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.