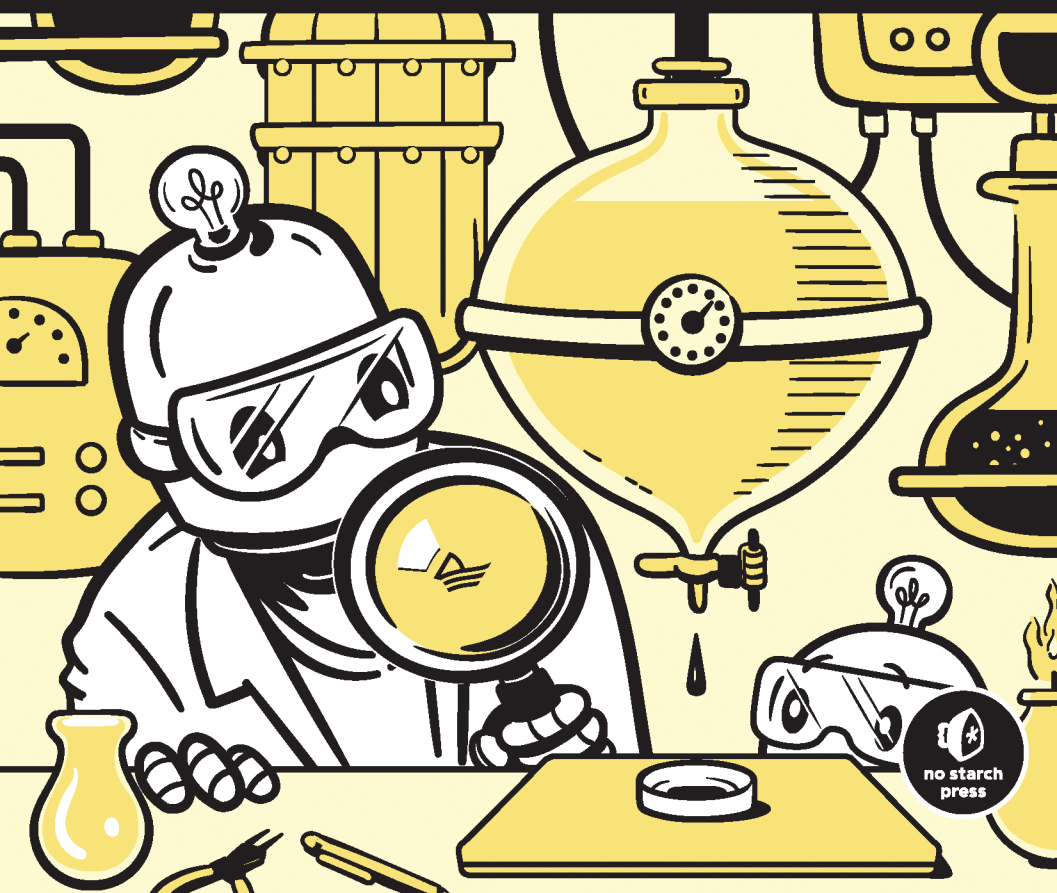
ИСКУССТВО ЧИСТОГО КОДА

КАК ИЗБАВИТЬСЯ ОТ СЛОЖНОСТИ И УПРОСТИТЬ ЖИЗНЬ

КРИСТИАН МАЙЕР



ББК 32.973.2-018-02
УДК 004.415
М14

Майер Кристиан

М14 Искусство чистого кода. — СПб.: Питер, 2023. — 240 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2391-9

Большинство разработчиков ПО тратят тысячи часов на создание излишне сложного кода. Девять основных принципов книги «Искусство чистого кода» научат вас писать понятный и удобный в сопровождении код без ущерба для функциональности. Главный принцип — это простота: сокращайте, упрощайте и перенаправляйте освободившуюся энергию на самые важные задачи, чтобы сэкономить бесчисленное количество часов и облегчить зачастую очень утомительную задачу поддержки кода. Автор бестселлеров Кристиан Майер помог тысячам людей усовершенствовать навыки программирования и в своей новой книге делится опытом с читателями.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018-02
УДК 004.415

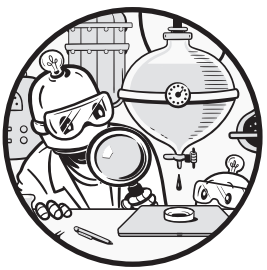
ISBN 978-1718502185 англ.

ISBN 978-5-4461-2391-9

Оглавление

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.