картины. Малое — прекрасно.

Каждые несколько лет появляются новые архитектурные шаблоны, предназначенные для разбиения крупных и монолитных приложений на красивые и небольшие для интенсификации разработки ПО. Недавними примерами являются CORBA (Common Object Request Broker Architecture — общая архитектура брокера объектных запросов), SOA (Service-Oriented Architecture — сервис-ориентированная архитектура) и микросервисы. Их суть заключается в том, чтобы разбить большой программный блок на ряд независимо развертываемых компонентов, доступ к которым могут получить не одна, а несколько программ. Предполагается, что это ускорит прогресс

в области разработки ПО за счет совместного использования и построения одних микросервисов на базе других.

В основе этих тенденций лежит идея написания модульного и многократно используемого кода. Изучив идеи, представленные в данной главе, вы сможете быстро и основательно разобраться в настоящих и будущих тенденциях с общим трендом на модульность. Стоит с самого начала применять разумные принципы, опережая события.

ПРИМЕЧАНИЕ

Более глубокое изучение этой захватывающей темы выходит за рамки данной книги, но я предлагаю вам ознакомиться с отличным ресурсом о микросервисах от Мартина Фаулера на сайте https://martinfowler.com/articles/microservices.html.

4. Создавайте прототип как можно быстрее

Команда Unix является ярым сторонником принципа, который обсуждался в главе 3, — это *создание MVP*. Он позволяет вам не зацикливаться на перфекционизме, добавляя все больше и больше функций и экспоненциально увеличивая сложность без всякой необходимости. Если вы работаете над большими приложениями, такими как операционные системы, вы просто не можете себе позволить идти по пути усложнения!

На рис. 7.2 показан пример раннего запуска приложения, которое, вопреки принципу MVP, напичкано ненужными функциями.

В приложении есть такие функции, как интерактивная проверка решения, голосование за головоломки, статистика пользователей, управление пользователями, премиум-функциональность и видеоролики, а также простые элементы, такие как логотип. Все это не нужно для первоначального запуска продукта. На самом деле MVP приложения Finxter должно быть изображением простой кодовой головоломки, опубликованным в социальных сетях. Этого

Рис. 7.2. Сравнение приложений Finxter.com и Finxter MVP

достаточно, чтобы подтвердить гипотезу о пользовательском спросе, не тратя годы на создание приложения. *Fail Fast, Fail Early, Fail Cheap*[1].

5. Предпочитайте портируемость эффективности

Портируемость (или *переносимость*) — это способность системы или программы нормально функционировать при перемещении ее из одной операционной среды в другую. Одним из основных преимуществ ПО является возможность его переноса между платформами: вы можете написать программу на своем компьютере, и миллионы пользователей смогут запустить ее на своих компьютерах, никак ее не адаптируя.

[1] «Быстрый провал, ранний провал, дешевый провал» — принцип, предполагающий быструю проверку гипотезы с минимальными затратами. — *Примеч. ред.*

Однако за удобство переноса приходится платить эффективностью. Этот *компромисс между портируемостью и производительностью* хорошо описан в технической литературе. Вы можете достичь более высокой эффективности, адаптируя ПО только к одному типу среды, при этом жертвуя переносимостью. *Виртуализация* — отличный пример такого компромисса: благодаря размещению дополнительного уровня ПО между частью программы и базовой инфраструктурой, на которой она выполняется, ваша программа сможет работать практически на любой *физической машине*. Кроме того, виртуальная машина может переносить текущее состояние выполнения программы с одного компьютера на другой. Это облегчает портируемость ПО. Однако дополнительный уровень, необходимый для виртуализации, увеличивает время выполнения и снижает эффективность использования памяти из-за дополнительных затрат на посредничество между программами и физической машиной.

В философии Unix предпочтение отдается портируемости, а не эффективности; это вполне логично, поскольку операционная система используется очень многими.

Но универсальное правило отдавать предпочтение переносимости применимо и к более широкой аудитории разработчиков ПО. Ограничение возможности переноса вашего приложения снижает его ценность. Сегодня принято радикально повышать портируемость — пусть и ценой уменьшения эффективности. Ожидается, что веб-приложения будут работать на любом компьютере с браузером, будь то macOS, Windows или Linux. Они также становятся все более доступными — например, для людей с нарушениями зрения — даже несмотря на то что их производительность при этом может снижаться. Многие ресурсы являются гораздо более ценными, чем скорость вычислений: человеческие жизни, временны́е затраты и другие последствия второго порядка, возникающие при использовании компьютеров.

Но что означает портируемость с точки зрения программирования помимо этих общих соображений? В листинге 7.4 реализована

функция, которая вычисляет среднее значение указанных аргументов — в представленном виде она не оптимизирована для переноса.

Листинг 7.4. Функция усреднения, не совсем оптимизированная с точки зрения возможности переноса

```
import numpy as np

def calculate_average_age(*args):
    a = np.array(args)
    return np.average(a)


print(calculate_average_age(19, 20, 21))
# 20.0
```

Этот код не портируется по двум причинам. Во-первых, имя функции `calculate_average_age()` не является достаточно общим, чтобы его можно было использовать в любом другом контексте, несмотря на то что она просто вычисляет среднее значение. Ведь вам не придет в голову использовать ее, например, для вычисления среднего количества посетителей сайта. Во-вторых, в коде без необходимости используется библиотека, несмотря на то что вы легко можете вычислить среднее значение без нее (листинг 7.5). Вообще, применение библиотек — отличная идея, но только если это приносит пользу. В данном случае добавление библиотеки снижает переносимость, так как у пользователя она может быть не установлена; к тому же это не слишком увеличивает эффективность.

В листинге 7.5 представлена функция с отличными возможностями для переноса.

Листинг 7.5. Функция усреднения с возможностью переноса

```
def average(*args):
    return sum(args) / len(args)

print(average(19, 20, 21))
# 20.0
```

Здесь мы дали функции более общее название и избавились от ненужного импорта библиотеки. Теперь вам не нужно беспокоиться о предустановке библиотеки и вы можете портировать этот код в другие проекты.

6. Храните данные в плоских текстовых файлах

Философия Unix предусматривает использование *плоских текстовых файлов* для хранения данных. Это простые текстовые или двоичные файлы без расширенных механизмов доступа к их содержимому — в отличие от многих других более эффективных, но и более сложных форматов, используемых, например, сообществом баз данных. Это простые, понятные человеку файлы данных. Распространенным примером плоского формата файла является CSV (comma-separated values), где каждая строка относится к одной записи данных, а значения разделены запятыми (листинг 7.6). Человек, впервые знакомящийся с этой информацией, может с первого взгляда составить о ней свое представление.

Листинг 7.6. Данные с сайта Data.gov о похищенном оружии, представленные в формате плоского файла (CSV)

```
Property Number,Date,Brand,Model,Color,Stolen,Stolen
From,Status,Incident number,Agency
P13827,01/06/2016,HI POINT,9MM,BLK,Stolen Locally,Vehicle,Recovered
Locally,B16-00694,BPD
P14174,01/15/2016,JENNINGS J22,,COM,Stolen Locally,Residence,Not
Recovered,B16-01892,BPD
P14377,01/24/2016,CENTURY ARMS,M92,,Stolen
Locally,Residence,Recovered Locally,B16-03125,BPD
P14707,02/08/2016,TAURUS,PT740 SLIM,,Stolen Locally,Residence,Not
Recovered,B16-05095,BPD
P15042,02/23/2016,HIGHPOINT,CARBINE,,Stolen
Locally,Residence,Recovered Locally,B16-06990,BPD
P15043,02/23/2016,RUGAR,,,Stolen Locally,Residence,Recovered
Locally,B16-06990,BPD
P15556,03/18/2016,HENRY ARMS,.17 CALIBRE,,Stolen
Locally,Residence,Recovered Locally,B16-08308,BPD
```

Вы можете легко обмениваться плоскими файлами, открывать их в любом текстовом редакторе и изменять их вручную. Однако за это удобство приходится платить эффективностью: формат, заточенный под конкретную цель, может хранить и считывать данные гораздо эффективнее. Базы данных, например, используют свои собственные файлы на диске, в которых применяются различные виды оптимизации типа подробных индексов и схем сжатия для представления данных. Если бы вы открыли их, вы бы ничего не поняли. Такая оптимизация позволяет программам считывать данные с меньшими затратами памяти и ресурсов, чем из обычных плоских текстовых файлов. В плоском файле системе приходится сканировать весь файл, чтобы считать конкретную строку. Веб-приложениям также требуется более эффективное оптимизированное представление данных, чтобы обеспечить пользователям быстрый доступ с малой задержкой, поэтому они редко используют плоские файлы и базы данных.

Однако использовать оптимизированные представления данных стоит только в том случае, если вы уверены в их необходимости. Например, при создании приложения, чувствительного к производительности, такого как поисковая система Google (способная находить веб-документы, наиболее релевантные запросу пользователя, за миллисекунды!). Для многих небольших приложений, таких как обучение модели в рамках машинного обучения (МО) на основе реального набора данных из 10 000 записей, формат CSV — рекомендуемый способ хранения. Использование базы данных со специализированным форматом снизит портируемость и добавит излишнюю сложность.

В листинге 7.7 приведен пример такой ситуации, в которой плоский формат предпочтителен. Здесь используется Python, один из самых популярных языков для приложений в области обработки данных и МО. Выполняется анализ набора данных, представляющих собой изображения (лица). Данные загружаются из плоского CSV-файла и обрабатываются, при этом предпочтение отдается переносимости, а не эффективности.

Листинг 7.7. Загрузка данных из плоского файла для задачи анализа данных на языке Python

```
From sklearn.datasets import fetch_olivetti_faces
From numpy.random import RandomState

rng = RandomState(0)

# Загрузка данных по лицам
faces, _ = fetch_olivetti_faces(...)
```

Функция `fetch_olivetti_faces` загружает набор данных *Olivetti faces* из библиотеки Scikit-learn, который содержит ряд изображений лиц. Обычно функции загрузки просто считывают данные и помещают их в память, прежде чем приступить к реальным вычислениям. Нет необходимости создавать базу или иерархическую структуру данных. Программа является автономной, не требующей установки базы данных или настройки дополнительных соединений с ней.

ПРИМЕЧАНИЕ

Я создал интерактивный блокнот Jupyter, чтобы вы могли запустить этот пример: https://blog.finxter.com/clean-code/#Olivetti_Faces/.

7. Используйте эффект рычага в своих интересах

Применение *«рычага» (leverage)* означает, что, прикладывая небольшие усилия, вы можете получить значительный результат. В финансовой сфере, например, под понятием «кредитный рычаг» (или «леверидж») подразумевают использование чужих денег для инвестирования и роста. В крупной корпорации это может быть размещение продукции в магазинах по всему миру с помощью обширной дистрибьюторской сети. Как программист вы должны применять накопленный опыт предыдущих поколений специалистов: задействовать библиотеки для реализации сложной функциональности, а не разрабатывать код с нуля, использовать StackOverflow и коллективный разум для исправления багов или просто просить других программистов проверить ваш код. Это те формы «рычага», которые позволят вам достичь гораздо большего меньшими усилиями.

Второй пример «рычага» — использование вычислительных мощностей. Компьютер может работать намного быстрее (и с гораздо меньшими затратами), чем человек. Создавайте полезное ПО, делитесь им с большим количеством людей, задействуйте больше вычислительных мощностей и чаще пользуйтесь чужими библиотеками и программами. Хорошие программисты быстро создают приличный исходный код, а великие — используют множество доступных им «рычагов» для усовершенствования своего кода.

В качестве примера в листинге 7.8 приведена однострочная программа из моей книги «Python One-Liners» (No Starch Press, 2020), которая просматривает заданный HTML-документ и находит все упоминания URL-адреса, содержащего подстроку `'finxter'` и либо `'test'`, либо `'puzzle'`.

Листинг 7.8. Однострочное решение для анализа ссылок на веб-страницы

```
## Зависимости
import re


## Данные
page = '''
<!DOCTYPE html>
<html>
<body>

<h1>My Programming Links</h1>
<a href="https://app.finxter.com/">test your Python skills</a>
<a href="https://blog.finxter.com/recursion/">Learn recursion</a>
<a href="https://nostarch.com/">Great books from NoStarchPress</a>
<a href="http://finxter.com/">Solve more Python puzzles</a>

</body>
</html>
'''


## Однострочник
practice_tests = re.findall("(<a.*?finxter.*?(test|puzzle).*?>)",
                 page)


## Результат
```

```
print(practice_tests)
# [('<a href="https://app.finxter.com/ "> проверьте свои знания
              языка Python </a>',
'test'),
# ('<a href="http://finxter.com/"> Решить еще головоломки на Python
             </a>', 'puzzle')]
```

Импортируя библиотеку `re`, мы используем мощную технологию регулярных выражений, которая мгновенно задействует тысячи строк кода и позволяет нам уложить всю программу в одну строку. Использование «рычагов» — это один из наиболее эффективных путей стать великим программистом. Например, использовать библиотеки в коде, вместо того чтобы реализовывать все самостоятельно, — это все равно что скачать приложение для планирования своего путешествия, а не прорабатывать каждую деталь с помощью бумажной карты.

ПРИМЕЧАНИЕ

Видеоролик с пояснением этого тезиса вы найдете на сайте https://pythononeliners.com/.

8. Отделяйте UI от функциональности

Некоторые UI (user interface, интерфейс пользователя) требуют от пользователя взаимодействия с программой, прежде чем запустится основной процесс (их называют *captive user interfaces, CUI*[1]). Примерами могут служить такие мини-программы, как Secure Shell (SSH), `top`, `cat` и `vim`, а также функции языка программирования, например функция `input()` в Python. CUI ограничивают юзабилити кода, поскольку они предназначены для запуска программы только при участии человека. Однако часто функциональность, заложенная в коде, скрытом за CUI, также полезна и для автоматизированных программ, которые должны иметь возможность работать без дополнительных действий со стороны пользователей. Грубо говоря,

[1] Captive — букв. «пленник». — *Примеч. ред.*

если вы поместите хороший код за таким интерфейсом, он будет недоступен без вмешательства человека!

Предположим, вы создаете на Python простой калькулятор ожидаемой продолжительности жизни, который принимает в качестве входных данных возраст пользователя и возвращает ожидаемое количество оставшихся лет, основываясь на простых эвристических методах.

«Если вам меньше 85 лет, ожидаемая оставшаяся продолжительность жизни равна 72 минус 80 % от вашего возраста. В противном случае она равна 22 минус 20 % от вашего возраста».

ПРИМЕЧАНИЕ

Эвристический анализ (а не сам код) основан на информации с сайта Decision Science News.

Ваш изначальный код на языке Python может выглядеть примерно так, как показано в листинге 7.9.

Листинг 7.9. Калькулятор ожидаемой продолжительности жизни — простой эвристический анализ, реализованный в виде CUI

```
def your_life_expectancy():
    age = int(input('how old are you? '))

    if age<85:
        exp_years = 72 - 0.8 * age
    else:
        exp_years = 22 - 0.2 * age

    print(f'People your age have on average {exp_years} years left -
            use them wisely![1]')


your_life_expectancy()
```

[1] «Людям вашего возраста осталось жить в среднем {exp_years} лет — распорядитесь ими мудро!». — *Примеч. ред.*

Вот несколько вариантов выполнения кода, представленного в листинге 7.9.

```
> how old are you? 10
People your age have on average 64.0 years left - use them wisely!
> how old are you? 20
People your age have on average 56.0 years left - use them wisely!
> how old are you? 77
People your age have on average 10.399999999999999 years left - use
them wisely!
```

Если вы хотите попробовать эту программу сами, я выложил ее в блокноте Jupyter на сайте https://blog.finxter.com/clean-code/#Life_Expectancy_Calculator/. Но, пожалуйста, не относитесь к этому слишком серьезно!

В листинге 7.9 мы использовали функцию Python `input()`, которая блокирует выполнение программы до тех пор, пока от пользователя не будут получены входные данные. Без пользовательского ввода код ничего не делает. Такой интерфейс ограничивает рамки использования данного кода. Если вы захотите рассчитать продолжительность жизни для каждого возраста от 1 до 100 и построить кривую, вам придется вручную ввести 100 значений и сохранить результаты в отдельном файле. Затем нужно скопировать и вставить их в новый скрипт, чтобы отобразить их на графике. На самом деле в представленном виде функция делает две вещи: обрабатывает ввод и вычисляет продолжительность жизни, что к тому же еще и нарушает первый принцип Unix: пусть каждая функция делает хорошо что-то одно.

Чтобы привести код в соответствие с этим принципом, мы отделим UI от функциональности — часто это отличный способ улучшения кода (листинг 7.10).

Листинг 7.10. Калькулятор продолжительности жизни — чисто эвристический анализ без CUI

```
# Функциональность
def your_life_expectancy(age):
    if age<85:
```

```
        return 72 - 0.8 * age
    return 22 - 0.2 * age


# Пользовательский интерфейс
age = int(input('how old are you? '))

# Объединяем пользовательский ввод данных и функциональность
# и выводим результат
exp_years = your_life_expectancy(age)
print(f'People your age have on average {exp_years} years left - use
      them wisely!')
```

Код в листинге 7.10 функционально идентичен коду в листинге 7.9 с одним существенным преимуществом: теперь мы можем использовать новую функцию в разных ситуациях, даже в таких, которые первоначальный разработчик не мог себе представить. В листинге 7.11 мы применяем функцию для вычисления ожидаемой продолжительности жизни при входных возрастах от 0 до 99 и строим график результатов; отметим, что благодаря удалению UI мы дополнительно добились портируемости программы.

Листинг 7.11. Код для построения графика ожидаемой продолжительности жизни в диапазоне от 0 до 99 лет

```
import matplotlib.pyplot as plt


def your_life_expectancy(age):
    '''Возвращает ожидаемое количество оставшихся лет.'''
    if age<85:
        return 72 - 0.8 * age
    return 22 - 0.2 * age


# График на первые 100 лет
plt.plot(range(100), [your_life_expectancy(i) for i in range(100)])

# Стиль графика
plt.xlabel('Age')
plt.ylabel('No. Years Left')
plt.grid()

# Показать и запомнить график
```

```
plt.savefig('age_plot.jpg')
plt.savefig('age_plot.pdf')
plt.show()
```

На рис. 7.3 показан результирующий график.

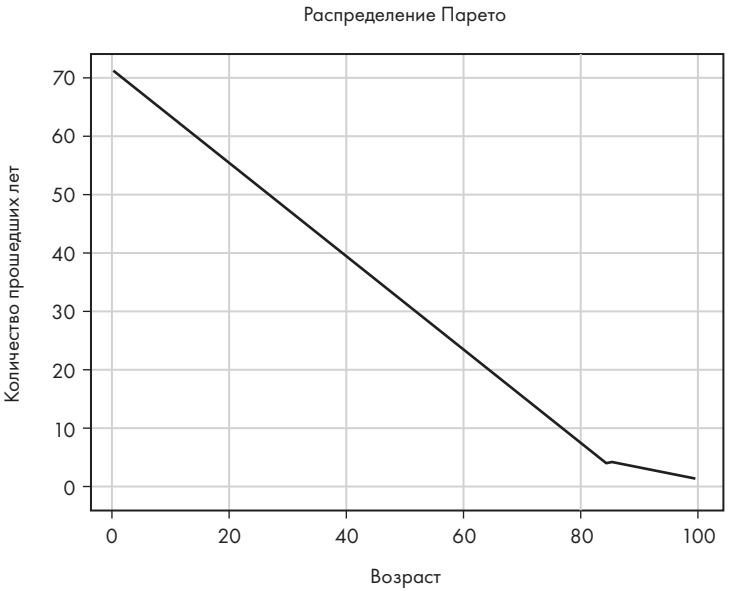


Рис. 7.3. Как работает эвристический анализ для входных данных в диапазоне от 0 до 99 лет

Конечно, любой эвристический анализ неточен по своей сути, но здесь основное внимание уделено тому, как отказ от CUI помог нам использовать код для построения графика. Если бы мы не придерживались восьмого принципа Unix, мы не смогли бы повторно задействовать исходную кодовую функцию `your_life_expectancy`, потому что CUI требовал бы ввода данных для каждого года в диапазоне от 0 до 99. С учетом данного принципа мы упростили код и открыли возможность для разнообразных будущих программ использовать и развивать эвристические методы. Вместо того чтобы оптимизировать код для одного конкретного случая, мы написали

его в общем виде, и теперь его можно применять в сотнях различных приложений. Почему бы не создать на его основе целую библиотеку?

9. Делайте каждую программу фильтром

Есть хороший аргумент в пользу того, что каждая программа уже является фильтром. Она преобразует входные данные в выходные с помощью определенного механизма фильтрации. Это позволяет легко объединять несколько программ, используя вывод одной из них в качестве ввода для другой, тем самым значительно повышая возможность многократного применения кода. Например, обычно не рекомендуется выводить результат вычислений в самой функции — вместо этого, согласно философии Unix, программа должна возвращать строку, которую затем можно вывести, записать в файл или использовать в качестве входных данных в другой программе.

Например, программу, сортирующую список, можно рассматривать как фильтр: она «фильтрует» неотсортированные элементы в упорядоченный список, как показано в листинге 7.12.

Листинг 7.12. Этот алгоритм сортировки методом вставок «фильтрует» неотсортированный список в отсортированный

```
def insert_sort(lst):

    # Проверка, пуст ли список
    if not lst:
        return []

    # Начинаем с первого элемента отсортированного списка
    new = [lst[0]]

    # Вставляем каждый последующий элемент
    for x in lst[1:]:
        i = 0
        while i<len(new) and x>new[i]:
            i = i + 1
        new.insert(i, x)

    return new

print(insert_sort([42, 11, 44, 33, 1]))
```

```
print(insert_sort([0, 0, 0, 1]))
print(insert_sort([4, 3, 2, 1]))
```

Алгоритм создает новый список и устанавливает каждый последующий элемент на ту позицию, где все элементы слева меньше него. Функция использует сложный фильтр для изменения порядка элементов, преобразуя входной список в отсортированный выходной.

Если какая-либо программа по сути является фильтром, то вы должны спроектировать ее соответствующим образом, используя интуитивно понятное сопоставление ввода и вывода. Позвольте, я поясню это ниже.

Золотым стандартом для фильтров является *однородное* отображение данных ввода/вывода, когда один тип входных данных сопоставляется с таким же типом выходных. Например, если кто-то говорит с вами по-английски, он ожидает, что вы ответите ему на английском, а не на другом языке. Аналогично, если функция берет значение входного параметра, то прогнозируемый результат — это возвращаемое значение. Если программа считывает данные из файла, то в ответ также ожидается файл. Если она считывает входные данные со стандартного ввода, она должна записать их на стандартный вывод. Вы уловили суть: самый интуитивный способ реализации фильтра состоит в том, чтобы все данные были одной категории.

В листинге 7.13 показан пример плохого кода с *неоднородным* отображением ввода/вывода, где реализована функция `average()`, преобразующая входные параметры в их среднее значение, но, вместо того чтобы возвращать среднее значение, функция `average()`выводит результат в оболочку.

Листинг 7.13. Отрицательный пример неоднородного характера входных и выходных данных

```
def average(*args):
    print(sum(args)/len(args))


average(1, 2, 3)
# 2.0
```

Более удачный подход (листинг 7.14) состоит в том, что функция `average()` возвращает среднее значение (однородное сопоставление данных ввода/вывода), которое затем можно вывести стандартным образом — посредством отдельного вызова этой функции с помощью `print()`. Этот вариант гораздо лучше, потому что позволяет, например, записать выходные данные в файл, а не выводить их на экран, или даже использовать их в качестве входных для другой функции.

Листинг 7.14. Положительный пример однородного отображения ввода/вывода

```
def average(*args):
    return sum(args)/len(args)


avg = average(1, 2, 3)
print(avg)
# 2.0
```

Конечно, некоторые программы выполняют фильтрацию из одной категории в другую — например, отображают файл на стандартном устройстве вывода или осуществляют перевод с английского языка на испанский. Но, следуя основному принципу создания ПО — «делай только одну вещь, но делай ее хорошо» (см. принцип Unix номер 1), — эти программы не должны выполнять ничего другого. Таким образом, золотое правило написания интуитивно понятных и органичных программ — разрабатывать их как фильтры!

10. Чем хуже, тем лучше

Этот принцип предполагает, что разработка кода с меньшей функциональностью на практике часто оказывается лучшим подходом. Когда ресурсы ограничены, разумнее запустить плохой продукт и быть первым на рынке, чем постоянно улучшать его, не выпуская первый релиз. Принцип был сформулирован в конце восьмидесятых годов разработчиком языка обработки списков LISP (list processing) Ричардом Габриэлем (Richard Gabriel). Он похож на принцип MVP,

рассмотренный в главе 3. Не воспринимайте этот парадоксальный принцип слишком буквально. С точки зрения качества худшее не может быть лучшим. Если бы вы обладали бесконечным временем и ресурсами, логично было бы всегда делать программу совершенной. Однако в мире с ограниченными ресурсами выпуск чего-то не очень хорошего часто оказывается более эффективным. Например, грубое и прямолинейное решение задачи дает вам преимущество первопроходца, позволяя получить быструю обратную связь от первых последователей, а также набрать популярность и привлечь внимание к своему программному продукту на ранней стадии разработки. Многие специалисты-практики утверждают, что идущий следом за первопроходцем должен вложить гораздо больше сил и ресурсов на создание намного более совершенного продукта, способного отвлечь пользователей от оригинала.

11. Чистый код лучше умного

Я немного изменил исходную формулировку этого принципа философии Unix *«Ясность лучше заумности»*, во-первых, для того чтобы сфокусироваться на программировании, а во-вторых, чтобы соотнести его с рассмотренными ранее принципами *написания чистого кода* (см. главу 4).

Этот принцип подчеркивает компромисс между чистым и умным кодом: код не должен быть умным в ущерб простоте.

Например, посмотрите на простой алгоритм пузырьковой сортировки (листинг 7.15). Он сортирует список, итерационно просматривая его и меняя положение каждых двух соседних элементов, которые еще не тронуты: меньший элемент перемещается влево, а больший — вправо. После каждой итерации список становится немного более отсортированным.

Листинг 7.15. Алгоритм пузырьковой сортировки на языке Python

```
def bubblesort(l):
    for boundary in range(len(l)-1, 0, -1):
        for i in range(boundary):
```

```
            if l[i] > l[i+1]:
                l[i], l[i+1] = l[i+1], l[i]
    return l

l = [5, 3, 4, 1, 2, 0]
print(bubblesort(l))
# [0, 1, 2, 3, 4, 5]
```

Алгоритм в листинге 7.15 хорошо читается и достаточно понятен, он успешно выполняет задачу и не содержит лишнего кода.

Теперь предположим, что ваш умный коллега утверждает, что вы могли бы сократить код на одну строку, используя *условные присваивания* вместо оператора `if` (листинг 7.16).

Листинг 7.16. «Умный» алгоритм пузырьковой сортировки на языке Python

```
def bubblesort_clever(l):
    for boundary in range(len(l)-1, 0, -1):
        for i in range(boundary):
            l[i], l[i+1] = (l[i+1], l[i]) if l[i] > l[i+1] else
                           (l[i], l[i+1])
    return l


print(bubblesort_clever(l))
# [0, 1, 2, 3, 4, 5]
```

Эта хитрость не улучшает код, а снижает удобство его чтения и ясность. Использование условного присваивания на первый взгляд кажется умным ходом, однако в результате оно приведет к невозможности выражения ваших идей в виде чистого кода. За советами о том, как писать такой код, обратитесь к главе 4.

12. Создавайте программы так, чтобы они могли взаимодействовать с другим ПО

Ваши программы не живут отдельно от остальных. Они вызываются для выполнения определенной задачи либо человеком, либо

другой программой. Поэтому вам необходимо разработать API для работы с внешним миром. Придерживаясь принципа 9 философии Unix *«Делайте каждую программу фильтром»*, который гласит, что необходимо обеспечить интуитивно понятное отображение ввода/вывода, вы уже тем самым создаете программы с возможностью взаимодействия друг с другом, а не заставляете их жить изолированно. Великий программист — это в равной степени и архитектор, и ремесленник, а новое ПО — уникальное сочетание старых и новых функций и чужих программ. В результате создание интерфейсов может стать основным этапом в цикле разработки.

13. Делайте свой код робастным

Кодовая база является *робастной* (надежной, устойчивой), если нарушить ее функционирование не так просто. Существует два взгляда на робастность кода: взгляд программиста и взгляд пользователя.

Как программист вы потенциально можете испортить код, просто изменив его. Поэтому кодовая база должна быть *устойчива к изменениям* (robust against changes), чтобы даже небрежный программист мог работать с кодовой базой, не нарушая ее функциональности. Допустим, у вас есть большой монолитный блок кода и каждый программист в вашей организации имеет доступ к его редактированию. Любое небольшое изменение может навредить всей системе. Теперь сравним эту ситуацию с разработкой кода в таких организациях, как Netflix или Google, где каждое вмешательство должно пройти несколько уровней утверждения, прежде чем код будет запущен в реальных условиях; все тщательно тестируется, поэтому развернутый код защищен от критических изменений. Добавляя уровни защиты, Google и Netflix сделали свой код более робастным, чем хрупкая монолитная кодовая база.

Одним из способов обеспечения надежности кодовой базы является управление правами доступа, чтобы отдельные разработчики не могли навредить приложению. Ни одно изменение не должно быть внесено, если не получено подтверждение (по крайней мере, еще от

одного человека), что это изменение скорее улучшит, чем испортит код. Такой прием приводит к снижению гибкости процесса разработки, но эту цену стоит заплатить, если у вас не стартап, которым вы занимаетесь в одиночку. Другие принципы обеспечения робастности кода мы уже рассматривали ранее: малое — прекрасно; создавайте функции, выполняющие только одну задачу, но делающие это хорошо; используйте разработку через тестирование; не усложняйте код без особой нужды. Вот еще несколько приемов повышения робастности, легко применимых на практике:

- Используйте системы управления версиями, такие как Git, чтобы можно было восстановить предыдущие варианты кода.
- Регулярно создавайте резервные копии данных вашего приложения, чтобы их можно было восстановить (данные не сохраняются системой управления версиями).
- Используйте распределенные системы, чтобы избежать единой точки отказа: запускайте приложение на нескольких машинах, чтобы снизить вероятность того, что поломка одной машины негативно повлияет на работу вашего приложения. Скажем, вероятность сбоя компьютера составляет 1 % в день — он будет выходить из строя примерно раз в 100 дней. Создание распределенной системы из пяти компьютеров, которые работают независимо друг от друга, теоретически может снизить вероятность отказа до $0.01^5 \times 100\ \% = 0.00000001\ \%$.

С точки зрения пользователя приложение является надежным, если вы не можете легко вывести его из строя, всего лишь предоставив неправильные или даже вредоносные входные данные. Предположим, что ваши пользователи ведут себя как стая горилл, бьющих по клавиатуре и вводящих случайные последовательности символов. Или, напротив, они — высококвалифицированные хакеры, которые разбираются в вашем приложении лучше вас и готовы использовать даже малейшую брешь в безопасности. Ваша программа должна быть надежно защищена от обоих типов пользователей.

Защититься от первой группы относительно просто. Юнит-тесты — это мощный инструмент: проверяйте каждую функцию на соответствие любым входным данным, которые только можно придумать, особенно на граничные случаи. Например, если ваша функция берет на входе целое число и вычисляет квадратный корень, проверьте, может ли она обработать отрицательные числа и 0, потому что необрабатываемые исключения могут нарушить последовательную работу надежных, простых, связанных в цепочку программ. Однако такие исключения приводят к другой, более коварной проблеме, на которую обратил мое внимание эксперт по безопасности и научный редактор этой книги Ноа Спан: предоставление возможности ввода данных для нарушения работы программы может дать злоумышленникам плацдарм для проникновения в операционную систему хоста. Поэтому проверьте способность вашей программы обрабатывать все виды ввода и, таким образом, сделайте свой код более робастным!

14. Исправляйте то, что можете, но лучше, если сбой случится раньше и громко

Хотя вы должны устранять проблемы в коде везде, где это возможно, вы не должны прятать ошибки, которые не можете устранить. Скрытая ошибка быстро усугубляется, становясь тем серьезнее, чем дольше она остается в тени.

Ошибки могут накапливаться. Предположим, что система распознавания речи в вашем приложении помощи водителю получила ошибочные обучающие данные и классифицирует два совершенно разных фонетических звучания как одно и то же слово (рис. 7.4). Из-за этого ваш код выдает ошибку, пытаясь сопоставить их с одним английским словом (например, ошибка может возникнуть при попытке сохранить эту противоречивую информацию в обратный указатель, который сопоставляет английские термины со звучанием). Вы можете написать код двумя способами: скрыть ошибку или передать ее на обработку приложению, пользователю или программисту. Хотя многие разработчики кода интуитивно хотят спрятать

проблемы, чтобы повысить юзабилити, это не самый разумный подход. Сообщения об ошибках несут полезную информацию. Если ваш код предупредит вас о проблеме на ранней стадии, вы быстрее найдете решение. Лучше узнать об ошибках раньше, прежде чем их последствия накопятся и уничтожат миллионы долларов или даже человеческие жизни.

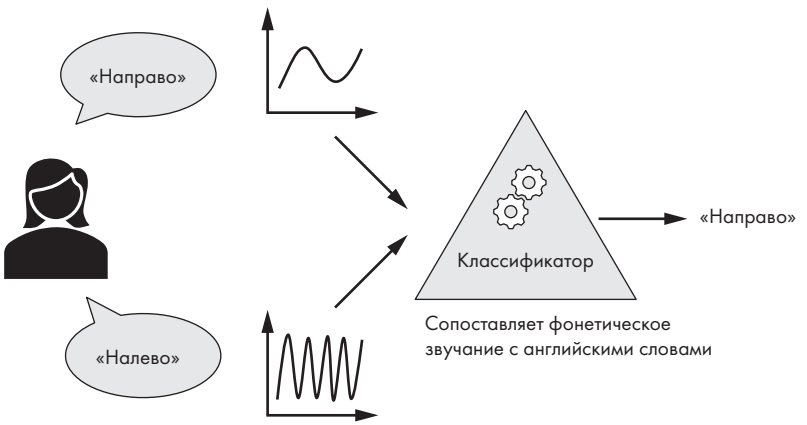


Рис. 7.4. Классификатор на этапе обучения сопоставляет два различных фонетических звучания с одним и тем же английским словом

Лучше выявлять неустранимые проблемы и предупреждать о них пользователя, чем умалчивать о них, даже если людям не нравятся сообщения об ошибках, и юзабилити приложения при этом снижается. Альтернатива — скрывать ошибки до тех пор, пока они не станут такими огромными, что с ними уже невозможно будет справиться.

В продолжение примера некорректных обучающих данных в листинге 7.17 приведен код, в котором функция `Python classify()` имеет на входе один аргумент — классифицируемое звучание — и возвращает соответствующее английское слово. Допустим, вы реализовали функцию `duplicate_check(wave, word)`. С помощью пары аргументов `wave` и `word` она проверяет, приводит ли к тому же результату классификации какое-то значительно отличающееся звучание из вашей

базы данных. В этом случае результат неоднозначен, поскольку два совершенно разных звучания соответствуют одному и тому же слову. Вы обязаны сообщить об этом пользователю, вызвав функцию `ClassificationError`, вместо того чтобы возвращать случайно угаданное слово. Да, пользователь будет раздражен, но, по крайней мере, у него будет шанс самостоятельно справиться с последствиями ошибки. *Исправляйте то, что можете, но лучше, если сбой случится раньше и громко!* (Fail Early and Noisily).

Листинг 7.17. Фрагмент кода с громким провалом вместо случайного угадывания, если результат классификации фонетических звучаний неоднозначен

```
def classify(wave):
    # Провести классификацию
      word = wave_to_word(wave) # реализовать

    # Проверка, не приводит ли другое звучание
    # к тому же слову
    if duplicate_check(wave, word):

        # Не возвращайте случайно угаданное значение,
        # умалчивая об ошибке!
        raise ClassificationError('Not Understood')

    return word
```

15. Избегайте написания кода вручную: если можете, пишите программы для создания программ

Этот принцип предполагает, что код, который может быть сгенерирован автоматически, *должен быть сгенерирован*, поскольку люди, как известно, склонны к ошибкам, особенно когда занимаются повторяющейся и скучной работой. Есть множество способов добиться этого — на самом деле современные языки программирования высокого уровня, такие как Python, компилируются в машинный код именно с помощью такого рода ПО. Авторы программ для написания программ помогли специалистам высокого уровня создавать все виды прикладного ПО, не заморачиваясь с низкоуровневыми аппаратными языками. Без этих инструментов, пишущих программы

за нас, компьютерная отрасль все еще находилась бы в зачаточном состоянии.

Генераторы и компиляторы кода уже сегодня создают исходный код в больших объемах. Рассмотрим этот принцип еще с одной стороны. Сегодня технологии машинного обучения и искусственного интеллекта поднимают концепцию создания программ для написания программ на новый уровень. Интеллектуальные машины (модели МО) собираются людьми по частям, а затем они переписывают (и настраивают) сами себя на основе данных. Технически модель МО — это программа, которая много раз меняла сама себя, пока ее функция приспособленности (обычно задаваемая человеком) не достигла максимума. По мере того как машинное обучение все больше проникает в информационные технологии (и начинает там преобладать), этот принцип становится все более доминирующим в сфере современных вычислений. Программисты по-прежнему будут играть важную роль в использовании этих мощных инструментов; в конце концов, компиляторы не заменили человеческий труд, а открыли новый мир приложений, создаваемых людьми. Я ожидаю, что то же самое произойдет и в программировании: инженеры машинного обучения и архитекторы ПО будут разрабатывать продвинутые приложения, комбинируя различные низкоуровневые программы, такие как модели МО. Что ж, это всего лишь мой взгляд на эту тему — ваш может быть более или менее оптимистичным!

Заключение

В этой главе вы изучили 15 принципов, разработанных создателями Unix для того, чтобы писать более эффективный код. Стоит их повторить — читая список, подумайте, как каждый из них можно применить к вашему текущему проекту кода.

- Пусть каждая функция делает хорошо что-то одно.
- Простое лучше сложного.
- Малое — прекрасно.

- Создавайте прототип как можно быстрее.
- Предпочитайте портируемость эффективности.
- Храните данные в плоских текстовых файлах.
- Используйте эффект рычага в своих интересах.
- Отделяйте UI от функциональности.
- Делайте каждую программу фильтром.
- Чем хуже, тем лучше.
- Чистый код лучше умного.
- Создавайте программы так, чтобы они могли взаимодействовать с другим ПО.
- Делайте свой код робастным.
- Исправляйте то, что можете, но лучше, если сбой случится раньше и громко.
- Пишите программы для создания программ.

В следующей главе вы узнаете о значении минимализма в дизайне и о том, как разработать приложение, вызывающее у пользователей восторг, затрачивая минимум усилий.

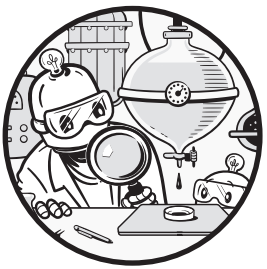
Источники

Mike Gancarz, *The Unix Philosophy*, Boston: Digital Press, 1994.

Eric Raymond, *The Art of Unix*, Boston: Addison-Wesley, 2004, http://www.catb.org/~esr/writings/taoup/html/.

8

В дизайне лучше меньше, да лучше



Простота — это стиль жизни разработчика кода. Хотя вы, возможно, не считаете себя дизайнером, есть вероятность, что в карьере программиста вы неоднократно столкнетесь с разработкой UI. Может быть, вы специалист по обработке и анализу данных и хотите создать визуально привлекательный дашборд. Или, будучи инженером баз данных, мечтаете разработать удобный API. А возможно, вы разработчик блокчейна и ваша цель — обычный веб-интерфейс для ввода данных в смарт-контракт. В любом случае знание основных принципов дизайна может сэкономить целый день для вас и вашей команды, и их легко усвоить! Принципы дизайна, рассматриваемые в этой главе, универсальны.

В частности, вы познакомитесь с одной крайне важной областью computer science, которая больше всего выигрывает от минималистического мышления, — это дизайн и взаимодействие с пользователем

(user experience, UX). Чтобы получить представление о важности минимализма в этой сфере, подумайте о различиях между Yahoo Search и Google Search, Blackberry и iPhone, Facebook Dating и Tinder: победившие на рынке технологии часто имеют абсолютно простой пользовательский интерфейс. Может, это потому, что и в дизайне царит принцип *«лучше меньше, да лучше»*?

Сначала мы кратко рассмотрим некоторые продукты, которые выиграли оттого, что их создатели были полностью сфокусированы на главном. А затем покажем, как использовать минимализм в собственных дизайнерских разработках.

Минимализм в эволюции мобильных телефонов

Ярким примером минимализма в дизайне вычислительной техники служит эволюция мобильных телефонов (рис. 8.1). Nokia Mobira Senator, один из первых коммерческих «мобильных» телефонов, был выпущен в 1980-х годах, весил 10 кг и был довольно сложным в обращении. Год спустя компания Motorola выпустила на рынок свою модель DynaTAC 8000X, которая была в 10 раз легче и весила всего 1 кг. Компании Nokia пришлось включиться в игру. В 1992 году Nokia выпускает модель 1011, которая весит вдвое меньше, чем DynaTAC 8000X. Почти десятилетие спустя, в 2000 году, в соответствии с законами Мура, Nokia добилась мегауспеха выпустив культовый телефон Nokia 3310, вес которого составил всего 88 г. По мере того как технологии становились все более совершенными и сложными, пользовательский интерфейс, включая размер, вес и даже количество кнопок, упрощался. Эволюция мобильных телефонов доказывает, что радикально минималистичный дизайн возможен, даже если сложность при этом возрастает на порядки. Можно даже утверждать, что он обеспечил успех приложений для смартфонов и их стремительное развитие в современном мире. Вам было бы трудно просматривать веб-страницы, пользоваться картографическими сервисами или отправлять видеосообщения с помощью Nokia Senator!

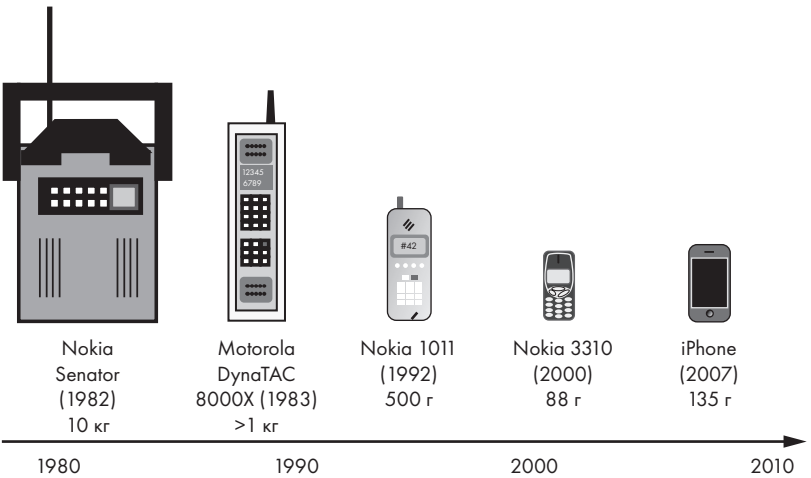


Рис. 8.1. Некоторые этапы эволюции мобильных телефонов

Преимущества минимализма в дизайне наглядно проявляются во многих продуктах помимо смартфонов. Компании используют его для оптимизации UX и создания целевых приложений. Что может быть лучшим примером, чем поисковая система Google?

Минимализм в поисковых системах

На рис. 8.2 изображен минималистичный дизайн, напоминающий то, как Google (и его подражатели) оформляют основной пользовательский интерфейс — в виде сильно упрощенного входа в интернет. Не заблуждайтесь, минималистичный и чистый дизайн — не случайность. Эту целевую страницу ежедневно посещают миллиарды пользователей. Возможно, она могла бы стать *важнейшей* продающей площадкой в интернете. Небольшая реклама на главной странице Google принесла бы миллиарды кликов и, вероятно, миллиарды долларов. Однако компания не позволяет объявлениям загромождать свою целевую страницу, несмотря на потерю возможности получения дохода в краткосрочной

перспективе. Менеджеры Google знают, что сохранение целостности и направленности бренда, выраженное в минималистичном дизайне, намного ценнее доходов от продажи этой превосходной рекламной площадки.

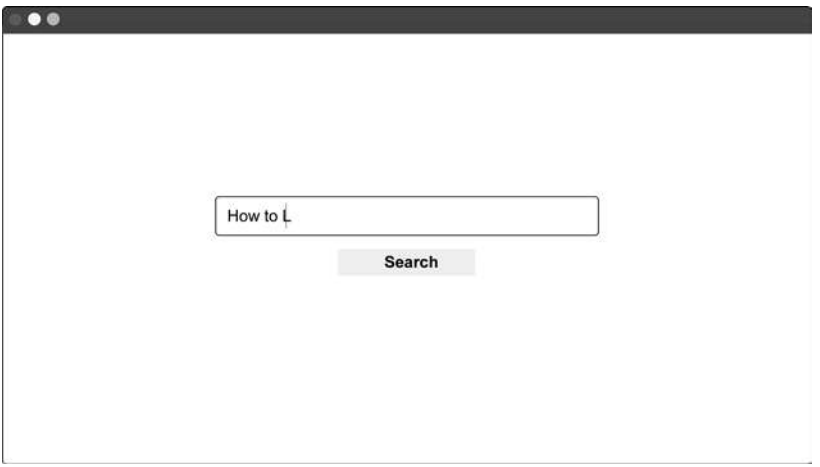


Рис. 8.2. Пример современной поисковой системы с минималистичным дизайном

Теперь сравним этот чистый сфокусированный дизайн с тем, который используют альтернативные поисковые системы, такие как Bing и Yahoo!, зарабатывая на своей основной площадке (рис. 8.3).

Даже при организации обычных поисковых сайтов такие компании, как Yahoo!, пошли по стандартному пути: они загромоздили свою главную страницу новостями и рекламой, чтобы повысить краткосрочный заработок. Но эти доходы оказались преходящими, так как дизайн оттолкнул тех, кто их приносил: пользователей. Снижение юзабилити привело к падению конкурентоспособности и постепенному разрушению привычного поискового поведения. Любой дополнительный элемент сайта, не связанный с поиском, увеличивает когнитивную нагрузку на пользователя, который вынужден игнорировать отвлекающие внимание заголовки, рекламу

и изображения. Удобство поиска — одна из причин, по которой Google постоянно увеличивает свою долю рынка. Окончательный вердикт выносить еще рано, но рост популярности узкоспециализированных поисковых систем в последние десятилетия свидетельствует о превосходстве минималистичного дизайна, сфокусированного на одной цели.

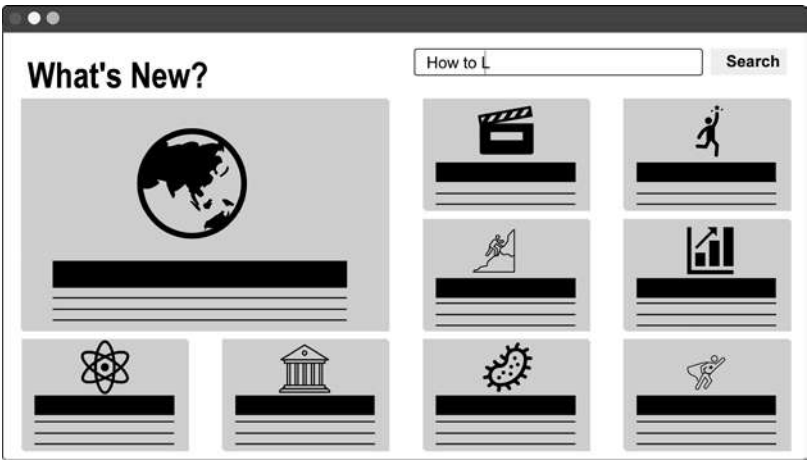


Рис. 8.3. Поисковая система или новостной агрегатор?

Стиль Material Design

Компания Google разработала и в настоящее время твердо придерживается философии *Material Design (материальный дизайн)* и языка дизайна, который описывает способ организации и оформления элементов экрана в соответствии с тем, что пользователи уже и так интуитивно понимают: с объектами физического мира. Это могут быть бумага, карточки, ручки или тень. В предыдущем разделе на рис. 8.3 показан пример материального дизайна. Веб-сайт структурирован в виде карточек, каждая из которых представляет собой часть контента, что создает разбивку, напоминающую газетную — с изображением и текстом заголовка. Внешний вид сайта создает

ощущение материальности, хотя трехмерный (3D) эффект является чистой иллюзией на двумерном (2D) экране.

На рис. 8.4 сравниваются материальный (слева) и нематериальный дизайн с удаленными необязательными элементами (справа). Вы можете возразить, что второй дизайн более минималистичен, и в некотором смысле вы будете правы. Он занимает не так много места, в нем меньше цветов и нефункциональных визуальных эффектов, таких как затенение. Однако, не имея привычно очерченных границ и интуитивно знакомой разбивки текста, нематериальный дизайн часто сбивает читателя с толку. Истинный минималист всегда использует меньше ценных ресурсов для выполнения одной и той же задачи. В некоторых случаях это означает снижение количества визуальных элементов сайта. В других, наоборот, их добавление для сокращения времени, необходимого человеку для принятия решения. Как правило, время пользователя — это гораздо более ценный ресурс, чем экранное пространство.

Полное введение в философию Material Design со множеством красивых примеров можно найти на сайте https://material.io/design/.

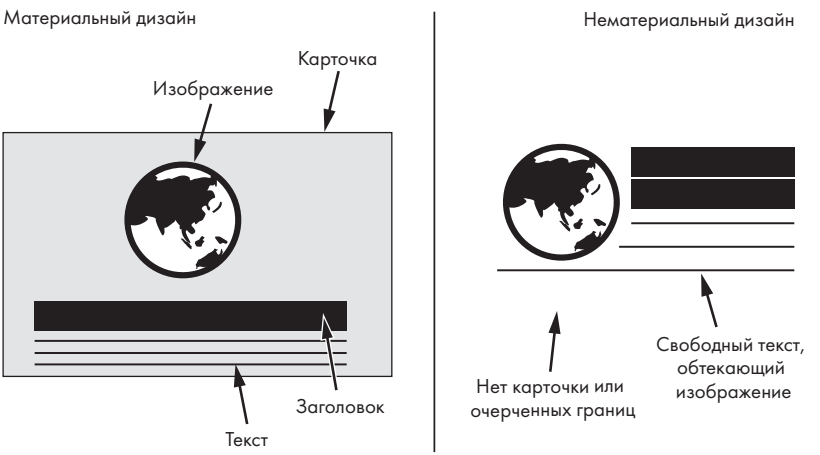


Рис. 8.4. Материальный и «нематериальный» дизайн

Со временем появятся новые тренды в дизайне, и пользователи будут все больше привыкать к цифровым произведениям искусства, поэтому материальные образы вполне могут потерять свою актуальность для следующих поколений пользователей компьютеров. Пока же просто отметим, что минимализм требует тщательного учета соответствующих ресурсов: времени, пространства и денег, и вы должны оценивать их с точки зрения потребностей вашего приложения. Подводя итог, можно сказать, что минимализм в дизайне избавляет от всех ненужных элементов и приводит к созданию прекрасного продукта, который наверняка понравится пользователям.

Далее вы узнаете, как этого добиться.

Как достичь минимализма в дизайне

В этом разделе вы познакомитесь с некоторыми полезными советами и техническими приемами, чтобы создать минималистичный дизайн, сфокусированный на конкретной цели.

Свободное пространство

Свободное (пустое) пространство — один из ключевых элементов минималистичного дизайна. Появление пустого места в вашем приложении может показаться бесполезной тратой драгоценной полезной площади. Нужно быть сумасшедшим, чтобы не использовать каждый квадратный сантиметр часто посещаемого сайта, верно? Вы можете использовать его для рекламы, призывов к действию, продажи большего количества товаров, размещения дополнительной информации о привлекательных предложениях или для более персонализированных сообщений. Чем успешнее становится ваше приложение, тем больше стейкхолдеров будут бороться за каждую толику внимания, которую они могут получить от пользователей, и вряд ли кто-нибудь попросит вас убрать эти рекламные элементы, заполняющие собой все свободное пространство.

«Субтрактивное»[1] мышление, возможно, кажется не совсем естественным, однако замена элементов дизайна пустым пространством повышает наглядность представления информации и приводит к более сфокусированному UX. С помощью этого подхода успешным компаниям удается сохранить главное в своих приложениях — сфокусированность и заточенность на определенную задачу. Например, на целевой странице Google много свободного пространства, а Apple использует его на презентациях продукции. Думая о своих пользователях, помните следующее: если вы собьете их с толку, вы их потеряете. Свободное пространство добавляет строгости пользовательским интерфейсам.

На рис. 8.5 представлена простая идея дизайна онлайн-сервиса доставки пиццы. Свободное пространство фокусирует внимание клиентов на главном: заказе пиццы. К сожалению, не всякая служба доставки окажется настолько смелой, чтобы использовать пустое пространство таким экстремальным образом.

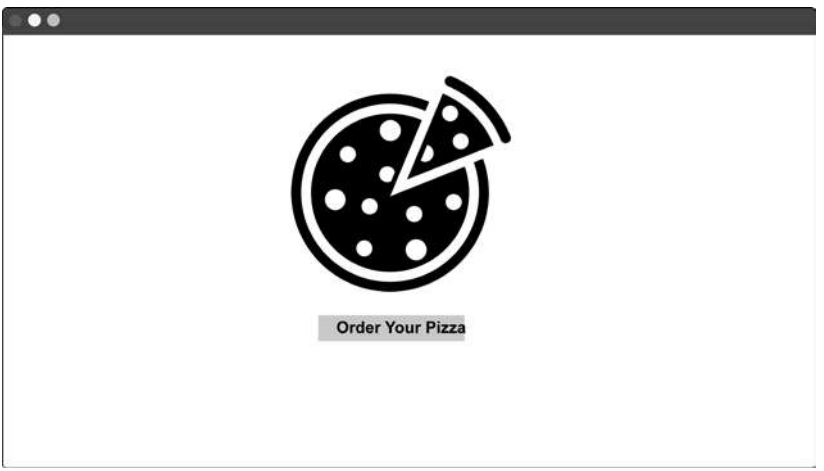


Рис. 8.5. Используйте много свободного пространства

[1] Субтрактивный метод — метод вычитания. — *Примеч. ред.*

Свободное пространство также улучшает читаемость текста. Обратите внимание на рис. 8.6, где сравниваются два способа форматирования абзацев.

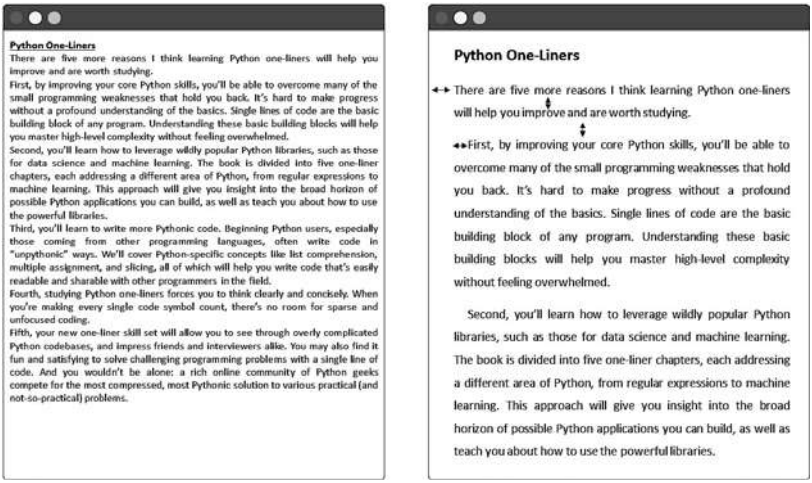


Рис. 8.6. Свободное пространство в тексте

В левой части рис. 8.6 текст гораздо менее удобен для чтения. В правой части для улучшения читаемости текста и UX добавлено свободное пространство: появились поля слева и справа от текстового блока, отступы абзацев; увеличены межстрочное расстояние, верхнее и нижнее поле вокруг абзацев и размер шрифта. Цена за это незначительна: прокрутка не отнимает много ресурсов, ведь нам не нужно физически вырубать деревья для использования большего количества бумаги, когда публикация осуществляется в цифровом формате. С другой стороны, преимущества вполне ощутимы: UX вашего сайта или приложения значительно улучшился!

Сокращайте количество элементов дизайна

Принцип прост: просмотрите все элементы дизайна один за другим и по возможности исключите ненужные. *Элементы дизайна* — это

любые видимые компоненты UI, такие как пункты меню, призывы к действию, рекомендованные списки, кнопки, изображения, окна, тени, поля форм, всплывающие элементы, видео и все остальное, что занимает место. Буквально пройдитесь по всем элементам дизайна и спросите себя: *можно ли это убрать?* Вы удивитесь, как часто ответ оказывается положительным!

Не заблуждайтесь: удаление элементов дизайна — дело непростое! Вы потратили время и усилия на их создание, и необъективность, связанная с невозвратностью издержек, вызывает у вас искушение сохранить свои творения, даже если они не нужны. На рис. 8.7 показан идеализированный процесс редактирования, при котором вы рассматриваете каждый элемент с точки зрения его важности для UX. Например, помогает ли пользователю в процессе оформления заказа пункт меню со ссылкой на блог вашей компании? Нет, поэтому его следует классифицировать как неважный. Так, компания Amazon убрала все ненужные элементы дизайна из процесса оформления заказа, введя кнопку «Купить в один клик». Когда я впервые узнал о таком методе на семинаре по написанию научных статей, мое представление о редактировании полностью изменилось. Удаление неважных и менее важных элементов дизайна гарантирует повышение юзабилити с минимальным риском. Но только по-настоящему великие дизайнеры имеют смелость удалять *важные* элементы дизайна и оставлять только *самые важные*. Именно это и отличает крутой дизайн от просто хорошего.

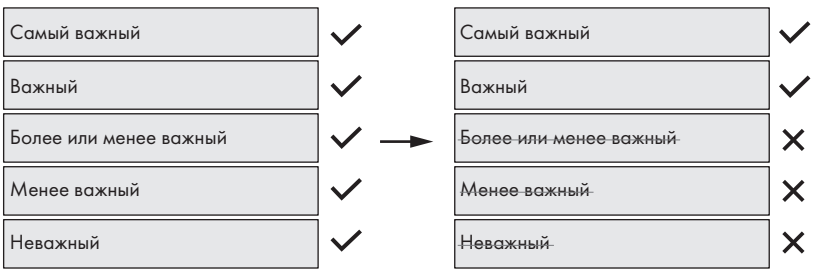


Рис. 8.7. Идеализированный процесс редактирования

На рис. 8.8 показаны примеры перегруженного и минималистичного, отредактированного дизайна. Страница заказа слева — это то, что обычно можно увидеть в онлайн-службе доставки пиццы. Некоторые элементы необходимы, например адрес доставки и кнопка заказа, но такие элементы, как слишком подробный список ингредиентов и информационное окно «Что нового?» (What's New?), менее важны. Справа вы видите отредактированную версию этой страницы заказа. Мы убрали лишние элементы, сосредоточились на самых популярных предложениях, объединили список ингредиентов с заголовком и совместили метки с элементами формы. Это позволило нам добавить больше свободного пространства и даже увеличить размер очень важного элемента дизайна — изображения вкусной пиццы! Уменьшение хаоса и усиление фокуса, скорее всего, повысят коэффициент конверсии страницы заказа за счет улучшения UX.

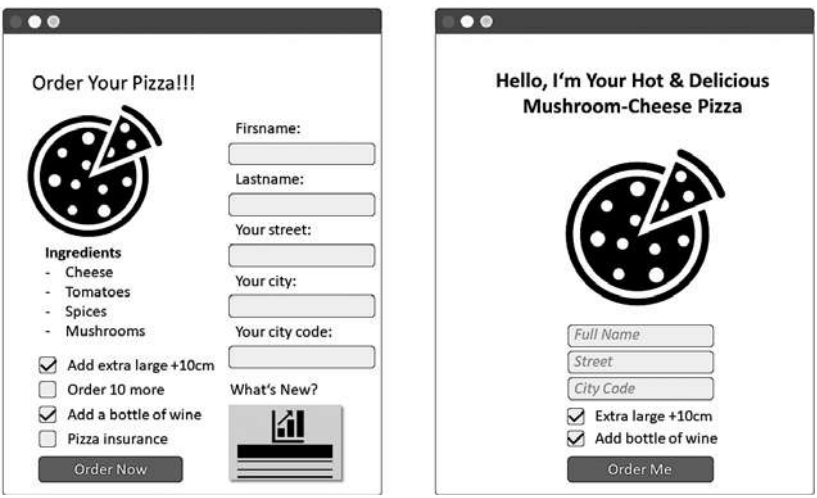


Рис. 8.8. Удаление несущественных элементов: непродуманная страница заказа с большим количеством элементов дизайна (*слева*); оптимизированная страница заказа с удаленными ненужными элементами дизайна (*справа*)

Удаляйте функции

Лучший способ реализовать минималистичный дизайн — это удалить из приложения функции целиком! Вы уже ознакомились с этой идеей в главе 3, в которой рассказывается о создании MVP с минимальным количеством функций, необходимых для проверки предположений. Сокращение числа функций (возможностей) также полезно и для устоявшегося бизнеса, чтобы помочь переориентировать свой ассортимент.

Со временем приложения имеют тенденцию накапливать функционал — это явление известно как *«расползание возможностей»* или *«ползучий улучшизм»*. В результате поддержка существующих возможностей требует все больше и больше внимания. «Ползучий улучшизм» приводит к раздуванию ПО, что в свою очередь ведет к техническим недоработкам и незавершенности. Это уменьшает гибкость в работе организации. Принцип удаления функций заключается в том, чтобы высвободить энергию, время и ресурсы и перенаправить их на те немногие возможности, которые наиболее важны для ваших пользователей.

Популярными примерами «расползания» функционала и его негативного влияния на юзабилити продукта являются Yahoo!, AOL и MySpace, которые так или иначе потеряли свои строго ориентированные продукты, добавляя слишком много элементов в пользовательский интерфейс.

Напротив, самые успешные в мире продукты сфокусированы и стараются противостоять «расползанию возможностей», даже если на первый взгляд кажется, что это не так. Компания Microsoft — отличный пример того, как создание *узкоспециализированных продуктов* помогает стать суперуспешной компанией. Есть мнение, что продукты Microsoft, такие как Windows, медленные, неэффективные и напичканы слишком большим количеством функций. Однако все с точностью до наоборот! *Вы видите только то, что реально существует*, — вы не *видите* мириады функций, которые Microsoft

удалила. Хотя компания Microsoft огромна, она на самом деле очень сфокусированна, учитывая ее размеры. Каждый день сотни тысяч разработчиков ПО пишут новый код Microsoft. Вот что говорит Эрик Траут (Eric Traut), известный инженер, работавший как в Apple, так и в Microsoft, о целенаправленном подходе Microsoft к разработке:

> *Многие люди считают Windows слишком большой и раздутой операционной системой, и это, надо признать, справедливо. Она действительно большая. В ней много всего. Но по существу ядро и компоненты, составляющие ее основу, как ни странно, отлично оптимизированы.*

Подводя итог, можно сказать, что при создании приложения, которым долгое время пользуется огромное количество людей, удаление функций должно стать основной задачей вашей повседневной жизни. Это высвобождает ресурсы, время, энергию и пространство UI: их можно перенаправить на совершенствование значимых функций.

Снижайте количество шрифтов и цветов

Широкая палитра возможностей приводит к избыточной сложности. Если вы слишком часто меняете вид шрифта, его размер и цвет, вы вносите когнитивный диссонанс, повышаете сложность восприятия UI и при этом теряете ясность представления информации. Будучи разработчиком-минималистом, вы не захотите, чтобы эти психологические эффекты повлияли на ваше приложение. В удачном дизайне без излишеств обычно используется только один или два шрифта в одном или двух цветах и не более чем в двух размерах. На рис. 8.9 продемонстрировано последовательное и минималистичное применение шрифтов, размеров, цветов и контрастов. Тем не менее следует отметить, что существует много подходов к дизайну и способов добиться сконцентрированности и минимализма на всех уровнях. Например, даже в сдержанном дизайне может использоваться большая палитра цветов с целью выделить веселые и яркие характерные черты приложения.

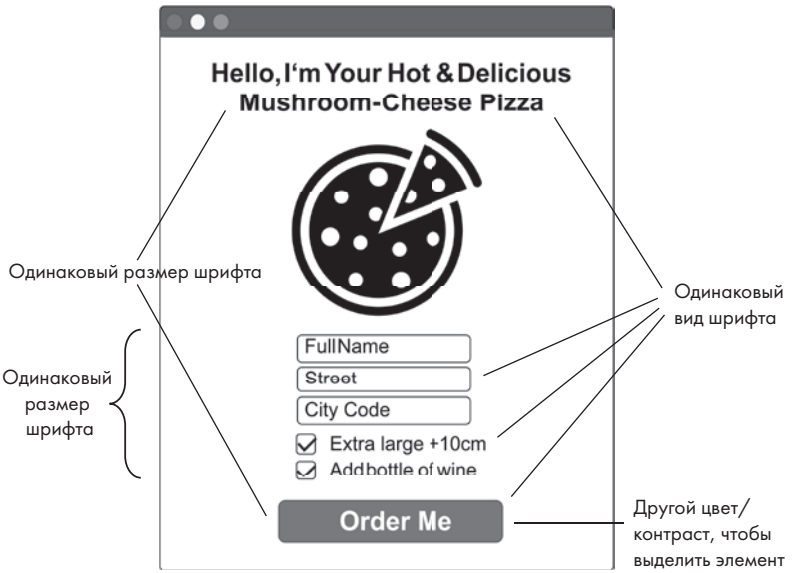


Рис. 8.9. Минимализм в выборе размера шрифта, вида шрифта, цвета и контраста

Будьте последовательны

Приложение обычно состоит не из одного UI, а из серии интерфейсов, осуществляющих взаимодействие с пользователем. Это подводит нас к еще одной стороне минимализма в дизайне: *последовательности (консистентности)*. Определим ее как степень, до которой мы минимизировали вариативность дизайнерских решений в рамках данного приложения. Вместо того чтобы предоставлять пользователю разные «впечатления и ощущения» на каждом этапе взаимодействия с продуктом, последовательный (консистентный) дизайн предполагает, что приложение будет выглядеть как единое целое. Например, компания Apple выпускает множество приложений для iPhone, таких как браузеры, приложения для здоровья и картографические сервисы, но все они имеют схожий внешний вид и узнаваемы как продукты Apple. Добиться от разных разработчиков приложений согласованного последовательного дизайна непросто,

но это чрезвычайно важно для укрепления бренда Apple. Для обеспечения последовательного дизайна в рамках бренда компании-разработчики ПО выпускают *руководства по использованию корпоративного стиля (гайдлайны)*, которых должны придерживаться все вовлеченные специалисты. Обязательно обратите внимание на этот пункт при создании собственного приложения. Последовательности в дизайне можно, в частности, добиться с помощью использования шаблонов и таблиц стилей (CSS).

Заключение

В этой главе основное внимание было уделено тому, как минимализм стал доминировать в мире дизайна, примером чему служат некоторые из самых успешных компаний по разработке ПО, такие как Apple и Google. Чаще всего ведущие технологии и пользовательские интерфейсы максимально просты. Никто не знает, что ждет нас в будущем, но похоже, что широкое внедрение систем распознавания речи и виртуальной реальности приведет к еще большему упрощению UI. Предельно минималистичный дизайн невидим. Я думаю, что в ближайшие десятилетия с развитием цифровизации и высоких технологий — например, таких, как голосовые помощники Alexa и Siri, — мы увидим еще более простые и сфокусированные UI. Итак, отвечая на поставленный в начале вопрос: *именно так, в дизайне лучше меньше, да лучше!*

В следующей и последней главе этой книги мы обсудим тему фокуса на результате и ее актуальность для современных программистов.

Источники

Apple's documentation of human interface design: https://developer.apple.com/design/human-interface-guidelines/.

Documentation for the material design style: https://material.io/design/introduction/.

9

Фокус

В этой небольшой главе мы кратко ответим на самый важный вопрос данной книги: как сфокусироваться на цели. Мы начинали с обсуждения сложности, которая является источником множества помех для продуктивности. Здесь мы суммируем все ваши знания по борьбе со сложностью, полученные благодаря этой книге.

Оружие против сложности

Главный посыл этой книги заключается в том, что сложность ведет к хаосу. Хаос — это обратная сторона сфокусированности. Чтобы ответить на вызовы, порождаемые сложностью, необходимо использовать мощное оружие — *фокус*.

Чтобы обосновать этот аргумент, рассмотрим понятие *энтропии*, хорошо известное во многих научных областях, таких как термодинамика и теория информации. Энтропия определяет уровень

произвольности, беспорядка и неопределенности в системе. Высокая энтропия подразумевает значительную степень случайности и хаоса. Низкая энтропия означает порядок и предсказуемость. Понятие энтропии лежит в основе знаменитого второго закона термодинамики, который гласит, что *энтропия системы со временем возрастает*.

На рис. 9.1 энтропия проиллюстрирована на примере расположения фиксированного числа частиц. Слева показано состояние с низкой энтропией, где собранные вместе частицы напоминают дом. Расположение каждой из них предсказуемо: согласно порядку и структуре более высокого уровня. Существует большой план местоположения частиц. Справа вы видите состояние с высокой энтропией: структура дома разрушилась. Схема расположения частиц утратила порядок, уступив место хаосу. Со временем энтропия будет расти (если никакая внешняя сила не приложит энергию для ее уменьшения) — и порядку придет конец. Разрушенные замки, например, являются свидетельством второго закона термодинамики. Вы можете спросить: какое отношение эта наука имеет к продуктивности написания кода?

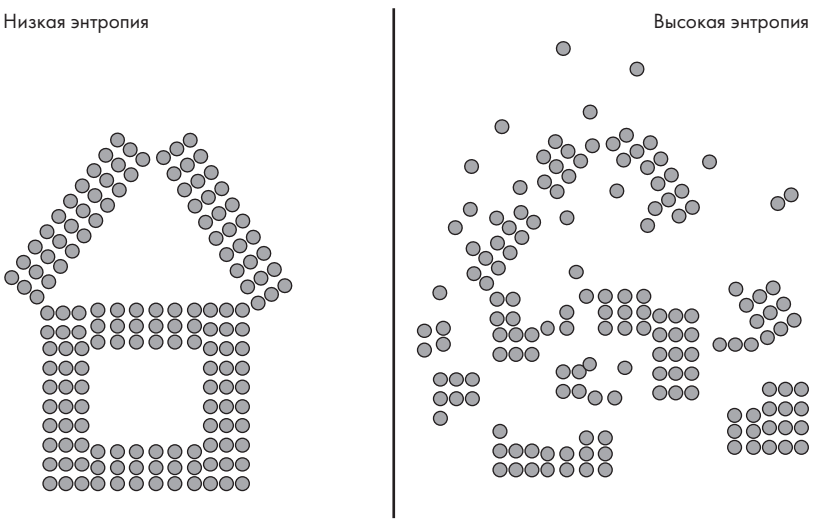


Рис. 9.1. Противоположные состояния с низкой и высокой энтропией

Это станет ясно немного позже. Давайте продолжим размышлять, отталкиваясь от основных принципов.

Продуктивность означает создание чего-либо, будь то строительство дома, написание книги или приложения. По сути, чтобы быть продуктивным, вы должны *уменьшить энтропию*, то есть сделать так, чтобы ресурсы были распределены наилучшим образом для воплощения в жизнь вашего большого плана.

На рис. 9.2 показана взаимосвязь между энтропией и продуктивностью. Вы — творец и строитель. Вы берете исходные ресурсы и переводите их из состояния с высокой энтропией в состояние с низкой, используя целенаправленные усилия для достижения задуманного. Вот и все! В этом весь секрет и то единственное, что вам нужно в жизни для суперпродуктивности и успеха: уделите время тщательному *планированию* своих действий, поставьте конкретные цели, разработайте систематические привычки и практические шаги, которые приведут вас к желаемому результату. Затем *сосредоточьте все усилия*, используя все имеющиеся ресурсы: время, энергию, деньги и людей, — пока ваш план не воплотится в жизнь.

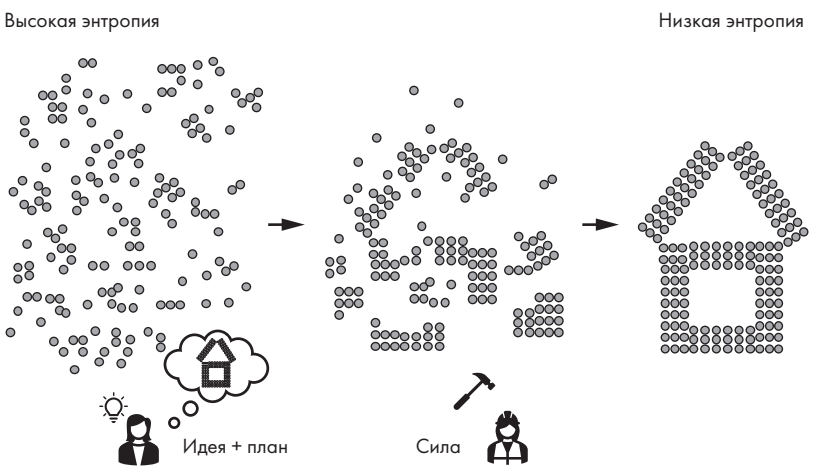


Рис. 9.2. Взаимосвязь между энтропией и продуктивностью

Это может звучать банально, но большинство людей не делают этого. Возможно, некоторые из них никогда не смогут сосредоточить усилия на реализации своей идеи, и она так и останется только в мечтах. Другие живут изо дня в день, никогда не планируя ничего нового. Только если вы будете делать и то и другое, то есть тщательно планировать и фокусировать свои усилия, вы станете продуктивной личностью. Итак, чтобы создать, скажем, приложение для смартфона, вы должны навести порядок в хаосе, планируя свои шаги и прилагая целенаправленные усилия до тех пор, пока не достигнете цели.

Если это так просто, почему все так не делают? Основным препятствием, как вы уже догадались, является сложность, которая часто возникает из-за недостатка фокуса. Если у вас несколько планов или вы позволяете им меняться чаще, чем следует, за короткий промежуток времени вы, скорее всего, продвинетесь всего на несколько шагов к своей цели, а затем все бросите. Только если вы достаточно долго сконцентрированы на *одном* плане, вы сможете его выполнить. Это касается и небольших достижений, таких как прочтение книги (вы уже почти сделали это!), и крупных, например написания и публикации вашего первого приложения. Фокус — обычно недостающий элемент.

На рис. 9.3 представлена простая и понятная схема, объясняющая силу фокусировки на цели.

Ваши время и энергия ограничены. Допустим, в течение дня у вас есть восемь часов для полноценной работы. Вы можете сами решить, как их использовать. Большинство людей расходуют время по чуть-чуть на множество разных дел. Например, Боб может потратить по одному часу: на совещания, на написание кода, на просмотр социальных сетей, на обсуждение проекта, на болтовню, на редактирование документации по коду, на обдумывание нового проекта и на написание романа. Боб, скорее всего, достигнет в лучшем случае средних результатов во всех видах деятельности, которыми он занимается, потому что он тратит слишком мало времени и усилий на каждый из них. А Алиса проводит все восемь часов за одним занятием: за написанием кода. Она делает это каждый день и быстро продвигается

к своей основной цели — публикации успешного приложения. Алиса старается стать исключительной в выполнении нескольких вещей, а не средней во многих. На самом деле она преуспевает только в одном полезном навыке: написании кода. И ее движение к этой цели не остановить.

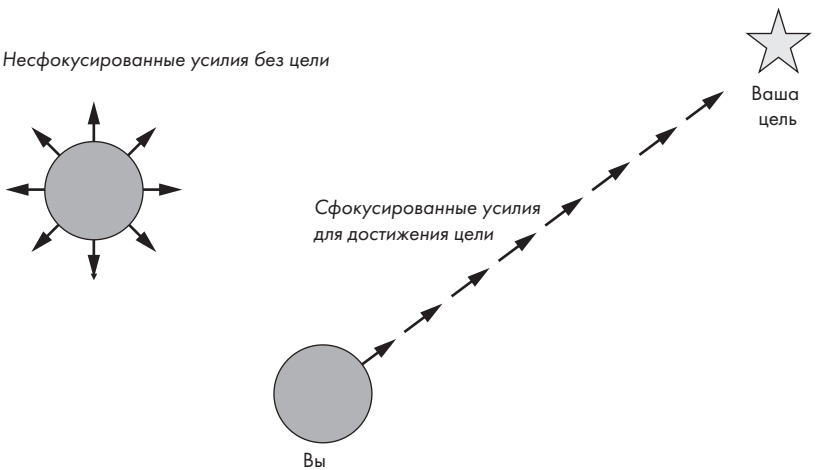


Рис. 9.3. Одинаковые усилия, разный результат

Обобщим все принципы

Я начал писать эту книгу, предполагая, что фокус на главном — это всего лишь один из многих принципов продуктивности, но быстро понял, что это объединяющее правило для всех принципов, изложенных в данной книге. Давайте попробуем обобщить:

Принцип 80/20

Сосредоточьтесь на жизненно важном меньшинстве: помните, что 20 % усилий дают 80 % результатов, и игнорируйте тривиальное большинство, повышая свою продуктивность на один или два порядка.

Создавайте MVP

При создании MVP фокусируйтесь только на одном предположении, тем самым снижая сложность, предотвращая «расползание возможностей» и максимально ускоряя процесс приведения продукта в соответствие рынку. Прежде чем написать хоть строчку кода, сформулируйте четкую гипотезу о том, чего хочет пользователь. Удалите все функции, кроме абсолютно необходимых. Лучше меньше, да лучше! Тратьте больше времени на обдумывание того, какие функции следует реализовывать, чем на их фактическую реализацию. Выпускайте MVP быстро и часто и все время улучшайте его, тестируя и постепенно добавляя новые функции. Используйте сплит-тестирование для проверки реакции пользователей сразу на два варианта продукта и отказывайтесь от функций, которые не приводят к улучшению ключевых пользовательских метрик.

Пишите чистый и простой код

Сложность затрудняет понимание кода и увеличивает риск ошибок. Как мы узнали от Роберта К. Мартина, «соотношение времени, затрачиваемого на чтение и написание кода, значительно превышает 10 к 1. Мы постоянно читаем старый код, пытаясь написать новый». Если ваш код легко читать, то писать новый будет проще. В своей знаменитой книге «The Elements of Style» («Элементы стиля») (Macmillan, 1959) Уильям Странк и Элвин Уайт (William Strunk Jr.; Elwyn B. White) сформулировали бессмертный принцип улучшения качества написания текстов: *опускайте ненужные слова*. Я предлагаю вам распространить этот принцип на создание программ и *опускать ненужный код*.

Преждевременная оптимизация — корень всех зол

Сосредоточьте свои усилия по оптимизации на том, что действительно важно. Преждевременная оптимизация — это пустая трата бесценных ресурсов на то, что в конечном итоге оказывается

ненужным. Как говорит Дональд Кнут, «нас в 97 % случаев не должна заботить низкая эффективность: преждевременная оптимизация — корень всех зол». Я поведал о шести лучших, с моей точки зрения, советах по отладке производительности: используйте метрики для сравнения, учитывайте принцип 80/20, вкладывайтесь в улучшение алгоритмов, применяйте принцип «лучше меньше, да лучше», кэшируйте повторяющиеся результаты и знайте, когда остановиться, — все эти советы можно объединить одной фразой: *«фокусируйтесь на цели»*.

Состояние потока

В состоянии потока вы полностью поглощены выполнением поставленной задачи — вы сосредоточенны и сконцентрированы. Исследователь этого состояния Чиксентмихайи определил три условия для его достижения. (1) Ваши цели должны быть четко сформулированы. Каждая строчка кода приближает вас к успешному завершению большого проекта. (2) Механизм обратной связи с вашим окружением должен быть постоянным и желательно незамедлительным. Найдите людей (лично или в интернете), которые будут оценивать вашу работу, и следуйте принципу MVP. (3) Существует баланс между потенциалом и возможностями. Если задача примитивна, вы быстро растеряете энтузиазм; если она чересчур трудна, вы рано сдадитесь. Соблюдая эти условия, вы с большей вероятностью достигнете состояния истинной сосредоточенности. Ежедневно спрашивайте себя: что я могу сделать *сегодня*, чтобы вывести свой программный проект на новый уровень? Этот вопрос является сложным, но не слишком.

Делай что-то одно, но делай это хорошо (Unix)

Основная идея философии Unix заключается в создании простого, ясного, лаконичного, модульного кода, который легко расширять и поддерживать. Это может означать что угодно, но основная цель состоит в том, чтобы позволить разным людям совместно работать над кодовой базой, отдавая приоритет

эффективности человека, а не компьютера, предпочитая композиционность монолитному дизайну. Вы ориентируете каждую функцию только на одну задачу. Вы изучили 15 принципов Unix для того, чтобы писать более совершенный код, среди них: «малое — прекрасно», «пусть каждая функция делает хорошо что-то одно», «создавайте прототип как можно быстрее» и «лучше, если сбой случится раньше и громко». Необязательно держать в голове все эти правила: запомнив принцип *фокуса*, вы и так будете работать в соответствии с ними.

В дизайне лучше меньше, да лучше

Здесь речь идет об использовании минимализма в дизайне. Подумайте о различиях между Yahoo Search и Google Search, Blackberry и iPhone, OkCupid и Tinder: победителями часто становятся технологии с радикально простым пользовательским интерфейсом. Используя минималистичный дизайн веб-сайта или приложения, вы сосредоточиваетесь на том, что у вас получается лучше всего. Сфокусируйте внимание пользователя на уникальных возможностях, которые предоставляет ваш продукт!

Заключение

Сложность — ваш враг, потому что она повышает энтропию, которую вы как создатель и творец стремитесь снизить: истинный акт творения — это минимизация энтропии. Вы достигаете этого, прилагая целенаправленные усилия. Фокус — секрет успеха каждого творца. Помните о том, что Уоррен Баффет и Билл Гейтс считали секретом своего успеха: *фокус*.

Чтобы применить принцип сфокусированности в работе, задайте себе следующие вопросы:

- На каком программном проекте я хочу сосредоточить усилия?
- На каких функциях я хочу сфокусироваться, чтобы создать MVP?

- Какое минимальное количество элементов дизайна мне понадобится, чтобы проверить жизнеспособность продукта?
- Кто будет использовать мой продукт и почему?
- Что я могу удалить из своего кода?
- Выполняет ли каждая из моих функций какую-то одну задачу?
- Как достичь того же результата за меньшее время?

Если вы продолжаете задавать себе эти или подобные вопросы, помогающие вам фокусироваться, значит, деньги и время, которые вы потратили на эту книгу, того стоили.

От автора

Поздравляю, вы прочитали книгу и получили представление о том, как на деле улучшить свои навыки программирования. Вы познакомились с тактиками написания чистого и простого кода и стратегиями успешных специалистов-практиков. Позвольте мне завершить эту книгу личным комментарием!

Изучив концепцию сложности, вы можете спросить: если идея простоты так сильна, почему же ее не используют все? Дело в том, что ее реализация, несмотря на значительные преимущества, требует огромного мужества, энергии и силы воли. Крупные и небольшие компании часто упорно сопротивляются удалению лишнего и упрощению сложного. Кто-то ведь отвечал за реализацию, поддержку и управление этими функциями, и эти люди будут бороться изо всех сил за сохранение своей работы, даже если знают, что она по большому счету не имеет значения. Проблема заключается в неприятии потерь — трудно отказаться от того, что приносит хоть какую-то пользу. С этим нужно бороться; я никогда не жалел ни об одном упрощении, на которое я пошел в своей жизни. Практически все имеет свою ценность, но важно учитывать, сколько вы заплатите за ее приобретение. Когда я создавал свой образовательный сайт Finxter, я сознательно полностью отказался от социальных сетей и сразу же увидел заметные положительные результаты — появилось дополнительное время, которое можно было потратить на развитие проекта. Упрощение полезно в любых сферах, а не только при разработке кода; оно способно сделать вашу жизнь продуктивнее и одновременно спокойнее. Надеюсь, прочитав эту книгу, вы стали

более открытыми к простоте, исключению лишнего и фокусировке на цели. Если вы все-таки пойдете по пути упрощения, вы окажетесь в хорошей компании: Альберт Эйнштейн считал, что «простой и непритязательный образ жизни полезен для всех; это справедливо и для тела, и для ума». Генри Дэвид Торо (Henry David Thoreau) советовал: «Простота, простота, простота! Сведите свои дела к двум-трем, а не сотням и тысячам». А Конфуций говорил: «Жизнь действительно проста, но мы настаиваем на том, чтобы ее усложнить».

Чтобы помочь вам в дальнейшей борьбе со сложностью, я собрал основные тезисы этой книги в виде одной страницы в формате PDF, которую можно скачать по адресу https://blog.finxter.com/simplicity/, распечатать и прикрепить на стену в своем офисе. Не забудьте также бесплатно подписаться на сервис *Finxter email academy*, где вы сможете получить несколько коротких и простых уроков по программированию. Как правило, мы уделяем внимание таким интересным областям, как язык Python, анализ и обработка данных, разработка блокчейна и машинное обучение. Также мы даем советы и рекомендации по повышению продуктивности, связанные с минимализмом, работой фрилансера и созданием бизнес-стратегии.

Прежде чем мы расстанемся, позвольте мне выразить глубокую признательность за то, что вы провели со мной так много времени. Цель всей моей жизни — помогать людям добиваться большего с помощью кода, и я рассчитываю, что эта книга поможет вам в этом. Я надеюсь, что вы получили представление о том, как повысить продуктивность написания кода, затрачивая меньше усилий. И я верю, что вы начнете создавать свой первый или очередной проект кода сразу, как только перевернете последнюю страницу, а также станете активным участником сообщества программистов-единомышленников Finxter. Желаю вам добиться успеха!