Марк Симан

Код, который умещается в голове

Эвристики для разработчиков

Лучшие практики разработки программного обеспечения, незаменимые советы по написанию кода в устойчивом темпе и по контролю сложности.

Роберт Мартин рекомендует

ББК 32.973.2-018
УДК 004.41
С37

Симан Марк

С37 Роберт Мартин рекомендует. Код, который умещается в голове: эвристики для разработчиков. — СПб.: Питер, 2023. — 400 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2293-6

Незаменимые практические советы по написанию кода в устойчивом темпе и по управлению сложностью, из-за которой проекты часто выходят из-под контроля. В книге описываются методы и процессы, позволяющие решать ключевые вопросы: от создания чек-листов до организации командной работы, от инкапсуляции до декомпозиции, от проектирования API до модульного тестирования. Автор иллюстрирует свои выводы фрагментами кода, взятыми из готового проекта. Написанные на языке C#, они будут понятны всем, кто использует любой объектно-ориентированный язык, включая Java, C++ и TypeScript. Для более глубокого изучения материала вы можете загрузить весь код и подробные комментарии к коммитам.

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

ББК 32.973.2-018
УДК 004.41

Права на издание получены по соглашению с Pearson Education Inc.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-0137464401 англ.

Authorized translation from the English language edition, entitled Code that Fits in your Head: Heuristics for Software Engineering, 1st Edition by Mark Seemann, published by Pearson Education, Inc, publishing as Addison Wesley Professional.

© 2022 Pearson Education, Inc

ISBN 978-5-4461-2293-6

© Перевод на русский язык ООО «Прогресс книга», 2023
© Издание на русском языке, оформление ООО «Прогресс книга», 2023
© Серия «Библиотека программиста», 2023

ОГЛАВЛЕНИЕ

Предисловие Роберта Мартина 20

Введение 24
- Для кого эта книга 25
  - Исходные требования 25
  - Примечание для архитекторов ПО 26
- Структура книги 26
  - О стиле кода 27
  - Типизировать явно или неявно 27
  - Примеры кода 28
- Примечание к библиографии 28
  - О моих книгах 29
- Благодарности 29
- От издательства 29

Об авторе 30

ЧАСТЬ I
РАЗВИТИЕ

Глава 1. Искусство или наука? 32
1.1. Строительство здания 33
1.1.1. Проблема проекта 33
1.1.2. Этапы разработки 34
1.1.3. Зависимости 35
1.2. Возделывание сада 36
1.2.1. Что заставляет сад расти? 36
1.3. С точки зрения инженерии 37
1.3.1. Программирование как ремесло 37
1.3.2. Эвристика 39
1.3.3. Ранние представления о разработке ПО 40
1.3.4. Становление и развитие программной инженерии 41
1.4. Заключение 43

Глава 2. Чек-листы: история, виды, преимущества 45
2.1. Как ничего не забыть 45
2.2. Чек-лист для новой кодовой базы 47
2.2.1. Использовать Git 48
2.2.2. Автоматизировать сборку 50
2.2.3. Включить все сообщения об ошибках 54
2.3. Добавление проверок в существующие кодовые базы 61
2.3.1. Постепенное улучшение 61
2.3.2. «Взломайте» свою организацию 62
2.4. Заключение 63

Глава 3. Преодоление трудностей 65
3.1. Цель 66
3.1.1. Надежность 67
3.1.2. Ценность 68

3.2. Почему программировать так сложно? 70
3.2.1. Аналогия с мозгом 70
3.2.2. Код больше читается, чем пишется 72
3.2.3. Удобочитаемость 73
3.2.4. Интеллектуальный труд 74
3.3. Навстречу программной инженерии 76
3.3.1. Связь с computer science 77
3.3.2. Гуманный код 78
3.4. Заключение 79

Глава 4. Вертикальный срез 80
4.1. Начните с рабочего ПО 81
4.1.1. От поступления данных до их сохранения 81
4.1.2. Минимальный вертикальный срез. 82
4.2. «Ходячий скелет» 84
4.2.1. Характеризационные тесты 85
4.2.2. Паттерн AAA (Arrange-Act-Assert) 87
4.2.3. Модерация статического анализа. 88
4.3. Модель тестирования «от общего к частному» (outside-in). . . 92
4.3.1. Получение данных JSON. 93
4.3.2. Размещение бронирования. 96
4.3.3. Модульное тестирование. 101
4.3.4. DTO и модель предметной области (доменная модель) 103
4.3.5. Fake Object, или фиктивный объект 106
4.3.6. Интерфейс Repository 107
4.3.7. Работа с интерфейсом Repository. 108
4.3.8. Настройка зависимостей. 109
4.4. Завершение среза 111
4.4.1. Схема 111
4.4.2. Репозиторий SQL. 113

4.4.3. Конфигурация базы данных 115
4.4.4. Дымовой тест, или smoke-тестирование 116
4.4.5. Граничный тест с фиктивной базой данных 117
4.5. Заключение 119

Глава 5. Инкапсуляция 120
5.1. Сохранение данных 120
5.1.1. Предпосылки приоритета трансформации (TPP) 121
5.1.2. Параметризованные тесты 123
5.1.3. Копирование данных dto в модель предметной области 124
5.2. Валидация 126
5.2.1. Невалидные данные 127
5.2.2. Цикл «красный, зеленый, рефакторинг» 129
5.2.3. Натуральные числа 132
5.2.4. Закон Постела (принцип надежности) 136
5.3. Защита инвариантов 139
5.3.1. Постоянная валидность 140
5.4. Заключение 143

Глава 6. Триангуляция 144
6.1. Кратковременная и долговременная память 145
6.1.1. Легаси-код и память 146
6.2. Объем памяти 147
6.2.1. Переполнение 148
6.2.2. Метод «Адвокат дьявола» 152
6.2.3. Существующее резервирование 155
6.2.4. Метод «Адвокат дьявола» и цикл «красный, зеленый, рефакторинг» 157
6.2.5. Когда тестов будет достаточно? 160
6.3. Заключение 161

Глава 7. Декомпозиция 163
7.1. Деградация кода 163
7.1.1. Пороговые значения 164
7.1.2. Цикломатическая сложность 166
7.1.3. Правило 80/24 168
7.2. Код, который умещается в вашей голове 170
7.2.1. Гексагональные цветки 170
7.2.2. Связность 173
7.2.3. «Завистливые функции» 177
7.2.4. Потери при передаче 179
7.2.5. Анализ вместо валидации 180
7.2.6. Фрактальная архитектура 183
7.2.7. Подсчет переменных 188
7.3. Заключение 189

Глава 8. Проектирование API 191
8.1. Принципы проектирования API 192
8.1.1. Аффорданс (возможность) 192
8.1.2. Poka-Yoke, или «защита от ошибок» 194
8.1.3. Пишите для читателей 196
8.1.4. Предпочитайте комментариям хорошо написанный код 197
8.1.5. Исключение имен 198
8.1.6. Command Query Separation (CQS), или разделение команд и запросов 201
8.1.7. Иерархия коммуникации 204
8.2. Проектирование API: примеры 205
8.2.1. Класс MaitreD (метрдотель) 206
8.2.2. Взаимодействие с инкапсулированным объектом 209
8.2.3. Детали реализации 212
8.3. Заключение 214

Глава 9. Командная работа 216
9.1. Git 217
9.1.1. Сообщение коммита 218
9.1.2. Непрерывная интеграция 221
9.1.3. Малые коммиты 224
9.2. Коллективное владение кодом 227
9.2.1. Парное программирование 230
9.2.2. Моб-программирование. 231
9.2.3. Задержка код-ревью 232
9.2.4. Отклонение набора изменений 235
9.2.5. Код-ревью 236
9.2.6. Пул-реквесты. 238
9.3. Заключение 240

ЧАСТЬ II
УСТОЙЧИВОСТЬ

Глава 10. Расширение кодовой базы 242
10.1. Функциональные флаги 243
10.1.1. Календарь 244
10.2. Паттерн Strangler («Душитель») 249
10.2.1. Паттерн Strangler. Уровень метода 251
10.2.2. Паттерн Strangler. Уровень класса. 255
10.3. Версионирование 259
10.3.1. Заблаговременное предупреждение 260
10.4. Заключение 261

Глава 11. Редактирование модульных тестов 262
11.1. Рефакторинг модульных тестов 262
11.1.1. Смена подушки безопасности 263

11.1.2. Добавление нового тестового кода . . . 264
11.1.3. Разделяйте рефакторинг тестового и продакшен-кода . . . 267
11.2. Непройденные тесты . . . 273
11.3. Заключение . . . 273

Глава 12. Устранение неполадок . . . 275
12.1. Понимание . . . 275
12.1.1. Научный подход . . . 276
12.1.2. Упрощение . . . 277
12.1.3. Метод утенка . . . 278
12.2. Дефекты . . . 280
12.2.1. Воспроизведение дефектов в виде тестов . . . 281
12.2.2. Медленные тесты . . . 284
12.2.3. Недетерминированные дефекты . . . 287
12.3. Метод бисекции . . . 292
12.3.1. Метод бисекции с Git . . . 292
12.4. Заключение . . . 297

Глава 13. Разделение ответственности . . . 299
13.1. Композиция . . . 300
13.1.1. Вложенная композиция . . . 301
13.1.2. Последовательная композиция . . . 304
13.1.3. Ссылочная прозрачность . . . 306
13.2. Сквозная функциональность . . . 310
13.2.1. Логирование . . . 310
13.2.2. Паттерн проектирования Decorator («Декоратор») . . . 311
13.2.3. Что регистрировать . . . 315
13.3. Заключение . . . 317

Глава 14. Организация рабочего процесса 319
14.1. Индивидуальный процесс работы 320
14.1.1. Тайм-боксинг 320
14.1.2. Делайте перерывы 322
14.1.3. Используйте время разумно 323
14.1.4. Метод слепой печати 325
14.2. Рабочий процесс в команде 326
14.2.1. Регулярное обновление зависимостей 326
14.2.2. Планирование других действий 328
14.2.3. Закон Конвея. 329
14.3. Заключение 330

Глава 15. Очевидные аспекты 331
15.1. Производительность 332
15.1.1. Устаревшие знания 332
15.1.2. Удобочитаемость 334
15.2. Безопасность 337
15.2.1. Модель угроз STRIDE 337
15.2.2. Спуфинг 338
15.2.3. Незаконное изменение. 339
15.2.4. Отказ от авторства. 340
15.2.5. Раскрытие информации 341
15.2.6. Отказ в обслуживании 343
15.2.7. Повышение привилегий 344
15.3. Прочие техники 345
15.3.1. Тестирование на основе свойств. 345
15.3.2. Поведенческий анализ кода 351
15.4. Заключение 354

Глава 16. Краткий обзор . . . 356
16.1. Навигация . . . 356
16.1.1. Общее представление . . . 358
16.1.2. Организация файлов . . . 361
16.1.3. Поиск деталей . . . 364
16.2. Архитектура . . . 366
16.2.1. Монолитная архитектура . . . 366
16.2.2. Циклы . . . 367
16.3. Использование . . . 371
16.3.1. Обучение на тестах . . . 372
16.3.2. Прислушивайтесь к своим тестам . . . 374
16.4. Заключение . . . 375

Приложение. Перечень методов . . . 377
П.1. Правило 50/72 . . . 377
П.2. Правило 80/24 . . . 378
П.3. Шаблон Arrange-Act-Assert (AAA) . . . 378
П.4. Бисекция . . . 378
П.5. Чек-лист для новой кодовой базы . . . 379
П.6. Разделение команд и запросов (CQS) . . . 379
П.7. Подсчет переменных . . . 379
П.8. Цикломатическая сложность . . . 380
П.9. Паттерн проектирования Decorator для сквозной функциональности . . . 380
П.10. Метод «Адвокат дьявола» . . . 380
П.11. Функциональный флаг . . . 381
П.12. Функциональное ядро, императивная оболочка . . . 381
П.13. Иерархия отношений . . . 382
П.14. Обоснование исключений из правил . . . 382

П.15. Анализировать, а не проверять . . . 382
П.16. Закон Постела . . . 383
П.17. Цикл «красный, зеленый, рефакторинг» . . . 383
П.18. Регулярное обновление зависимостей . . . 384
П.19. Воспроизведение дефектов в виде тестов . . . 384
П.20. Код-ревью . . . 384
П.21. Семантическое версионирование . . . 385
П.22. Раздельный рефакторинг тестового и продакшен-кода . . . 385
П.23. Срез . . . 385
П.24. Паттерн Strangler . . . 386
П.25. Модель угроз STRIDE . . . 386
П.26. Предпосылки приоритета трансформации (TPP) . . . 387
П.27. X-ориентированная разработка . . . 387
П.28. Исключение имен . . . 388

Библиография . . . 389

1 ИСКУССТВО ИЛИ НАУКА?

Вы ученый или художник? Инженер или ремесленник? Садовник или повар? Поэт или архитектор? Наконец, вы программист или разработчик ПО? Кем вы себя считаете?

Мой ответ таков: «Никем из вышеперечисленного. Я программист, но в то же время немного и повар, и садовник, и художник, и строитель».

Важно задавать такие вопросы. Индустрии разработки ПО в общей сложности около 70 лет, и мы все еще не до конца разобрались в этой области. Основная проблема здесь в том, *как мы думаем* о ней. Процесс разработки программного обеспечения похож на строительство дома? Или он напоминает стихосложение?

Десятилетиями программирование сравнивали с чем угодно, даже с возделыванием сада, но лучшей метафоры так и не нашли.

Я считаю, что то, как мы думаем о разработке ПО, влияет на то, как мы работаем. Программист должен подстраиваться под свои проекты. Он должен понимать, что и для чего он пишет.

1.1. Строительство здания

Многие годы разработку ПО сравнивали со строительством здания.

Кент Бек сказал об этом так:

> *«К сожалению, разработка программного обеспечения была скована метафорами физического проектирования» [5].*

Это одно из самых распространенных, неоднозначных и вводящих в заблуждение мнений.

1.1.1. Проблема проекта

Полагая, что разработка ПО похожа на строительство здания, вы совершаете первую ошибку — думаете об этом процессе как о *проекте*. У проекта есть начало и конец. Как только вы дойдете до конца, работа будет сделана.

Полностью завершить можно только неудачное ПО, успешное же будет долговечным. Качественное программное обеспечение предполагает постоянное развитие, которое может длиться годами, а иногда и десятилетиями[1].

После того как здание построено, люди могут в него заселиться. Чтобы поддерживать дом в исправном состоянии, его нужно обслуживать, но затраты на это будут в разы меньше по сравнению с затратами на его проектирование. Конечно, такой софт есть. Например, вы создали внутреннее бизнес-приложение для какой-нибудь корпорации, оно завершено, и пользователи привязаны к нему. По завершении разработки такое ПО переходит в режим обслуживания и сопровождения.

Но большинство конкурентоспособных программных продуктов никогда не будут завершены. Если вы все еще связываете процесс разработки со строительством здания, можете сравнивать его с серией проектов. Вы можете запланировать выпуск следующей версии своего

[1] Эта книга сверстана в LaTeX — ПО, первая версия которого была выпущена в 1984 году!

продукта через девять месяцев, но, к своему ужасу, обнаружите, что ваш конкурент внедряет улучшения каждые три.

Вы начинаете усердно работать над тем, чтобы сократить свои «проекты». И к моменту, когда у вас наконец получится выпускать продукт каждые три месяца, ваш конкурент будет внедрять обновления уже ежемесячно. Вы понимаете, к чему все идет?

Это может превратиться в бесконечную погоню за наращиванием функциональности и выпуском новых версий [49] или привести к разорению. В книге «Ускоряйся!» [29] приводятся научно подкрепленные аргументы того, что ключевым свойством, отличающим высокоэффективные команды от низкоэффективных, служит способность мгновенно обновлять и распространять информацию.

Если вы будете способны это сделать, понятие *проекта* разработки программного обеспечения потеряет свою актуальность.

1.1.2. Этапы разработки

Еще одно заблуждение, связанное с метафорой строительства: ПО нужно разрабатывать в несколько *этапов*. Перед началом работ архитектор создает чертеж. Далее оценивается логистика, на площадку поставляются материалы, и только после этого можно приступать к постройке здания.

В случае если метафора уместна, вы назначаете *архитектора программного обеспечения*, в обязанности которого входит создание плана. Только после этого можно бужет начать разработку ПО. Этот этап — этап проектирования — довольно сложный интеллектуальный процесс. Если возвращаться к аналогии со строительством, то он похож на фактический этап самих строительных работ, где разработчики — это взаимозаменяемые сотрудники[1], вроде машинистов.

Но это очень отдаленное сравнение. Как указал Джек Ривз в 1992 г. [87], этап *создания* программного обеспечения — это когда вы компилируете исходный код. По сравнению со строительством здания этот про-

[1] Ничего не имею против строителей — мой отец был каменщиком.

цесс можно назвать почти бесплатным. Вся работа происходит на этапе проектирования, или, как выразился Кевлин Хенни:

> *«Недвусмысленное описание программы и программирование — это один и тот же процесс» [42].*

В рамках разработки ПО мы не можем говорить об этапе строительства. Это не означает, что проектирование бесполезно, но указывает на то, что метафора с с постройкой здания здесь неприменима.

1.1.3. Зависимости

Строительство ведется в соответствии с определенными нормами и требованиями: сначала нужно заложить фундамент, затем возвести стены и только потом можно устанавливать крышу. Другими словами, все эти процессы взаимосвязаны и взаимозависимы.

Такая аналогия внушает ложную идею того, что зависимостями можно управлять. Я знаком с менеджерами, которые для планирования проекта составляли сложные диаграммы Ганта.

Я работал со многими командами, и большинство из них начинают любой проект с разработки схемы реляционной базы данных (БД). БД — основа большинства онлайн-сервисов, и ни один разработчик не будет спорить с тем, что можно спроектировать пользовательский интерфейс еще до появления базы данных.

Некоторым командам иногда даже не удается создать полностью рабочее ПО. После того как БД спроектирована, они решают, что необходимо создать так называемый каркас приложения, или *фреймворк*. Они продолжают заново изобретать ORM (Object-Relational Mapping, объектно-реляционное отображение), этот Вьетнам computer science [70].

Метафора строительства дома вредна — она заставляет вас думать о разработке ПО определенным образом. Вы упустите возможности, которых не видите из-за того, что ваша точка зрения не совпадает с реальностью. Образно говоря, разработку программного обеспечения вы *можете* начать с установки крыши. Немного позже я подкреплю эти слова примером.

1.2. ВОЗДЕЛЫВАНИЕ САДА

Мы выяснили, что аналогия со строительством не подходит, но, возможно, другие подойдут больше. В 2010-х годах становится популярной метафора возделывания сада. Не случайно Нат Прайс и Стив Фримен назвали свою книгу *Growing Object-Oriented Software, Guided by Tests* [36].

В этом примере ПО сравнивается с живым организмом, который требует особенного отношения, вложения большого количества сил и внимания. Это еще одна вполне убедительная метафора. Вы когда-нибудь думали о том, что кодовая база живет своей жизнью?

Возможно, будет правильнее рассматривать программное обеспечение именно с этой точки зрения? По крайней мере, так мы сможем не ограничиваться только строительством здания.

Теперь, в рамках аналогии с взращиванием сада, мы делаем акцент на обрезке (сокращении). Если не ухаживать за зеленью, она начнет разрастаться. Предотвратить этот процесс может садовник, избавляясь от сорняков и поддерживая культурные растения. При разработке ПО это помогает сосредоточиться на действиях, *предотвращающих* «загнивание» кода, таких как рефакторинг и удаление мертвого кода.

Но мне кажется, что эта метафора тоже не дает нам полного представления о процессе разработки ПО.

1.2.1. Что заставляет сад расти?

Мне нравится, что аналогия с садоводством делает упор на действиях, предотвращающих беспорядок. Точно так же, как вы должны ухаживать за своим садом, вам необходимо проводить рефакторинг и погашать технический долг в своих кодовых базах.

С другой стороны, эта метафора мало говорит о том, откуда берется код. В саду растения растут сами по себе: все, что им нужно, — удо-

брения, солнечный свет и вода. ПО само по себе развиваться не будет. Вы не можете просто забросить компьютер, чипсы и колу в темную комнату и ожидать, что из этого вырастет программное обеспечение. Все это не будет работать без самой важной составляющей — программистов.

Код должен быть написан кем-то. Это активный процесс, и здесь аналогия с садом становится уже не такой актуальной. Как вы решаете, что писать, а что — нет? Как принимаете решение о том, *как* структурировать код?

Мы должны ответить на эти вопросы, если хотим улучшить индустрию разработки программного обеспечения.

1.3. С ТОЧКИ ЗРЕНИЯ ИНЖЕНЕРИИ

Для разработки ПО есть и другие метафоры. Например, термин *«технический долг»*, который я упоминал ранее, можно сравнить с финансовым кредитом. А процесс *написания* кода напоминает некоторые виды авторской деятельности. Все эти аналогии в какой-то степени верны, но ни одна из них не будет абсолютно правильной.

Но я начал эту книгу именно с аналогии со строительством здания. И на то есть несколько причин. Во-первых, это сравнение довольно распространено. Во-вторых, оно кажется настолько неправильным, что его уже невозможно спасти.

1.3.1. Программирование как ремесло

К выводу, что аналогия со строительством вредна, я пришел много лет назад. И как правило, после отказа от одной теории вы обычно начинаете искать другую. Я нашел ее в *ремесле программного обеспечения.*

Давайте рассмотрим разработку ПО как ремесло, ведь, по сути, это и есть *квалифицированный труд.* Вы *можете* получить образование

в области computer science, но это вовсе не обязательно. У меня, например, его нет[1].

Навыки, необходимые профессиональным разработчикам ПО, обычно зависят от ситуации. Изучите, как устроена конкретная кодовая база, научитесь использовать конкретный фреймворк, потратьте время на исправление ошибок в продакшене. От вас будет требоваться что-то вроде этого.

Чем больше вы что-то делаете, тем опытнее вы становитесь. Если вы останетесь в одной компании и будете годами работать с одной и той же кодовой базой, вы можете стать специалистом. Но как это поможет вам при устройстве на другую работу?

Вы будете развиваться быстрее, переходя от одной кодовой базы к другой. Освойте бэкенд- и фронтенд-разработку. Изучите программирование игр или машинное обучение. Так вы гарантированно сможете накопить полезный опыт.

Становление разработчика ПО подобно старой традиции *странствующего подмастерья* в Европе. Ремесленник, плотник или кровельщик путешествовал по разным городам и странам, вкладывая все свои силы в ремесло. Все это открывало большие возможности и позволяло оттачивать свои навыки. В книге «Программист-прагматик» даже есть раздел «Путь от подмастерья к мастеру» [50].

Если это утверждение верно, мы должны соответствующим образом структурировать нашу отрасль. У нас должны быть ученики, работающие вместе с мастерами. Мы могли бы даже организовать гильдии.

Так ведь?

Программирование как ремесло — еще одна метафора. Это похоже на ослепляющий свет истины, но он может таить в себе тени скрытого подтекста. Как говорится, чем ярче свет, тем темнее кажутся тени (рис. 1.1).

---

[1] Если вам любопытно, у меня есть высшее образование в сфере экономики, но, кроме как для работы в министерстве экономики Дании, оно мне больше не пригодилось.

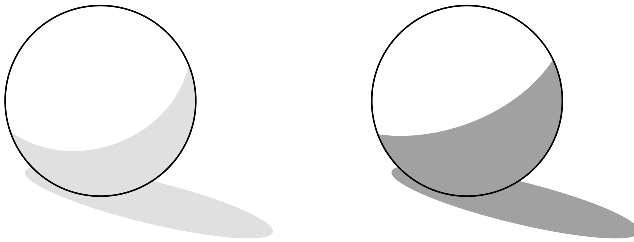

Рис. 1.1. Чем ярче освещена фигура, тем темнее отбрасываемая ею тень

Кажется, на рисунке все еще чего-то не хватает.

1.3.2. Эвристика

Годы, когда я занимался разработкой ПО, в некотором смысле были периодом полнейшего разочарования. Я рассматривал ремесло исключительно с точки зрения накопления опыта. Мне казалось, что нет никаких методологий и правил, что все зависит только от обстоятельств. Я думал, что не существует правильного или неправильного способа что-то делать.

Программирование было своего рода творчеством, что меня вполне устраивало. Мне всегда нравилось искусство. В молодости я даже хотел стать художником[1].

С этой точкой зрения проблема кажется невероятно сложной. Чтобы «взрастить» новых программистов, вам придется их обучать. А на то, чтобы преодолеть путь от подмастерья до мастера, у них может уйти несколько лет.

Еще одна проблема, связанная с отношением к программированию как к искусству или ремеслу, состоит в том, что эта аналогия тоже

[1] Я давно мечтал стать художником комиксов в европейской традиции. Позже, будучи подростком, я взял в руки гитару и решил стать рок-звездой. Но выяснилось, что, хотя мне нравилось и рисовать, и играть, я не был особо талантлив.

не соответствует действительности. Примерно в 2010 году я начал задумываться над тем [106], что все то время, когда программировал, я следовал эвристике — эмпирическим правилам и рекомендациям.

Поначалу я не придал этому большого значения. Но в дальнейшем, в процессе обучения других разработчиков, я часто определенным образом формулировал доводы для написания кода.

Я начал понимать, что ошибался в своих смелых утверждениях, и понял: рекомендации могут стать ключом к превращению программирования в техническую дисциплину.

1.3.3. Ранние представления о разработке ПО

О программной инженерии заговорили ближе к концу 1960-х годов[1]. Это было связано с *кризисом ПО* того времени — с появлением идеи о том, что программировать *сложно*.

На тот момент программисты действительно хорошо понимали, что делают. Многие выдающиеся деятели нашей отрасли трудились в то время: Эдсгер Дейкстра, Тони Хоар, Дональд Кнут, Алан Кей. Если бы вы тогда у них спросили, станет ли в 2020-х годах программирование отдельным предметом для изучения, вероятно, они бы ответили утвердительно.

Вы могли заметить, что я рассматриваю разработку ПО как амбициозную цель, а не как повседневную рутину. Вполне возможно, что в мире есть центры реальной разработки программного обеспечения[2], но, по моему опыту, бóльшая часть разработки программного обеспечения ведется в иной манере.

Я не одинок, предполагая, что разработка ПО — это наше будущее.

---

[1] Термин может быть старше. Я не могу ничего об этом рассказать, так как тогда я еще не родился. Но тот факт, что две конференции НАТО, в 1968 и 1969 годах, популяризировали термин «программная инженерия», не вызывает сомнений [4].

[2] НАСА выглядит достаточно близким к тому, чтобы быть одним из таких центров.

Адам Барр сказал следующее:

> *«Если вы похожи на меня, то вы мечтаете о том дне, когда разработка программного обеспечения будет изучаться вдумчиво и методично, а руководство, предоставленное программистам, будет основываться на экспериментальных результатах, а не на зыбучих песках индивидуального опыта» [4].*

Адам Барр объясняет, что программная инженерия стремительно развивалась, но затем появилось нечто, что помешало ей, — персональные компьютеры. Благодаря их развитию стали появляться разработчики, которые научились программировать самостоятельно. Поскольку они сами могли разбираться с компьютерами, они оставались в неведении относительно уже существовавших знаний.

Такая ситуация сохраняется и по сей день. Алан Кей описывает *компьютерную поп-культуру* так:

> *«Поп-культура презирает историю. Она дает ощущение самобытности и вовлеченности, причастности. Это не имеет ничего общего с сотрудничеством, прошлым или будущим, — это жизнь в настоящем. Я думаю, то же относится и к большинству людей, которые пишут код за деньги. Они понятия не имеют, откуда взялась их культура» [52].*

Возможно, мы потеряли полвека, добившись незначительного развития программной инженерии, но я думаю, что мы могли достичь прогресса в других направлениях.

1.3.4. Становление и развитие программной инженерии

Что делает инженер? Проектирует и контролирует каждый этап строительных работ, начиная от больших сооружений, таких как мосты (рис. 1.2), туннели, небоскребы и электростанции, до крошечных объектов, таких как микропроцессоры[1]. Инженер помогает создавать физические объекты.

---

[1] У меня когда-то был друг, инженер-химик по образованию. После окончания университета он стал пивоваром в Carlsberg. Так что инженеры могут даже варить пиво.

Рис. 1.2. Мост королевы Александрины (Queen Alexandrine Bridge) — арочный мост через пролив Ульвсунн между островами Зеландия и Мён в Дании. Был открыт в 1943 году

Но программисты так не делают, ведь ПО неосязаемо. Как сказал Джек Ривз [87], поскольку физического объекта нет, проектирование не будет ничего вам стоить. Разработка программного обеспечения — это прежде всего проектная деятельность. Написание кода больше похоже на построение плана инженером, а не на работу строителей на объекте.

Настоящие инженеры следуют методологиям, которые, как правило, приводят к успешным результатам. Мы, программисты, тоже хотим так делать. Но мы должны быть предельно внимательны и выполнять только те действия, которые будут целесообразны в нашем контексте. Когда проектируется физический объект, реальная конструкция стоит дорого. Вы не можете просто взять и попробовать построить мост, проверить его в течение какого-то времени, а потом решить, что он не особо хорош, разрушить его и начать все сначала. Поскольку на реальное строительство уходит много средств, инженеры изначально

занимаются расчетами и моделированием. Для расчета прочности моста нужно меньше времени и материалов, чем для его строительства.

Есть целая инженерная дисциплина, связанная с логистикой. Люди занимаются тщательным планированием — самым безопасным и наименее затратным способом создания физических объектов.

Это часть инженерии, которую нам *не нужно* копировать.

Но есть и много других приемов, которые могут нас вдохновить. Инженеры способны и на творческую работу, но она, как правило, четко структурирована. За одними конкретными действиями должны следовать другие. Специалисты контролируют и утверждают работу друг друга, следуют контрольному списку (чек-листу) [40].

Вы тоже можете так сделать.

Я считаю, что изучение эвристики — весьма полезное и занимательное занятие, но в то же время и очень сложное. Адам Барр называет это *зыбучими песками индивидуального опыта*.

Я считаю, что это отражает текущее состояние нашей отрасли. Любой, кто полагает, что у нас есть четкие научные доказательства, должен прочесть The Leprechauns of Software Engineering («Лепреконы программной инженерии») [13].

1.4. ЗАКЛЮЧЕНИЕ

Думая об истории разработки ПО, вы, вероятно, представляете себе успешное развитие отрасли и большое количество достижений. Но многие из этих достижений связаны с аппаратным, а не с программным обеспечением. И все-таки в разработке ПО за последние 50 лет мы добились существенного прогресса.

Сегодня у нас есть гораздо более продвинутые языки программирования, чем полвека назад, доступ к интернету (включая сервисы взаимопомощи наподобие Stack Overflow), объектно-ориентированное и функциональное программирование, автоматизированные среды тестирования, Git, интегрированные среды разработки и пр.

С другой стороны, мы все еще боремся с *программным кризисом*. Хотя можно ли назвать кризисом то, что длится уже полвека?

Несмотря на серьезные усилия, индустрия разработки ПО все еще не похожа на инженерную дисциплину. Между инженерией и программированием есть некоторые фундаментальные различия, и пока мы этого не поймем, мы не сможем добиться прогресса.

Хорошая новость в том, что программисты могут многое почерпнуть у инженеров: образ мыслей и набор процессов, которым можно следовать.

Как отметил научный фантаст Уильям Гибсон, будущее уже наступило, просто оно еще неравномерно распределено[1].

Как говорится в книге «Ускоряйся!», одни организации сегодня используют передовые методики, а другие отстают [29]. Будущее действительно распределено неравномерно. Хорошая новость в том, что прогрессивные возможности — в свободном доступе и, как вы будете их использовать, зависит только от вас.

В главе 2 вы познакомитесь с конкретными методами, которые сможете применить на практике.

[1] Это довольно расплывчатая цитата. Идея и формулировка в целом принадлежат У. Гибсону, но неясно, когда именно он впервые заявил об этом [76].

2 ЧЕК-ЛИСТЫ: ИСТОРИЯ, ВИДЫ, ПРЕИМУЩЕСТВА

Как программисту стать инженером-программистом? Я не утверждаю, что в книге есть однозначный ответ на этот вопрос, но надеюсь, что, прочитав ее, вы сможете выбрать верный для себя путь.

Что касается разработки ПО, то я думаю, что многое нам еще не дано понять. С другой стороны, мы не можем ждать, пока во всем разберемся. Мы извлекаем уроки и учимся на собственном опыте. Действия и методологии из этой книги уже давно придуманы великими людьми[1]. Эти практики до сих пор актуальны для меня и многих специалистов, которых я обучал. Надеюсь, что они будут полезными для вас или вдохновят вас на разработку своих, более совершенных методов.

2.1. КАК НИЧЕГО НЕ ЗАБЫТЬ

Основная проблема разработки ПО в том, что происходит огромное количество процессов, а наш мозг не способен решать несколько задач одновременно.

[1] Достойных людей слишком много, поэтому я не буду их всех здесь перечислять, но вы можете обратиться к библиографии. Я сделал все возможное, чтобы отдать должное этим людям за их вклад, и приношу свои извинения, если кого-то забыл.

Еще мы склонны игнорировать дела, которые не кажутся нам важными в данный момент.

Проблема не в том, что вы не знаете, как делать, а в том, что вы забываете это сделать, хотя знаете, что должны.

Эта проблема касается не только программирования. От этого страдают и, например, пилоты. Последние придумали простое решение проблемы: *чек-листы*.

Я понимаю, что это звучит невероятно скучно, но советую вам ознакомиться с историей происхождения чек-листа. Согласно Атулу Гаванде [40], идея чек-листов появилась в 1935 году, когда во время испытаний разбился прототип бомбардировщика B-17, что едва не привело к закрытию проекта. По сравнению с предыдущими самолетами B-17 оказался слишком сложным и дорогим в производстве. Управлять бомбардировщиком было непросто, что привело к трагедии, в результате которой погибли два члена экипажа, включая пилота.

Расследование авиакатастрофы показало, что причиной крушения стала ошибка пилота. Учитывая, что пилот был одним из самых опытных летчиков-испытателей армейской авиации, вряд ли это можно было бы списать на непрофессионализм. Как позже написали в одной из газет, самолет был слишком сложен, чтобы на нем мог летать один человек [40].

В результате группой летчиков-испытателей было придумано простое решение: создать чек-лист *основных* операций выполнения этапов взлета и посадки.

Простые чек-листы значительно расширяют возможности квалифицированных специалистов, таких как пилоты. При решении сложной задачи вы неизбежно забываете учитывать одно или несколько действий. Контрольный список поможет вам сосредоточиться на сложных частях вашей задачи, отвлекая внимание от разных мелочей. Вам не нужно будет прилагать усилия для выполнения всех простых действий — вы просто будете следовать *пунктам*.

Важно понимать, что чек-листы должны помогать, поддерживать и освобождать специалиста. Они не предназначены для мониторинга

или аудита. Сила таких списков в том, что работа с ними всегда выполняется качественно, единообразно и результативно. Хороший чек-лист поможет добиться необходимого результата и избежать ошибок. Это могут быть просто списки на плакате, в буфере обмена, в скоросшивателе и т. д.

Чек-листы не должны ограничивать вас в действиях. Они предназначены для улучшения организации процесса и повышения качества достигаемого результата.

Американский хирург и общественный деятель Атул Гаванде разработал чек-лист для врачей и медперсонала, который напоминает мыть руки перед операцией, проверять, давно ли введен антибиотик, и проводить рабочие совещания. Результаты оказались впечатляющими [40].

Если пилоты и хирурги смогли использовать чек-листы, то сможете и вы. Суть в том, чтобы *улучшить результат, не улучшая навыки*.

Далее я периодически буду приводить примеры чек-листов. Это не единственный «инженерный подход», который вы изучите, но он самый простой. Пусть это будет хорошим началом!

Контрольный список просто помогает вам не забыть о чем-то. Он не должен ограничивать вас — он существует, чтобы помочь вам не забывать выполнять тривиальные, но важные действия, такие как мытье рук перед операцией.

2.2. ЧЕК-ЛИСТ ДЛЯ НОВОЙ КОДОВОЙ БАЗЫ

Чек-листы из этой книги основаны на моем подходе к программированию и носят рекомендательный характер. Ваши ситуации могут отличаться, поэтому мои рекомендации могут подойти вам только частично. Так же как чек-лист по проверке взлета Airbus A380 отличается от чек-листа для B-17.

Так что просто ознакомьтесь с предложенными примерами и используйте наиболее подходящие для вашей конкретной ситуации либо создавайте свои наподобие.

Вот чек-лист для запуска новой кодовой базы.

1. Использовать Git.
2. Автоматизировать сборку.
3. Включить все сообщения об ошибках.

Я преднамеренно создал такой короткий список. Чек-лист — это не сложная блок-схема с подробными инструкциями. Это простой перечень, который вы можете охватить за несколько минут.

Чек-листы бывают двух видов: *«прочитал — сделал»* и *«сделал — отметил»* [40]. Цель первого — читать каждый пункт и выполнять действия строго друг за другом. Во втором вы выполняете все действия, а затем проверяете и подтверждаете завершенные отметкой.

Я намеренно оставил приведенный выше список расплывчатым и абстрактным, но, так как в нем использовано повелительное наклонение, он будет считаться чек-листом *«прочитал — сделал»*. Но вы можете легко сделать из него чек-лист *«сделал — отметил»*, и я настоятельно советую, чтобы его просмотрел другой человек. Именно так делают пилоты: один читает чек-лист, а другой подтверждает. В одиночку очень легко пропустить важный пункт, но товарищ всегда может вас подстраховать.

Как именно *использовать Git*, *автоматизировать сборку* и *включать все сообщения об ошибках*, зависит от вас. Далее мы разберем пример создания более конкретного чек-листа на основе списка, представленного выше.

2.2.1. Использовать Git

Git — стандартная система контроля версий для проектов с открытым исходным кодом, поэтому я рекомендую использовать именно его[1].

---

[1] Несмотря на то что Git превосходит большинство альтернативных систем, у него есть и ряд недостатков. Самый серьезный из них — это сложный и непоследовательный интерфейс командной строки. Если в будущем появится улучшенная распределенная система управления версиями, то лучше будет выбрать ее. Но на момент написания моей книги такой альтернативы не было.

По сравнению с централизованными системами управления версиями, такими как CVS или Subversion, распределенная система (Git) дает вам огромное преимущество. Но Git может быть полезна, только если вы знаете, как ее использовать.

Это не самая удобная система в мире, но вы — программист. Вам удалось выучить как минимум один язык программирования, и на фоне этого опыта изучить основы Git будет для вас проще простого. Но обязательно потратьте на это время. Не на изучение графического пользовательского интерфейса, а на изучение основного функционала Git, того, как система работает на самом деле.

Git дает возможность смело *экспериментировать* с кодом. Вы можете написать код и, если он не сработает, просто отменить изменения. Именно способность работать в качестве системы контроля версий на *вашем жестком диске* ставит Git выше централизованных систем.

У Git еще есть несколько графических пользовательских интерфейсов (GUI), но в этой книге я остановлюсь на командной строке. Это не только основа Git, но и способ, которым я обычно предпочитаю с ним работать. И так как моя ОС — Windows, я работаю в командной строке Git Bash.

Первое, что нужно сделать в новой кодовой базе, — инициализировать локальный репозиторий Git[1]. Откройте окно командной строки в папке, в которую хотите поместить код. Сейчас вам не нужно беспокоиться об онлайн-сервисах Git вроде GitHub. Вы всегда можете подключить репозиторий позже. Затем пропишите следующую команду[2]:

```
$ git init
```

---

[1] Я бы придерживался этого правила для любой кодовой базы, которая должна прожить больше недели. Иногда я не беспокоюсь об инициализации репозитория Git для действительно эфемерного кода, но мой порог для создания репозитория Git довольно низок. Вы всегда можете все отменить, удалив папку .git.

[2] Символ $ указывает на приглашение командной строки, его не нужно писать перед командой. Я буду использовать его в примерах в этой книге.