

С примерами на C#

Кодер с улицы

Правила нарушать
рекомендуется

Седат Капаноглу



Street Coder

THE RULES TO BREAK AND HOW TO BREAK THEM

SEDAT KAPANOĞLU



MANNING
SHELTER ISLAND

Кодер с улицы

Правила нарушать рекомендуется

Седат Капаноглу



Санкт-Петербург · Москва · Минск

2023

ББК 32.973.2-018

УДК 004.421

К20

Капаноглу Седат

К20 Кодер с улицы. Правила нарушать рекомендуется. — СПб.: Питер, 2023. — 320 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2268-4

Джунаам и вчерашним выпускникам вузов катастрофически не хватает «уличного» опыта. Чтобы стать отличным разработчиком, понадобятся вполне конкретные навыки, позволяющие превратить теорию в практику, а также понимание того, в какие моменты можно нарушать казавшиеся незыблемыми правила. Эта книга — справочник по выживанию для начинающего разработчика.

«Кодер с улицы» научит васправляться с реальными задачами. Седат Капаноглу честно делится советами, основанными на личном опыте, а не на абстрактной теории. Вы узнаете, как адаптировать знания, полученные из книг и курсов, к повседневным рабочим задачам.

Пора узнать, как использовать антипаттерны и «плохие» методы программирования. Эта книга построена на конкретных задачах, с которыми вы столкнетесь на работе, — от чисто технических аспектов, таких как создание функции поиска, до законов выживания в проблемной команде с менеджером-параноиком.

Все это превратит вас в настоящего уличного бойца, готового в любой момент приступить к созданию эффективного программного обеспечения.

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

ББК 32.973.2-018

УДК 004.421

Права на издание получены по соглашению с Manning Publications. 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 978-1617298370 англ.

ISBN 978-5-4461-2268-4

© 2022 by Manning Publications Co. All rights reserved

© Перевод на русский язык ООО «Прогресс книга», 2023

© Издание на русском языке, оформление ООО «Прогресс книга», 2023

© Серия «Библиотека программиста», 2023

Краткое содержание

Предисловие	14
Благодарности	16
О книге	18
Об авторе	21
Иллюстрация на обложке.....	22
От издательства	23
Глава 1. На улицы!.....	24
Глава 2. Практическая теория	40
Глава 3. Полезные антипаттерны.....	89
Глава 4. Распробуйте тестирование	138
Глава 5. Вознаграждение за рефакторинг.....	172
Глава 6. Все внимание безопасности.....	193
Глава 7. Самостоятельная оптимизация.....	231
Глава 8. Приятная масштабируемость	264
Глава 9. Жизнь с ошибками.....	294

Оглавление

Предисловие	14
Благодарности	16
О книге.....	18
Для кого эта книга	18
Структура книги	18
О коде в книге	19
Форум liveBook	20
Об авторе	21
Иллюстрация на обложке.....	22
От издательства	23
Глава 1. На улицы!.....	24
1.1. Что важно на улицах.....	25
1.2. Кто такой уличный кодер?.....	26
1.3. Великие уличные кодеры.....	28
1.3.1. Любознательность	29
1.3.2. Нацеленность на результат	29
1.3.3. Высокая производительность.....	30
1.3.4. Умение справляться со сложностями и неоднозначностями	30

1.4. Проблемы современной разработки	31
1.4.1. Слишком много технологий.....	33
1.4.2. Парапланеризм на парадигмах.....	34
1.4.3. Черные ящики технологий	35
1.4.4. Недооценка накладных расходов	36
1.4.5. Не моя работа	37
1.4.6. Рутина — это гениально.....	37
1.5. Чего нет в этой книге	38
1.6. Основные темы книги	38
Итоги	39
Глава 2. Практическая теория	40
2.1. Краткий обзор алгоритмов	41
2.1.1. «О-большое» должно быть приемлемым	44
2.2. Структуры данных изнутри	46
2.2.1. Строки.....	48
2.2.2. Массив.....	51
2.2.3. Список.....	52
2.2.4. Связанный список	53
2.2.5. Очередь.....	55
2.2.6. Словарь.....	55
2.2.7. Хеш-множества.....	59
2.2.8. Стек	59
2.2.9. Стек вызовов.....	60
2.3. К чему весь этот ажиотаж с типами?.....	61
2.3.1. Сила типов.....	62
2.3.2. Проверка правильности.....	63
2.3.3. Используйте фреймворк с умом	69
2.3.4. Типы вместо опечаток	73

8 Оглавление

2.3.5. Быть nullable или non-nullable?	75
2.3.6. Высокая производительность бесплатно.....	82
2.3.7. Ссыльные типы и типы значений.....	83
Итоги.....	87
Глава 3. Полезные антипаттерны.....	89
3.1. Если не сломано, сломай	90
3.1.1. Лицом к лицу с жестью	91
3.1.2. Ломайте скорее	92
3.1.3. Соблюдайте границы	93
3.1.4. Выделение общей функциональности.....	94
3.1.5. Пример веб-страницы.....	96
3.1.6. Не оставляйте за собой долгов	97
3.2. Пишите с нуля.....	98
3.2.1. Стирайте и переписывайте	99
3.3. Чините, даже если ничего не сломано	100
3.3.1. Гонка за будущим	100
3.3.2. Качество кода и культура поведения	102
3.4. Не бойтесь повторяться	104
3.4.1. Повторное использование или копирование?	109
3.5. Изобретайте	111
3.6. Не используйте наследование	114
3.7. Не используйте классы	117
3.7.1. Enum – это ням!	117
3.7.2. Структуры рулят!.....	119
3.8. Пишите плохой код	125
3.8.1. Не используйте If/Else.....	125
3.8.2. Используйте goto	127
3.9. Не пишите комментарии к коду.....	131

3.9.1. Подбирайте длинные имена	133
3.9.2. Эффективно используйте функции	134
Итоги	136
Глава 4. Распробуйте тестирование	138
4.1. Типы тестов	139
4.1.1. Ручное тестирование.....	139
4.1.2. Автоматизированное тестирование.....	139
4.1.3. Опасная жизнь: тестирование в рабочей среде.....	141
4.1.4. Выбор правильной методологии тестирования	142
4.2. Как перестать беспокоиться и полюбить тесты.....	144
4.3. Не используйте TDD и другие сокращения.....	152
4.4. Пишите тесты для своего же блага	153
4.5. Как понять, что именно тестировать	154
4.5.1. Уважайте границы.....	155
4.5.2. Покрытие кода	157
4.6. Не пишите тесты	160
4.6.1. Не пишите код.....	160
4.6.2. Ограничьтесь выборочными тестами.....	160
4.7. Пусть тестированием займется компилятор	161
4.7.1. Как исключить проверки на null.....	161
4.7.2. Как исключить проверки диапазона	165
4.7.3. Как исключить проверки допустимых значений	167
4.8. Именование тестов	170
Итоги	171
Глава 5. Вознаграждение за рефакторинг.....	172
5.1. Зачем нужен рефакторинг?	173
5.2. Изменения архитектуры.....	174
5.2.1. Выделение компонентов.....	177
5.2.2. Оценка объема работы и риска	178

5.2.3. Престиж.....	179
5.2.4. Рефакторинг, чтобы упростить рефакторинг.....	181
5.2.5. Финальное усилие.....	188
5.3. Надежный рефакторинг.....	189
5.4. Когда рефакторинг не нужен	191
Итоги	192
Глава 6. Все внимание безопасности.....	193
6.1. Что еще, кроме хакеров	194
6.2. Моделирование угроз	196
6.2.1. Модели угроз карманного формата	198
6.3. Написание безопасных веб-приложений.....	200
6.3.1. Проектирование с учетом требований безопасности	201
6.3.2. Повышение безопасности через неясность.....	202
6.3.3. Не используйте собственные механизмы безопасности	203
6.3.4. Атаки путем внедрения SQL-кода	204
6.3.5. Межсайтовый скрипting	211
6.3.6. Межсайтовая подделка запроса (CSRF)	217
6.4. Флуд	219
6.4.1. Не используйте капчу.....	219
6.4.2. Альтернативы капче	220
6.4.3. Не применяйте кэш	221
6.5. Хранение секретов	222
6.5.1. Хранение секретов в исходном коде.....	222
Итоги	230
Глава 7. Самостоятельная оптимизация.....	231
7.1. Решаем правильную проблему	232
7.1.1. Простой бенчмаркинг	233
7.1.2. Производительность и время отклика.....	236

7.2. Анатомия медлительности.....	238
7.3. Начните сверху	239
7.3.1. Вложенные циклы	241
7.3.2. Строко-ориентированное программирование	243
7.3.3. Вычисление $2b \parallel !2b$	244
7.4. Разбиваем бутылку по горлышку.....	245
7.4.1. Не упаковывайте данные.....	246
7.4.2. Производите вычисления локально	247
7.4.3. Разделяйте зависимые процессы.....	248
7.4.4. Будьте предсказуемы	250
7.4.5. SIMD	252
7.5. Ввод и вывод	255
7.5.1. Ускоряйте ввод/вывод.....	255
7.5.2. Делайте ввод/вывод неблокирующим	257
7.5.3. Архаичные способы.....	259
7.5.4. Современные операторы <code>async/await</code>	260
7.5.5. Подводные камни асинхронного ввода/вывода.....	261
7.6. Если ничего не помогает, кэшируйте	262
Итоги.....	262
Глава 8. Приятная масштабируемость	264
8.1. Не используйте блокировки.....	266
8.1.1. Блокировка с двойной проверкой	274
8.2. Смиритесь с несоответствиями.....	277
8.2.1. Страшный NOLOCK.....	278
8.3. Не кэшируйте подключения к базе данных.....	280
8.3.1. В виде ORM	284
8.4. Не используйте потоки	285
8.4.1. Подводные камни асинхронного кода	289
8.4.2. Многопоточность и асинхронность	290

8.5. Уважайте монолит	291
Итоги.....	292
Глава 9. Жизнь с ошибками.....	294
9.1. Не исправляйте ошибки	296
9.2. Ужас ошибок	297
9.2.1. Неприятная правда об исключениях	298
9.2.2. Не перехватывайте исключения	300
9.2.3. Устойчивость к исключениям.....	303
9.2.4. Устойчивость без транзакций	308
9.2.5. Исключения и ошибки.....	309
9.3. Не занимайтесь отладкой.....	311
9.3.1. Отладка printf()	312
9.3.2. Дамп-дайвинг	313
9.3.3. Продвинутая отладка с помощью резиновой уточки	317
Итоги.....	318

*Моему брату Музafferу,
который открыл для меня фантастический мир компьютеров*

Предисловие

Будучи самоучкой, я пробовал освоить разработку ПО разными способами (вместо чтения книг): пытался изучить машинный язык, помещая случайные числа в память компьютера и стараясь отследить результат; студентом ночевал в прокуренных кабинетах, тайком улизнув из университетского кампуса после работы в лаборатории; изучал содержимое двоичных файлов в надежде, что эти байты волшебным образом помогут мне понять, как работает код; из-за отсутствия документации запоминал коды операций и пробовал каждую комбинацию порядка аргументов в функции, чтобы определить их правильное расположение.

Еще в 2013 году мой друг Азиз Кеди (Aziz Kedi), владелец книжного магазина в Стамбуле, попросил меня написать книгу о разработке на основе моего опыта. И я впервые задумался о том, чтобы написать о своей профессии. Тогда мне пришлось отложить эту идею, потому что Азиз закрыл свой магазин и переехал в Лондон.

Я продолжал раздумывать о книге, которая помогла бы новичкам в программировании, только вступившим на карьерный путь, заполнять пробелы в опыте, расширяя при этом свой кругозор. Начальное представление о сфере разработки формируется в основном на базе учебных планов, собственных убеждений и так называемых лучших практик. Новоиспеченный программист, естественно, считает полученные знания основой и не хочет далеко от них отходить.

В какой-то момент я окончательно решил написать книгу, но дело продвигалось очень медленно. Я назвал ее «Кодер с улицы» и начал записывать случайные идеи, которые могли бы облегчить жизнь новым разработчикам.

Это не обязательно были лучшие практики, это могли быть даже плохие практики, заставляющие разработчиков принципиально по-новому подходить к проблемам, с которыми они сталкивались. Документ постепенно разрастался, но в какой-то момент я забыл о нем, пока однажды мне не позвонили из Лондона.

На этот раз это был не Азиз Кеди. Тогда он, скорее всего, был занят сценариями, и я уверен, что он работает над очередным из них, пока я пишу этот текст. Звонок был от Энди Уолдрона (Andy Waldron) из Manning Publications. Он спросил меня: «Есть у тебя идея для книги?». Сначала я ничего не мог придумать и хотел только выиграть время, ответив вопросом на вопрос: «Эм-м, что ты имел в виду?». А потом меня осенило. Я вспомнил свои записи и название, которое дал им: «Кодер с улицы».

Это название отражает то, чему я научился на улицах мира профессиональной разработки путем множества проб и ошибок. Это позволило мне взглянуть на разработку pragmatичным, практическим взглядом, как на ремесло. В этой книге рассказывается, как менялась моя точка зрения. А вам с ней будет проще начать свою карьеру!

Благодарности

Эта книга не появилась бы на свет без моей жены Гюньюз. Пока я был занят писательством, она заботилась обо всем остальном. Спасибо, детка. Я тебя люблю.

Спасибо Эндрю Уолдрону, который подтолкнул меня к написанию книги. Это был феноменальный опыт. Энди всегда был терпимым и понимающим, даже когда я обвинил его в том, что он тайно проник в мой дом и изменил текст книги. С меня пиво, Энди.

Спасибо моим редакторам: Тони Арритоле (Toni Arritola), который научил меня всему, что я знаю о написании книг по программированию, и Бекки Уитни (Becky Whitney), которая великодушно довела до ума плохо написанные главы. На самом деле эти главы — дело рук Энди, я серьезно.

Спасибо научному редактору Фрэнсису Буонтемпо (Frances Buontempo) за безукоризненно конструктивные и точные замечания о технических аспектах. Спасибо также Орландо Мендесу Моралесу (Orlando Méndez Morales) за проверку того, что код, которым я делаюсь в книге, действительно имеет смысл.

Спасибо моим друзьям Мурату Гиргину (Murat Girgin) и Волкану Севиму (Volkan Sevim), которые просмотрели первые черновики и признались, что мои шутки были бы смешными, если бы читатель был знаком со мной.

Я благодарю Дональда Кнута (Donald Knuth) за то, что он позволил мне процитировать его. Я считаю везением получить от него любой личный ответ, даже если им было всего лишь «OK». Я также благодарю Фреда Брукса (Fred Brooks)

за напоминание о том, что в законе об авторском праве есть пункт о добросовестном использовании. Наверное, мне не нужно было звонить ему каждый день, чтобы спросить разрешение на цитирование, а также ломиться в дом в 3 часа ночи. Но и вызывать копов было не обязательно, Фред, я же уже уходил! Также спасибо Леону Бэмбрику (Leon Bambrick) за то, что он без эксцессов позволил процитировать себя.

Спасибо читателям МЕАР, особенно Джихату Имамоглу (Cihat İmamoğlu), с которым мы лично не знакомы, но это не помешало ему прислать множество подробных комментариев. Я благодарю всех рецензентов Manning: Адаила Ретамала (Adail Retamal), Алена Кунью (Alain Couniot), Андреаса Шабуса (Andreas Schabus), Брента Хонадела (Brent Honadel), Кэмерона Пресли (Cameron Presley), Дениз Вехби (Deniz Vehbi), Гэвина Бауманиса (Gavin Baumanis), Герта Ван Лаэтема (Geert Van Laethem), Илью Сакаева (Ilya Sakayev), Янека Лопеса Романива (Janek López Romaniv), Джереми Чена (Jeremy Chen), Джонни Нисбета (Jonny Nisbet), Джозефа Перению (Joseph Perenia), Картикеяраджана Раджендрана (Karthikeyarajan Rajendran), Кумара Унникришнана (Kumar Unnikrishnan), Марчина Сенка (Marcin Sęk), Макса Садрие (Max Sadrieh), Михаила Рыбинцева (Michael Rybintsev), Оливера Кортена (Oliver Korten), Онофри Джорджа (Onofrei George), Роберта Уилка (Robert Wilk), Сэмюэла Босха (Samuel Bosch), Себастьяна Феллинга (Sebastian Felling), Тиклю Гангули (Tiklu Ganguly), Винсента Делькоайна (Vincent Delcoigne) и Сюй Яна (Xu Yang) — ваши предложения помогли сделать эту книгу лучше.

И наконец, я благодарю своего папу, научившего меня, что игрушки можно мастерить самостоятельно.

О книге

«Кодер с улицы» заполняет пробелы в профессиональном опыте разработчика, обращаясь к известным парадигмам, демонстрируя антишаблоны и, на первый взгляд, плохие или малоизвестные практики, которые могут быть полезны на улицах профессионального мира. Цель книги — привить читателю любознательность и практическое мышление и помочь понять, что создавать программный продукт — это не только гуглить и печатать код. Я также показываю, что рутинная работа может сэкономить больше времени, чем она сама требует. В целом книга призвана изменить точку зрения на процесс разработки.

ДЛЯ КОГО ЭТА КНИГА

Эта книга предназначена для разработчиков начального и среднего уровня, изучавших программирование и вышедших за пределы обычной учебной программы, но которым все еще не хватает широкого взгляда на парадигмы и лучшие практики разработки. Примеры написаны на C# и .NET, поэтому знакомство с этими языками поможет при чтении. Однако автор стремился, чтобы книга была, насколько это возможно, независима от конкретного языка и его структуры.

СТРУКТУРА КНИГИ

- Глава 1 разъясняет понятие «уличного кодера» — разработчика с профессиональным опытом — и описывает качества, которые помогут стать таким специалистом.

- В главе 2 обсуждается значение теории в практической разработке программных продуктов и почему стоит обращать внимание на структуры данных и алгоритмы.
- В главе 3 объясняется, как некоторые антишаблоны или плохие практики во многих случаях могут быть полезны или даже предпочтительны.
- В главе 4 рассматривается таинственный мир модульного тестирования и то, как оно поможет писать меньше кода и выполнять меньше работы, даже если на первый взгляд кажется, что дело обстоит с точностью до наоборот.
- В главе 5 обсуждаются приемы рефакторинга, как проводить его легко и безопасно и когда его стоит избегать.
- Глава 6 знакомит с основными концепциями и методами обеспечения безопасности и демонстрирует средства защиты от наиболее распространенных атак.
- В главе 7 разбираются некоторые методы жесткой оптимизации, рекомендуется использовать преждевременную оптимизацию и описывается методический подход к устранению проблем с производительностью.
- В главе 8 описываются методы повышения масштабируемости кода, рассматриваются механизмы распараллеливания и их влияние на производительность и скорость отклика.
- Глава 9 посвящена лучшим практикам обработки сбоев и ошибок. В частности, она рекомендует не обрабатывать ошибки и описывает методы написания отказоустойчивого кода.

О КОДЕ В КНИГЕ

Большая часть кода включена в книгу для вспомогательных целей, и в нем могут быть опущены детали реализации, чтобы сосредоточиться на рассматриваемой теме. Полный рабочий код для нескольких проектов находится в онлайн-репозитории GitHub (<https://github.com/ssg/streetcoder>) и на веб-сайте Manning (<https://www.manning.com/books/street-coder>), так что его можно запустить и тестировать локально. В одном примере рассматривается сценарий миграции из .NET Framework, что означает совместимость только с Windows. Альтернативный файл решения для других платформ представлен в репозитории, поэтому создание проекта не должно вызвать проблем.

Книга включает множество примеров кода как в пронумерованных листингах, так и внутри текста абзацев. В обоих случаях код отформатирован моноширинным

шрифтом, чтобы отделить его от обычного текста. Иногда код также выделен **жирным монотипом**, чтобы выделить изменения по сравнению с предыдущими вариантами, например, когда новая функция добавляется к приведенной ранее строке кода.

Во многих случаях исходный код был переформатирован: мы добавили разрывы строк и переработали отступы, чтобы они соответствовали доступному пространству книжной страницы. В редких случаях даже этого было недостаточно, и в листинги пришлось добавить маркеры продолжения строки (➡). Кроме того, комментарии в исходном коде удалялись из листингов, если код описан в тексте. Ко многим листингам добавлены аннотации, чтобы обратить внимание читателей на важные концепции.

ФОРУМ LIVEBOOK

Приобретая книгу «Кодер с улицы», вы получаете бесплатный доступ к закрытому веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу <https://livebook.manning.com/#!/book/street-coder/discussion>. Информацию о форумах Manning и правилах поведения на них см. на <https://livebook.manning.com/#!/discussion>.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы он не терял интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

Об авторе

СЕДАТ КАПАНОГЛУ — разработчик-самоучка из Эскишехира, Турция. Работал инженером в корпорации Microsoft в Сиэтле (США), в подразделении Windows Core Operating System. Его профессиональная карьера в области разработки ПО насчитывает три десятилетия.

Седат — младший из пятерых детей в боснийской семье, эмигрировавшей из бывшей Югославии в Турцию. Он основал популярную турецкую пользовательскую платформу *Ekşi Sözlük* (<https://eksisozluk.com>), что дословно переводится как «кислый словарь». В 1990-х годах активно участвовал в деятельности турецкой демосцены — международного сообщества в сфере цифрового искусства, члены которого занимаются созданием компьютерных графических и музыкальных произведений.

Связаться с ним можно в Twitter (@esesci) или в его авторском блоге, посвященном программированию, на <https://ssg.dev>.

Иллюстрация на обложке

Иллюстрация под названием *Lépero* (что означает «бродяга»), помещенная на обложку, взята из книги Клаудио Линати (1708–1832) «Trajes civiles, militares y religiosos de México», опубликованной в 1828 году. Линати — художник и литограф из Италии, основавший первую литографическую мастерскую в Мексике. В его книге собраны изображения гражданских, военных и религиозных костюмов мексиканцев. Это была одна из первых многокрасочных книг о Мексике, а также первая книга о мексиканцах, написанная иностранцем. В нее вошли 48 раскрашенных вручную литографий с краткими описаниями. Иллюстрации показывают, насколько сильны были культурные различия между регионами, городами, деревнями и районами всего 200 лет назад. Изолированные друг от друга, люди говорили на разных диалектах и языках. По одежде прохожих на улицах городов и деревень было легко определить, где они живут и какова их профессия и социальный статус.

Манера одеваться с тех пор сильно изменилась, и богатое местное разнообразие постепенно ушло в небытие. Сегодня бывает непросто отличить друг от друга людей с разных континентов, не говоря уже о жителях соседних городов или областей. Можно сказать, что мы променяли культурное разнообразие на разнообразие личной жизни, и уж точно на большее разнообразие и динамику технологий.

Во времена, когда одну цифровую книгу трудно отличить от другой, Manning создает обложки, которые напоминают о культурном разнообразии двухвековой давности и возвращают к жизни рисунки из изданий, подобных книге Линати.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

На улицы!

В этой главе

- ✓ Реалии улиц
- ✓ Кто такой кодер с улицы?
- ✓ Проблемы современной разработки
- ✓ Как вам помогает мудрость улиц

Мне везет. Свою первую программу я написал в 1980-х годах. Я всего лишь включил компьютер, написал две строчки кода, добавил RUN — и вуаля! На экране загорелось мое имя.

Открывшиеся возможности меня поразили. Если я смог добиться этого всего двумя строками, то что я смогу сделать шестью или даже двадцатью строками! В мой девяностолетний мозг проникло столько дофамина, что я мгновенно пристрастился к программированию.

Современная разработка значительно сложнее. Примитивные задачи 1980-х, когда взаимодействие с пользователем ограничивалось выводом указания «Нажмите любую клавишу, чтобы продолжить» (Press any key to continue), ушли в прошлое. Хотя уже тогда многие ломали голову, пытаясь найти клавишу «апу» на клавиатуре. Не было ни окон, ни мышек, ни веб-страниц, ни элементов

интерфейса, ни библиотек, ни фреймворков, ни сред выполнения, ни мобильных устройств. Был только набор команд и статическая аппаратная конфигурация.

Для каждого используемого на практике уровня абстракции есть обоснование, и оно не в том, что мы мазохисты (кроме, пожалуй, программистов на Haskell¹). Эти абстракции существуют, потому что являются единственным способом соответствовать современным программным стандартам. Программирование больше не ограничивается выводом имени на экран. Теперь имя должно отображаться определенным шрифтом и располагаться в окне, чтобы можно было перетаскивать его и изменять его размер.

Программа должна выглядеть привлекательно и поддерживать функции копирования и вставки, а также различные имена для обеспечения гибкости конфигурации. Имена, скорее всего, следует хранить в базе данных, может быть, в облаке. Так что выводить на экран свое имя уже не так весело.

К счастью, есть ресурсы, помогающие справляться со сложными задачами: университетские курсы, хакатоны, учебные лагеря, онлайн-курсы и *резиновые уточки*.

ДЛЯ СПРАВКИ Метод утенка, или метод резиновой уточки (rubber duck debugging), — это эзотерический прием поиска решений задач по программированию. Он состоит в том, чтобы разговаривать с желтой игрушечной птичкой. Подробнее я расскажу об этом в главе об отладке.

Полезно вооружиться всеми имеющимися ресурсами, но надо понимать, что этого фундамента не всегда достаточно для высококонкурентной и требовательной среды, в которой трудится разработчик, — среды *улиц*.

1.1. ЧТО ВАЖНО НА УЛИЦАХ

Мир профессиональной разработки неисповедим. Некоторые заказчики клянутся, что заплатят вам через пару дней, каждый раз, когда вы звоните им, уже несколько месяцев кряду. Некоторые клиенты вообще ничего не платят, но обещают, что будут платить, «как только заработают деньги». Вселенная случайным образом решает, кто зайдет кабинет руководителя. Некоторые ошибки исчезают при использовании отладчика. Некоторые команды разработчиков вообще не используют систему контроля версий. Да, это пугает. Но придется смотреть правде в глаза.

¹ Haskell — язык, созданный с целью впихнуть в него как можно больше понятий из академических трудов.

На улицах ясно одно: важнее всего скорость разработки. Никому нет дела до изысков дизайна, знаний алгоритмов или качества кода. Значение имеет только то, сколько кода вы сможете выдать в конкретный момент. Но как ни странно, хороший дизайн, правильное использование алгоритмов и качественный код могут существенно повысить производительность, причем именно это упускают из виду многие программисты. Такие вещи обычно воспринимаются как помехи на пути между кодером и дедлайном. Подобное мышление может превратить вас в зомби, у которого ноги связаны цепью.

На самом деле есть и те, кому небезразлично качество вашего кода. Это ваши коллеги, они не хотят нянчиться с вашим кодом. Они заинтересованы, чтобы ваш код работал, его было легко понять и удобно обслуживать. Это ваш долг, потому что как только вы сделаете коммит своего кода в репозитории, он станет общим кодом. В команде общая производительность важнее, чем производительность отдельного ее члена. Если вы пишете плохой код, то замедляете работу коллег. Некачественный код вредит команде, медлительная команда вредит продукту, а невыпущенный продукт вредит вашей карьере.

Самое простое, что можно описать с нуля, — это идея, а следующий шаг — это дизайн. Вот почему хороший дизайн имеет значение. Хороший дизайн — это не то, что красиво выглядит на бумаге. Можно мысленно создать работающий дизайн и столкнуться с людьми, которые не верят в проектирование и просто выдают код. Эти люди не ценят свое время.

Согласно тому же принципу, хороший шаблон проектирования или хороший алгоритм увеличивает производительность. Если этого не происходит, значит, они бесполезны. Поскольку почти все можно выразить в денежном эквиваленте, любой труд можно оценить по его результатам.

Высокая производительность возможна и с плохим кодом, но только в первой итерации. Когда клиент запросит изменения, придется поддерживать ужасный код. В этой книге я разберу ситуации, когда вы закапываете себя в яму, и расскажу, как выбраться из нее раньше, чем потребуется помочь психиатра.

1.2. КТО ТАКОЙ УЛИЧНЫЙ КОДЕР?

При приеме на работу Microsoft выбирает кандидатов из двух категорий: выпускников профильных факультетов computer science и отраслевых экспертов с хорошим опытом разработки.

Как программисту-самоучке, так и человеку, изучавшему компьютерные науки, в начале карьеры не хватает одного: *мудрости улиц*, то есть знания самого

важного. У программиста-самоучки за плечами множество проб и ошибок, но ему может не хватать знания формальной теории и ее применения в повседневной работе. Выпускник университета хорошо знаком с теорией, но ему не хватает практики, а иногда и критического отношения к своим знаниям (рис. 1.1).

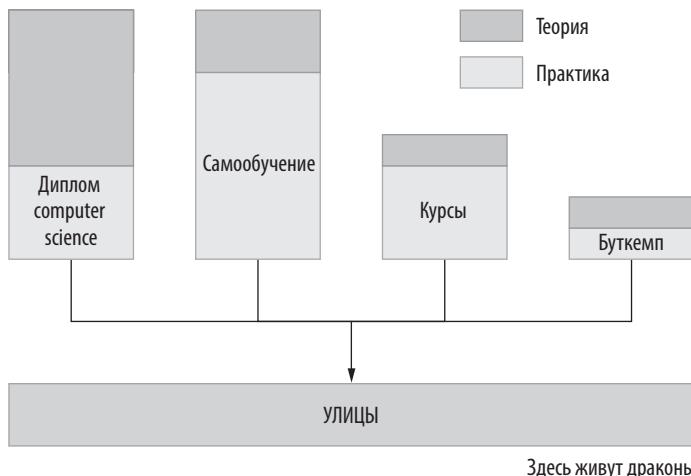


Рис. 1.1. Варианты начала карьеры

Когда вы учитесь, то получаете знания не в порядке их важности. Вы следуете учебному плану, а не приоритетам. Вы даже не представляете, как некоторые навыки могут пригодиться на улицах, где конкуренция беспощадна, сроки нереальные, а кофе вечно холодный. В лучшем фреймворке в мире обнаруживается единственная ошибка, которая превращает в тыкву ваши недельные труды. Ваша идеально спроектированная абстракция рассыпается под давлением заказчика, который постоянно меняет свои требования.

Вам удалось быстро реорганизовать код, используя копирование и вставку, но теперь придется отредактировать 15 его фрагментов только для того, чтобы изменить одно значение конфигурации.

С годами вы развиваете новые навыки, чтобы справляться с неоднозначностью и сложностью. Программисты-самоучки изучают алгоритмы, которые им помогают, а выпускники университетов со временем понимают, что лучшая теория не всегда имеет практическую пользу.

Кодер с улицы — это человек с опытом разработки, чьи убеждения и навыки сформировались под влиянием неадекватного босса, который хочет к утру

получить результат недельной работы. Кодер с улицы научился создавать резервные копии любого содержимого на нескольких носителях после того, как потерял тысячи строк кода и был вынужден переписывать все с нуля. Он видел сияние горящих жестких дисков в серверной и дрался у ее двери с системным администратором, чтобы добраться до продакшена, потому что кто-то только что развернул непроверенный фрагмент кода. Он тестировал новый алгоритм программного сжатия на собственном исходном коде, только чтобы выяснить, что все сжимается в один байт со значением 255, а алгоритм декомпрессии еще только предстоит изобрести.

Вы только что закончили университет и ищете работу или увлеклись программированием, но понятия не имеете, что вас ждет? Вы прошли буткемп и ищете возможности трудоустройства, но не уверены в своих знаниях? Вы выучили язык программирования, но сомневаетесь в своих навыках? Добро пожаловать на улицы.

1.3. ВЕЛИКИЕ УЛИЧНЫЕ КОДЕРЫ

Помимо авторитета, репутации и преданности делу, в идеале уличному кодеру нужны следующие качества:

- Любознательность.
- Желание добиться результата («нацеленность на результат» на языке специалистов по подбору персонала).
- Высокая скорость работы.
- Умение справляться со сложностями и неоднозначностями.

ВЕЛИКИЕ РАЗРАБОТЧИКИ — ЭТО НЕ ПРОСТО ВЕЛИКИЕ КОДЕРЫ

Чтобы стать надежным командным игроком, требуется гораздо больше, чем просто вводить биты и байты в компьютер. Необходимо уметь общаться, давать конструктивную обратную связь и правильно воспринимать критику. Даже Линус Торвальдс¹ признал, что ему нужно поработать над своими коммуникативными способностями. Однако разбор таких навыков выходит за рамки этой книги. Вам придется завести друзей.

¹ Линус Торвальдс создал операционную систему Linux и программное обеспечение для управления исходным кодом Git. Он считал, что ругать волонтеров проекта последними словами — это нормально, если они делают ошибки.

1.3.1. Любознательность

Того, кто разговаривает сам с собой, в лучшем случае считают чудаком, особенно если у него нет ответов на вопросы, которые он себе задает. Однако сомневаясь, задавая вопросы о себе и широко известных понятиях, анализируя их, можно многое понять.

Книги, отраслевые эксперты и Славой Жижек¹ подчеркивают важность критического подхода и любознательности, но не многие из их советов имеют практическую ценность. В этой книге вы найдете примеры очень известных методов и лучших практик, а еще узнаете, когда они бывают не так эффективны, как обещано.

Критиковать какую-то методику не означает заявлять, что она бесполезна. Но критика поможет расширить кругозор и понять, в каких случаях лучше использовать альтернативный метод.

Цель этой книги не в том, чтобы подробно разобрать каждую технику программирования, а в том, чтобы научить вас оценивать возможности лучших практик и взвешивать плюсы и минусы альтернативных подходов.

1.3.2. Нацеленность на результат

Вы можете быть лучшим программистом в мире, превосходно разбирающимся в тонкостях профессии и способным создавать идеальный дизайн кода, но это окажется бесполезным, если вы не доведете разработку до готового продукта.

Согласно парадоксу Зенона², чтобы достичь конечной цели, необходимо сначала преодолеть половину пути до нее. Это парадокс, потому что затем необходимо преодолеть половину оставшегося пути, затем половину той четверти, что осталась, и так далее, что делает невозможным достижение чего-либо. Зенон был прав: чтобы получить конечный продукт, нужно уложиться в окончательные и промежуточные сроки. Иначе цели достичь не получится. Нацеленность на результат означает также ориентацию на промежуточные точки, на прогресс.

«Как можно задержать проект на год? ... День за днем».

Фред Брукс. «Мифический человеко-месяц»

¹ Славой Жижек — современный философ, занимающийся в основном критикой всего на свете.

² Зенон — грек, который жил тысячи лет назад и любил задавать обескураживающие вопросы. Естественно, ни одно из его письменных произведений не сохранилось.

Нацеленность на результат может вести к жертвам в качестве кода, элегантности исполнения и техническом совершенстве. Важно иметь четкое представление о том, что вы делаете и ради кого, и не сходить с пути.

Жертвовать качеством кода не значит жертвовать качеством продукта. Если у вас хорошие тесты и четкий набор требований, можно даже написать все на PHP¹. Однако учтите, что в таком случае вам потом будет больно от плохого кода. Это называется кодовой кармой.

Некоторые приемы, которые вы изучите в книге, помогут вам принимать взвешенные решения, чтобы добиваться нужных результатов.

1.3.3. Высокая производительность

Важнейшими факторами, влияющими на скорость разработки, являются опыт, хорошая и понятная документация, а также использование механической клавиатуры. Шучу: вопреки популярному мнению, механическая клавиатура не повышает скорость. Она просто круто выглядит и бесит вашу вторую половинку. На самом деле я не думаю, что скорость печати как-либо влияет на скорость разработки. Скорее она может подтолкнуть к написанию более сложного кода, чем нужно.

Кое-какой опыт можно получить, учась на чужих ошибках. В этой книге вы найдете описания таких примеров. Полученные знания позволят вам писать меньше кода и быстрее принимать решения, а также иметь как можно меньше технических долгов, поэтому вам не придется тратить несколько суток, разбираясь в коде, который вы написали всего полгода назад.

1.3.4. Умение справляться со сложностями и неоднозначностями

Сложность пугает, а неоднозначность — тем более, потому что неизвестна серьезность угрозы, и это пугает еще больше.

Работа с неоднозначностями — один из основных навыков, который рекрутеры Microsoft стараются проверить на собеседованиях. Обычно это вопросы вроде «Сколько мастерских по ремонту скрипок в Нью-Йорке?», «Сколько

¹ Когда-то PHP был примером того, как не нужно разрабатывать языки программирования. Тем не менее он прошел долгий путь от предмета насмешек до фантастически крутого инструмента, которому осталось только решить кое-какие проблемы с имиджем.

заправочных станций в Лос-Анджелесе?» или «Сколько агентов секретной службы у президента и какой у них график работы? Назовите их имена и, желательно, укажите их маршруты на этом чертеже Белого дома».

Чтобы корректно ответить, нужно выяснить все, что известно о проблеме, и дать приблизительную оценку, основанную на фактах. Например, можно выяснить, сколько всего жителей в Нью-Йорке и какая часть из них умеет играть на скрипке. Это даст представление о размере рынка.

Точно так же, решая задачу с несколькими неизвестными параметрами, такими как оценка затрат времени на разработку функции, всегда можно сузить диапазон возможных значений на основе известных данных. Используйте то, что вы знаете, в своих интересах и как можно эффективнее, поскольку это уменьшит неоднозначность до минимума.

Интересно, что работа со сложностями очень похожа. То, что выглядит чрезвычайно сложным, можно разделить на более управляемые, менее сложные и, в конце концов, более простые части. Чем больше ясности вы вносите, тем проще справиться с неизвестным. Методы, которые вы изучите в этой книге, дадут понимание и уверенность в работе с неоднозначностями и сложностями.

1.4. ПРОБЛЕМЫ СОВРЕМЕННОЙ РАЗРАБОТКИ

Кроме неоправданной сложности, бесчисленных слоев абстракций и необходимости модерации Stack Overflow, в современной разработке есть и другие проблемы:

- Слишком много технологий — языков программирования и фреймворков. И уж точно слишком много библиотек. Например, в прт (менеджере пакетов для фреймворка Node.js) была библиотека с названием `left-pad`, предназначенная исключительно для добавления пробелов в конец строки.
- Зацикленность на парадигме и, следовательно, консервативность. Многие программисты считают языки программирования, лучшие практики, шаблоны проектирования, алгоритмы и структуры данных древними инопланетными реликвиями и понятия не имеют, как они работают.
- Технологии становятся все более непрозрачными, как автомобили. Раньше люди могли ремонтировать свои автомобили сами. Теперь, когда двигатели делаются все совершеннее, все, что мы видим под капотом, — это металлическая крышка, как у гробницы фараона, насылающей проклятия на любого, кто ее откроет. Технологии разработки ПО ничем не отличаются. Хотя почти все решения сейчас имеют открытый исходный код, кажется, что новые

технологии менее понятны, чем код, восстановленный из двоичного файла 1990-х, поскольку сложность программ чрезвычайно возросла.

- Люди не заботятся о накладных расходах кода, потому что в их распоряжении на порядки больше ресурсов, чем ранее. Написали новое простое приложение для чата? Почему бы не объединить его с полнофункциональным браузером, потому что это просто сэкономит время, и никто и глазом не моргнет, ведь вы все равно используете гигабайты памяти!
- Программисты сосредоточены на своей нише, не обращая внимания на работу остальных, и это правильно: им нужно подать еду к столу, а учиться некогда. Я называю это «проблемой обедающих разработчиков». Многие вещи, влияющие на качество продуктов, остаются незамеченными из-за имеющихся ограничений. Веб-разработчики обычно понятия не имеют, как работают сетевые протоколы. Они принимают задержку при загрузке страницы как данность и учатся с ней жить, поскольку не знают, что на нее влияет такая не-значительная техническая деталь, как чересчур длинная цепочка сертификатов.
- Заученное вместе с парадигмами неприятие рутинных операций, таких как повторение, копирование и вставка. От вас ожидают решения в стиле DRY¹. Такая культура заставляет вас сомневаться в себе и своих способностях и, как следствие, вредит вашей продуктивности.

ИСТОРИЯ NPM И LEFT-PAD

В последнее десятилетие npm стал де-факто экосистемой библиотек JavaScript. Люди могли добавлять свои собственные пакеты в экосистему, а другие пакеты могли их использовать, что упрощало разработку крупных проектов. Азер Кочulu (Azer Koçulu) был одним из таких разработчиков, и Left-pad был одним из 250 пакетов, которые он добавил в npm. У этой библиотеки была единственная функция: добавлять пробелы к строке, чтобы она всегда имела фиксированный размер.

Однажды он получил электронное письмо от npm, в котором сообщалось об удалении одного из его пакетов — Kik, потому что поступила жалоба от компании с таким же названием. Это так разозлило Азера, что он удалил все свои библиотеки, включая Left-pad. Однако сотни масштабных проектов по всему миру прямо или косвенно использовали этот пакет, и его удаление привело к остановке всех этих проектов. Это была настоящая катастрофа и хороший урок доверия к открытым платформам.

Мораль этой истории в том, что жизнь на улицах полна неприятных сюрпризов.

¹ DRY. Не повторяйтесь (англ. Don't Repeat Yourself). Суеверие, гласящее, что если кто-то повторит строку кода, вместо того чтобы обернуть ее функцией, он тут же превратится в жабу.

В этой книге я предлагаю некоторые решения перечисленных проблем, включая разбор основных концепций, которые вы, возможно, считали скучными. Ключами к успеху являются упор на практичность, простоту, отказ от некоторых укоренившихся убеждений и, что самое важное, сомнение во всем, что мы делаем. А самое ценное качество — умение задавать вопросы перед началом работы.

1.4.1. Слишком много технологий

Мы находимся в постоянном поиске лучшей технологии — несуществующей серебряной пули¹, которая может увеличить производительность на порядок. Возьмем, например, Python² — интерпретируемый язык, код которого не нужно компилировать. Более того, не нужно указывать типы для объявляемых переменных, что делает работу еще быстрее. Следовательно, Python — лучшая технология, чем C#? Не обязательно.

Поскольку вы не тратите время на аннотирование кода с помощью типов и его компиляцию, вы упускаете ошибки. Это значит, что вы можете обнаружить их только во время тестирования или в готовом продукте, а это обойдется намного дороже, чем простая компиляция кода. Большинство технологий — это компромиссы, а не стимуляторы производительности. Вашу производительность действительно повышает знание нужной технологии и методов, а не то, какие именно технологии вы используете. Да, есть технологии, которые превосходят аналоги, но они редко лучше на порядок.

Когда в 1999 году я захотел разработать свой первый интерактивный веб-сайт, то совершенно не представлял, как писать веб-приложение. Если бы я сначала попытался найти лучшую технологию, то пришлось бы изучать VBScript или Perl. Вместо этого я использовал то, что знал тогда лучше всего: Pascal³. Это был один из самых неподходящих для поставленной задачи языков, но он сработал. Конечно, не обошлось без проблем. Всякий раз при зависании процесс оставался активным в памяти на случайном сервере в Канаде, и пользователю приходилось просить провайдера перезапустить физический сервер. Тем не менее я смог

¹ Отсылка к статье Фредерика Брукса 1986 г. «Серебряной пули нет», в которой автор утверждает, что не существует универсального метода — так называемой серебряной пули, увеличивающей на порядок производительность, надежность и простоту. — *Примеч. ред.*

² Python — это результат коллективных усилий по раскрутке пробелов, замаскированных под практичный язык программирования.

³ Ранний исходный код *Ekşi Sözlük* доступен на GitHub: <https://github.com/ssg/sozluk-cgi>.

быстро создать прототип, потому что инструментарий был мне удобен. Вместо того чтобы тратить месяцы на обучение, я написал и выпустил код за три часа.

В этой книге я покажу вам, как эффективно использовать имеющиеся инструменты.

1.4.2. Парапланеризм на парадигмах

Первой *парадигмой*, с которой я столкнулся, было структурное программирование в 1980-х годах. Структурное программирование — это, по сути, написание кода блоками, такими как функции и циклы, вместо номеров строк, операторов GOTO, крови, пота и слез. Такой код было легче читать и поддерживать без ущерба для производительности. Структурное программирование пробудило во мне интерес к таким языкам, как Pascal и C.

Следующей парадигмой, с которой я познакомился, стало объектно-ориентированное программирование, или ООП. Оно появилось примерно через пять лет после того, как я узнал о структурном программировании. Я помню, что компьютерные журналы того времени не могли им насытиться. Это была выдающаяся методология, благодаря которой программы стали еще лучше.

После ООП я думал, что новые парадигмы будут появляться каждые пять лет или около того. Однако это стало происходить чаще. Девяностые познакомили нас с JIT-компилируемыми¹ управляемыми языками программирования, когда появились Java, веб-скрипты на JavaScript и функциональное программирование. К концу 1990-х они стали мейнстримом.

Потом наступили 2000-е. В следующие десятилетия мы стали свидетелями широкого применения термина «*N-уровневые приложения*», появились толстые клиенты, тонкие клиенты, дженерики, MVC², MVVM³ и MVP⁴. Асинхронное

¹ JIT (just-in-time compilation) — своевременная компиляция. Миф, созданный Sun Microsystems, разработавшей Java. Считается, что если скомпилировать код во время его выполнения, он станет быстрее, потому что оптимизатор соберет больше данных. Этот миф существует до сих пор.

² MVC (Model — View — Controller, «модель — представление — контроллер») — классический шаблон проектирования в ООП. — *Примеч. пер.*

³ MVVM (Model — View — ViewModel, «модель — представление — модель представления») — шаблон проектирования архитектуры приложения, производный от MVC. — *Примеч. пер.*

⁴ MVP (Model — View — Presenter, «модель — представление — представитель») — шаблон проектирования пользовательских интерфейсов, производный от MVC. — *Примеч. пер.*

программирование начало распространяться с помощью промисов, фьючерсов и, наконец, реактивного программирования. Не забудем и о микросервисах. Более функциональные концепции программирования, такие как LINQ¹, со-поставление с образцом и неизменяемость, стали основными понятиями. Нас накрыло цунами модных словечек.

Здесь я даже не упоминал шаблоны проектирования или лучшие практики. У нас есть бесчисленное множество лучших практик, советов и приемов чуть ли не по каждой теме. Существуют манифесты, посвященные тому, следует ли использовать символы табуляции или пробелы для отступов в исходном коде, хотя ответ очевиден — пробелы².

Мы считаем, что проблемы решаются с помощью парадигм, шаблонов, структур или библиотек. Учитывая сложность современных проблем, это не лишено оснований.

Однако слепое использование инструментов может создать еще больше проблем в будущем: они могут замедлять вашу работу, поскольку добавляют новые знания в предметной области, которые необходимо изучать, и специфические наборы ошибок. Они могут даже побудить изменить дизайн. Эта книга даст вам уверенность в том, что вы правильно используете шаблоны, пытливо их исследуете и собираете хорошие примеры их применения.

1.4.3. Черные ящики технологий

Фреймворк или библиотека — это программный пакет. Разработчики устанавливают его, читают документацию и используют, но обычно не знают, как именно он работает. Они одинаково подходят к алгоритмам и структурам данных, используют словарь, потому что это удобно для хранения ключей и значений, не задумываясь о последствиях.

Безоговорочное доверие к экосистемам и фреймворкам чревато серьезными ошибками. Оно может стоить дней отладки, если никто не знал, что добавление элементов в словарь с одним и тем же ключом с точки зрения скорости поиска не отличается от использования списка. Если мы используем генераторы C#,

¹ LINQ (Language Integrated Query) — проект Microsoft по добавлению синтаксиса языка запросов, напоминающего SQL, в языки программирования платформы .NET Framework. — *Примеч. пер.*

² Я писал о спорах вокруг табуляции и пробелов с прагматической точки зрения: <https://medium.com/@ssg/tabs-vs-spaces-towards-a-better-bike-shed-686e111a5cse>.

когда достаточно простого массива, то страдаем от значительного снижения производительности.

В 1993 году приятель подарил мне звуковую карту для моего компьютера. Раньше нам приходилось устанавливать дополнительные карты, чтобы добиться нормального звука, потому что иначе мы слышали только отдельные сигналы. До этого я еще ни разу не залезал во внутренности компьютера, так как боялся что-нибудь сломать. Я попросил: «Можешь сделать это для меня?», на что приятель ответил: «Тебе надо просто открыть его, чтобы посмотреть, как он работает».

Я понял, что волновался из-за отсутствия практики, а не из-за неспособности. Сняв крышку, я успокоился — там была пара плат. Звуковая карта стала в один из разъемов, и задача была решена. Позже я использовал ту же технику, обучая студентов художественной школы основам работы на компьютере. Я открыл мышь и показал им ее шарик. Да, у мышь тогда были шарики, хотя это звучит двусмысленно. Я снял крышку корпуса ПК: «Видите, это не страшно, тут платы и разъемы».

Позже это стало моим девизом в работе со всем новым и сложным. Я перестал бояться открывать крышку и обычно делал это сразу, чтобы увидеть все, что пугало.

Точно так же нюансы работы библиотеки, фреймворка или компьютера могут оказать огромное влияние на понимание систем, в основе которых они лежат. Открыв коробку и рассмотрев детали, вы сможете правильно ими пользоваться. На самом деле не обязательно читать весь код или изучать теорию от корки до корки, но нужно хотя бы понять, что куда вставляется и как это может повлиять на использование вашего продукта.

Вот почему мы рассмотрим несколько фундаментальных или низкоуровневых тем. Чтобы принимать лучшие решения на высоком уровне, нужно открыть коробку и посмотреть, как все работает.

1.4.4. Недооценка накладных расходов

Я рад, что с каждым днем мы видим все больше облачных приложений. Они не только экономически эффективны, но и действительно помогают понять фактическую стоимость кода. Когда вы начинаете платить за каждое свое неправильное решение, вас начинают беспокоить накладные расходы.

Фреймворки и библиотеки обычно помогают избежать накладных расходов, что делает их полезными абстракциями. Однако делегировать весь процесс принятия

решений фреймворкам не получится. Иногда приходится принимать решения и учитывать накладные расходы самостоятельно. В масштабных приложениях эти расходы еще более важны. Каждая сэкономленная миллисекунда может помочь сбережению драгоценных ресурсов.

Разработчик не должен ставить во главу угла сокращение накладных расходов. Однако если он будет знать, как их избежать в определенных случаях, то сэкономит и свое время, и время пользователя, который ждет, глядя на спиннер¹ на веб-странице.

В этой книге вы встретите сценарии и примеры того, как легко избежать накладных расходов, не ставя это главной целью.

1.4.5. Не моя работа

Один из способов справиться со сложностью — сосредоточиться исключительно на своей области задач: компоненте, которым мы владеем, коде, который мы пишем, ошибках, которые мы вызвали, а иногда и на взорвавшейся лазанье в офисной микроволновке. Такой метод работы кажется самым эффективным, но код, как и все в этом мире, имеет взаимосвязи.

Изучив, как работает конкретная технология, библиотека и зависимости, мы сможем принимать лучшие решения при написании кода. Примеры в этой книге позволят вам сосредоточиться не только на конкретной области задач, но и на ее зависимостях и проблемах. Они могут находиться вне вашей зоны комфорта, но при этом оказывают влияние на судьбу вашего кода.

1.4.6. Рутина — это гениально

Все советы в области разработки ПО сводятся к одному: тратьте меньше времени на работу. Избегайте повторяющихся, бессмысленных задач, таких как копирование и вставка, а также написания с нуля кода, который лишь незначительно отличается от имеющегося. Во-первых, это отнимает время, а во-вторых, такой код сложно обслуживать.

¹ Спиннеры — это современная замена песочным часам. Раньше, чтобы продемонстрировать пользователю необходимость подождать, использовалось анимированное изображение таких часов. Спиннер — современный эквивалент этой анимации. Обычно он представляет собой вращающуюся дугу, которая отвлекает пользователя, чтобы тот окончательно не пал духом.

Но не все подобные задачи плохи. Даже копирование и вставка. Их репутация сильно подмочена, но они могут быть более эффективными, чем некоторые из лучших практик, которым вас учили.

Кроме того, не весь код, который вы пишете, работает как код для реального продукта. Часть его будет использована для разработки прототипа, часть — для тестов, а часть — для разогрева перед работой над реальной задачей. Я разберу некоторые из этих сценариев и то, как использовать эти задачи в своих интересах.

1.5. ЧЕГО НЕТ В ЭТОЙ КНИГЕ

Эта книга не является исчерпывающим руководством по программированию, алгоритмам или любой другой конкретной теме. Я не считаю себя экспертом в отдельных узких темах, но имею достаточно опыта в разработке. Книга в основном рассказывает о том, что осталось неочевидным после прочтения известных, популярных и замечательных изданий. Это определенно не учебное пособие по программированию.

Опытным программистам книга вряд ли будет очень полезна, потому что они уже приобрели достаточно знаний и стали уличными кодерами. Тем не менее некоторые идеи, представленные здесь, все еще могут их удивить.

Эта книга также является своего рода экспериментом в области подачи материала. Я хотел представить программирование в первую очередь как веселое занятие. Некоторые вещи не следует воспринимать слишком серьезно, и эта книга не исключение. Если вы получите удовольствие от ее чтения и почувствуете себя увереннее как разработчик, я буду считать, что моя цель достигнута.

1.6. ОСНОВНЫЕ ТЕМЫ КНИГИ

Некоторые темы будут повторяться на протяжении всей книги:

- Базовые знания, достаточные, чтобы ориентироваться на улицах. Эти темы раскрыты не исчерпывающе, но так, чтобы заинтересовать вас, если раньше вы считали их скучными. Эти основы помогают принимать правильные решения.
- Известные и зарекомендовавшие себя лучшие практики и методы, которые я дополнил антишаблонами, более эффективными в некоторых случаях. Чем

больше вы о них знаете, тем лучше будет ваша интуиция при критическом анализе методов программирования.

- Некоторые, казалось бы, неуместные приемы, такие как оптимизация на уровне ЦП, которые могут повлиять на принятие решений и написание более качественного кода. Знание внутренних механизмов имеет огромную ценность, даже если эта информация прямо не используется.
- Проверенные автором методы, которые помогут повысить вашу производительность и успешно выполнять повседневные рабочие задачи. В их числе: грызть ногти и вовремя стать невидимым для босса.

Эти темы позволят вам взглянуть на вопросы программирования по-новому, изменят отношение к «скучным» предметам и, возможно, к некоторым общепринятым убеждениям. Вы начнете получать больше удовольствия от работы.

ИТОГИ

- Суровая реальность улиц, мир профессиональной разработки, требует навыков, которым не учат или которые не считаются приоритетными в системах формального образования. А при самообучении их иногда вовсе упускают из виду.
- Разработчики-новички, как правило, либо чересчур углубляются в теорию, либо полностью ее игнорируют. В конце концов вы сами найдете золотую середину, но ее определение можно несколько ускорить.
- Разработка стала намного сложнее, чем была всего пару десятилетий назад. Чтобы разработать простое работающее приложение, требуются прочные знания на многих уровнях.
- Программистам приходится выбирать между практикой и изучением нового. С этим можно справляться, переосмысливая задачи в более практическом ключе.
- Отсутствие четкого представления о том, над чем вы работаете, делает программирование рутинным и скучным, снижая продуктивность. Понимание того, что вы делаете, повысит удовольствие от работы.

Практическая теория

В этой главе

- ✓ Почему знания computer science важны для выживания
- ✓ Как заставить типы работать на вас
- ✓ Понимание особенностей алгоритмов
- ✓ Структуры данных и их странные свойства, о которых вам не рассказали родители

Вопреки широко распространенному мнению, программисты тоже люди. И они, так же как остальные люди, заблуждаются, думая о практике разработки. Они сильно переоценивают преимущества необязательного использования типов, не заботясь о правильных структурах данных или считая, что алгоритмы важны только для авторов библиотек.

Вы не исключение. От вас ожидают, что вы сделаете качественный продукт вовремя и с улыбкой. Как гласит поговорка, программист — это организм, который получает кофе на входе и создает программы на выходе. С таким же успехом вы можете использовать копирование и вставку, код, который вы нашли на Stack Overflow, простые текстовые файлы для хранения данных или заключить сделку с дьяволом,

если еще не продали душу, подписав NDA¹. Только ваших коллег действительно волнует, как вы работаете, всем остальным нужен надежный готовый продукт.

Теория может быть избыточной и неуместной. Алгоритмы, структуры данных, теория типов, нотация «O-большое» и полиномиальная сложность могут показаться чересчур мудреными и неприменимыми для разработки. Существующие библиотеки и фреймворки и так уже справляются со всем этим, к тому же они оптимизированы и надежно протестированы. Вас всячески мотивируют не реализовывать алгоритмы с нуля, особенно ввиду вопросов информационной безопасности или сжатых сроков.

Тогда зачем нужна теория? Потому что знание теории computer science позволяет не только разрабатывать алгоритмы и структуры данных с нуля, но и правильно определять, когда их стоит использовать. Это помогает понять стоимость компромиссов и свойства масштабируемости кода, который вы пишете. Это заставляет смотреть вперед. Возможно, вы никогда не создадите структуру данных или алгоритм целиком, но знание того, как они работают, сделает вас эффективным разработчиком. Оно повысит ваши шансы выжить на улицах.

В этой книге будут рассмотрены только критически важные разделы теории, которые вы могли пропустить при ее изучении, — не самые известные свойства типов данных, оценка сложности алгоритмов и внутренних механизмов работы определенных структур данных. Если вы раньше не знали о типах, алгоритмах или структурах данных, эта глава подскажет, с чего начать подробное знакомство с ними.

2.1. КРАТКИЙ ОБЗОР АЛГОРИТМОВ

Алгоритм — это набор правил и шагов для решения задачи. Спасибо, что слушали меня на конференции TED². Вы ожидали более сложного определения, не так ли? Примером простого алгоритма может служить перебор элементов массива, чтобы выяснить, содержит ли он заданное число:

```
public static bool Contains(int[] array, int lookFor) {
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
        }
    }
}
```

¹ Non-disclosure agreement — Соглашение о неразглашении, запрещающее сотрудникам говорить о своей работе, если только они не начинают разговор со слов «Я вам этого не говорил, но...».

² TED Talks — научно-популярные конференции в разных сферах знания. — *Примеч. пер.*

```

        }
    }
    return false;
}
}

```

Его можно было бы назвать *алгоритмом Седата*, если бы он был изобретен мной, но скорее всего, это один из первых алгоритмов, придуманных человеком. Он не совершенен, но работает и имеет смысл. Это одно из важных и необходимых свойств алгоритма: он должен работать для ваших нужд и не обязательно творить чудеса. Когда вы ставите посуду в посудомоечную машину и нажимаете кнопку запуска, то следуете алгоритму. Алгоритм не обязательно должен быть сложным.

Тем не менее существуют и более интеллектуальные алгоритмы. В предыдущем примере кода, если известно, что список содержит только положительные целые числа, можно использовать специальную обработку неположительных чисел:

```

public static bool Contains(int[] array, int lookFor) {
    if (lookFor < 1) {
        return false;
    }
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
        }
    }
    return false;
}

```

Так алгоритм будет работать намного быстрее при вызовах с отрицательными числами. В лучшем случае функция всегда будет вызываться с отрицательными числами или нулем и немедленно возвращаться, даже если массив содержит миллиарды целых чисел. В худшем — функция всегда будет вызываться с положительными числами и дополнительная проверка будет только снижать производительность. Здесь на помощь придет беззнаковый тип целых чисел в C# — `uint`. При его использовании компилятор выполнит проверку только для положительных чисел, что исключит лишние проверки:

```

public static bool Contains(uint[] array, uint lookFor) {
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
        }
    }
    return false;
}

```

Мы обеспечили положительность проверяемого числа, не меняя алгоритм, за счет ограничения типа данных. Но можно ускорить работу алгоритма и путем изменения формы данных. Что мы знаем о данных? Массив отсортирован? Если да, можно ускорить поиск числа. Если сравнить число с любым элементом в отсортированном массиве, легко исключить огромное количество элементов (рис. 2.1).



Рис. 2.1. Одна операция сравнения позволяет исключить левую или правую часть отсортированного массива

Если искомое число — 3 и мы сравниваем его с элементом отсортированного по возрастанию массива, который оказался равным 5, можно быть уверенными, что наше число не может располагаться правее этого элемента. Это значит, что можно спокойно игнорировать все элементы справа от 5.

Таким образом, если мы выбираем элемент из середины массива, то после сравнения гарантированно можно исключить как минимум половину массива. Этую же логику можно применить к оставшейся половине, выбрать в ней среднее значение и продолжить. Таким образом, для отсортированного массива из 8 элементов потребуется максимум три сравнения, чтобы определить, имеется ли в нем искомое число. Что еще более важно, потребуется не более 10 операций для поиска числа в массиве из 1000 элементов.

Деление массива пополам — мощный инструмент. Его реализация представлена в листинге 2.1. Мы постоянно находим средний элемент и после сравнения исключаем половину элементов в зависимости от того, в какую из них попадает искомое значение. Для расчета среднего элемента используется формула, которую можно было бы упростить до $(\text{start} + \text{end}) / 2$. Но мы этого упрощения не делаем, потому что $(\text{start} + \text{end})$ может привести к переполнению при больших значениях `start` и `end`, тогда среднее будет вычислено неверно. Формула в листинге ниже позволяет избежать такого переполнения.

Листинг 2.1. Двоичный поиск в отсортированном массиве

```
public static bool Contains(uint[] array, uint lookFor) {
    int start = 0;
    int end = array.Length - 1;
    while (start <= end) {
```

```

int middle = start + ((end - start) / 2); ← Определение среднего элемента
uint value = array[middle];
if (lookFor == value) {
    return true;
}
if (lookFor > value) { ← Исключение левой части массива
    start = middle + 1;
} else { ← Исключение правой части массива
    end = middle - 1;
}
return false;
}

```

Здесь мы реализовали двоичный поиск — значительно более быстрый алгоритм, чем *алгоритм Седата*. Поскольку теперь понятно, почему двоичный поиск может быть быстрее, чем простой перебор, можно переходить к известной нотации «O-большое».

2.1.1. «O-большое» должно быть приемлемым

Понимание закономерностей — отличный навык для разработчика. Когда вы знаете, как быстро что-то увеличивается в размере или количестве, то можете предвидеть будущее и, следовательно, прогнозировать возможные проблемы, прежде чем потратите на них время. Это особенно полезно, когда свет в конце туннеля становится все ярче, даже если вы неподвижны.

Нотация «O-большое» — условное обозначение оценки сложности, но не все это понимают. Когда я впервые увидел O(N), то подумал, что это обычная функция, которая должна возвращать число. Это не так. С ее помощью математики выражают рост сложности расчетов. Она дает базовое представление о масштабируемости алгоритма. Последовательный обход каждого элемента (*алгоритм Седата*) подразумевает совершение количества операций, которое прямо пропорционально числу элементов в массиве. Запишем это утверждение в виде O(N), где N — количество элементов. O(N) не дает понять, сколько именно шагов потребуется алгоритму для выполнения, но показывает, что это число растет линейно. Благодаря этому можно оценить производительность алгоритма в зависимости от размера данных и предположить, в какой момент он станет неэффективным.

Реализованный нами двоичный поиск имеет сложность O(log²ⁿ). Логарифм — это функция, противоположная экспоненте, поэтому логарифмическую сложность можно считать замечательной, если только вопрос не касается денег. Если

бы алгоритм сортировки из нашего примера волшебным образом приобрел логарифмическую сложность, для сортировки массива из 500 000 элементов потребовалось бы всего 18 сравнений. Это великолепная эффективность.

«O-большое» используется для измерения роста не только количества вычислений, то есть *временной сложности*, но и объема используемой памяти, то есть *пространственной сложности*. Алгоритм может быть быстрым, но требовать полиномиального увеличения объема используемой памяти как в нашем примере с сортировкой. Необходимо понимать это.

ДЛЯ СПРАВКИ Вопреки распространенному мнению, $O(N^x)$ не означает экспоненциальную сложность. Она обозначает полиномиальную сложность, которая хоть и плоха, но не настолько, как экспоненциальная $O(x^n)$. В массиве из 100 элементов $O(N^2)$ будет означать 10 000 итераций, а $O(2^n)$ – абсолютно умопомрачительное число из 30 цифр, которое даже выговорить сложно. Существует также факториальная сложность, которая еще хуже экспоненциальной. Мне почти не приходилось с ней сталкиваться, кроме алгоритмов вычисления перестановок и алгоритмов, где она сочетается с другими видами сложности – наверное, потому, что никто не хочет иметь с ней дела.

Поскольку нотация «O-большое» описывает рост, самое важное для нее – это сравнение функций роста. На практике $O(N)$ эквивалентна $O(4N)$, но не $O(N.M)$ (точка – оператор умножения), если и N и M возрастают. Последняя может быть эквивалентной $O(N^2)$. $O(N.\log N)$ несколько хуже, чем $O(N)$, но не так плоха, как $O(N^2)$.

Функция $O(1)$ удивительна. Она означает, что производительность алгоритма не зависит от количества элементов в имеющейся структуре данных. Поэтому его называют алгоритмом *постоянного времени*.

Представьте, что вы написали функцию поиска в базе данных, которая перебирает все записи. Это означает, что время выполнения алгоритма будет расти прямо пропорционально количеству элементов в базе данных. Предположим, что мы до сих пор пользуемся счетами и обработка каждой записи у нас занимает секунду. Это означает, что поиск в базе данных из 60 элементов займет до минуты. Это сложность $O(N)$. Другие разработчики в команде могут предложить другие алгоритмы, как показано в табл. 2.1.

Чтобы принимать обоснованные решения при выборе алгоритма обработки данных, вы должны иметь представление о том, как нотация «O-большое» описывает рост сложности алгоритма и использование памяти. Прикиньте значение

«О-большого», даже если вам не нужно реализовывать алгоритм. Не пренебрегайте оценкой сложности.

Таблица 2.1. Влияние сложности на производительность

Алгоритм поиска	Оценка сложности	Время нахождения записи среди 60 строк
Самодельный квантовый компьютер из гаража	$O(1)$	1 секунда
Двоичный поиск	$O(\log N)$	6 секунд
Линейный поиск (потому что ваш босс попросил вас написать код за час до презентации)	$O(H)$	60 секунд
Стажер по ошибке использовал вложенные циклы	$O(N^2)$	1 час
Код, найденный на Stack Overflow, параллельно ищет решение открытой математической проблемы	$O(2^N)$	36,5 миллиарда лет
Алгоритм пытается найти порядок данных, расшифровывающий запись, которую вы ищете. Хорошая новость в том, что разработчик этого здесь больше не работает	$O(N!)$	До конца вселенной, но все же раньше, чем обезьяна закончит печатать пьесу Шекспира ¹

2.2. СТРУКТУРЫ ДАННЫХ ИЗНУТРИ

Вначале была пустота. Когда первые электрические импульсы достигли первого бита памяти, появились данные. Данные свободно плавали в виде байтов. Эти байты соединились и создали структуру.

Начало 0:1

Структуры данных — это о том, как данные располагаются. Люди обнаружили, что данные более полезны, если они расположены определенным образом. Список покупок на листе бумаги легче читать, если каждый пункт начинается с новой строки. Таблица умножения более полезна, если она представлена в виде сетки. Чтобы стать хорошим программистом, важно понимать,

¹ Отсылка к теореме о бесконечных обезьянах, согласно которой абстрактная обезьяна, случайным образом ударяя по клавиатуре, рано или поздно напечатает пьесу Шекспира. — Примеч. пер.

как работает та или иная структура данных. Для этого надо изучить, как все устроено «под капотом».

Возьмем, к примеру, массивы. Массив в программировании — одна из простейших структур данных. В памяти он располагается в виде непрерывного набора элементов. Допустим, у вас есть такой массив:

```
var values = new int[] {1, 2, 3, 4, 5, 6, 7, 8};
```

Вы представляете, что в памяти он располагается так, как показано на рис. 2.2.



Рис. 2.2. Размещение массива в памяти — неверное представление

На самом деле это не так, потому что у каждого объекта в .NET имеется заголовок, указатель на таблицу виртуальных методов и информация о длине, как показано на рис. 2.3.



Рис. 2.3. Фактическое размещение массива в памяти

Структура данных становится еще понятнее, если представлять, как массив размещен в оперативной памяти, потому что последняя не состоит из целых чисел (рис. 2.4). Я рассказываю об этом, потому что хочу, чтобы вы не боялись низкоуровневых концепций. Их понимание помогает на всех уровнях программирования.

В реальности современные операционные системы выделяют для каждого процесса свой фрагмент оперативной памяти. И это придется принять как данность, если только вы не станете разрабатывать свою собственную операционную систему или собственные драйверы устройств.

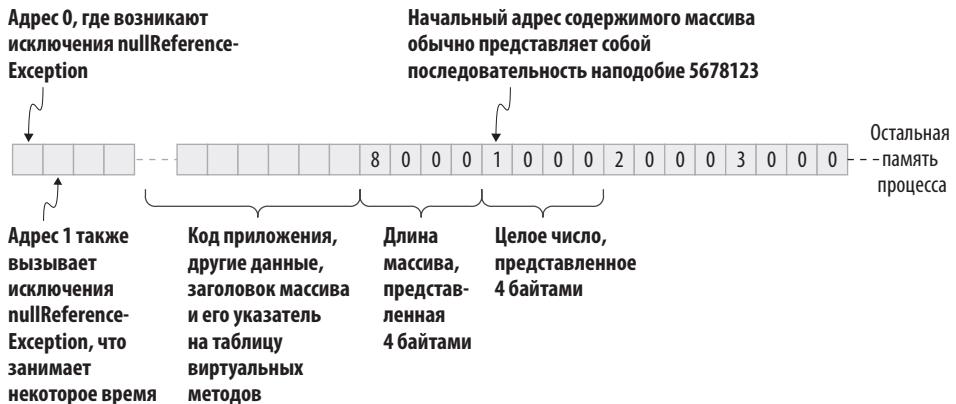


Рис. 2.4. Процессы и массивы в памяти

В целом способ организаций данных может как ускорить операции и сделать работу эффективнее, так и наоборот. Поэтому необходимо знать некоторые основные структуры данных и принципы их работы.

2.2.1. Строки

Строки — самый понятный тип данных в программировании, поскольку они представляют собой текст. Не стоит использовать строки, если для ваших целей лучше подходит другой тип данных, но обойтись без них невозможно. Вдобавок они удобны. Однако не все даже основные свойства строк очевидны.

Строки в .NET неизменяемы, хотя и напоминают массивы по типу использования и структуре. Неизменяемость означает, что содержимое структуры данных не может быть изменено после инициализации. Предположим, что нам необходимо объединить имена и фамилии и получить единую строку, разделенную запятыми, и что мы отброшены на два десятка лет назад, поэтому нет лучшего способа сделать это, кроме следующего:

```
public static string JoinNames(string[] names) {
    string result = String.Empty; ← Неинициализированная строка имела бы по умолчанию нулевое значение, которое можно было бы обнаружить проверкой на допустимость такого значения
    int lastIndex = names.Length - 1; ← Индекс последнего элемента
    for (int i = 0; i < lastIndex; i++) {
        result += names[i] + ", ";
    }
}
```

```

result += names[lastIndex]; ← Благодаря этому строка не будет оканчиваться запятой
return result;
}

```

На первый взгляд может показаться, что мы модифицируем в цикле одну и ту же строку с именем `result`, но это не так. Каждый раз, когда мы присваиваем `result` новое значение, мы создаем новую строку в памяти. .NET необходимо определить длину новой строки, выделить для нее память, скопировать в эту память содержимое других строк и вернуть `result`. Это довольно затратная операция, и ее стоимость увеличивается по мере увеличения длины строки и сопутствующего мусора.

Во фреймворке есть бесплатные инструменты, позволяющие избежать этой проблемы. Благодаря им не нужно менять логику или из кожи вон лезть, чтобы повысить производительность. Один из таких инструментов — `StringBuilder`, с помощью которого за один раз можно создать финальную строку и извлечь ее с помощью вызова `ToString`:

```

public static string JoinNames(string[] names) {
    var builder = new StringBuilder();
    int lastIndex = names.Length - 1;
    for (int i = 0; i < lastIndex; i++) {
        builder.Append(names[i]);
        builder.Append(", ");
    }
    builder.Append(names[lastIndex]);
    return builder.ToString();
}

```

`StringBuilder` использует последовательные блоки памяти, вместо того чтобы перераспределять и копировать их каждый раз, когда нужно увеличить строку. Обычно это более эффективно, чем создавать строку с нуля.

Очевидно, что естественное и значительно более короткое решение доступно уже давно, хотя не всегда применимо на практике:

```
String.Join(", ", names);
```

Конкатенация обычно допустима при инициализации строки, потому что это требует только одного выделения буфера после вычисления требуемой общей длины. Например, если у вас есть функция, которая соединяет имя и фамилию с пробелом между ними с помощью оператора сложения, вы создаете одну новую строку за один шаг:

```
public string ConcatName(string firstName, string middleName,
    string lastName) {
    return firstName + " " + middleName + " " + lastName;
}
```

Это может показаться полной ерундой, если посчитать, что `firstName + " "` сначала создаст новую строку, затем — новую строку с `middleName` и так далее, но на самом деле компилятор превращает все это в один вызов функции `String.Concat()`, которая выделяет новый буфер длиной, равной сумме длин всех строк и возвращает его за один раз. Поэтому все происходит быстро. Но объединение строк за несколько шагов с промежуточными условиями `if` или циклами компилятор уже не может оптимизировать. Нужно знать, когда можно объединять строки, а когда нет.

Тем не менее неизменяемость — это не нерушимый Святой Грааль. Существуют способы изменения строк и других неизменяемых структур. В основном они подразумевают использование небезопасного кода и вызов духов и обычно не рекомендуются, потому что строки дедулицируются средой выполнения .NET и некоторые их свойства, такие как хеш-коды, кэшируются. Внутренняя реализация в значительной степени зависит от особенностей неизменяемой структуры.

Строковые функции по умолчанию работают с текущей системной культурой. Поэтому может оказаться неприятным сюрпризом то, что приложение перестанет работать в другой стране.

ПРИМЕЧАНИЕ *Культура*, в некоторых языках программирования также называемая *локалью*, представляет собой набор правил для выполнения операций, зависящих от региона, таких как сортировка строк, формат даты и времени, раскладка столовых приборов и т. д. Текущая культура — это то, чем пользуется операционная система.

Понимание культурного контекста может повысить скорость и безопасность операций со строками. Например, рассмотрим код, определяющий, содержит ли имя файла расширение `.gif`:

```
public bool isGif(string fileName) {
    return fileName.ToLower().EndsWith(".gif");
}
```

Как видите, задав строке нижний регистр, мы предусмотрели случаи, когда расширение может быть набрано как `.GIF`, `.Gif` или другой комбинацией регистров. Но не во всех языках буква в нижнем регистре соответствует той же букве

в верхнем. Например, в турецком языке строчная буква для «İ» — это не «i», а «ı», также известная как I без точки. Поэтому код в приведенном примере не сработает в Турции и, возможно, в Азербайджане и некоторых других странах. Переводя строку в нижний регистр, мы фактически создаем новую строку, что, как мы узнали, неэффективно.

.NET предоставляет не привязанные к культуре версии некоторых строковых методов, таких как `ToLowerInvariant`, а также возможности перегрузки метода при получении значения `StringComparison`, имеющего инвариантную и порядковую разновидности. Поэтому можно сделать метод более безопасным и быстрым:

```
public bool isGif(string fileName) {
    return fileName.EndsWith(".gif",
        StringComparison.OrdinalIgnoreCase);
}
```

Этот метод позволяет избежать создания новой строки, а сравнение строк происходит более безопасно и быстро без учета особенностей текущей культуры. Можно использовать и `StringComparison.InvariantCultureIgnoreCase`, но, в отличие от порядкового сравнения, в этом случае добавляются правила транслитерации, такие как замена немецких умлятуотов и специфических графем их латинскими аналогами (ß на ss), что может вызвать проблемы с именами файлов и другими идентификаторами ресурсов. При порядковом сравнении значения символов сравниваются напрямую, без транслитерации.

2.2.2. Массив

Мы уже видели, как массив размещается в памяти. Массивы удобны для хранения набора элементов, числовые значения которых не превышают размеров массива. Массивы — статические структуры. Поэтому если потребуется больший массив, придется создать новый и скопировать в него содержимое старого. Рассмотрим несколько особенностей массивов, о которых следует знать.

Массивы, в отличие от строк, изменяемы. Это их суть. Их содержимым можно свободно манипулировать. На самом деле очень сложно сделать массивы неизменяемыми, поэтому они не годятся для интерфейсов. Рассмотрим такое свойство:

```
public string[] Usernames { get; }
```

Хотя здесь нет сеттера, это массив, и он изменяем. Ничто не мешает сделать следующее:

```
Usernames[0] = "root";
```

Однако теперь все усложнилось, даже если этот класс используете только вы. Никогда не вносите изменения в состояние, если только в этом нет крайней необходимости. Корень всех зол — состояние, а не нулевое значение. Чем меньше состояний у приложения, тем меньше проблем.

Старайтесь придерживаться типов с минимальной необходимой функциональностью. Если вам нужно только просматривать элементы по порядку, используйте `IEnumerable<T>`. Если вдобавок нужно периодически вести подсчет, используйте `ICollection<T>`. Обратите внимание, что метод расширения `LINQ .Count()` содержит специальный код обработки для типов, поддерживающих `IReadOnlyCollection<T>`, поэтому даже если вы используете его для `IEnumerable`, есть вероятность, что он вернет кэшированное значение.

Массивы лучше всего подходят для использования внутри локальной области видимости функции. Для любых других целей в дополнение к `IEnumerable<T>` существует более подходящий тип или интерфейс, например `IReadOnlyCollection<T>`, `IReadOnlyList<T>` или `ISet<T>`.

2.2.3. Список

Список ведет себя как массив, который может постепенно увеличиваться, подобно тому как работает `StringBuilder`. Списки можно использовать вместо массивов практически везде, но это приведет к ненужному снижению производительности, поскольку индексированный доступ в списке представляет собой виртуальный вызов, в то время как массив использует прямой доступ.

Дело в том, что в объектно-ориентированном программировании есть такая замечательная вещь, как полиморфизм. Это означает, что объект может вести себя в соответствии с базовой реализацией, не меняя интерфейс. Если у вас есть, скажем, переменная `a` с типом интерфейса `IOpenable`, то `a.Open()` может открыть файл или установить сетевое подключение в зависимости от типа назначенного ей объекта. Это реализуется путем предоставления ссылки на таблицу, в которой вызываемые виртуальные функции сопоставлены с типом, указанным в начале объекта, — таблицу виртуальных методов `vtable`. Таким образом, хотя `Open` сопоставляется с одной и той же записью в таблице для каждого объекта одного и того же типа, вы не будете знать, куда это приведет, пока не увидите фактическое значение в таблице.

Поскольку мы не знаем, что именно вызываем, такие вызовы называются виртуальными. Виртуальный вызов подразумевает дополнительный поиск в таблице виртуальных методов, поэтому он немного медленнее, чем обычные вызовы

функций. Это может не представлять проблемы, если вызовов несколько, но когда они осуществляются внутри сложного алгоритма, затраты могут расти полиномиально. Таким образом, если список не будет увеличиваться после инициализации, вместо него в локальной области видимости можно использовать массив.

Обычно о таких деталях думать не приходится. Но зная разницу, можно определить, когда массив предпочтительнее списка.

Списки похожи на `StringBuilder`. В обоих случаях структуры являются динамически растущими, но механизм роста в списках менее эффективен. Всякий раз, когда список должен увеличиться, он выделяет новый массив большего размера и копирует в него существующее содержимое. `StringBuilder`, напротив, сохраняет цепочку фрагментов памяти, что не требует копирования. Область буфера для списков увеличивается всякий раз при достижении предела, но размер нового буфера удваивается, поэтому потребность в увеличении со временем снижается. Тем не менее это пример того, как специальный класс может быть более эффективен, чем общий.

Улучшить производительность списка можно, указав емкость. Если ее не указать, список начнется с пустого массива. Затем он увеличит свою вместимость до нескольких элементов. После заполнения он удвоит вместимость. Задание емкости при создании списка позволит избежать ненужных операций увеличения и копирования. Используйте эту возможность, если заранее знаете максимальное количество элементов в списке.

Однако не стоит задавать емкость списка без необходимости. Это может вызвать ненужные затраты памяти, которые могут накапливаться. Возьмите за правило принимать продуманные решения.

2.2.4. Связанный список

Связанные списки — это списки, в которых элементы расположены в памяти не последовательно, но каждый элемент указывает на адрес следующего. Они полезны, поскольку производительность при их вставке и удалении равна $O(1)$. Вы не сможете получить доступ к отдельным элементам по индексу, потому что они хранятся в разных местах памяти и вычислить их местоположение не получится. Но если вы обращаетесь к началу либо концу списка или если нужно только пересчитать элементы, производительность будет максимальной. Приверка же того, существует ли элемент в связанным списке, является операцией $O(N)$, как в массивах и обычных списках. На рис. 2.5 показан пример представления связанныного списка.



Рис. 2.5. Представление связанного списка

Однако сказанное выше не означает, что связанные списки во всех случаях быстрее обычных. Индивидуальное выделение памяти для каждого элемента вместо выделения общего блока памяти и дополнительные поиски ссылок также могут снизить производительность.

Связанный список может быть полезен, когда требуется структура очереди или стека, но .NET учитывает это. Так что если только вы не занимаетесь системным программированием, нет необходимости использовать связанные списки в повседневной работе. Однако о связанных списках часто задают головоломные вопросы на собеседованиях, поэтому все-таки важно иметь о них представление.

НЕЛЬЗЯ ИГНОРИРОВАТЬ СВЯЗАННЫЕ СПИСКИ

Ответы на вопросы по программированию на собеседованиях — это часть обряда посвящения в разработчики. Большинство вопросов, касающихся кода, затрагивают структуры данных и алгоритмы. Связанные списки тоже попадают в сферу внимания, и вас могут попросить обратить связанный список или инвертировать двоичное дерево.

В реальной работе вы, скорее всего, никогда не столкнетесь с такими задачами. Но на собеседовании проверяют вашу способность выбирать подходящие структуры данных и алгоритмов. Также проверяют ваши навыки аналитического мышления и решения задач, поэтому важно «думать вслух» и делиться мыслями с интервьюером.

Не всегда требуется обязательно решить поставленную задачу. Обычно выбирают того, кто увлечен предметом, твердо знаком с основными понятиями и может найти дорогу, даже если заблудится.

Я, к примеру, обычно задавал кандидатам в Microsoft дополнительные вопросы по поиску ошибок в их коде. Это помогало, потому что они чувствовали, что ошибки ожидаются и человека оценивают не по тому, насколько близок к идеалу его код, а по тому, как он находит ошибки.

Интервью — это не только поиск подходящего сотрудника, но и поиск коллеги, с которым комфортно работать. Кандидат должен быть любознательным, страстным, настойчивым и легким в общении человеком, который действительно поможет решать поставленные задачи.

Связанные списки были популярны на заре программирования, когда эффективность использования памяти стояла на первом месте. Тогда невозможно было выделять килобайты памяти только потому, что список увеличивался. Размеры хранилищ приходилось ограничивать. Связанные списки идеально подходили для этого. И они по-прежнему часто используются в ядрах операционных систем из-за своей чудесной производительности $O(1)$ при операциях вставки и удаления.

2.2.5. Очередь

Очередь — это базовая форма жизни, то есть структура данных. Она позволяет читать элементы из списка в порядке их добавления. Очередь может быть просто массивом, пока для чтения следующего элемента и вставки нового используются отдельные ячейки. Если расположить числа в очереди по возрастанию, она будет похожа на рис. 2.6.



Рис. 2.6. Общее представление очереди

Буфер клавиатуры на ПК в эпоху MS-DOS использовал массив байтов для хранения «нажатий клавиш». Буфер предотвращал потери введенной информации из-за медленной работы или зависания программы. Когда буфер заполнялся, BIOS издавал звуковой бип, оповещающий пользователя, что нажатия больше не записываются. К счастью, в .NET существует класс `Queue<T>`, который можно использовать, не беспокоясь о деталях реализации и производительности.

2.2.6. Словарь

Словари, также известные как *хеш-карты* или иногда *пары «ключ — значение»*, — одна из самых полезных и наиболее часто используемых структур данных. Мы принимаем их возможности как должное. Словарь — это контейнер, в котором

хранятся пары ключей и значений. Поиск значения по ключу занимает постоянное время $O(1)$, то есть происходит чрезвычайно быстро. Но как? В чем магия?

Магия — в понятии «хеш». Хеширование — это генерация одного числа из набора данных. Сгенерированное число должно быть детерминированным, то есть для одних и тех же данных должно генерироваться одно и то же неуникальное значение. Существует множество способов вычислить значение хеш-функции. Логика хеширования объекта реализована в методе `GetHashCode`.

Хеши хороши, потому что их значения можно использовать для поиска. Если, например, у вас есть массив всех возможных хеш-значений, вы можете находить их по индексу. Но такой массив занял бы около 16 Гбайт для каждого словаря, потому что каждое число типа `int` занимает 4 байта и может иметь около 4 миллиардов возможных значений.

Словари используют значительно меньший массив, основываясь на равномерном распределении хеш-значений. Вместо того чтобы искать значение хеш-функции, они ищут длину массива остатков от деления хеш-значения по модулю. Предположим, что словарь с целочисленными ключами выделяет массив из шести элементов для сохранения индекса, а метод `GetHashCode()` для целого числа просто вернет его значение `value`. Это означает, что формула для определения места отображения элемента — `value % 6`, поскольку индексы массива начинаются с нуля. Массив чисел от 1 до 6 будет распределен как показано на рис. 2.7.



Рис. 2.7. Распределение элементов в словаре

Что произойдет, если элементов будет больше, чем вместимость словаря? Безусловно, будут возникать наложения, поэтому словари сохраняют перекрывающиеся элементы в динамически растущем списке. Массив элементов с ключами от 1 до 7 будет выглядеть, как показано на рис. 2.8.

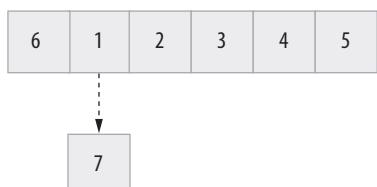


Рис. 2.8. Хранение перекрывающихся элементов в словаре

Почему я об этом говорю? Потому что производительность поиска по ключу в словаре обычно составляет $O(1)$, но накладные расходы на поиск в связанных списках составляют $O(N)$. Таким образом, по мере увеличения количества наложений скорость поиска будет снижаться. Допустим, имеется функция `GetHashCode`, которая всегда возвращает 4, например¹:

```
public override int GetHashCode() {
    return 4; // выбирается честным броском костей
}
```

В этом случае внутренняя структура словаря при добавлении в него элементов будет напоминать рис. 2.9.

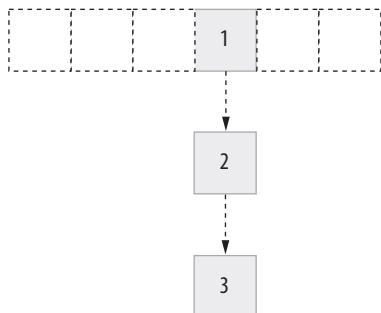


Рис. 2.9. Словарь с плохой функцией `GetHashCode()`

Словарь не лучше связанных списков, если хеш-значения неподходящие. Он может иметь даже худшую производительность из-за дополнительной необходимости управлять этими значениями. Это подводит нас к ключевой мысли: функция `GetHashCode` должна быть максимально уникальной. При большом количестве наложений пострадают словари, которые заставят страдать приложение, а последнее заставит страдать всю компанию. В конце концов дело дойдет и до вас. Из-за отсутствия гвоздя погибло государство².

Иногда требуется объединить значения нескольких свойств в классе, чтобы вычислить уникальное хеш-значение. Например, имена репозиториев уникальны для каждого пользователя на GitHub. То есть разные пользователи могут иметь репозитории с одинаковыми именами, поэтому только имени репозитория

¹ Навеяно комиксом xkcd о случайных числах: <https://xkcd.com/221>.

² Отсылка к английской народной песне *For want of a nail the shoe was lost...* (в переводе С. Я. Маршака — «Гвоздь и подкова»). — *Примеч. пер.*

недостаточно, чтобы сделать его уникальным. Предположим, вы используете только имя: это вызовет дополнительные конфликты. Следовательно, нужно комбинировать хеш-значения. Точно такая же проблема была бы на веб-сайте, если бы он предоставлял уникальные значения для каждой темы.

Чтобы эффективно комбинировать хеш-значения, необходимо знать их диапазоны и работать с их побитовым представлением. Если просто использовать оператор сложения или простые операции OR/XOR, можно столкнуться с гораздо большим количеством конфликтов, чем ожидалось. Придется также использовать битовые сдвиги. Правильная функция `GetHashCode` будет использовать побитовые операции, чтобы добиться хорошего разброса по всем 32 битам целого числа.

Код для такой операции может выглядеть как сцена взлома из низкопробного фильма про хакеров. Его сложно понять даже тем, кто разбирается в предмете. По сути, мы сдвигаем одно из 32-битных целых чисел на 16 бит, так что его младшие байты перемещаются к середине, и выполняем операцию XOR (^) для этого значения и другого 32-битного целого числа, что значительно снижает вероятность конфликтов. Выглядит это ужасно:

```
public override int GetHashCode() {
    return (int)((TopicId & 0xFFFF) << 16)
        ^ (TopicId & 0xFFFF0000 >> 16)
        ^ PostId;
}
```

К счастью, с появлением .NET Core и .NET 5 объединение хеш-значений было абстрагировано от класса `HashCode` таким образом, чтобы минимизировать конфликты. Вот все, что нужно сделать, чтобы объединить два значения:

```
public override int GetHashCode() {
    return HashCode.Combine(TopicId, PostId);
}
```

Хеш-коды используются не только в словарных ключах, но и в других структурах данных, таких как множества. Поскольку хорошо написать `GetHashCode` значительно проще, используя вспомогательные функции, вам не будет оправдания, если вы не воспользуетесь этой возможностью. Всегда имейте ее в виду.

Когда не стоит использовать словарь? Он не даст преимуществ и даже может снизить производительность, если все, что вам нужно, — это последовательно пройтись по парам «ключ — значение». Тогда лучше использовать `List<KeyValuePair<K, V>>`, чтобы избежать ненужных накладных расходов.

2.2.7. Хеш-множества

Множество похоже на массив или список, с единственным отличием: оно может содержать только уникальные значения. Его преимущество перед массивами или списками в том, что производительность поиска в множестве — $O(1)$, как у ключей словаря. Это достигается благодаря хешированному отображению, которое мы только что рассмотрели. Таким образом, если требуется неоднократная проверка, содержится ли элемент в данном массиве или списке, быстрее будет использовать множество. В .NET он называется `HashSet` и предоставляется бесплатно.

Поскольку `HashSet` обеспечивает быстрый поиск и вставку, он подходит для операций пересечения и объединения. Более того, он поставляется с методами, обеспечивающими эту функциональность. Чтобы воспользоваться всеми преимуществами множества, тщательно продумайте реализацию `GetHashCode()`.

2.2.8. Стек

Стеки представляют собой очереди LIFO (Last In First Out — последним пришел, первым вышел). Они полезны, когда требуется сохранить состояние и восстановить его в порядке, обратном тому, в котором оно сохранялось. Когда вы отправляйтесь в дорожную полицию, то иногда попадаете внутрь стека. Сначала вы подходите к окну 5, где сотрудник проверяет ваши документы и видит, что вы не оплатили пошлину, поэтому отправляет вас к окну 13. Сотрудник в окне 13 видит, что в ваших документах нет фотографии, и отправляет вас к окну 47, чтобы вы сфотографировались. Затем вам нужно вернуться к окну 13, взять там квитанцию об оплате и снова отправиться к окну 5, чтобы получить документ. Список окон и их посещение в определенном порядке (LIFO) — это операция, подобная стеку, причем от стеков обычно больше пользы, чем от дорожной полиции.

Стек может быть представлен массивом. Отличие заключается в том, куда помещаются и откуда читаются элементы. Стек с числами, добавленными в порядке возрастания, показан на рис. 2.10.

Добавление в стек обычно называется *вталкиванием*, а чтение значения из стека называется *извлечением (выталкиванием)*. Стеки полезны для возврата по следам совершенных шагов. Возможно, вы уже знакомы со *стеком вызовов*, поскольку он показывает не только место возникновения исключения, но и путь выполнения. Выполненная с использованием стека функция знает, куда возвращаться, так как перед ее вызовом в стек добавляется адрес возврата. Когда

функция возвращается к вызывающей стороне, считывается последний адрес, помещенный в стек, и ЦП продолжает выполнение по этому адресу.



Рис. 2.10. Общее представление стека

2.2.9. Стек вызовов

Стек вызовов — это структура данных, в которой функции хранят адреса возврата, чтобы знать, куда возвращаться после завершения выполнения. На каждый поток приходится один стек вызовов.

Каждое приложение выполняется в одном или нескольких отдельных процессах. Процессы позволяют изолировать память и ресурсы. Каждый процесс содержит один или несколько потоков. Поток — это единица выполнения. В современных операционных системах потоки выполняются параллельно друг другу, из чего возник термин *многопоточность*. Даже с четырехъядерным процессором в операционной системе могут выполняться тысячи потоков параллельно, так как потоки большую часть времени ожидают завершения какой-либо операции. Поэтому можно заполнить их слоты другими потоками, реализуя параллельное выполнение. Благодаря этому многозадачность возможна даже на одном процессоре.

В старых системах UNIX процесс был и контейнером для ресурсов приложения, и единицей выполнения. Хотя такой подход прост и элегантен, при нем часто возникали *зомби-процессы*. Потоки требуют меньше ресурсов и не имеют такой проблемы, поскольку привязаны к времени выполнения.

Каждый поток имеет свой собственный стек вызовов — фиксированный объем памяти. По традиции стек растет сверху вниз в пространстве памяти процесса, где верх означает конец области памяти, а низ — это нулевой указатель

(нулевой адрес). Если элемент помещается в стек вызовов, то указатель стека уменьшается.

Как и все хорошее, стек может закончиться, так как имеет фиксированный размер. При его превышении ЦП вызывает исключение `StackOverflowException`. Вы столкнетесь с ним, если вызовете функцию из самой себя. Однако для большинства задач размера стека хватит, поэтому нет нужды беспокоиться о его переполнении.

Стек вызовов содержит не только адреса возврата, но также параметры функций и локальные переменные. Поскольку локальные переменные занимают мало памяти, использование стека для них очень эффективно и позволяет избегать дополнительных шагов по управлению памятью, таких как выделение и освобождение.

Стек работает быстро, но имеет такое же время жизни, как и использующая его функция. Когда функция завершается, поток выполнения возвращается к месту вызова и пространство стека освобождается для следующей функции. Вот почему стек идеален для хранения только небольшого количества локальных данных. Поэтому управляемые среды выполнения, такие как C# и Java, хранят в стеке не данные классов, а только ссылки на них.

Это еще одна причина того, что типы значений иногда обеспечивают большую производительность, чем ссылочные типы. Типы значений существуют в стеке только при локальном объявлении, хотя и передаются при копировании.

2.3. К ЧЕМУ ВЕСЬ ЭТОТ АЖИОТАЖ С ТИПАМИ?

Программисты принимают типы данных как должное. Есть даже мнение, что программировать на языках с динамической типизацией, например JavaScript или Python, быстрее, потому что не приходится тратить время на такие сложности, как определение типа каждой переменной.

ПРИМЕЧАНИЕ *Динамическая типизация* означает, что типы переменных или членов класса в языке программирования могут изменяться во время выполнения. В JavaScript можно присвоить переменной вначале строку, а затем целое число, потому что это язык с динамической типизацией. В статически типизированных языках, например C# или Swift, такой возможности нет. Подробнее об этом мы поговорим позже.

Да, указание типа для каждой переменной и параметра — это рутинा, но ключом к быстрой работе является целостный подход. Скорость важна не только при написании кода, но и при его обслуживании. Вам, скорее всего, не придется беспокоиться об обслуживании, если вас уволили. В остальном же разработка ПО — это марафон, а не спринт.

Быстрая ошибка — одна из лучших практик разработки. А знание типов данных — один из первых уровней защиты от того, чтобы не забуксововать. Это знание позволяет совершить ошибку быстрее и исправить ее до того, как она станет критической. Кроме того что вы не спутаете строку с целым числом, у определения типов есть и другие преимущества.

2.3.1. Сила типов

Типы есть в большинстве языков программирования. Даже в самых простых языках, таких как BASIC, были типы: строки и целые числа, а в некоторых из его разновидностей даже действительные числа. Существует несколько бестиповых языков, например Tcl, REXX, Forth и др. Эти языки работают только с одним типом, обычно со строкой или целым числом. Программирование без необходимости думать о типах удобнее, но написанные таким образом программы работают медленнее и в них больше ошибок.

Типы — это, по сути, бесплатная проверка на правильность кода, поэтому их понимание поможет стать программистом высокого класса. Способ реализации типов сильно зависит от того, интерпретируемый язык или компилируемый.

- *Интерпретируемые языки программирования*, такие как Python или JavaScript, позволяют запускать код без компиляции. Поэтому переменные в этих языках, как правило, имеют гибкие типы: можно присвоить строковое значение целочисленной переменной и даже складывать строки и числа. Такие языки обычно называют языками с *динамической типизацией* — по способу реализации типов. На интерпретируемых языках код пишется намного быстрее, потому что не требуется объявлять типы.
- *Компилируемые языки программирования*, как правило, более строгие. Насколько они строги, зависит от того, сколько неприятностей хотел причинить вам создатель языка. Например, язык Rust можно считать образцом немецкой инженерии: необычайно строгий, перфекционистский и, как следствие, безошибочный. Язык C тоже можно считать немецким продуктом, но больше похожим на Volkswagen: он позволяет нарушать правила и расплачиваться за это позже. Оба языка типизированы статически. После объявления переменной

её тип невозможно изменить, но Rust считается *сильно типизированным*, как C#, а C – *слабо типизированным*.

Степень типизации показывает, насколько легко в языке преобразовывать один тип переменных в другой. В этом смысле в C меньше ограничений: можно без проблем присвоить указатель целому числу или наоборот. C# более строг: указатели/ссылки и целые числа являются несовместимыми типами. В табл. 2.2 приведены категоризации некоторых языков программирования.

Таблица 2.2. Строгость типов в языках программирования

	Статически типизированные Тип переменной не может меняться во время выполнения	Динамически типизированные Тип переменной может меняться во время выполнения
Сильно типизированные Различные типы невзаимо- заменяемы	C#, Java, Rust, Swift, Kotlin, TypeScript, C++	Python, Ruby, Lisp
Слабо типизированные Различные типы могут заменять друг друга	Visual Basic, C	JavaScript, VBScript

Строгие языки программирования могут повергнуть в уныние. А такие, как Rust, могут даже заставить задуматься о жизни и о том, зачем мы существуем во Вселенной. Объявление и явное преобразование типов в этих языках могут показаться слишком бюрократизированными. Например, в JavaScript не нужно объявлять типы каждой переменной, аргумента или члена.

Зачем обременять себя явными типами, если многие языки программирования могут работать без них? Ответ прост: типы помогают писать более безопасный, быстрый и простой в обслуживании код. Время, которое мы потратили на объявление типов переменных и аннотирование классов, компенсируется при отладке благодаря меньшему количеству ошибок и отсутствию проблем с производительностью.

Помимо очевидных преимуществ, у типов есть и неявные достоинства. Давайте рассмотрим их.

2.3.2. Проверка правильности

Проверка правильности – одно из неочевидных преимуществ предопределенных типов. Предположим, вы разрабатываете платформу для микроблогов, на

которой существует ограничение на количество символов в посте и никто никого не осуждает за нежелание формулировать мысли длиннее одного предложения. На этой гипотетической платформе можно упоминать других пользователей в постах, используя префикс @, и другие посты, используя префикс #, за которым следует идентификатор поста. Любой пост можно найти, введя его идентификатор в поле поиска. А введя в поле поиска имя пользователя с префиксом @, можно найти профиль этого пользователя.

Многообразие вариантов пользовательского ввода доставляет массу хлопот с проверкой. Что произойдет, если пользователь введет буквы после префикса #? Что, если он введет число недопустимой длины? Может показаться, что эти проблемы решаются сами собой, но обычно в приложении происходит сбой, потому что где-то в коде элемент, получив недопустимый ввод, выдаст исключение. Это худший сценарий для пользователя: он не понимает, что пошло не так и что делать дальше. Проблема может даже перейти в плоскость безопасности, если введенные данные отображаются без очистки.

Организовать проверку данных по всему коду достаточно сложно. Вы можете проверить ввод в клиенте, но, к примеру, стороннее приложение может отправить запрос без проверки. Вы можете проверить код, обрабатывающий веб-запросы, но другое ваше приложение, например API, будет вызывать служебный код без проверки. Точно так же код базы данных может получать запросы из нескольких источников, скажем, с сервисного уровня и уровня обслуживания, поэтому необходимо проверять, что вы добавляете в базу данных верные записи. На рис. 2.11 показано, в каких точках приложению может потребоваться проверка ввода.

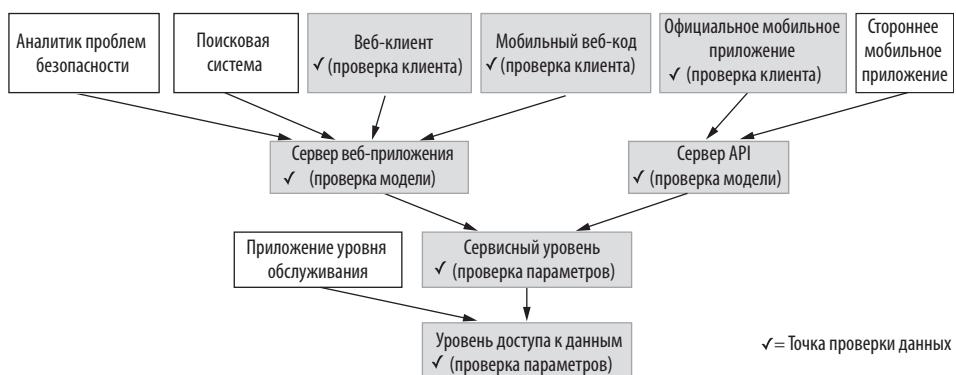


Рис. 2.11. Непроверенные источники данных и места, в которых необходима регулярная проверка данных

В конечном итоге потребуется проверять ввод по всему коду, причем последовательно. Вы же не хотите, чтобы идентификатор поста оказался равен `-1`, а имя пользователя '`OR 1=1--`' (это обычная атака с внедрением SQL-кода, которую мы рассмотрим в главе о безопасности).

СТИЛЬ ПРИМЕРОВ КОДА

Использование фигурных скобок — вторая по популярности тема в программировании, консенсус по которой еще не достигнут. Она уступает только выбору между табуляциями и пробелами. Для большинства С-подобных языков, особенно C# и Swift, я предпочитаю стиль Олмана. В этом стиле каждый символ фигурной скобки располагается на отдельной строке.

Swift официально рекомендует использовать 1TBS (One True Brace Style), также известный как улучшенный стиль K&R, где открывающая скобка располагается на одной строке с объявлением. Однако необходимость в пустых строках после каждого объявления блока остается, потому что иначе код в стиле 1TBS плохо читаем. Добавление пустых строк фактически превращает любой стиль в стиль Олмана, но в этом никто себе не признается.

В C# стиль Олмана используется по умолчанию. Я считаю, что его гораздо легче читать, чем 1TBS или K&R. Между прочим, Java использует 1TBS.

Мне пришлось форматировать код в 1TBS из-за ограничений печатного издания в отношении набора текста, но при написании кода в C# я советую использовать стиль Олмана. Дело не только в том, что его проще читать, но и в том, что это самый популярный стиль для C#.

Типы могут содержать проверку правильности. Вместо того чтобы передавать целые числа для идентификаторов постов в блоге или строки для имен пользователей, можно использовать классы или структуры, которые проверяют ввод при своем создании, что делает невозможным добавление в них недопустимого значения. Это простой, но мощный инструмент. Любая функция, которая получает идентификатор поста в качестве параметра, запрашивает класс `PostId` вместо целого числа. Это позволяет проводить проверку правильности после контроля в конструкторе. Целое число необходимо проверять, а `PostId` уже проверен. Контролировать его содержимое больше не нужно, потому что этот класс невозможно создать без проверки, как видно в следующем фрагменте кода. Единственный способ создать `PostId` в этом случае — вызвать конструктор, который проверяет его значение и выдает исключение в случае сбоя. Это значит, что получить недопустимый экземпляр `PostId` невозможно:

```

public class PostId
{
    public int Value { get; private set; } ← Значение не может быть изменено
    public PostId(int id) { ← Конструктор — единственный способ создать этот объект
        if (id <= 0) {
            throw new ArgumentOutOfRangeException(nameof(id));
        }
        Value = id;
    }
}

```

На практике этот путь не так прост, как в рассмотренном выше примере. Например, сравнение двух разных объектов `PostId` с одним и тем же значением не будет работать как ожидалось, потому что по умолчанию сравниваются только ссылки, а не содержимое классов (ссылки и значения мы обсудим ниже в этой главе). Придется добавлять целые леса конструкций вокруг, чтобы все работало как надо.

Вот некоторые рекомендации и полезные факты:

- Для сравнения двух экземпляров класса необходимо как минимум переопределить метод `Equals`, потому что от него могут зависеть некоторые библиотеки и функции фреймворка.
- Если вы планируете писать собственный код для сравнения значений с помощью операторов `==` и `!=`, предусмотрите их *перегрузку* в классе.
- Если вы планируете использовать класс в качестве ключа в `Dictionary<K, V>`, переопределите метод `GetHashCode`. Позже в этой главе я объясню, как связаны хеширование и словари.
- Функции форматирования строк, такие как `String.Format`, используют метод `ToString` для получения строкового представления класса, пригодного для вывода.

В листинге 2.2 показан класс `PostId` с настройками, необходимыми для проверки, что он работает во всех сценариях равенства. Мы переопределили `ToString()`, чтобы этот класс стал совместим с форматированием строк и его значение было проще проверять во время отладки. Мы переопределили `GetHashCode()`, чтобы он возвращал `Value` напрямую, потому что ее значение идеально соответствует типу `int`. Также мы переопределили метод `Equals()`, чтобы проверки на равенство в коллекциях этого класса работали правильно, если нам нужны уникальные значения или мы хотим выполнить поиск по значению. Наконец, мы переопределили операторы `==` и `!=`, чтобы напрямую сравнивать классы `PostId`, не обращаясь к их значениям.

НЕ ИСПОЛЬЗУЙТЕ ПЕРЕГРУЗКУ ОПЕРАТОРОВ БЕЗ НЕОБХОДИМОСТИ

Перегрузка операторов — это способ изменить в языке программирования поведение таких операторов, как `==`, `!=`, `+` и `-`. Узнав о возможности перегрузки, некоторые разработчики увлекаются ею настолько, что стремятся создать свой собственный язык со странным поведением ненужных классов. Например, они перегружают оператор `+=` для вставки записи в таблицу с синтаксисом `db += Record`. Практически невозможно понять назначение такого кода, если не прочитать документацию. В среде IDE нет функции, позволяющей определить, какие операторы перегружаются определенным типом. Не используйте перегрузку оператора без надобности, даже если вы забыли, что именно он делает. Используйте перегрузку только при необходимости создания альтернатив операторам равенства и приведения типов. Не тратьте время на реализацию перегрузки, если она не понадобится.

В некоторых примерах мы будем использовать перегрузку операторов, чтобы сделать классы семантически эквивалентными значениям, которые они представляют. Тогда класс будет работать с оператором `==` так же, как и число, которое он представляет.

ПРИМЕЧАНИЕ Неизменяемый класс, предназначенный исключительно для представления значений, на улицах называется типом значения. Знать жаргонные словечки не помешает, но не стоит зацикливаться на этом. Обращайте внимание только на их пользу.

Листинг 2.2. Полная реализация класса, содержащего значение

СИНТАКСИС СТРЕЛКИ

Синтаксис стрелки появился в C# версии 6.0 и соответствует обычному синтаксису метода с одним оператором `return`. Вы можете использовать обозначение стрелки, если благодаря этому код станет легче читать. Синтаксис стрелки сам по себе не хорош и не плох — хорош код, который легко читать, а плох код, который понять невозможно.

Описание метода без стрелок:

```
public int Sum(int a, int b) {
    return a + b;
}
```

То же самое:

```
public int Sum(int a, int b) => a + b;
```

Обычно этого не требуется, но если класс необходимо поместить в контейнер, который сортируется или сравнивается, реализуйте эти две дополнительные функции:

- Обеспечьте упорядочивание с помощью `IComparable<T>`, поскольку равенства для этого недостаточно. В листинге 2.1 мы этого не делали, потому что идентификаторы не ранжировались.
- Если вы планируете сравнивать значения с помощью операторов «меньше» или «больше» (`<`, `>`, `<=`, `>=`), вам необходимо реализовать перегрузки и для них.

Это требует массы времени и сил, особенно когда можно просто передать целое число, но затраты окупаются в больших проектах, когда вы работаете в команде. Еще больше преимуществ вы увидите, прочитав следующие главы.

Чтобы эффективно использовать контекст проверки, не всегда требуется создавать новые типы. Можно применить наследование для создания базовых типов, содержащих определенные примитивы с общими правилами. Например, имеющийся общий тип идентификатора можно адаптировать к другим классам. Можно просто переименовать класс `PostId` в листинге 2.1 в `DbId` и вывести из него все типы.

Всякий раз, когда вам нужен новый тип, такой как `PostId`, `UserId` или `TopicId`, можно наследовать его от `DbId` и расширять по мере необходимости. В этом случае можно создать полнофункциональные разновидности идентификаторов одного типа, чтобы лучше отличать их от других типов. Кроме того, можно сделать классы специализированными:

```

public class PostId: DbId {
    public PostId(int id): base(id) { }
}
public class TopicId: DbId {
    public TopicId(int id) : base(id) { }
}
public class UserId: DbId {
    public UserId(int id): base(id) { }
}

```

Отдельные типы для элементов вашего дизайна, если они применяются совместно и часто, упрощают семантическое разделение вариантов использования типа `DbId`, а также предотвращают передачу неверного типа идентификатора в функцию.

ПРИМЕЧАНИЕ Всякий раз, когда вы видите хорошее решение проблемы, убедитесь, что знаете также, когда это решение использовать не нужно. Рассмотренный сценарий повторного использования не исключение. Для простого прототипа может не понадобиться такая сложность и даже не требуется специальный класс. Когда вы замечаете, что часто передаете одно и то же значение функциям, и забываете, нужно ли его проверять, может быть полезнее включить его в класс и передавать класс.

Пользовательские типы данных эффективны, потому что объясняют дизайн лучше, чем примитивы, а также помогают избежать повторных проверок и, как следствие, ошибок. Но их реализация может быть затратной. Более того, необходимые типы могут предоставляться в используемом фреймворке.

2.3.3. Используйте фреймворк с умом

.NET, как и многие другие фреймворки, поставляется с набором полезных абстракций для определенных типов данных, но об этих возможностях обычно никто не знает или их игнорируют. В результате пользовательские текстовые значения, такие как URL-адреса, IP-адреса, имена файлов или даже даты, хранят в виде строк. Рассмотрим некоторые из таких готовых типов и способы их эффективного использования.

Возможно, вы уже знаете о классах на основе .NET для разных типов данных, но до сих пор предпочитаете строки, потому что оперировать ими проще. Проблема в том, что функциям неизвестно, проверена ли строка. Это ведет либо к непредвиденным ошибкам, либо к лишнему коду повторной проверки, что замедляет работу. В таких случаях лучше использовать готовый класс для определенного типа данных.

Когда из инструментов у вас имеется только молоток, любая проблема становится гвоздем. То же самое относится и к строкам.

Строки — отличное универсальное хранилище, и их легко анализировать, разделять, объединять или изменять. Все это заманчиво. Но такой подход ведет к тому, что время от времени вы пытаетесь изобрести велосипед — стараетесь использовать функции обработки строк, даже если это совершенно не нужно.

Рассмотрим такой пример: вам поручили написать службу поиска для компании под названием Supercalifragilisticexpialidocious, занимающейся сокращением URL-адресов. У этой компании возникли финансовые проблемы, и вы их единственная надежда. Их сервис работает по следующей схеме:

1. Пользователь предоставляет длинный URL-адрес, например:

<https://llanfair.com/pwllgw/yngyll/gogerych/wyrndrobwll/llan/tysilio/gogo/goch.html>

2. Сервис создает короткий код для URL-адреса и новый короткий URL-адрес, например:

<https://su.pa/mK61>

3. Всякий раз, когда пользователь в своем браузере переходит по короткому URL-адресу, происходит перенаправление на длинный URL-адрес.

Функция, которую необходимо разработать, должна извлекать короткий код из сокращенного URL.

Строковый подход будет выглядеть следующим образом:

```
public string GetShortCode(string url)
{
    const string urlValidationPattern =
        @"^https://([\w-]+.)+([\w-]+(/[\w- ./?%&=])?")?$/"; ← Регулярное выражение.
    if (!Regex.IsMatch(url, urlValidationPattern)) {           Используется в парсинге строк
        return null; ← Недопустимый URL-адрес
    }
    // взять часть после последнего слеша
    string[] parts = url.Split('/');
    string lastPart = parts[^1]; ← Новый синтаксис C# 8.0, который ссылается
    return lastPart;                                              на предпоследний элемент в диапазоне
}
```

На первый взгляд с кодом все в порядке, но если исходить из гипотетического ТЗ, он содержит ошибки. Шаблон проверки URL-адресов, неполный и пропускает недопустимые URL-адреса. В частности, он не учитывает возможность наличия нескольких слешей в URL-адресе и даже создает ненужный массив строк, только чтобы получить последнюю часть URL-адреса.

ПРИМЕЧАНИЕ Говорить о наличии ошибок можно, только имея ТЗ. Если ТЗ отсутствует, назвать какую-то операцию ошибочной нельзя. Это позволяет компаниям избегать скандалов, отмахиваясь от ошибок со словами: «Это не баг, а фича». ТЗ не обязательно должно быть письменным — оно может существовать только в вашей голове, пока вы способны ответить на вопрос: «Функция точно должна работать именно так?».

Что еще более важно, неочевидна логика кода. Лучше использовать класс `Uri` из .NET Framework:

```
public string GetShortCode(Uri url) ← Понятно, чего ожидать
{
    string path = url.AbsolutePath; ← Смотри-ка, никаких регулярных выражений!
    if (path.Contains('/')) {
        return null; ← Недопустимый URL-адрес
    }
    return path;
}
```

Здесь мы не парсим строки сами. К моменту вызова функции парсинг уже выполнен. Такой код нагляднее и его легче писать, и все только потому, что мы использовали `Uri` вместо `string`. Поскольку парсинг и проверка выполняются раньше, такой код становится легче отлаживать. В этой книге есть целая глава об отладке, но лучшая отладка — это ее отсутствие.

В дополнение к примитивам, таким как `int`, `string`, `float` и т. д., .NET предоставляет множество других полезных типов данных. Тип `IPAddress` — лучшая альтернатива строке для хранения IP-адресов не только потому, что в него встроена проверка, но и поскольку он поддерживает современный протокол IPv6. Это невероятно, я знаю. В классе `IPAddress` также содержатся элементы быстрого доступа для определения локального адреса:

```
var testAddress = IPAddress.Loopback;
```

Таким образом, вам не потребуется вводить `127.0.0.1` всякий раз, когда нужен петлевой адрес, что позволит работать быстрее. Если вы допустите ошибку в IP-адресе, то обнаружите ее раньше, чем в строке.

Еще один полезный тип — `TimeSpan`. Как следует из названия, он выражает продолжительность во времени. Эта характеристика используется почти во всех проектах разработки, особенно в механизмах кэширования и контроля срока действия. Мы определяем продолжительность в константах времени компиляции. Худший из возможных способов такой:

```
const int cacheExpiration = 5; // минуты
```

Не сразу понятно, что срок действия кэша измеряется в минутах. Единицу измерения невозможно узнать, не посмотрев исходный код. Надо хотя бы включить ее в название, чтобы вашим коллегам, а через какое-то время и вам самим не пришлось лишний раз заглядывать в исходный код:

```
public const int cacheExpirationMinutes = 5;
```

Так лучше, но если вам понадобится значение того же параметра, но в другой функции с другой единицей измерения, то придется выполнить преобразование:

```
cache.Add(key, value, cacheExpirationMinutes * 60);
```

Это дополнительная работа, которую нужно не забыть сделать. И здесь тоже могут возникать ошибки. Можно случайно ошибиться в цифре 60 и получить на выходе неверное значение, а потом потратить дни на отладку или тщетно пытаться оптимизировать производительность, хотя все дело в ошибке расчета.

`TimeSpan` в этом смысле поражает воображение. Ничего другого для представления продолжительности не требуется, даже если функция, которую вы вызываете, не принимает `TimeSpan` в качестве параметра:

```
public static readonly TimeSpan cacheExpiration = TimeSpan.FromMinutes(5);
```

Только посмотрите! Вы знаете, что это продолжительность, и она объявлена. Еще лучше, что для использования в других функциях вам не требуется знать единицу измерения, вы просто передаете `TimeSpan`. Если функция получает значение времени как целое число в определенных единицах измерения, скажем, в минутах, можно вызвать ее следующим образом:

```
cache.Add(key, value, cacheExpiration.TotalMinutes);
```

При этом значение конвертируется в минуты. Гениально!

Существуют и другие полезные типы, например `DateTimeOffset`, который, как и `DateTime`, передает дату и время, но дополнительно включает информацию о часовом поясе.

На практике всегда лучше использовать именно `DateTimeOffset`, потому что его легко преобразовать в/из `DateTime`, а при внезапном изменении информации о часовом поясе компьютера или сервера данные не потеряются. Выполнив перегрузку операторов, вместе с `TimeSpan` и `DateTimeOffset` можно использовать даже арифметические операторы:

```

var now = DateTimeOffset.Now;
var birthDate =
    new DateTimeOffset(1976, 12, 21, 02, 00, 00,
        TimeSpan.FromHours(2));
TimeSpan timePassed = now - birthDate;
Console.WriteLine($"It's been {timePassed.TotalSeconds} seconds since I was
➥ born!");

```

ПРИМЕЧАНИЕ Работать с датой и временем достаточно сложно, а ошибиться легко, особенно в глобальных проектах. Вот почему существуют отдельные сторонние библиотеки для различных сценариев, такие как Noda Time Джона Скита (Jon Skeet).

.NET похож на гору золота, в которую прыгает дядя Скрудж. В ней полно замечательных утилит, облегчающих жизнь разработчика. Их изучение может показаться бесполезным или скучным занятием, но в итоге обеспечит выигрыш во времени по сравнению с использованием строк или изобретением собственных решений.

2.3.4. Типы вместо опечаток

Писать комментарии к коду может быть утомительно, и далее в книге я буду стремиться этого не делать, но прежде чем бросаться в меня клавиатурой, дочитайте этот раздел до конца. Даже без комментариев код должен оставаться описательным. Объяснить код помогут типы.

Представьте, что в бескрайних подземельях кодовой базы проекта вы наткнулись на такой фрагмент кода:

```

public int Move(int from, int to) {
    // ... quite a code here
    return 0;
}

```

Что делает эта функция? Что она перемещает? Какие параметры принимает? Какой результат возвращает? Без типов ответы на эти вопросы будут расплывчатыми. Можно попытаться разобраться в коде или найти внешний класс, но это займет время. Все было бы проще, если бы имя функции было удачным:

```

public int MoveContents(int fromTopicId, int toTopicId) {
    // ... quite a code here
    return 0;
}

```

Так намного лучше, но осталось непонятным, какой результат она возвращает. Это код ошибки, количество перемещенных элементов или новый идентификатор, созданный в результате конфликта перемещения? Как передать эту информацию, не оставляя комментарии? С типами, конечно. Рассмотрите такой фрагмент кода:

```
public MoveResult MoveContents(int fromTopicId, int toTopicId) {
    // ... still quite a code here
    return MoveResult.Success;
}
```

Появилась какая-то ясность. Этот фрагмент не очень информативен, поскольку мы уже знали, что `int` — результат перемещения, но теперь можно посмотреть, что на самом деле делает тип `MoveResult`, просто нажав клавишу `F12` в Visual Studio и VS Code:

```
public enum MoveResult
{
    Success,
    Unauthorized,
    AlreadyMoved
}
```

Теперь стало значительно понятнее. Изменения не только упростили понимание API метода, но и улучшили сам код функции, потому что вместо констант или, что еще хуже, жестко закодированных целочисленных значений вы видите понятный `MoveResult.Success`. В отличие от констант в классе, перечисления ограничивают значения, которые можно передавать, и имеют собственное имя типа, что позволяет подробнее описать свои действия.

Поскольку функция получает в качестве параметров целые числа, в нее необходимо включать проверки, так как это общедоступный API. Вы скажете, что проверки необходимы даже для внутреннего или личного кода, поскольку они стали повсеместной практикой. В нашем случае логику проверки целесообразно включить в исходный код:

```
public MoveResult MoveContents(TopicId from, TopicId to) {
    // ... still quite a code here
    return MoveResult.Success;
}
```

Как видите, типы могут работать на вас, перемещая код в нужное место и облегчая его понимание. А поскольку компилятор еще и проверяет, правильно ли написано имя типа, это позволяет избежать опечаток.

2.3.5. Быть nullable или non-nullable?

Все разработчики рано или поздно столкнутся с `NullReferenceException`. Хотя Тони Хоар (Tony Hoare), в простонародье известный как «изобретатель `null`», называет его создание «ошибкой на миллиард долларов», не все так безнадежно.

КРАТКАЯ ИСТОРИЯ NULL

`Null`, или `nil` в некоторых языках, — это значение, символизирующее отсутствие значения или апатию программиста. Обычно это синоним нуля. Поскольку адрес с нулевым значением означает недопустимую область памяти, современные процессоры могут перехватывать такие обращения, преобразовывая их в любезные сообщения об исключении. В средние века вычислительной эры, когда обращения к нулевому адресу памяти не проверялись, компьютеры зависали, ломались или просто перезагружались.

Проблема не в `null` как таковом — нам все равно нужно описывать отсутствующее значение в коде. У него есть свой смысл. Проблема в том, что всем переменным может быть присвоено значение `null` по умолчанию, а проверки незапланированного присваивания `null` отсутствуют, что в конце концов и приводит к сбою.

В JavaScript в дополнение к другим проблемам с системой типов предусмотрено два разных значения: `null` и `undefined`. Первое выражает отсутствие значения переменной, а `undefined` — то, что переменная не определена. Я знаю, это больно. Примите JavaScript таким, какой он есть.

В C# 8.0 появилась новая возможность — *ссылки, допускающие null-значение (nullable references)*. Ее суть на первый взгляд проста: по умолчанию ссылки не могут иметь значение `null`. Вот и все. Это, пожалуй, самое существенное изменение в C# после появления универсальных шаблонов. Все остальные возможности таких ссылок связаны с этим новшеством.

Название нового вида ссылок может вводить в заблуждение, поскольку ссылки допускали значение `null` и до C# 8.0. Вместо этого стоило использовать термин «*ссылки, не допускающие null*» (*non-nullable references*), чтобы было более понятно. Мне ясна логика названия: оно указывает на способ представления типов значений, допускающих значение `null`, но многие разработчики могут подумать, что ничего нового в этом нет.

Когда все ссылки допускали значение `null`, функции, которые принимали ссылки, могли получать два разных значения: действительную ссылку и `null`.

Любая функция, не ожидавшая `null`, выдавала ошибку при попытке сослаться на это значение.

Все изменилось, когда ссылки стали `non-nullable` по умолчанию. Функции никогда не получат значение `null`, если в проекте есть вызывающий код. Рассмотрим следующий код:

```
public MoveResult MoveContents(TopicId from, TopicId to) {
    if (from is null) {
        throw new ArgumentNullException(nameof(from));
    }
    if (to is null) {
        throw new ArgumentNullException(nameof(to));
    } // .. фактический код здесь
    return MoveResult.Success;
}
```

СОВЕТ Синтаксис `is null` в приведенном выше коде может показаться неуместным. Я начал использовать его вместо `x == null` после того, как прочитал твиты senior-инженеров Microsoft. Очевидно, что оператор `is` не может быть перегружен, поэтому он гарантированно возвращает правильный результат. Аналогичным образом можно использовать синтаксис `x is object` вместо `x != null`. Проверки значений на `non-nullable` избавляют от проверок на `null`, но внешний код по-прежнему может вызывать ваш код с `null`, например, если вы публикуете библиотеку. В этом случае вам все равно может потребоваться явная проверка значений на `null`.

ЗАЧЕМ НУЖНА ПРОВЕРКА НА NULL, ЕСЛИ ВЫПОЛНЕНИЕ КОДА ВСЕ РАВНО ЗАВЕРШИТСЯ СБОЕМ?

Если не проверить аргументы на значение `null` в самом начале, функция продолжит выполнение до тех пор, пока не сошлется на такое значение. В результате она может остановиться в нежелательном состоянии, таком как недописанная запись, или не остановиться, но выполнить недопустимую операцию. Быстрый сбой во избежание необрабатываемых состояний — это лучше, чем такое состояние. Не надо бояться сбоев, потому что они дают возможность найти ошибки.

При быстром сбое трассировка стека будет более понятной. Вы будете точно знать, какой параметр вызвал сбой функции.

Не все нулевые значения нужно проверять. Возможно, вы получаете необязательное значение, и `null` — самый простой способ выразить необязательность. В главе об обработке ошибок мы обсудим это более подробно.

Вы можете включить проверку на `null` во всем проекте или в отдельных файлах. Я всегда рекомендую первый вариант, если проект новый, потому что это стимулирует писать правильный код с самого начала и тратить меньше времени на исправление ошибок. Чтобы включить проверку на `null` в отдельном файле, добавьте в его начало строку `#nullable enable`.

ПРОФЕССИОНАЛЬНЫЙ СОВЕТ Всегда заканчивайте директиву компилятора `enable/disable` шаблоном `restore`, а не противоположной директивой. Так вы не затронете глобальные настройки проекта, если вы с ними работаете. В противном случае вы можете пропустить ценную обратную связь.

С включенными проверками на `null` код выглядит так:

```
#nullable enable
public MoveResult MoveContents (TopicId from, TopicId to) {
    // ... фактический код здесь
    return MoveResult.Success;
}
(nullable restore
```

В этом случае попытка вызвать функцию `MoveResult` с нулевым значением или значением, допускающим `null`, завершится немедленным предупреждением от компилятора, а не неожиданной ошибкой в рабочей версии продукта. Вы обнаружите ошибку еще до того, как начнете использовать код. Вы можете игнорировать предупреждения и продолжать, но лучше никогда так не делайте.

Поначалу ссылки, допускающие значение `null`, могут раздражать. Объявлять классы так же легко, как раньше, не получится. Предположим, что мы создаем страницу сайта, на которой будут регистрироваться участники конференции. Страница получает имя и адрес электронной почты участника и записывает результаты в БД. В классе есть поле для источника, принимающее строку в произвольной форме из рекламной сети. Если эта строка не имеет значения, значит, участник перешел на страницу напрямую, а не из объявления. Создадим такой класс:

```
#nullable enable
class ConferenceRegistration
{
    public string CampaignSource { get; set; }
    public string FirstName { get; set; }
    public string? MiddleName { get; set; } ← Отчество необязательно
    public string LastName { get; set; }
```

```

public string Email { get; set; }
public DateTimeOffset CreatedOn { get; set; } ← Для ревью удобно, если в БД
}                                         сохраняется дата создания
(nullable restore)

```

При попытке скомпилировать класс вы получите предупреждение компилятора для всех строк, объявленных ненулевыми, то есть для всех свойств, кроме `MiddleName` и `CreatedOn`:

Non-nullable property '...' is uninitialized. Consider declaring the property
as nullable.

Отчество необязательно, поэтому мы объявили `MiddleName` как обнуляемое. Вот почему оно не получило ошибку компиляции.

ПРИМЕЧАНИЕ Никогда не используйте пустые строки для обозначения необязательности. Используйте `null`. Ваши коллеги не поймут, зачем нужна пустая строка. Это действительное значение или указание на необязательность? Непонятно. `null` же однозначен.

О ПУСТЫХ СТРОКАХ

На протяжении всей карьеры вам придется объявлять пустые строки, причем не для того, чтобы указать на необязательность. Не используйте для объявления пустых строк нотацию `""`. Поскольку код можно просматривать в множестве различных сред, таких как текстовый редактор, окно вывода в исполнителе тестов или веб-страница непрерывной интеграции, эту нотацию легко спутать со строкой, состоящей из одного пробела `" "`. Явно объявляйте пустые строки с помощью `String.Empty`, чтобы эффективно использовать существующие типы. Имя класса может быть и в нижнем регистре — `string.Empty`, если это позволяет кодовым соглашением. Код должен передавать ваши намерения.

`CreatedOn` — это структура, так что компилятор просто заполняет ее нулями. Поэтому он не предупреждает об ошибке, но тем не менее, лучше этой ошибки избежать.

Первая реакция разработчика на ошибку компиляции — согласиться на предложение компилятора. В примере выше это означало бы объявить свойства обнуляемыми, но это противоречит нашей задаче. Неверно делать так, чтобы свойства имени и фамилии стали необязательными. Но нам нужно подумать о том, как применить семантику необязательности.

Если вы хотите, чтобы свойство не было нулевым, ответьте на несколько вопросов. Первый: есть ли у свойства значение по умолчанию? Если есть, установите это значение во время создания свойства. Это поможет лучше понять поведение класса при изучении кода. Если полю источника задано значение по умолчанию, это можно выразить так:

```
public string CampaignSource { get; set; } = "organic";
public DateTimeOffset CreatedOn { get; set; } = DateTimeOffset.Now;
```

Тем самым мы избежим предупреждения компилятора и сообщим о своих намерениях всем, кто читает код.

Имя и фамилия не могут быть необязательными и не могут иметь значений по умолчанию. Нет, не стоит устанавливать в качестве значений по умолчанию «Джон» и «Смит». Следующий вопрос: как будет инициализироваться этот класс?

Если вы хотите, чтобы класс был инициализирован в пользовательском конструкторе и никогда не принимал недопустимые значения, присвойте значения свойств в таком конструкторе и объявите их неизменяемыми с помощью `private set`. Мы обсудим это подробнее в разделах о неизменяемости. В конструкторе можно также указать на необязательность с помощью параметра со значением по умолчанию `null`, как показано ниже.

Листинг 2.3. Пример неизменяемого класса

```
class ConferenceRegistration
{
    public string CampaignSource { get; private set; }
    public string FirstName { get; private set; }
    public string? MiddleName { get; private set; }           Все свойства находятся
    public string LastName { get; private set; }             в закрытом наборе
    public string Email { get; private set; }
    public DateTimeOffset CreatedOn { get; private set; } = DateTimeOffset.Now;

    public ConferenceRegistration(
        string firstName,
        string? middleName,
        string lastName,
        string email,
        string? campaignSource = null) { ← null указывает
                                         на необязательность
        FirstName = firstName;
        MiddleName = middleName;
        LastName = lastName;
        Email = email;
        CampaignSource = campaignSource ?? "organic";
    }
}
```

Я уже слышу ваши жалобы: «Но это же так долго». Согласен. Создавать неизменяемый класс не должно быть сложно. К счастью, для простоты в C# 9.0 добавлена новая конструкция — *тип записи*. А если вы не используете C# 9.0, то придется определиться, что для вас главное: меньшее количество ошибок или скорость работы?

ТИПЫ ЗАПИСИ В ПОМОЩЬ

В C# 9.0 появился тип записи, что значительно упрощает создание неизменяемых классов. Класс в листинге 2.3 можно записать следующим образом:

```
public record ConferenceRegistration(  
    string CampaignSource,  
    string FirstName,  
    string? MiddleName,  
    string LastName,  
    string Email,  
    DateTimeOffset CreatedOn);
```

Будут автоматически созданы и сделаны неизменяемыми свойства с теми же именами, что и аргументы в списке параметров. Таким образом, код выше эквивалентен записи класса в листинге 2.3. Вы также можете добавлять в тело блока кода методы и дополнительные конструкторы как обычный класс, вместо того чтобы заканчивать объявление точкой с запятой. Это феноменально. Потрясающая экономия времени.

Ответ неочевиден, потому что мы, люди, очень плохо оцениваем будущие расходы и обычно заботимся только о ближайшей перспективе. Так, я пишу эту книгу, сидя в самоизоляции в Сан-Франциско из-за пандемии COVID-19, потому что человечество не смогло предвидеть последствия небольшой вспышки в китайском Ухане. Оценщики рисков из нас так себе. Придется это принять.

Итак, у вас есть выбор. Можно устранить целый класс ошибок, вызванных отсутствием проверок на `null` и неправильным состоянием, потратив время на добавление нужного конструктора. Или можно оставить все как есть и потом разбираться с последствиями каждой зафиксированной ошибки: составлять отчеты, отслеживать проблемы, обсуждать их с менеджером проекта, проводить сортировку и исправлять ошибку только для того, чтобы столкнуться с еще одной того же класса... и в конце концов махнуть рукой: «OK, с меня хватит, я сделаю, как сказал Седат». Что вы выберете?

Как я уже говорил, на практике, чтобы определиться с выбором, требуется интуитивно понимать, сколько ошибок может быть в той или иной части кода. Не спешите слепо применять предложенное. Вы должны представлять, сколько

кода напишете зря, то есть сколько изменений придется вносить. Чем больше код изменится в будущем, тем больше будет в нем ошибок.

Предположим, вы подумали и решили: «Нет, игра не стоит свеч, оставим все как есть». В этом случае все еще можно избежать ошибок, сохранив проверки на допустимость значений `null`, но предварительно инициализировав поля следующим образом:

```
class ConferenceRegistration
{
    public string CampaignSource { get; set; } = "organic";
    public string FirstName { get; set; } = null!;
    public string? MiddleName { get; set; }
    public string LastName { get; set; } = null!;
    public string Email { get; set; } = null!;
    public DateTimeOffset CreatedOn { get; set; }
}
```

Объявление `null!` в качестве новой конструкции

Оператор `!` однозначно сообщает компилятору: «Я знаю, что делаю». Конкретно в этом случае: «Я позабочусь о том, чтобы инициализировать эти свойства сразу после создания класса. Я знаю, что если этого не сделать, проверки на допустимость значений `null` не будут работать». Таким образом, вы сохраняете защиту от ошибок с `null`, если выполняете обещание немедленно инициализировать эти свойства.

Поступать так рискованно, потому что очень сложно скоординировать всех членов команды, и они все равно могут инициализировать свойства позже. Если вы уверены, что управляете рисками в полной мере, используйте этот способ. Иногда по-другому и не получится, например, при использовании библиотек вроде Entity Framework, для которых требуется конструктор по умолчанию и устанавливаемые свойства объектов.

MAYBE<T> УМЕР, ДА ЗДРАВСТВУЕТ NULLABLE<T>!

Поскольку в C# компилятор ранее не поддерживал проверку правильности типов, допускающих значение `null`, а ошибка приводила к сбою всей программы, их старались не использовать для указания на необязательность. Поэтому разработчики применяли собственные необязательные типы `Maybe<T>` или `Option<T>`, без риска выдачи `NullReferenceException`. В C# 8.0 проверки безопасности для нулевых значений вышли на новый уровень, поэтому эра собственных необязательных типов официально закончилась. Теперь компилятор проверяет и оптимизирует типы, допускающие значение `null`, лучше, чем в случае пользовательских самоделок. Дополнительно вы получаете синтаксическую поддержку языка с операторами и распознавание шаблонов. Да здравствует `Nullable<T>`!

Проверки на допустимость значений `null` помогают обдумать код, который вы пишете. Например, вы будете более четко представлять, действительно ли то или иное значение необязательно. Это уменьшит количество ошибок и сделает вас лучше как разработчика.

2.3.6. Высокая производительность бесплатно

Производительность — не главная проблема при написании прототипа, но иметь представление о характеристиках производительности типов, структур данных и алгоритмов полезно, чтобы работать быстрее. Вы можете писать более быстрый код, используя специальные типы вместо общих, и даже не подозревать об этом.

Существующие типы могут обеспечить более эффективное хранение данных, причем *бесплатно*. Например, допустимая строка IPv6 может содержать максимум 65 символов. Адрес IPv4 имеет длину не менее семи символов. Это означает, что строковое хранилище будет занимать от 14 до 130 байт, а при включении в заголовки объектов — от 30 до 160 байт. Тип `IPAddress`, с другой стороны, хранит IP-адрес в виде последовательности байтов и использует от 20 до 44 байт. На рис. 2.12 показана разница в организации памяти при хранении IP-адресов.

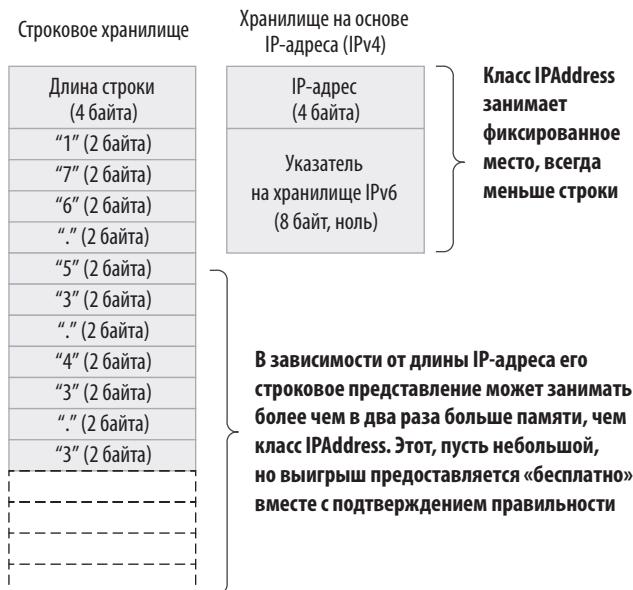


Рис. 2.12. Различия в хранении типов данных (без включения в общие заголовки объектов)

Разница может показаться непринципиальной, но помните: это бесплатно. Чем длиннее IP-адрес, тем больше места вы экономите. А еще получаете подтверждение правильности, поэтому можете быть уверены, что переданный объект содержит действительный IP-адрес во всем коде. Ваш код становится более понятным, потому что типы также описывают логику, стоящую за данными.

С другой стороны, все мы знаем, что бесплатный сыр бывает только в мышеловке. В чем тут подвох? Когда не стоит использовать специальный тип? Что ж, он предполагает некоторые расходы на парсинг при деконструкции строки на байты. Один фрагмент кода просматривает строку и определяет, представляет она адрес IPv4 или IPv6, а затем соответствующим образом анализирует ее, используя другой, оптимизированный код. С другой стороны, проверки строки после парсинга избавляют от необходимости проверять остальной код, что компенсирует эти накладные расходы. Если тип будет верным с самого начала, вы избежите затрат на проверку типов переданных аргументов. И последнее, но не менее важное: правильные типы позволяют эффективно использовать типы значений, когда это целесообразно. Подробнее о преимуществах типов значений мы поговорим в следующем разделе.

Производительность и масштабируемость — не одномерные понятия. Например, оптимизация хранения данных в некоторых случаях может приводить к снижению производительности, как я объясню в главе 7. И при всех преимуществах специальных типов переменных практически всегда оправданно использование специальных типов данных.

2.3.7. Ссылочные типы и типы значений

Основное различие между ссылочными типами и типами значений заключается в способе хранения в памяти. Простыми словами: содержимое типов значений хранится в стеке вызовов, тогда как ссылочные типы хранятся в куче, а в стеке вызовов хранится только ссылка на их содержимое. Вот простой пример кода:

```
int result = 5; ← Примитивный тип значения
var builder = new StringBuilder(); ← Ссылочный тип
var date = new DateTime(1984, 10, 9); ← Все структуры являются типами значений
string formula = "2 + 2 = "; ← Примитивный ссылочный тип
builder.Append(formula);
builder.Append(result);
builder.Append(date.ToString());
Console.WriteLine(builder.ToString()); ← Выводит ужасную математику
```

В Java нет других типов значений, кроме примитивов, таких как `int`. C# позволяет в дополнение к ним задавать собственные типы значений. Зная отличия между ссылочными типами и типами значений, вы сможете работать эффективнее, поскольку будете использовать подходящий тип. Разобраться в этом несложно.

Ссылка похожа на управляемый указатель. Указатель — это адрес в памяти. Обычно я представляю память в виде очень-очень длинного массива байтов, как показано на рис. 2.13.

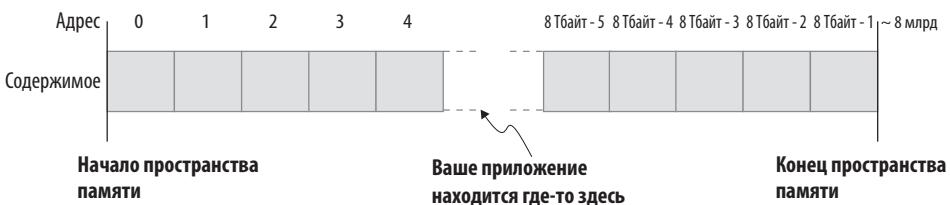


Рис. 2.13. Структура памяти 64-битного процесса, который может адресовать до 8 Тбайт

Это не вся оперативная память; это структура памяти только одного процесса. Физическая оперативная память устроена намного сложнее, но операционные системы скрывают ее беспорядок за аккуратным, чистым и непрерывным пространством памяти для каждого процесса, которое в реальности может и вовсе отсутствовать на вашем ОЗУ. Вот почему она называется виртуальной памятью. По состоянию на 2020 год ни на одном компьютере нет 8 Тбайт ОЗУ, но в 64-разрядных операционных системах доступно 8 Тбайт памяти. Я уверен, что в будущем над этими цифрами будут смеяться так же, как я смеюсь над своим старым компьютером из 1990-х годов с 1 Мбайт памяти.

ПОЧЕМУ 8 ТБАЙТ? Я ДУМАЛ, ЧТО 64-РАЗРЯДНЫЕ ПРОЦЕССОРЫ МОГУТ АДРЕСОВАТЬ 16 ЭКСАБАЙТ!

Процессоры действительно имеют такую возможность, но ограничение пользовательского пространства в основном продиктовано практикой. Создание таблицы отображения для меньшего объема виртуальной памяти требует меньших ресурсов ОС и времени. К примеру, переключение процессов требует полного перераспределения памяти, и большое адресное пространство замедлит его. В будущем, когда 8 Тбайт оперативной памяти станут обычным делом, можно будет расширить пользовательское пространство, но до тех пор 8 Тбайт — наш предел.

Указатель — это, по сути, число, указывающее на адрес в памяти. Преимущество использования указателей вместо фактических данных в том, что это избавляет от ненужного дублирования, которое может быть весьма дорогостоящим. Можно просто передавать гигабайты данных от функции к функции в адресе, то есть в указателе. В противном случае нам пришлось бы копировать гигабайты памяти при каждом вызове функции. Вместо этого мы просто копируем одно число.

Очевидно, что нет смысла использовать указатели для чего-то, что меньше по размеру, чем сам указатель. 32-битное целое число (`int` в C#) вдвое меньше указателя в 64-битной системе. Таким образом, примитивы, такие как `int`, `long`, `bool` и `byte`, считаются типами значений. Это означает, что вместо указателя на их адрес функциям передается их значение.

Ссылка является синонимом указателя, за исключением того, что доступ к ее содержимому управляет средой выполнения .NET. Значение ссылки также неизвестно. Это позволяет сборщику мусора по мере необходимости самостоятельно очищать память, на которую указывает ссылка. Указатели можно использовать в C#, но только в небезопасном контексте.

C# позволяет использовать сложные типы значений, называемые структурами. Структура по определению очень похожа на класс, но передается иначе. Если у вас есть структура и вы отправляете ее в функцию, создается копия структуры, а когда функция передаст структуру другой функции, будет создана еще одна копия. Структуры всегда копируются. Рассмотрим следующий пример.

Листинг 2.4. Пример неизменяемости

```
struct Point
{
    public int X;
    public int Y;
    public override string ToString() => $"X:{X},Y:{Y}";
}

static void Main(string[] args) {
    var a = new Point() {
        X = 5,
        Y = 5,
    };
    var b = a;
    b.X = 100;
    b.Y = 200;
    Console.WriteLine(b);
    Console.WriteLine(a);
}
```

СБОРКА МУСОРА

Программист должен следить за выделением памяти и освобождать (перераспределять) память, когда она больше не используется. В противном случае приложение будет использовать все больше памяти и произойдет ее утечка. Выделение и освобождение памяти вручную чревато ошибками. Программист может забыть об этом или, что еще хуже, попытаться освободить уже свободную память, что является причиной многих ошибок безопасности.

Одним из первых решений проблем ручного управления памятью был подсчет ссылок. Это примитивная форма сборки мусора. Среда выполнения хранит секретный счетчик для каждого выделенного объекта. Каждая ссылка на объект увеличивает счетчик, а каждый раз, когда переменная, ссылающаяся на объект, выходит за пределы области видимости, счетчик уменьшается. Если счетчик достигает нуля, значит, переменные, ссылающиеся на объект, отсутствуют, и объект высвобождается.

Подсчет ссылок отлично подходит для многих сценариев, но у него есть недостатки: он работает медленно, потому что операция удаления объекта выполняется каждый раз, когда ссылка на него выходит за пределы области видимости. Обычно это менее эффективно, чем, скажем, совместное освобождение соответствующих блоков памяти. Проблемы, требующие дополнительных усилий разработчика, возникают также при подсчете циклических ссылок.

Еще одно решение — сборка мусора, или, если точнее, сборка помеченного мусора (Mark and Sweep), поскольку подсчет ссылок — это тоже форма сборки мусора. Сборка мусора — это компромисс между подсчетом ссылок и ручным управлением памятью. При сборке мусора отдельные счетчики ссылок не сохраняются. Вместо этого отдельный процесс просматривает все дерево объектов и помечает как мусор те из них, на которые отсутствуют ссылки. Мусор хранится какое-то время, и когда его количество превышает определенный порог, приходит сборщик мусора и освобождает всю неиспользуемую память за один раз. Операции микроосвобождения снижают накладные расходы на освобождение и фрагментацию памяти. Отсутствие счетчиков также делает код быстрее.

В языке Rust появилось новое средство управления памятью, называемое проверкой заимствования, когда компилятор отслеживает, в какой момент выделенная память больше не нужна. Это означает, что в Rust выделение памяти не требует дополнительных затрат при выполнении, но придется писать код определенным образом и пофиксить множество ошибок компилирования, прежде чем вы разберетесь, что к чему.

Как вы думаете, что эта программа выведет в консоль? Когда вы присваиваете `a` переменной `b`, среда выполнения создает копию `a`. То есть, изменяя `b`, вы изменяете новую структуру со значениями `a`, а не саму `a`. Что, если бы `Point` был классом? Тогда `b` и `a` будут иметь одинаковую ссылку, а изменение содержимого `a` будет означать одновременное изменение `b`.

Типы значений иногда более эффективны, чем ссылочные, как с точки зрения хранения, так и с точки зрения производительности. Мы уже обсуждали, что тип размером со ссылкой или меньше эффективнее передавать по значению. Кроме того, ссылочные типы имеют один уровень косвенности. Обращаясь к полю ссылочного типа, среда выполнения .NET должна сначала прочесть значение ссылки, затем перейти по адресу, указанному этой ссылкой, и только потом прочесть фактическое значение. Для типа значения среда выполнения считывает значение напрямую, что ускоряет доступ.

ИТОГИ

- Теория computer science может быть скучной, но ее знание поможет усовершенствовать навыки разработки.
- Типы обычно рассматриваются как шаблонный код в строго типизированных языках, но их можно использовать, чтобы писать меньше кода.
- .NET предоставляет улучшенные и более эффективные структуры для определенных типов данных. Эти структуры сделают код быстрее и надежнее.
- Благодаря использованию типов код станет более понятным, и, как следствие, к нему понадобится меньше комментариев.
- Добавленная в C# 8.0 функция ссылок, допускающих значение `null`, делает код намного более надежным и позволяет тратить меньше времени на отладку приложения.
- Разница между типами значений и ссылочными типами довольно велика, и вы будете работать эффективнее, если в ней разберетесь.
- Строки становятся полезнее, если знать внутренние принципы их работы.
- Массивы — быстрый и удобный инструмент, но они плохо подходят для общедоступного API.
- Структуру списка хорошо использовать, если он будет расширяться, но массивы более эффективны, если не требуется динамически увеличивать их содержимое.

- Связанный список — нишевая структура данных, но знание его особенностей поможет понять недостатки словарей.
- Словари отлично подходят для быстрого поиска ключей, но их производительность сильно зависит от правильной реализации `GetHashCode()`.
- Список уникальных значений можно представить с помощью `HashSet`, чтобы добиться отличной производительности поиска.
- Стеки — отличные структуры данных для отслеживания совершенных шагов. Стек вызовов конечен.
- Знание принципов работы стека вызовов, в дополнение к использованию правильных типов значений и ссылочных типов, поможет повысить производительность приложения.

3

Полезные антипаттерны

В этой главе

- ✓ Общеизвестные плохие практики, от которых может быть толк
- ✓ Действительно полезные антипаттерны
- ✓ Как понять, когда использовать лучшую практику, а когда ее антипод

Литература по программированию полна описаний лучших практик и шаблонов проектирования. Некоторые из них считаются эталонными, и на вас будут косо смотреть, если вы вздумаете усомниться в их эффективности. Со временем они становятся догмами. Время от времени кто-нибудь критикует одну из этих практик в блоге, и если статья получает одобрение сообщества Hacker News¹, ее признают обоснованной и открываются двери для новых идей. В остальных случаях такие догмы лучше даже не начинать обсуждать. Если бы у меня была возможность обратиться к разработчикам всего мира с одним-единственным

¹ Hacker News — это платформа для обмена новостями из мира технологий, где каждый является экспертом во всех сферах: <https://news.ycombinator.com>.

посланием, я бы попросил их подвергать сомнению все, чему их учат, — пользу, смысл, преимущества и стоимость использования инструментов.

Догмы и непреложные законы ограничивают наше видение, и чем дольше мы их придерживаемся, тем сильнее эти ограничения. Из-за подобных слепых зон от нас ускользают полезные методы, которые в определенных условиях могут стать еще более полезными.

Антипаттерны, или *плохие практики*, если хотите, имеют плохую репутацию вполне заслуженно, но это не значит, что мы должны шарахаться от них, как от радиации. Я разберу некоторые из таких шаблонов, которые могут оказаться более полезными, чем их аналоги из лучших практик. Тогда вы будете использовать практики и шаблоны проектирования, понимая, когда они полезны, а когда нет. Вы увидите, чего вам не хватает и какие сокровища скрыты в слепой зоне.

3.1. ЕСЛИ НЕ СЛОМАНО, СЛОМАЙ

Первое, что я усваивал в компаниях, где работал, — ну, после того, как узнавал, где находится буфет, — избегать изменений кода любой ценой. Каждое изменение несет в себе риск регресса, возникновения ошибки, нарушающей уже работающий сценарий. Ошибки в новой функциональности обходятся дорого, а их исправление требует времени. Но регресс хуже, чем выпуск новой функциональности с ошибками, — это шаг назад. Промахнуться по чужим воротам в футболе — ошибка. Забить в свои, принеся очко сопернику, — это регресс. Время — самый ценный ресурс в разработке, и его потеря влечет за собой самое серьезное наказание. Регрессы отнимают больше всего времени. Лучше избегать их и не ломать код.

Однако стремление избегать изменений может в итоге завести вас в порочный круг, потому что если для создания новой функции требуется сломать и переделать старую, вы будете сопротивляться такой разработке. Вы привыкнете ходить на цыпочках вокруг существующего кода и будете стараться поместить новое только в новый код, не трогая тот, что уже есть. Это приведет к неоправданному росту объема кода, который необходимо обслуживать.

Необходимость менять существующий код — большая проблема. Такое изменение может быть очень сложным, потому что код тесно связан с определенным способом работы, и изменение в одном месте потребует исправлений во многих других местах. Сопротивление кода изменениям называется *жесткостью*. Чем жестче код, тем большую его часть придется сломать, чтобы изменить.

3.1.1. Лицом к лицу с жестью

Жесткость кода зависит от множества факторов, и один из них — слишком большое количество зависимостей. Зависимость может относиться к сборке фреймворка, к внешней библиотеке или к другому объекту в самом коде. Все типы зависимостей могут создать проблемы, если код запутан. Зависимость — это и благословение, и проклятие. На рис. 3.1 представлена ужасная схема зависимостей. В ней нарушены границы ответственности, и любой сбой в одном из компонентов потребует изменения почти всего кода.

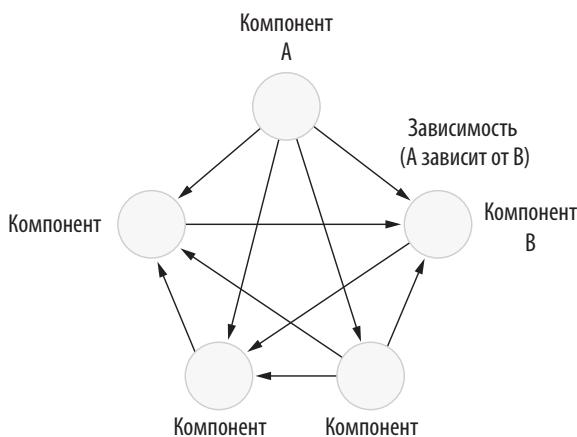


Рис. 3.1. Адская пентаграмма зависимостей

Почему с зависимостями столько проблем? Когда вы собираетесь добавить зависимость, рассматривайте каждый компонент как отдельного клиента или каждый уровень как отдельный сегмент рынка с разными потребностями. Обслуживание клиентов из нескольких сегментов требует большей ответственности, чем обслуживание только одного типа клиентов. У клиентов разные потребности, и вы будете вынуждены их удовлетворять, хотя вам это не нужно. Взвесьте все, принимая решение, касающееся цепочки зависимостей. В идеале постараитесь обслуживать как можно меньше типов клиентов. Это поможет сохранить компонент или целый слой максимально простым.

Без зависимостей обойтись невозможно. Они необходимы для повторного использования кода. Повторное использование кода — это договор из двух пунктов. Если компонент А зависит от компонента В, первый пункт звучит так: «В предоставляет услуги А». Существует также второй пункт, который часто упускают из виду: «А потребует обслуживания всякий раз, когда В вносит критическое

изменение». С зависимостями, которые обусловлены необходимостью повторного использования кода, все будет в порядке, если их цепочка будет организована и систематизирована.

3.1.2. Ломайте скорее

Нужно ли ломать код, чтобы он не компилировался или проваливал тесты? Пересечение зависимостей повышает жесткость кода, что делает его малопригодным к изменениям. Это как крутая гора, на которую вы будете карабкаться все медленнее и в конечном итоге остановитесь. В начале работы справиться с последствиями ломки легче, поэтому лучше найти проблемы и сломать код, даже если он работает. На рис. 3.2 показано, как рост числа зависимостей заставляет действовать быстрее.

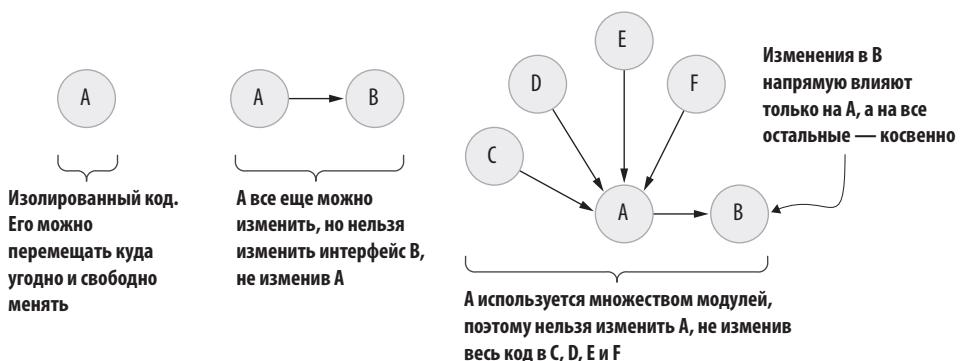


Рис. 3.2. Пригодность к изменениям обратно пропорциональна количеству зависимостей

Компонент без связей изменить проще всего. Ничего другого, кроме него, сломать невозможно. Наличие у компонента зависимости уже создает некоторую жесткость. Если вы меняете интерфейс B, вам придется изменить и A. Если вы измените реализацию B без изменения интерфейса, вы все равно нарушите работу A, потому что сломаете B. Проблема становится серьезнее, когда от одного компонента зависят несколько других.

Изменить A становится сложнее, потому что это требует изменения зависимых компонентов и влечет риск поломки любого из них. Разработчики склонны считать, что чем больше они повторно используют код, тем больше времени экономят. Но какой ценой? Подумайте об этом.

3.1.3. Соблюдайте границы

Первая привычка, которую вы должны усвоить, — избегать нарушения границ абстракции для зависимостей. Граница абстракции — это логическая граница вокруг слоев кода, набор функций данного слоя. Например, веб-уровень, а также уровни бизнеса и базы данных в коде могут быть абстракциями. В такой структуре кода уровень базы данных ничего не знает о веб-уровне или бизнес-уровне, а веб-уровень не знает о базе данных, как показано на рис. 3.3.

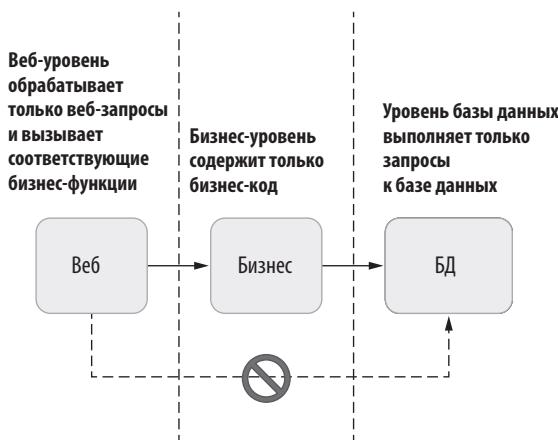


Рис. 3.3. Нарушение границ абстракции, которого следует избегать

Почему нарушать границы плохо? Потому что это лишает вас преимуществ абстракции. Когда вы переносите сложность нижних слоев на более высокие уровни, то берете на себя ответственность за обслуживание изменений на всех нижних слоях. Подумайте о членах команды, которые отвечают за свои слои. Внезапно разработчику веб-уровня придется изучить SQL. Кроме того, об изменениях на уровне БД придется сообщать большему количеству людей, чем необходимо. Это добавляет разработчику лишнюю головную боль. Затраты времени на согласование между сотрудниками будут расти в геометрической прогрессии. В результате вы потеряете время и преимущества абстракций.

Если вы столкнетесь с такими проблемами, сломайте код, например деконструируйте его, чтобы он стал нерабочим, устранимте нарушение, проведите рефакторинг и ликвидируйте последствия. Исправьте зависимые части кода. Будьте внимательны и немедленно решайте такие проблемы, даже рискуя сломать код. Код, который страшно ломать, — плохой. Это не значит, что хороший код не ломается, просто осколки хорошего кода склеить значительно проще.

ВАЖНОСТЬ ТЕСТОВ

Вы должны уметь оценить, приведет ли изменение кода к сбою сценария. Вы можете полагаться на собственное понимание кода, но оно будет тем меньше, чем сложнее будет становиться код с течением времени.

Часто целесообразно провести тесты. Они могут представлять собой список инструкций на листе бумаги или быть полностью автоматизированы. Последний вариант обычно предпочтительнее, потому что такие тесты пишутся только один раз и на их выполнение разработчик не тратит времени. Благодаря тестовым фреймворкам писать тесты тоже довольно просто. Мы подробнее рассмотрим эту тему в главе о тестировании.

3.1.4. Выделение общей функциональности

Означает ли все сказанное, что веб-уровень на рис. 3.3 никогда не сможет иметь общих функций с БД? Нет конечно. Но в таких случаях необходим отдельный компонент. Например, оба уровня могут полагаться на общие классы моделей. В этом случае у вас будет диаграмма отношений как на рис. 3.4.

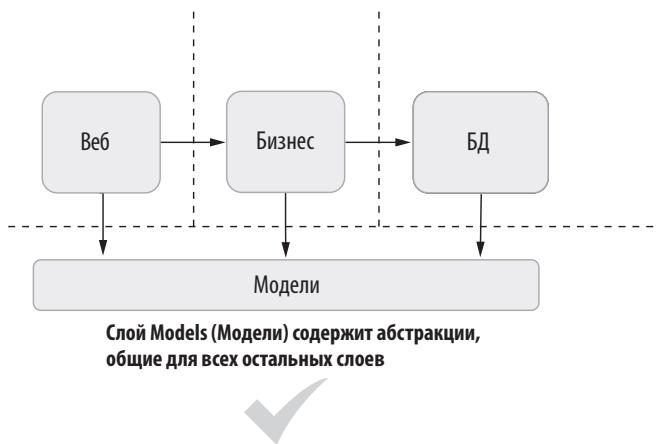


Рис. 3.4. Извлечение общей функциональности без нарушения абстракций

Рефакторинг кода может нарушить процесс сборки или привести к ошибкам тестирования, и теоретически этого нельзя допускать. Но нарушения, возникающие в ходе рефакторинга, можно назвать скрытыми сбоями. Они требуют немедленного исправления. Если же при этом появляется еще большее количество

сбоев и багов, это не значит, что во всем виноваты вы. Вероятно, ошибка уже была и теперь проявилась, так что с ней стало проще работать.

Рассмотрим пример. Предположим, что вы пишете API для чат-приложения, в котором общаться можно только с помощью эмодзи. Да, звучит ужасно, но существовал и чат, в котором можно было написать только «Йоу»¹. Так что наша идея круч.

Вы разрабатываете приложение с веб-уровнем, который принимает запросы от мобильных устройств и вызывает бизнес-уровень (также известный как *уровень логики*), который выполняет фактические операции. Такое разделение позволяет тестировать бизнес-уровень без веб-уровня. Затем эту же бизнес-логику можно использовать на других платформах, например на мобильном веб-сайте. Поэтому выделение бизнес-логики обоснованно.

ПРИМЕЧАНИЕ «Бизнес» в терминах «бизнес-логика» или «бизнес-уровень» не обязательно означает что-то, связанное с бизнесом, а скорее отсылает к основной логике приложения с абстрактными моделями. В принципе, чтение кода бизнес-уровня должно помочь получить более общее представление о том, как работает приложение.

Бизнес-уровень ничего не знает о базах данных или о методах хранения. Для этого он обращается к уровню базы данных. Уровень базы данных инкапсулирует функциональность базы данных независимо от последней. Такое разделение ответственности упрощает тестируемость бизнес-логики, поскольку к бизнес-уровню можно легко подключить имитацию уровня хранения. Что еще более важно, эта архитектура позволяет скрыто изменять БД, не меняя ни строки кода на бизнес- или веб-уровнях. Схематичное представление такой архитектуры показано на рис. 3.5.

Минус состоит в том, что каждый раз, когда вы добавляете новую функцию в API, вам придется создавать новый класс или метод бизнес-уровня и соответствующий класс и методы уровня БД. Это кажется нецелесообразным, особенно когда сроки сжатые, а функциональность довольно простая. «Зачем мне заморачиваться ради простого SQL-запроса?» — подумаете вы. Пойдем дальше и осуществим мечту многих разработчиков — нарушим существующие абстракции.

¹ Чат-приложение Yo, в котором можно было отправить только «Йоу» (Yo), когда-то оценивалось в 10 миллионов долларов. Компания работала до 2016 года: [https://en.wikipedia.org/wiki/Yo_\(app\)](https://en.wikipedia.org/wiki/Yo_(app)).

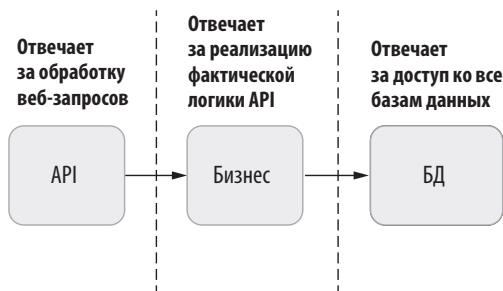


Рис. 3.5. Базовая архитектура API мобильного приложения

3.1.5. Пример веб-страницы

Предположим, ваш босс поручил вам внедрение новой функции — вкладки статистики, которая показывает, сколько всего сообщений отправил и получил пользователь. Это всего два простых SQL-запроса в серверной части:

```

SELECT COUNT(*) as Sent FROM Messages WHERE FromId=@userId
SELECT COUNT(*) as Received FROM Messages WHERE ToId=@userId
    
```

Вы можете выполнять эти запросы на своем уровне API. Даже если вы не знакомы с ASP.NET Core, с веб-разработкой или SQL, вам должен быть понятен код в листинге 3.1, определяющий модель, которая возвращается в мобильное приложение. Затем модель автоматически сериализуется в JSON. Получаем строку подключения к базе данных SQL-сервера. Используем эту строку, чтобы установить подключение, выполнить запросы к базе данных и вернуть результаты.

Класс `StatsController` в листинге 3.1 представляет собой абстракцию веб-обработки, в которой полученные параметры запроса находятся в аргументах функции, URL-адрес определяется именем контроллера, а результат возвращается в виде объекта. Таким образом, доступ к коду в листинге 3.1 можно получить из адреса вида `https://yourwebdomain/Stats/Get?userId=123`, при этом инфраструктура MVC автоматически сопоставляет параметры запроса с параметрами функции, а возвращаемый объект — с результатом JSON. Это упрощает написание кода для веб-обработки, поскольку вам не придется иметь дело с URL-адресами, строками запросов, заголовками HTTP и сериализацией JSON.

Листинг 3.1. Реализация функции путем нарушения абстракций

```

public class UserStats { ← Определяем модель
    public int Received { get; set; }
    
```

```

    public int Sent { get; set; }

}

public class StatsController: ControllerBase { ← Наш контроллер
    public UserStats Get(int userId) { ← Наша конечная точка API
        var result = new UserStats();
        string connectionString = config.GetConnectionString("DB");
        using (var conn = new SqlConnection(connectionString)) {
            conn.Open();
            var cmd = conn.CreateCommand();
            cmd.CommandText =
                "SELECT COUNT(*) FROM Messages WHERE FromId={0}";
            cmd.Parameters.Add(userId);
            result.Sent = (int)cmd.ExecuteScalar();
            cmd.CommandText =
                "SELECT COUNT(*) FROM Messages WHERE ToId={0}";
            result.Received = (int)cmd.ExecuteScalar();
        }
        return result;
    }
}

```

На написание этой реализации я потратил минут пять. Она выглядит просто. Зачем возиться с абстракциями? Давайте просто поместим все на уровень API?

Это можно сделать, если вы работаете над прототипами, которые не требуют идеального дизайна. Но при работе над конечным продуктом такие решения нужно принимать осторожно. Вы имеете право останавливать продакшен? Что, если сайт упадет на пару минут? Если вы считаете, что ничего страшного при этом не произойдет, продолжайте. А как насчет вашей команды? Готов ли человек, который обслуживает уровень API, к постоянным SQL-запросам? А что с тестированием? Как протестировать этот код и убедиться, что он работает правильно? А если добавить новые поля? Представьте, как вы приходите в офис на следующий день. Подумайте, как вас там встретят. Вас обнимут? Поприветствуют? Или подкинут на стул парочку канцелярских кнопок?

Вы добавили зависимость к физической структуре БД. Если в будущем потребуется изменить макет таблицы `Messages` или используемую технологию БД, вам придется просмотреть весь код и убедиться, что все работает с новой БД или новым макетом таблицы.

3.1.6. Не оставляйте за собой долгов

Мы, программисты, не умеем предсказывать будущие события и их стоимость. Когда мы принимаем плохие решения, только чтобы уложиться в срок, мы еще

больше усложняем себе жизнь, поскольку создаем беспорядок. Разработчики обычно называют это техническим долгом.

Технический долг — это решение, принятое сознательно. Бессознательные решения называются некомпетентностью. Долгом это решение называется потому, что либо вы его вернете, либо код рано или поздно придет за вами и сломает вам ноги монтировкой.

Технический долг накапливается по-разному. Может показаться, что проще передать произвольное значение, чем создавать для него константу. «Кажется, строка сюда подойдет», «от сокращения имени ничего не будет», «давайте я просто все скопирую и изменю кое-что», «я просто буду использовать регулярные выражения». Каждое маленькое неверное решение будет уменьшать вашу производительность и эффективность вашей команды. Ваша продуктивность будет становиться все меньше и меньше. Вы будете работать все медленнее, получая все меньше удовлетворения от того, что делаете, и все меньше одобрения от руководства. Ленясь неправильно, вы обрекаете себя на поражение. Ленитесь с умом: сначала поработайте.

Лучший способ справиться с техническим долгом — не становиться должником. У вас впереди большая работа? Считайте ее разогревом. Код может сломаться? Это хорошо, используйте поломку как возможность выявить жесткие части кода и сделать их детализированными и гибкими. Займитесь кодом, измените его, а затем, если он покажется вам недостаточно хорошим, откатите все изменения.

3.2. ПИШИТЕ С НУЛЯ

Если изменять код рискованно, то писать его с нуля рискованнее на порядок. Любой непроверенный сценарий может оказаться нерабочим. Это означает, что с нуля придется не только писать, но и исправлять ошибки. А такой способ устранения недостатков дизайна считается крайне неэффективным и затратным.

Однако все это верно только для кода, который уже работает. Начать же писать заново код, с которым вы уже повозились, может быть спасением. Как это, спросите вы? Это выход из спирали отчаяния. Она выглядит так:

1. Вы придумали простой и элегантный дизайн кода.
2. Вы начинаете писать код.
3. Возникают граничные случаи, которые вы не учли.

4. Вы начинаете пересматривать дизайн.
5. Вы замечаете, что текущий дизайн не соответствует требованиям.
6. Вы снова начинаете корректировать дизайн, но не переделываете его, потому что это приведет к слишком большому количеству изменений в коде. Теперь каждая строчка усиливает чувство стыда.
7. Ваш дизайн теперь похож на чудовище Франкенштейна, состоящее из разнородной смеси идей и кода. Потеряна элегантность, простота, а с ними и всяческая надежда.

На этой стадии вы попадаете в замкнутый круг, вообразив, что затраты не окупятся. Время, которое вы уже потратили на код, заставляет вас отказаться от его переделки. Хотя дизайн не решает основных задач, вы целыми днями пытаетесь убедить себя, что он рабочий. Может быть, в конце концов вы займетесь его исправлением, но потеряете недели только потому, что сами закопали себя в яму.

3.2.1. Стирайте и переписывайте

Начните с нуля, настаиваю я, перепишите. Отбросьте все, что вы уже сделали, и напишите заново. Вы не представляете, каким облегчением это станет. Вам кажется, что вы потратите вдвое больше времени, но это не так, потому что один раз это уже было сделано. Вы уже знаете, как обойти проблемы. Выигрыш от повторного выполнения задачи выглядит примерно так, как показано на рис. 3.6.



Рис. 3.6. Прелесть выполнения задачи снова и снова

Трудно переоценить ускорение, которого вы добиваетесь, делая что-то во второй раз. В отличие от киношных хакеров большую часть времени вы

проводите, глядя в монитор: не пишете, а обдумываете, как поступить. Программирование — это не столько созидание, сколько блуждание по лабиринту сложного дерева решений. Возвращаясь к началу лабиринта, вы уже знаете о возможных неудачах, ловушках и схемах, которые продумали, проходя его в прошлый раз.

Если вы чувствуете, что застряли, напишите все заново. Я бы посоветовал даже не сохранять прежний результат, но он вам может понадобиться, если вы не уверены, что сможете быстро повторить его. Хорошо, сохраните копию, но уверяю вас, в большинстве случаев она не пригодится. Она уже у вас в голове, направляет вас по быстрому пути и уводит от спирали отчаяния.

Что еще более важно, когда вы начнете заново, вы гораздо раньше поймете, что свернули не туда. На этот раз ваш внутренний детектор ошибок сработает. Вы выработаете чутье, которое подскажет, как должна работать создаваемая функция. Программирование в этом смысле очень похоже на консольную игру, такую как Spider-Man от Marvel или The Last of Us. Вы постоянно погибаете и начинаете проходить все сначала. Вы умираете и возрождаетесь. Вы становитесь лучше с каждым воскрешением в игре и с каждым повторением в программировании. Да, начиная с нуля, вы совершенствуете свои навыки разработки одной функции, но при этом улучшаются и ваши навыки разработки в целом. Они обязательно пригодятся вам в будущем. Не стесняйтесь стирать написанное и начинать заново. Не поддавайтесь заблуждению о неокупаемых затратах.

3.3. ЧИНИТЕ, ДАЖЕ ЕСЛИ НИЧЕГО НЕ СЛОМАНО

Справиться с негибкостью кода можно, в том числе поддерживая его в состоянии изменений, чтобы он не застыпал, насколько это возможно. Хороший код должно быть легко изменять, причем не меняя сотни фрагментов ради одного нужного. Можно вносить в код изменения, которые не требуются сейчас, но могут пригодиться в долгосрочной перспективе. Возьмите за привычку регулярно обновлять зависимости, поддерживать гибкость приложения и выявлять самые жесткие части, которые трудно изменить. Улучшайте код, устраивая также и мелкие недостатки.

3.3.1. Гонка за будущим

Вам не обойтись без использования экосистемы пакетов, и вы не будете их менять, потому что они работают на вас. Проблема в том, что когда вам

потребуется другой пакет, а для его использования — новая версия текущего пакета, обновить его будет значительно сложнее, чем если бы вы проводили его регулярное обновление и актуализацию. Схема этого конфликта представлена на рис. 3.7.

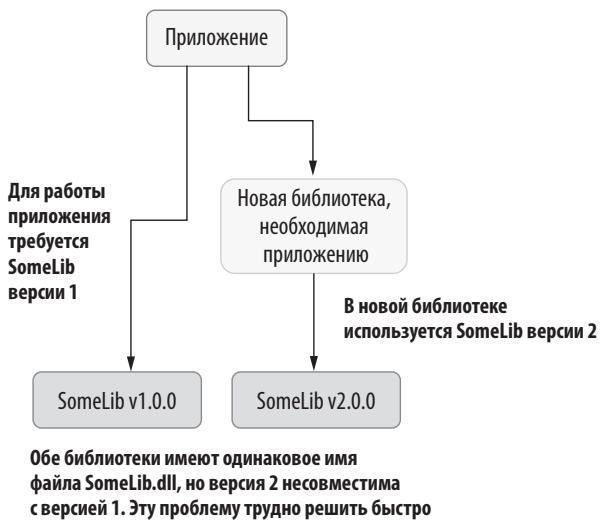


Рис. 3.7. Неразрешимые конфликты версий

Обычно специалисты по обслуживанию беспокоятся только о сценариях обновления основных версий, забывая о промежуточных. Например, популярная поисковая библиотека Elasticsearch требует, чтобы обновления выполнялись строго последовательно: обновить ее от одной основной версии до любой другой сразу не получится. .NET поддерживает перенаправление привязки, чтобы сгладить проблемы наложения версий одного и того же пакета. Перенаправление привязки — это директива в конфигурации приложения, которая указывает .NET на необходимость переадресовывать вызовы старой версии сборки на ее более новую версию, или наоборот. Конечно, это работает только тогда, когда оба пакета совместимы. Обычно вам не нужно заниматься перенаправлениями привязки самостоятельно, потому что Visual Studio делает это за вас, если в окне свойств проекта активирована опция «Автоматически создавать перенаправления привязки».

Регулярное обновление пакетов обеспечивает два важных преимущества. Во-первых, вы распределяете усилия по обновлению до текущей версии на период обслуживания. Каждый шаг будет менее сложным. Во-вторых, что более важно,

каждое небольшое обновление может вызвать незначительные и незаметные нарушения в коде или дизайне, и вам придется найти их и исправить, чтобы двигаться дальше. Это заставит вас улучшать код и дизайн постепенно, основываясь на тестах.

У вас может быть веб-приложение, использующее Elasticsearch для операций поиска и Newtonsoft.Json для парсинга и создания JSON. Это одни из самых популярных библиотек. Проблемы начинаются, когда вам нужно обновить пакет Newtonsoft.Json, чтобы использовать новую функцию, а Elasticsearch использует старую. Но чтобы обновить Elasticsearch, вам также нужно изменить код, который она обрабатывает. Как вы поступите?

Большинство пакетов поддерживают только последовательные обновления. Elasticsearch, например, можно обновить с версии 5 до версии 6, но не сразу до версии 7. Придется обновляться до каждой версии отдельно. Некоторые обновления также потребуют значительных изменений кода. Для Elasticsearch 7 придется писать его почти с нуля.

Конечно, можно оставаться на старых версиях под защитой проверенного кода, но их поддержка, как и документация и примеры кода, не вечно. Stack Overflow заполнен вопросами о проблемах совместимости, потому что, начиная проект, разработчики используют последнюю версию. Сообщества поддержки старых версий со временем исчезают. Это усложняет обновление с каждым годом, что подталкивает вас к спирали отчаяния.

Мой выбор — участвовать в гонке за будущим. Поддерживайте библиотеки в актуальном состоянии. Возьмите за правило регулярно их обновлять. Это время от времени будет вызывать ошибки в коде, но так вы узнаете, какая его часть более хрупкая, чтобы добавить туда больше тестов.

Главная идея состоит в том, что, работая с небольшими ошибками после обновлений, вы предотвращаете крупные сбои, с которыми было бы очень трудно справиться. Вы инвестируете не только в возможную будущую выгоду, но и в гибкость зависимостей приложения. Исправляйте код, чтобы при следующем изменении его было уже не так легко сломать, независимо от масштаба обновлений пакета. Чем лучше приспособлено приложение к изменениям, тем оно совершеннее с точки зрения дизайна и простоты обслуживания.

3.3.2. Качество кода и культура поведения

Что меня в первую очередь привлекло в компьютерах, так это их детерминизм. Все процессы всегда будут происходить гарантированно одинаково. Работающий

код будет работать всегда. Сначала я в это поверил, потому что был очень наивен. Но за свою карьеру я встретил множество ошибок, зависящих от скорости процессора или времени суток. Первое правило улиц: «Все меняется». Изменится ваш код, требования, документация, окружение. Вы не сможете поддерживать стабильно работающий код, не трогая его.

Избавившись от заблуждения о неизменности, можно расслабиться и принять тот факт, что менять код — нормально. Не стоит бояться перемен, потому что они все равно произойдут. Поэтому не медлите с улучшениями в работающем коде. Они могут быть небольшими: добавление нужных комментариев и удаление ненужных, улучшение имен. Поддерживайте код живым. Чем больше вы меняете код, тем менее устойчивым он становится, поскольку изменения приведут к сбоям, а сбои позволяют выявить слабые места и повысить управляемость кода. Вы должны разобраться, как и где ломается ваш код. В конце концов вы станете интуитивно понимать, какие изменения будут наименее рискованными.

Такая работа похожа на садоводство. Не обязательно добавлять функции или исправлять ошибки, но код должен становиться немного лучше после каждой доработки. Это поможет другому разработчику понять ваш код и улучшить покрытие тестами — как подарок от Санты, найденный рождественским утром, или оживший искусственный цветок в офисе.

Зачем заниматься этой неблагодарной работой? В идеале она должна оцениваться и вознаграждаться, но так бывает не всегда. Вас даже могут обругать, потому что коллег не устроят внесенные изменения. Вы можете нарушить их рабочий процесс и не нарушая код. Пытаясь улучшить первоначальный дизайн, вы можете сделать его хуже.

Все это должно быть ожидаемым. Единственный способ приобрести уверенность в обращении с кодом — менять его как можно больше. Убедитесь, что ваши изменения легко откатить, чтобы, если вы кого-то расстроили, вернуть исходный вариант. Также стоит научиться правильно сообщать об изменениях, которые могут повлиять на работу коллег. Хорошая коммуникация — лучший навык разработки.

Основное преимущество простого и постепенного улучшения кода состоит в том, что таким образом вы очень быстро погружаетесь в работу. Приступить к серьезной задаче бывает морально сложно. Вы не знаете, с чего начать и как справиться с большими изменениями. Вас охватывает пессимизм: «О, это так трудно, я буду страдать», — и вы откладываете проект. Но чем дольше откладываете, тем больше будете бояться начать писать код.

Небольшие улучшения — это уловка, которая заставит ваш мозг работать, чтобы разогреться для решения более серьезной задачи. Поскольку вы уже пишете код, мозг сопротивляется переключению меньше, чем если бы вы до этого зависали в соцсетях. Когнитивные процессы уже запущены и готовы к крупному проекту.

Если вы не видите, что можно легко улучшить, запустите анализатор кода. Это отличный инструмент для поиска мелких недочетов. Проверьте, что вы настроили параметры анализатора так, чтобы не задеть интересы других участников команды. Спросите у коллег, как они относятся к использованию анализатора. Если они не хотят отвлекаться на решение проблем, пообещайте им начать исправления самостоятельно и считайте это тренировкой. Используйте командную строку или собственный инструмент Visual Studio, чтобы анализировать код, не нарушая принятый порядок работы в команде.

Вам даже не обязательно применять внесенные изменения, потому что они нужны, только чтобы разогреть вас и настроить на написание нового кода. К примеру, вы сомневаетесь, применять ли исправление, которое выглядит рискованно. Вы уже немало поработали над ним, но понимаете, что лучше его отбросить. Всегда можно начать сначала и переделать все заново. Пусть вас не беспокоит, что работа пропадет зря. Если вам очень жалко результатов своего труда, сохраните резервную копию, но я бы не стал сильно переживать.

Если вы уверены, что команда не возражает против ваших изменений, публикуйте их. Удовлетворение даже от незначительного улучшения мотивирует вас на более серьезные изменения.

3.4. НЕ БОЙТЕСЬ ПОВТОРЯТЬСЯ

Повторение и использование копирования и вставки при написании кода — это концепции, на которые в кругу разработчиков принято смотреть свысока. Как и всякая разумная рекомендация, это со временем превратилось в религию, заставляющую людей страдать.

Теория выглядит так: вы пишете фрагмент кода. Этот же фрагмент вам нужен в другом месте. Новичок просто скопировал бы и вставил его. Пока все идет хорошо. Затем вы находите ошибку в скопированном коде. Теперь вам нужно изменить код в двух разных местах. Вы должны синхронизировать изменения. Это потребует дополнительных усилий, и вы не уложитесь в сроки.

Все так, верно? Решение проблемы — поместить повторяющийся код в общий класс или модуль и использовать его. Таким образом, меняя общий код, вы

волшебным образом меняете его везде, где на него есть ссылки, что экономит массу времени.

Пока все идет хорошо, но это временно. Проблемы возникают, когда вы начинаете слепо применять этот принцип, где только можно. Если вы пытаетесь реорганизовать код в повторно используемые классы, то упускаете, что, по сути, создаете новые зависимости, а зависимости влияют на дизайн и иногда даже диктуют свои условия.

Самая большая проблема общих зависимостей в том, что у разных частей ПО, использующих общий код, могут быть разные требования. Когда это выясняется, разработчик рефлекторно пытается удовлетворить их, используя один и тот же код. Он добавляет необязательные параметры, условную логику, чтобы общий код удовлетворял два разных требования. Это усложняет код и в конечном итоге вызывает больше проблем, чем решает. В какой-то момент вы начинаете подумывать о более сложном дизайне, чем при простых копировании-вставке.

Рассмотрим пример: вам поручили написать API для онлайн-магазина. Клиенту необходимо изменить адрес доставки, представленный классом `PostalAddress`, например, таким образом:

```
public class PostalAddress {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
    public string Notes { get; set; }
}
```

Необходимо провести нормализацию полей, например использовать заглавные буквы, даже если пользователь не вводит их там, где нужно. Функция обновления может выглядеть как последовательность операций нормализации и обновления базы данных:

```
public void SetShippingAddress(Guid customerId,
    PostalAddress newAddress) {
    normalizeFields(newAddress);
    db.UpdateShippingAddress(customerId, newAddress);
}

private void normalizeFields(PostalAddress address) {
    address.FirstName = TextHelper.Capitalize(address.FirstName);
    address.LastName = TextHelper.Capitalize(address.LastName);
    address.Notes = TextHelper.Capitalize(address.Notes);
}
```

Этот метод подстановки заглавных букв будет работать, если сделать первый символ прописным, а остальную часть строки оставить в нижнем регистре:

```
public static string Capitalize(string text) {
    if (text.Length < 2) {
        return text.ToUpper();
    }
    return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
}
```

Теперь, кажется, с примечаниями для курьера и именами все в порядке: вместо «джон» получаем «Джон», а вместо «ОСТАВЬТЕ ПОСЫЛКУ У ДВЕРЕЙ» – «Оставьте посылку у дверей», не пугая курьера излишней тревожностью. Спустя некоторое время вы хотите привести в порядок названия городов. Вы добавляете в функцию `normalizeFields` следующее:

```
address.City = TextHelper.Capitalize(address.City);
```

Но получив заказ из Сан-Франциско, вы замечаете, что название города записано неверно: «Сан-франциско». Теперь нужно доработать логику функции изменения регистра, чтобы каждое слово начиналось с заглавной буквы. Это также поможет правильно писать имена детей Илона Маска. Но затем вы замечаете, что примечание для курьера стало выглядеть так: «Оставьте Посылку У Дверей». Это лучше, чем только заглавные буквы, но босс хочет, чтобы все было идеально. Что вы сделаете?

Может показаться, что проще всего изменить функцию `Capitalize`, добавив в нее дополнительный параметр поведения. Код в листинге 3.2 получает дополнительный параметр `EveryWord`, который указывает использовать заглавные буквы в каждом слове или только в первом. Обратите внимание, что параметр не называется `isCity` или похожим образом, потому что его назначение не входит в область задач функции `Capitalize`. Имена должны заключать пояснения в контексте их характера, а не в контексте вызывающей функции. В итоге если `everyWord` принимает значение `true`, текст разбивается на слова, и для каждого слова вызывается `Capitalize`, а затем слова вновь объединяются в строку.

Листинг 3.2. Первоначальная реализация функции Capitalize

```
public static string Capitalize(string text,
    bool everyWord = false) { ← Новый параметр
    if (text.Length < 2) {
        return text;
    }
}
```

```

if (!everyWord) { ← Обрабатывает только первую букву
    return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
}
string[] words = text.Split(' ');
for (int i = 0; i < words.Length; i++) {
    words[i] = Capitalize(words[i]);
}
return String.Join(" ", words);
}

```

Одна и та же функция вызывается, чтобы каждое слово начиналось с заглавной буквы

Выглядит сложно, но потерпите — я действительно хочу, чтобы вы разобрались. Изменение поведения функции кажется самым простым решением. Вы просто добавляете параметр и операторы `if`. В результате вырабатывается плохая привычка, почти рефлекс, обрабатывать так каждое небольшое изменение, что чревато огромными сложностями.

Допустим, вам также нужны заглавные буквы в именах файлов, загружаемых в приложение, и у вас уже есть функция, которая исправляет регистр. Слова в именах файлов должны быть набраны с заглавной буквы и разделяться нижним подчеркиванием. Например, если API получает файл с названием *invoice report*, он должен преобразовать его в *Invoice_Report*. Поскольку у вас уже есть функция `Capitalize`, вы хотите снова немного изменить ее поведение. Добавляем новый параметр `filename`, потому что нет более общего имени, и проверяем его там, где он имеет значение. При смене регистров необходимо использовать версии функций `ToUpper` и `ToLower`, не зависящие от языка и региональных параметров, чтобы название файла на компьютере в Турции внезапно не превратилось в `?nvoice_Report?` (обратите внимание на I с точкой в `?nvoice_Report?`). Теперь реализация выглядит так, как показано в листинге ниже.

Листинг 3.3. Функция «швейцарский нож», которая может делать все что угодно

```

public static string Capitalize(string text,
    bool everyWord = false, bool filename = false) { ← Новый параметр
    if (text.Length < 2) {
        return text;
    }
    if (!everyWord) {
        if (filename) { ← Код имени файла
            return Char.ToUpperInvariant(text[0])
                + text.Substring(1).ToLowerInvariant();
        }
        return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
    }
    string[] words = text.Split(' ');

```

```

for (int i = 0; i < words.Length; i++) {
    words[i] = Capitalize(words[i]);
}
string separator = " ";
if (filename) {
    separator = "_" ; ← Код имени файла
}
return String.Join(separator, words);
}

```

Посмотрите, какого монстра вы создали. Нарушен принцип сквозной функциональности, поскольку функция `Capitalize` затрагивает соглашения об именовании файлов. Она потеряла общий характер и стала частью специфической бизнес-логики. Да, удалось повторно использовать код, насколько это было возможно, но тем самым вы очень усложнили себе жизнь.

Обратите внимание, что также создан новый регистр, которого нет в дизайне: новый формат имени файла, в котором не все слова пишутся с заглавной буквы. Это поведение задается условием, при котором `everyWord` принимает значение `false`, а `filename` — `true`. Вы не собирались вводить это поведение, но теперь оно присутствует. Другой разработчик может использовать это поведение, и ваш код со временем превращается в спагетти.

Я предлагаю соблюдать гигиену: не бойтесь повторяться. Не пытайтесь свести каждый бит логики в один код, используйте отдельные функции, даже если код в них будет отчасти повторяться. Создайте функции для каждого случая: пусть одна оставляет заглавной только первую букву первого слова, вторая делает заглавной первую букву каждого слова, а третья фактически форматирует имя файла. Эти функции не обязательно должны располагаться рядом друг с другом — код для имени файла может оставаться ближе к бизнес-логике, которой он необходим.

Итак, у вас есть три функции, которые намного лучше соответствуют своим задачам. Первая называется `CapitalizeFirstLetter`, и из названия («первая буква — заглавная») понятно, что она делает. Вторая — `CapitalizeEveryWord` — тоже информативное название («каждое слово с заглавной»). Она вызывает `CapitalizeFirstLetter` для каждого слова, что гораздо легче понять, чем пытаться вникнуть в рекурсию. Наконец, функция `FormatFilename`, у которой совершенно другое имя («форматировать имя файла»), потому что она служит не только для правильной расстановки заглавных букв. В ней заново реализована вся логика изменения регистра. Это позволяет свободно редактировать функцию при изменении соглашений о форматировании имен файлов и не беспокоиться о регистрах букв.

Листинг 3.4. Более гибкий повторяющийся фрагмент, который проще читать

```

public static string CapitalizeFirstLetter(string text) {
    if (text.Length < 2) {
        return text.ToUpper();
    }
    return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
}

public static string CapitalizeEveryWord(string text) {
    var words = text.Split(' ');
    for (int n = 0; n < words.Length; n++) {
        words[n] = CapitalizeFirstLetter(words[n]);
    }
    return String.Join(" ", words);
}

public static string FormatFilename(string filename) {
    var words = filename.Split(' ');
    for (int n = 0; n < words.Length; n++) {
        string word = words[n];
        if (word.Length < 2) {
            words[n] = word.ToUpperInvariant();
        } else {
            words[n] = Char.ToUpperInvariant(word[0]) +
                word.Substring(1).ToLowerInvariant();
        }
    }
    return String.Join("_", words);
}

```

Таким образом, не придется втискивать всю возможную логику в одну функцию. Это особенно важно, когда требования вызывающих сторон различаются.

3.4.1. Повторное использование или копирование?

Как выбрать между повторным использованием кода и его репликацией в другом месте? Важнейший фактор при выборе — то, как вы воспринимаете ответственность за корректные формулировки. При описании требований к функции, форматирующей имя файла, вам может мешать тот факт, что уже имеется похожая функция, изменяющая регистр первых букв слова. Мозг стремится использовать то, что есть. Это могло бы иметь смысл, если бы имена файлов необходимо было только переводить в другой регистр, но разница в требованиях слишком существенна.

В компьютерной науке сложны три вещи: инвалидация кэша, именование объектов и ошибки смещения на единицу¹. Правильное именование — один

¹ Так Леон Бэмбрик замечательно перефразировал (<https://twitter.com/secretGeek/status/7269997868>) знаменитую цитату Фила Карлтона (Phil Karlton), который не упоминал об «ошибках смещения на единицу».

из самых важных факторов, влияющих на понимание кода при его повторном использовании. Имя `Capitalize` правильно определяет упомянутую выше функцию. Ее можно было бы назвать и `NormalizeName`, но это помешает повторному использованию в других разработках. Имена объектов должны отражать их реальную функциональность. Благодаря этому функция сможет использоваться верно, не создавая путаницы в отношении ее назначения. На рис. 3.8 показано, как различные подходы к именованию влияют на восприятие фактического поведения функции.

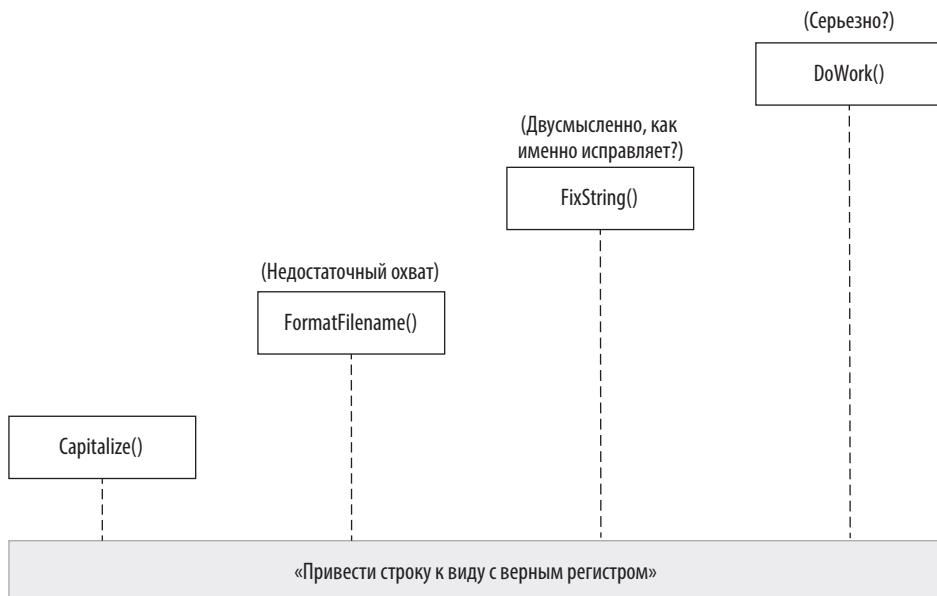


Рис. 3.8. Выбор имени, максимально отражающего реальную функциональность

Можно еще углубиться в детали, например: «Эта функция преобразует первые буквы каждого слова в строке в верхний регистр, а все остальные буквы — в нижний», но такую фразу трудно уместить в имени. Имена должны быть максимально краткими и однозначными. `Capitalize` в этом смысле отличный вариант.

Понимание, за что отвечает тот или иной фрагмент кода, — важный навык. Я обычно представляю, что функции и классы — это люди со своими интересами. И говорю о них как о людях: «Эту функцию не волнует....». Вы можете поступать так же с фрагментами кода. Параметр функции получил имя `EveryWord` («каждое слово»), а не `isCity` («город»), потому что этой функции все равно, город это или нет. Ее это не волнует.

Имена, отражающие зону ответственности объектов, помогают понять, как их использовать. Тогда почему мы в итоге назвали функцию форматирования имен файлов `FormatFilename`? Разве мы не должны были назвать ее `CapitalizeInvariantAndSeparateWithUnderscores` («сделать заглавным каждое слово и разделить нижним подчеркиванием»)? Нет. Функции могут выполнять несколько операций, но они решают только одну задачу и должны называться в соответствии с ней. Если вас так и тянет добавить в название функции союзы «и» и «или», то либо вы выбрали неверное название, либо возлагаете на функцию слишком большую ответственность.

Имена — это лишь одна из многих характеристик фрагмента кода. Расположение кода, его модуль и класс также могут повлиять на решение, использовать ли код повторно.

3.5. ИЗОБРЕТАЙТЕ

В Турции бытует поговорка, которая дословно переводится как «не изобретай ничего сейчас». Это означает «не доставляй нам проблем, у нас нет времени разбираться с новшествами». Изобретение велосипеда — это проблема. Разработчики даже придумали название этой болезни: синдром «изобрел не я». Ею страдают те, кто не может спать по ночам, пока не изобретет то, что уже создано.

Создавать заново что-то, у чего есть известная и работающая альтернатива, — это, безусловно, большая работа. И в ней тоже могут встречаться ошибки. Проблема возникает, когда повторное использование того, что уже есть, становится нормальным, а создание нового — невозможным. Такой подход со временем превращается в принцип «не изобретай ничего и никогда». Не бойтесь изобретать.

Во-первых, изобретатели любознательны. Если вы все время задаете вопросы, то неизбежно станете изобретателем. Когда вы перестаете задавать вопросы, вы начинаете скучать и превращаетесь в чернорабочего. Избегайте такого состояния, потому что человек, который не подвержен сомнениям, не способен оптимизировать свою работу.

Во-вторых, не у всех изобретений есть альтернативы. Ваши собственные абстракции тоже изобретения — ваши классы, ваш дизайн, вспомогательные функции, которые вы придумали. Все они служат для повышения производительности, но требуют изобретательности.

Я всегда хотел создать сайт, который предоставлял бы статистику Twitter о моих подписчиках и людях, на которых яписан. Но я не хочу изучать,

как работает API Twitter. Я знаю, что есть соответствующие библиотеки, но не хочу изучать их работу, или, что еще важнее, не хочу, чтобы их реализация влияла на мой дизайн. Если я использую определенную библиотеку, она привяжет к своему API, и, если я захочу изменить библиотеку, то придется полностью переписывать код.

Решение таких проблем требует изобретательности. Мы придумываем интерфейс мечты и помещаем его в качестве абстракции перед используемой библиотекой. Так мы избегаем привязки к определенному дизайну API. Если мы хотим изменить используемую библиотеку, то меняем только абстракцию, а не весь код. Я до сих пор понятия не имею, как работает веб-API Twitter, но полагаю, что это обычный веб-запрос, позволяющий проверить авторизацию для доступа к API Twitter. Это подразумевает получение элементов из Twitter.

Первая реакция разработчика — найти пакет и изучить его рабочую документацию, чтобы интегрировать в свой код. Вместо этого придумайте и запустите новый API, вызывающий библиотеку, которую вы используете. API должен быть максимально простым. Станьте своим собственным клиентом.

Прежде всего изучите требования API. Обычно веб-API предоставляет пользовательский веб-интерфейс для выдачи разрешений приложению. Он открывает, например, страницу в Twitter, которая запрашивает разрешения и возвращает обратно в приложение при получении подтверждения от пользователя. Следовательно, нам нужно узнать, какой URL открыть для авторизации и на какой URL перенаправить. Затем можно использовать данные со страницы перенаправления для дополнительных вызовов API.

После авторизации нам больше ничего не нужно. Итак, для нашей цели я придумал такой API:

Листинг 3.5. Воображаемый API Twitter

```
public class Twitter {
    public static Uri GetAuthorizationUrl(Uri callbackUrl) { ← Статические функции,
        string redirectUrl = "";                               которые обрабатывают
        // сделать что-нибудь, чтобы создать URL-адрес перенаправления
        return new Uri(redirectUrl);
    }
    public static TwitterAccessToken GetAccessToken( ← Статические функции, которые
        TwitterCallbackInfo callbackData) {                   обрабатывают поток авторизации
        // должно получиться что-то похожее на это
        return new TwitterAccessToken();
    }
}
```

```

public Twitter(TwitterAccessToken accessToken) {
    // это нужно где-то сохранить
}

public IEnumerable<TwitterUserId> GetListOfFollowers(←
    TwitterUserId userId) {
    // непонятно, как это будет работать
}
}

public class TwitterUserId {
    // кто знает, как Twitter определяет идентификаторы пользователей
}

public class TwitterAccessToken {
    // непонятно, что это будет
}

public class TwitterCallbackInfo {
    // и это
}

```

Мы создали с нуля API для Twitter, хотя почти не знаем, как на самом деле работает API Twitter. Возможно, это не лучший API, но мы сами себе клиенты, поэтому можем позволить себе роскошь разработать его в соответствии со своими потребностями. К примеру, мне вряд ли потребуется работать с пакетной передачей данных и для меня не важны связанные с этим задержки и блокировки, но в более универсальном API это может быть учтено.

ПРИМЕЧАНИЕ Подход с использованием собственных удобных интерфейсов, выступающих в качестве адаптеров, на улицах ожидаемо называется шаблоном адаптера. Я стараюсь делать упор не на название инструмента, а на его прикладную пользу, но теперь вы знаете термин на случай, если кто-то заговорит на эту тему.

Интерфейс из классов, которые мы определили, можно извлечь позже, поэтому мы не зависим от конкретных реализаций, что упрощает тестирование. Мы даже не знаем, поддерживает ли библиотека Twitter, которую собираемся использовать, простую замену реализации. Иногда дизайн вашей мечты оказывается несовместим с реальным продуктом. Тогда придется приспособливать свой дизайн, но это хорошо, поскольку означает, что он отражает ваше понимание базовой технологии.

Так что, возможно, я слукавил. Не пишите библиотеку Twitter с нуля. Но не переставайте думать как изобретатель. Изобретать мысленно и изобретать в реальности — это почти одно и то же, и важно уметь делать и то и другое.

3.6. НЕ ИСПОЛЬЗУЙТЕ НАСЛЕДОВАНИЕ

Объектно-ориентированное программирование (ООП) обрушилось на мир в 1990-х годах как молот на наковальню, вызвав смещение парадигмы со структурного программирования. Случилась революция. Наконец-то была решена многолетняя проблема повторного использования кода.

Характерной чертой ООП явилось наследование. Повторное использование кода определялось как набор унаследованных зависимостей. Это позволило упростить также и модификацию кода. Чтобы создать новый код с несколько иным поведением, не нужно было менять исходный код. Вы просто наследовали его и переопределяли соответствующий член.

В долгосрочной перспективе наследование создало больше проблем, чем решило. И множественное наследование стало одной из первых. Что, если надо повторно использовать код из нескольких классов, а все они имеют метод с одинаковым именем и, возможно, одинаковой сигнатурой? Как это будет работать? А как на счет проблемы ромбовидной зависимости, показанной на рис. 3.9? Реализация множественного наследования очень сложна, поэтому эта возможность имеется в очень немногих языках программирования.

Еще более серьезной проблемой наследования является *сильная зависимость*, также известная как *сильная связанность*. Я уже говорил, что зависимости — это корень всех зол. По своей природе наследование привязывает вас к конкретной реализации, что нарушает один из общепризнанных принципов ООП — принцип инверсии зависимостей, согласно которому код должен зависеть не от конкретной реализации, а от абстракции.

На чем основан этот принцип? Когда вы привязаны к конкретной реализации, ваш код становится жестким. Как мы видели, жесткий код очень сложно тестировать или изменять.

В таком случае как использовать код повторно? Как наследовать класс от абстракции? Очень просто — с помощью композиции. Вместо наследования от класса необходимо получить его абстракцию в качестве параметра в конструкторе. Представьте, что компоненты системы — это кубики лего, которые поддерживают друг друга, а не образуют иерархию объектов.

При обычном наследовании связь между общим кодом и его вариациями выражается в модели «родитель — потомок». Композиция же рассматривает общую функцию как отдельный компонент.



Рис. 3.9. Проблема ромбовидной зависимости:
как должен вести себя класс D?

ПО ПРИНЦИПАМ SOLID

Известная аббревиатура SOLID обозначает пять принципов ООП. Проблема в том, что эта аббревиатура, похоже, была подогнана, чтобы составить значимое слово¹, а не чтобы улучшить разработку. Я считаю, что эти принципы имеют разную значимость, более того, какие-то из них вообще не важны. И я категорически против того, чтобы руководствоваться какими-либо принципами, не будучи убежденным в их ценности.

Принцип единственной ответственности (single-responsibility) утверждает, что класс должен отвечать только за одну задачу, в отличие от класса, выполняющего несколько задач, известного как божественный объект. Это утверждение довольно расплывчато, потому что именно мы решаем, что подразумевать под одной задачей. Можно ли сказать, что класс с двумя методами по-прежнему отвечает за одну задачу? Даже божественный объект на определенном уровне отвечает за одну задачу: быть божественным объектом. Я бы изменил название на «принцип ясного имени»: имя класса должно как можно более ясно выражать его назначение. Если имя слишком длинное или слишком общее, класс необходимо разделить.

Принцип открытости-закрытости (open-closed) гласит, что класс должен быть открыт для расширения, но закрыт для модификации. Это означает, что необходимо проектировать классы так, чтобы их поведение можно было изменять извне. Требование, опять же, очень расплывчатое, его выполнение может отнимать чересчур много времени. Расширяемость — это конструктивное решение, которое может быть нежелательным, непрактичным и даже небезопасным. Оно похоже на совет использовать гоночные шины. Я бы сформулировал этот принцип так: «Относитесь к расширяемости как к функции».

¹ Solid — прочный, надежный. — Примеч. пер.

Принцип подстановки Лисков (Liskov substitution), введенный Барбарой Лисков (Barbara Liskov), гласит, что поведение программы не должно меняться, если один из используемых классов заменяется своим производным классом. Хотя этот принцип разумен, я не думаю, что он важен для повседневной разработки. Для меня он похож на совет не совершать ошибок. Если вы нарушите интерфейсный контракт, в программе будут ошибки. Если вы спроектируете плохой интерфейс, в программе тоже будут ошибки. Это естественно. Возможно, этот принцип можно превратить в более простой и действенный, например «Придерживайтесь контракта».

Согласно *принципу разделения интерфейсов* (interface segregation) предпочтение отдается меньшим и специализированным интерфейсам, а не интерфейсам общего назначения с широкой областью действия. Это утверждение чересчур сложно и расплывчато, если не сказать ошибочно. В некоторых случаях интерфейсы общего назначения больше подходят для работы, а интерфейсы с высокой детализацией обходятся слишком дорого. Разделение интерфейсов должно основываться не на области их действия, а на требованиях проекта. Если единый интерфейс не подходит для работы, не бойтесь разделить его, невзирая на несоответствие принципу детализации.

Последний — *принцип инверсии зависимостей* (dependency inversion). Опять же, это не очень удачное название, я бы предложил «зависимость от абстракций». Да, зависимость от конкретных реализаций создает сильную связанность, и мы уже встречались с ее нежелательными эффектами. Но это не значит, что надо создавать интерфейсы для каждой зависимости. Я утверждаю обратное: используйте зависимость от абстракций и обеспечивайте гибкость, чтобы попадать в зависимость от конкретной реализации только тогда, когда это не имеет значения. Код должен адаптироваться к дизайну, а не наоборот. Смело экспериментируйте с разными моделями.

Композиция больше похожа на отношения «клиент — сервер», чем «родитель — потомок». Вы вызываете повторно используемый код по ссылке, а не наследуете его методы. Вы можете создать класс, от которого зависите, в конструкторе или, что еще лучше, получить его как параметр и использовать как внешнюю зависимость. Так вы сделаете эту связь более конфигурируемой и гибкой.

У получения класса в качестве параметра есть дополнительное преимущество: оно упрощает модульное тестирование объекта, когда в объект внедряются имитации конкретных реализаций. Я подробнее расскажу о внедрении зависимостей в главе 5.

Использование композиции вместо наследования может привести к увеличению объема кода, потому что вам может понадобиться определять зависимости

с помощью интерфейсов вместо конкретных ссылок, но при этом код освободится от зависимостей. В любом случае стоит взвесить все за и против, прежде чем использовать композицию.

3.7. НЕ ИСПОЛЬЗУЙТЕ КЛАССЫ

Классы — отличная штука. Они делают свою работу, а затем уходят в тень. Но как я уже говорил в главе 2, они связаны с накладными расходами на поддержку косвенных ссылок и имеют немного более косвенный характер по сравнению с типами значений. В большинстве случаев это неважно, но вам следует знать плюсы и минусы классов, чтобы понимать код и то, как ваши неправильные решения влияют на него.

Типы значений полезны. Примитивные типы, поставляемые с C#, такие как `int`, `long` и `double`, уже являются типами значений. Но можно создавать и свои собственные типы значений с помощью таких конструкций, как `enum` и `struct`.

3.7.1. `Enum` — это ням!

Тип `enum` (перечисление) отлично подходит для хранения дискретных порядковых значений. Классы также можно использовать для определения дискретных значений, но им не хватает возможностей, которые есть у `enum`. И конечно, класс лучше, чем жестко заданные значения.

Если вы пишете код, который обрабатывает ответ на веб-запрос из приложения, вам придется обрабатывать разные числовые коды ответа. Допустим, вы за-прашиваете информацию о погоде в Национальной метеорологической службе (NWS) для местоположения пользователя и пишете функцию, получающую эту информацию. В листинге 3.6 мы используем `RestSharp` для запросов API и пакет `Newtonsoft.Json` для парсинга ответа путем проверки кода состояния HTTP, если запрос успешен. Обратите внимание, что мы используем жестко заданное значение `200` в строке `if` для проверки кода состояния. Затем мы используем библиотеку `Json.NET` для парсинга ответа в динамический объект и извлечения необходимой информации.

Листинг 3.6. Функция, возвращающая прогноз погоды от NWS для заданного местоположения

```
static double? getTemperature(double latitude,
    double longitude) {
```

```

const string apiUrl = "https://api.weather.gov";
string coordinates = $"{latitude},{longitude}";
string requestPath = $"{points}/{coordinates}/forecast/hourly";
var client = new RestClient(apiUrl);
var request = new RestRequest(requestPath);
var response = client.Get(request); ← Отправляет запрос в NWS
if (response.StatusCode == 200) { ← Проверка наличия успешного кода состояния HTTP
    dynamic obj = JObject.Parse(response.Content); ← Анализируем JSON
    var period = obj.properties.periods[0];
    return (double)period.temperature; ← Готово!
}
return null;
}

```

Самая большая проблема с жестко закодированными значениями — это неспособность людей запоминать числа. Мы не умеем этого делать. Мы не воспринимаем числа с первого взгляда, кроме количества нулей в чеке. Числа труднее вводить, чем слова, потому что их трудно с чем-то ассоциировать, и в то же время в них легче сделать опечатку. Вторая проблема заключается в том, что значения могут меняться. Если вы везде используете одно и то же значение, то для его изменения придется менять все.

Еще одна проблема с числами заключается в том, что они плохо выражают логику. Числовое значение, например 200, может означать что угодно. Поэтому не используйте жестко закодированные значения.

Классы — один из способов инкапсулирования значений. Можно инкапсулировать коды состояния HTTP в такой класс:

```

class HttpStatusCode {
    public const int OK = 200;
    public const int NotFound = 404;
    public const int ServerError = 500;
    // ... и т. д.
}

```

Можно изменить строку, которая проверяет успешный HTTP-запрос, например, так:

```

if (response.StatusCode == HttpStatusCode.OK) {
...
}

```

Этот вариант более описателен. Мы сразу понимаем контекст, значение и значение в контексте. Это наглядно.

Тогда для чего нужны типы `enum`? Разве нельзя обойтись классами? Допустим, у нас есть еще один класс для хранения значений:

```
class ImageWidths {  
    public const int Small = 50;  
    public const int Medium = 100;  
    public const int Large = 200;  
}
```

После компиляции этот код вернет `true`:

```
return HttpStatusCode.OK == ImageWidths.Large;
```

Но скорее всего, вам это не требуется. Напишем этот код через `enum`:

```
enum HttpStatusCode {  
    OK = 200,  
    NotFound = 404,  
    ServerError = 500,  
}
```

Так намного проще, правда? Этот вариант решает ту же задачу. При этом каждый определяемый тип `enum` уникален, что делает значения типобезопасными, в отличие от примера с классами `const`. В нашем случае `enum` имеет большие преимущества. Если попробовать провести сравнение двух разных типов `enum`, компилятор выдаст ошибку:

```
error CS0019: Operator '==' cannot be applied to operands of type  
  '(HttpStatusCode' and 'ImageWidths'
```

Потрясающе! Перечисления экономят время и не позволяют сравнивать яблоки с апельсинами во время компиляции. Они передают намерения, а также классы, содержащие значения. При этом `enum` является типом значений, поэтому он так же быстр, как передача целочисленного значения.

3.7.2. Структуры рулят!

Как было показано в главе 2, использование классов не связано с большими накладными расходами на хранение данных. Каждый экземпляр класса при инстанцировании должен сохранять заголовок объекта и таблицу виртуальных методов. Кроме того, классы распределяются в куче и обрабатываются сборщиком мусора.

.NET отслеживает каждый инстанцированный класс, а когда он становится не нужен — извлекает его из памяти. Этот процесс организован очень эффективно — обычно вы его даже не замечаете. Похоже на волшебство, не требующее ручного управления памятью. Поэтому не бойтесь классов.

Но как мы видели, полезно знать, когда можно пользоваться бесплатными преимуществами. Структуры подобны классам. В них можно определить свойства, поля и методы. Структуры также могут реализовывать интерфейсы. Однако структура не может быть унаследована, а также не может наследовать от другой структуры или класса. Дело в том, что структуры не имеют таблицы виртуальных методов или заголовка объекта. Они не удаляются сборщиком мусора, поскольку размещаются в стеке вызовов.

Как я уже говорил в главе 2, стек вызовов — это просто непрерывный блок памяти, в котором меняется только положение его верхнего указателя. Это делает стек очень эффективным механизмом хранения с быстрой автоматической очисткой. Фрагментации не происходит, поскольку данные организованы по принципу LIFO (Last In First Out).

Если стек такой быстрый, почему бы не использовать его всегда? Зачем нужны куча и сборка мусора? Дело в том, что стек живет, только пока выполняется функция. Когда функция возвращается, стековый фрейм пустеет и то же пространство могут использовать другие функции. Куча нужна для объектов, срок жизни которых превышает жизненный цикл функции.

Кроме того, размер стека ограничен. Вот почему существует целый веб-сайт под названием Stack Overflow: потому что приложение рухнет, если стек переполнится. Уважайте стек, помните пределы его возможностей.

Структуры — это облегченные классы. Они размещаются в стеках, поскольку представляют собой типы значений. Следовательно, значение структуры присваивается переменной путем копирования ее содержимого, так как на него отсутствуют ссылки. Необходимо это учитывать, поскольку если размер данных превышает размер указателя, копирование выполняется медленнее, чем передача ссылок.

Структуры все же могут содержать ссылочные типы. Например, если структура содержит строку, она является ссылочным типом внутри типа значения, так же как типы значений могут находиться внутри ссылочного типа. Я покажу это на схемах ниже.

Структура, содержащая только целочисленное значение, обычно занимает меньше места, чем ссылка на класс, содержащий целочисленное значение, как

показано на рис. 3.10. Представим, что наши структура и класс хранят идентификаторы, как это было описано в главе 2. Две разновидности одной структуры будут выглядеть так, как показано в следующем листинге.

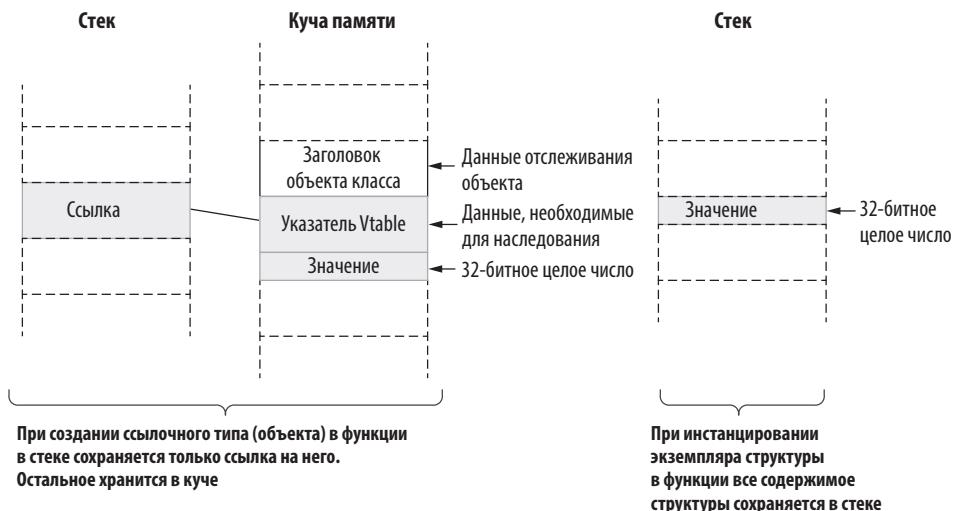


Рис. 3.10. Сравнение расположения классов и структур в памяти

Листинг 3.7. Сходство объявлений классов и структур

```
public class Id {
    public int Value { get; private set; }

    public Id (int value) {
        this.Value = value;
    }
}

public struct Id {
    public int Value { get; private set; }

    public Id (int value) {
        this.Value = value;
    }
}
```

Единственная разница между двумя вариантами кода заключается в ключевых словах **struct** и **class**, но обратите внимание на отличие способов сохранения, когда вы их создаете, например, в такой функции:

```
var a = new Id(123);
```

На рис. 3.10 показано, как они расположены в памяти.

Поскольку структуры являются типами значений, присвоение одной структуры другой подразумевает создание еще одной копии всего содержимого структуры вместо копирования ссылки:

```
var a = new Id(123);
var b = a;
```

На рис. 3.11 показано, почему структуры могут быть эффективны для хранения небольших типов.

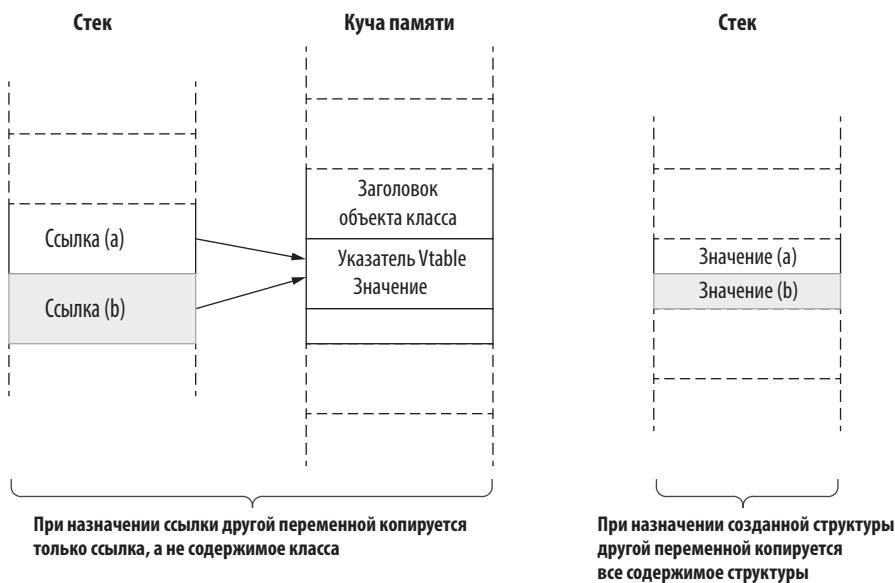


Рис. 3.11. Небольшие структуры эффективно используют память

Стековое хранилище создается только на время выполнения функции, и оно ничтожно мало по сравнению с кучей. Размер стека в .NET составляет 1 Мбайт, а куча может содержать терабайты данных. Стек обеспечивает высокую скорость, но может быстро заполниться при работе с большими структурами. Кроме того, копирование больших структур занимает больше времени, чем копирование ссылки. Предположим, что вместе с идентификаторами необходимо сохранить данные о пользователе. Реализация будет выглядеть следующим образом.

Листинг 3.8. Создание большего класса или структуры

```
public class Person {
    public int Id { get; private set; }
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public string City { get; private set; }

    public Person(int id, string firstName, string lastName,
        string city) {
        Id = id;
        FirstName = firstName;
        LastName = lastName;
        City = city;
    }
}
```

←
Здесь класс можно превратить в структуру,
изменив определение

Процессы создания класса и структуры будут различаться только ключевыми словами `struct` и `class`. Тем не менее создание и присваивание сущностей оказывает глубокое влияние на внутренние системные процессы. Рассмотрим простой код, где `Person` может быть либо структурой, либо классом:

```
var a = new Person(42, "Sedat", "Kapanoglu", "San Francisco");
var b = a;
```

Различие в схемах организации памяти после присвоения переменной `b` значения `a` показано на рис. 3.12.

Стек вызовов — очень эффективное хранилище. Он отлично подходит для работы с небольшими объемами данных при небольших накладных расходах, поскольку сборки мусора не происходит. Так как стеки не являются ссылочными типами, они не могут принимать нулевое значение, что делает невозможным обработку исключений нулевых ссылок для структур.

Структуры не универсальны, это очевидно из способа их хранения: невозможно использовать общую ссылку на них и, как следствие, невозможно изменить общий экземпляр с помощью нескольких ссылок. Это то, о чем мы не задумываемся. Допустим, что мы делаем структуру изменяемой и используем модификаторы `get; set;` вместо `get; private set;`. Таким образом можно изменять структуру на лету. Посмотрите на следующий пример.

Листинг 3.9. Изменяемая структура

```
public struct Person {
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
```

```

public string City { get; set; }

public Person(int id, string firstName, string lastName,
    string city) {
    Id = id;
    FirstName = firstName;
    LastName = lastName;
    City = city;
}
}

```

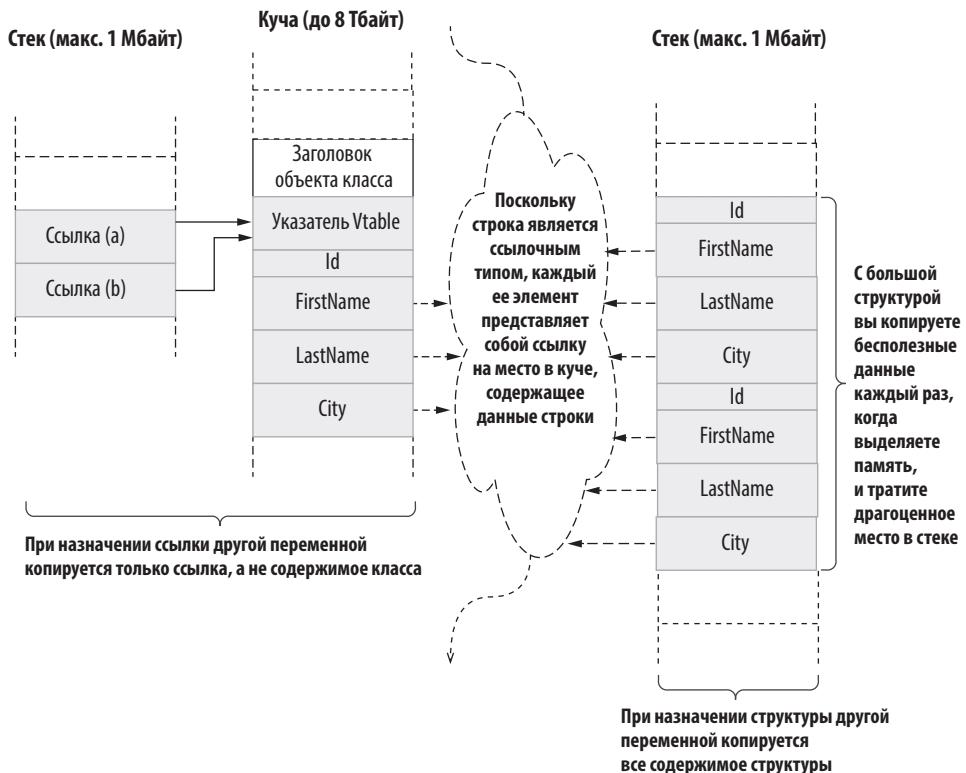


Рис. 3.12. Разница между типами значений и ссылочными типами при работе с объектами большого размера

Рассмотрим фрагмент кода с изменяемой структурой:

```

var a = new Person(42, "Sedat", "Kapanoglu", "San Francisco");
var b = a;

```

```
b.City = "Eskisehir";
Console.WriteLine(a.City);
Console.WriteLine(b.City);
```

Как вы думаете, что будет на выходе? Если бы это был класс, обе строки вывели бы «Эскишехир» (Eskisehir) как новый город. Но так как у нас есть два отдельных экземпляра, выводом будут «Сан-Франциско» и «Эскишехир». Поэтому полезно всегда делать структуры неизменяемыми, чтобы в них нельзя было случайно совершить ошибку.

Хотя для повторного использования кода композиция предпочтительнее наследования, последнее также может быть полезно при наличии зависимости. В этом случае классы обеспечат большую гибкость, чем структуры.

Классы более эффективны для хранения данных, если имеют большой размер, так как копируется только ссылка на класс. С учетом вышесказанного старайтесь использовать структуры для небольших неизменяемых типов данных, которые не нуждаются в наследовании.

3.8. ПИШИТЕ ПЛОХОЙ КОД

Лучшие практики рождаются из плохого кода, но плохой код может также стать результатом бездумного применения лучших практик. Структурное, объектно-ориентированное и даже функциональное программирование служит для того, чтобы разработчики писали более качественный код. При обучении лучшим практикам некоторые методы работы объявляются абсолютным злом и полностью отвергаются. Познакомимся с некоторыми из них поближе.

3.8.1. Не используйте `If/Else`

`If/Else` — одна из первых конструкций, которые изучает будущий программист. Она выражает одно из фундаментальных понятий программирования — логику. Мы любим `If/Else`. Эта конструкция позволяет представить логику программы в виде блок-схемы. Но она же делает код менее читаемым.

Как и многие конструкции программирования, блоки `If/Else` требуют добавления дополнительного отступа при наборе кода. Предположим, необходимо добавить в класс `Person` функциональность из последнего примера для обработки записи в БД. Мы хотим определить, было ли изменено свойство `City` класса `Person`, и изменить его в БД, если класс `Person` указывает на

действительную запись. Это не оптимальное решение, есть варианты и получше, но здесь я хочу показать, каким может получиться код, а не его реальную функциональность.

Листинг 3.10. Торжество If/Else

```
public UpdateResult UpdateCityIfChanged() {
    if (Id > 0) {
        bool isActive = db.IsPersonActive(Id);
        if (isActive) {
            if (FirstName != null && LastName != null) {
                string normalizedFirstName = FirstName.ToUpper();
                string normalizedLastName = LastName.ToUpper();
                string currentCity = db.GetCurrentCityByName(
                    normalizedFirstName, normalizedLastName);
                if (currentCity != City) {
                    bool success = db.UpdateCurrentCity(Id, City);
                    if (success) {
                        return UpdateResult.Success;
                    } else {
                        return UpdateResult.UpdateFailed;
                    }
                } else {
                    return UpdateResult.CityDidNotChange;
                }
            } else {
                return UpdateResult.InvalidName;
            }
        } else {
            return UpdateResult.PersonInactive;
        }
    } else {
        return UpdateResult.InvalidId;
    }
}
```

Даже если разобрать каждый шаг функции, невозможно не запутаться в ней снова, вернувшись через пять минут. Одна из причин путаницы — слишком много отступов. Люди не привыкли читать с отступами, кроме горстки пользователей Reddit. Трудно понять, к какому блоку относится строка, каков контекст. Трудно следовать логике.

Общее правило для избавления от ненужных отступов в коде — выйти из функции как можно раньше и не использовать `else`, если поток его уже подразумевает. В листинге 3.11 показано, как операторы `return` сигнализируют об окончании потока кода и устраниют необходимость в `else`.

Листинг 3.11. Никаких else!

```

public UpdateResult UpdateCityIfChanged() {
    if (Id <= 0) {
        return UpdateResult.InvalidId;
    }
    bool isActive = db.IsPersonActive(Id);
    if (!isActive) {
        return UpdateResult.PersonInactive;
    }
    if (FirstName is null || LastName is null) {
        return UpdateResult.InvalidName;
    }
    string normalizedFirstName = FirstName.ToUpper();
    string normalizedLastName = LastName.ToUpper();
    string currentCity = db.GetCurrentCityByName(
        normalizedFirstName, normalizedLastName);
    if (currentCity == City) {
        return UpdateResult.CityDidNotChange;
    }
    bool success = db.UpdateCurrentCity(Id, City);
    if (!success) {
        return UpdateResult.UpdateFailed;
    }
    return UpdateResult.Success;
}

```

Используемая здесь техника называется *счастливый путь* (*happy path*). Счастливый путь — это код, который выполняется, если нигде нет сбоев. Это идеальный сценарий выполнения. Поскольку счастливый путь в совокупности соответствует основной задаче функции, его должно быть легче всего читать. В коде, содержащем вызов `return` вместо операторов `else`, гораздо легче определить счастливый путь, чем в коде-матрешке из операторов `if`.

Проводите проверки быстро и возвращайте результат как можно раньше. Поместите исключения в операторы `if` и старайтесь вывести счастливый путь за пределы их блоков. Используйте эти два способа организации кода, чтобы обеспечивать его читаемость и удобство в обслуживании.

3.8.2. Используйте `goto`

Всю теорию программирования можно пересказать, используя память, базовую арифметику и операторы `if` и `goto`. Оператор `goto` переводит выполнение программы непосредственно в заданную точку. За ним трудно следить, и его не рекомендуют использовать после выхода статьи Эдсгера Дейкстры (Edsger

Dijkstra) «Go to statement is considered harmful» («Оператор `go to` вреден») (<https://dl.acm.org/doi/10.1145/362929.362947>). С этой статьей связано немало заблуждений, в первую очередь с ее названием. Дейкстра изначально назвал статью «A Case Against the Goto Statement» («Дело в отношении оператора `Goto`»), но его редактор, а также изобретатель языка Паскаль Никлаус Вирт (Niklaus Wirth) изменил название, что представило позицию Дейкстры более агрессивной и превратило разбор недостатков `goto` в охоту на ведьм.

Все это происходило до 1980-х годов. У языков программирования было достаточно времени для создания новых конструкций, реализующих функции оператора `goto`. Были созданы циклы `for/while`, операторы `return/break/continue` и даже исключения для конкретных сценариев, которые ранее реализовывались только с помощью `goto`. Программисты на Basic помнят известную инструкцию обработки ошибок `ON ERROR GOTO`, которая являлась примитивным механизмом обработки исключений.

Хотя во многих современных языках больше нет эквивалента `goto`, в C# он предусмотрен и отлично подходит для удаления лишних точек выхода в функциях. Оператор `goto` можно использовать, сделав код менее подверженным ошибкам и при этом сэкономив время, — как три связанных удара в Mortal Kombat.

Точка выхода — это оператор в функции, возвращающий ее к вызывающей стороне. В C# каждый оператор `return` является точкой выхода. В прежние времена удаление лишних точек выхода было важной процедурой, поскольку программисты тогда больше времени уделяли ручной очистке кода. Приходилось запоминать, какие ресурсы выделены и что нужно очистить, прежде чем возвращать функцию.

C# предоставляет отличные инструменты для очистки структуры, такие как блоки `try/finally` и операторы `using`. Однако для некоторых ситуаций ни один из них не подходит, и тогда для очистки следует использовать `goto`, хотя на самом деле он больше подходит для устранения избыточности.

Представим, что мы разрабатываем форму ввода адреса доставки для интернет-магазина. Веб-формы прекрасно подходят для демонстрации многоуровневых проверок. Допустим, мы остановили выбор на ASP.NET Core. Значит, в форме необходимо предусмотреть действие `submit` (отправить). Его код может выглядеть как показано в листинге 3.12. У нас есть проверка модели, которая выполняется на стороне клиента, и нам нужна еще проверка на сервере с формой через USPS API, чтобы убедиться, что адрес правильный. После проверки можно сохранить информацию в базу данных и, в случае успеха, перенаправить пользователя на страницу оплаты. В противном случае необходимо повторно вывести форму адреса доставки.

Листинг 3.12. Код обработки формы адреса доставки в ASP.NET Core

```
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
    if (!ModelState.IsValid) {
        return RedirectToAction("Index", "ShippingForm", form);
    }
    var validationResult = service.ValidateShippingForm(form);
    if (validationResult != ShippingFormValidationResult.Valid) {
        return RedirectToAction("Index", "ShippingForm", form);
    }
    bool success = service.SaveShippingInfo(form);
    if (!success) {
        ModelState.AddModelError("", "Problem occurred while " +
            "saving your information, please try again");
        return RedirectToAction("Index", "ShippingForm", form);
    }
    return RedirectToAction("Index", "BillingForm"); ←———— Частливый путь
}
```

Я уже затрагивал некоторые проблемы копирования и вставки, но множественные точки выхода в листинге 3.12 создают другую сложность. Вы заметили опечатку в третьем операторе `return`? Мы случайно удалили символ, не заметив этого, и, поскольку он находится в строке, эту ошибку почти невозможно обнаружить, пока не возникнет ошибка при сохранении формы на продакшен или пока мы не создадим сложные тесты контроллеров. Дублирование в этих случаях создает проблемы. Оператор `goto` может объединить операторы `return` под одной меткой, как показано в листинге 3.13. Мы создаем под счастливым путем новую метку для случая ошибки и с помощью `goto` переходим на нее из нескольких мест функции.

Листинг 3.13. Объединение общих точек выхода в один оператор `return`

```
[HttpPost]
public IActionResult Submit2(ShipmentAddress form) {
    if (!ModelState.IsValid) {
        goto Error;
    }
    var validationResult = service.ValidateShippingForm(form);
    if (validationResult != ShippingFormValidationResult.Valid) {
        goto Error;
    }
    bool success = service.SaveShippingInfo(form);
    if (!success) {
        ModelState.AddModelError("", "Problem occurred while " +
            "saving your shipment information, please try again");
        goto Error;
    }
    return RedirectToAction("Index", "BillingForm");
Error: ←———— Метка назначения
    return RedirectToAction("Index", "ShippingForm", form); ←———— Общий код выхода
}
```

Самое замечательное в этой консолидации то, что если вы когда-нибудь захотите добавить что-то в общий код выхода, то нужно будет сделать это только в одном месте. Допустим, вы хотите сохранить файл cookie для клиента при возникновении ошибки. Все, что вам нужно сделать, — это добавить его после метки `Error`, как показано ниже.

Листинг 3.14. Добавить новый код в общий код выхода просто

```
[HttpPost]
public IActionResult Submit3(ShipmentAddress form) {
    if (!ModelState.IsValid) {
        goto Error;
    }
    var validationResult = service.ValidateShippingForm(form);
    if (validationResult != ShippingFormValidationResult.Valid) {
        goto Error;
    }
    bool success = service.SaveShippingInfo(form);
    if (!success) {
        ModelState.AddModelError("", "Problem occurred while " +
            "saving your information, please try again");
        goto Error;
    }
    return RedirectToAction("Index", "BillingForm");
Error:
    Response.Cookies.Append("shipping_error", "1"); ← Код, сохраняющий cookie
    return RedirectToAction("Index", "ShippingForm", form);
}
```

Используя `goto`, мы сохранили код читаемым, сэкономили время и упростили внесение изменений в будущем, поскольку его достаточно исправить только в одном месте.

`goto` все равно может завести в тупик того, кто с ним не знаком. К счастью, в C# 7.0 появились локальные функции, которые можно использовать для выполнения работы `goto`, возможно, более простым для понимания способом. Мы объявляем локальную функцию с именем `error`, которая выполняет обычную операцию возврата ошибки и возвращает ее результат вместо использования `goto`. В следующем листинге показано, как это работает.

Листинг 3.15. Использование локальных функций вместо goto

```
[HttpPost]
public IActionResult Submit4(ShipmentAddress form) {
    IActionResult error() { ← Локальная функция
        Response.Cookies.Append("shipping_error", "1");
        return RedirectToAction("Index", "ShippingForm", form);
    }
```

```

if (!ModelState.IsValid) {
    return error(); ← Общие случаи возврата ошибок
}
var validationResult = service.ValidateShippingForm(form);
if (validationResult != ShippingFormValidationResult.Valid) {
    return error(); ← Общие случаи возврата ошибок
}
bool success = service.SaveShippingInfo(form);
if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
        "saving your information, please try again");
    return error();
}
return RedirectToAction("Index", "BillingForm");
}

```

Использование локальных функций также позволяет объявить обработку ошибок в верхней части функции, что является нормой для современных языков программирования, таких как Go, с операторами типа `defer`, хотя в нашем случае придется вызывать функцию `error()` для выполнения.

3.9. НЕ ПИШИТЕ КОММЕНТАРИИ К КОДУ

В XVI веке в Турции жил архитектор по имени Синан. Он построил знаменитую мечеть Сулеймание в Стамбуле и множество других зданий. С ним связана одна история: спустя сотни лет после смерти Синана группа архитекторов начала реставрацию одного из его зданий. Замковый камень в одной из арок необходимо было заменить. Реставраторы осторожно сняли каменный блок и обнаружили небольшой стеклянный сосуд, втиснутый между камнями. В нем была записка: «Этот камень простоял триста лет. Если вы читаете эти строки, значит, он разрушен или вы пытаетесь его заменить. Есть только один способ правильно установить новый камень». Дальше описывалось, как именно это сделать.

Архитектор Синан, возможно, первый человек в истории, который правильно использовал комментирование в коде. Рассмотрим противоположный случай, когда здание испещрено надписями. На каждой двери надпись «Это дверь», на каждом окне написано «Окно». Между кирпичами торчат сосуды с записками: «Это кирпичи».

Не нужно писать комментарии к коду, если код и так понятен. Избыточные комментарии могут навредить и затруднить понимание. Не пишите комментарии только ради комментариев. Используйте их с умом и только в случае необходимости.

Рассмотрим пример в следующем листинге. Если переборщить с комментариями, код будет выглядеть так.

Листинг 3.16. Комментарии к коду повсюду

```

/// <summary>
/// Получить модель адреса доставки и обновить ее в
/// базе данных, а затем перенаправить пользователя на страницу оплаты
/// в случае успеха. ← Это понятно из контекста и объявления функции
/// </summary>
/// <param name="form">Модель для получения.</param> ←
/// <returns>Перенаправить результат к форме ввода в случае
/// ошибки или на
/// страницу оплаты в случае успеха.</returns>
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
    // Общий код обработки ошибок, который сохраняет файл cookie
    // и перенаправляет обратно к форме ввода
    // информации о доставке.
    IActionResult error() {
        Response.Cookies.Append("shipping_error", "1");
        return RedirectToAction("Index", "ShippingForm", form); ←
    }
    // проверить, допустимо ли состояние модели | Вновь совершенно
    if (!ModelState.IsValid) { | ненужная информация
        return error(); | Еще один
    }
    // проверить форму, используя логику проверки на стороне сервера | повтор
    var validationResult = service.ValidateShippingForm(form); | Да ладно!
    // проверка прошла успешно?
    if (validationResult != ShippingFormValidationResult.Valid) { | Серьезно?
        return error(); | Мы пришли к этому?
    }
    // сохранить информацию о доставке | Без шуток,
    bool success = service.SaveShippingInfo(form); | Шерлок
    if (!success) {
        // не удалось сохранить. Сообщить об ошибке пользователю
        ModelState.AddModelError("", "Возникла проблема при" +
            "сохранении данных, попробуйте еще раз");
        return error(); | Никогда бы
    }
    // перейти к форме оплаты | не подумал
    return RedirectToAction("Index", "BillingForm");
}

```

Если мы умеем читать код, то он может быть понятен без комментариев. Попробуем найти скрытые подсказки (рис. 3.13).

Может показаться, что это трудно и долго, так как нужно собрать все части воедино, просто чтобы понять, что делает код. Но со временем это будет получаться

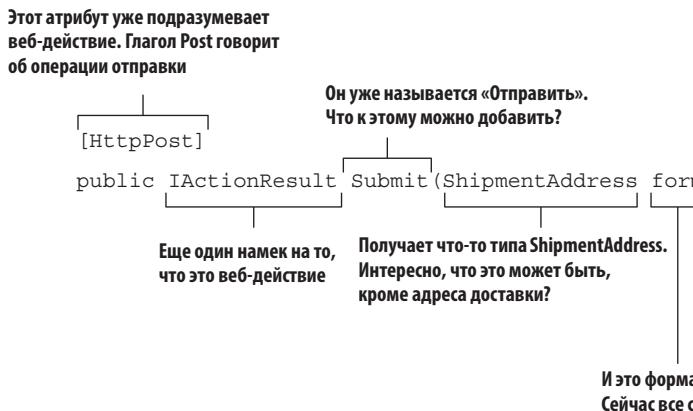


Рис. 3.13. Чтение подсказок в коде

лучше, и вы будете тратить все меньше усилий. Вы можете упростить жизнь тому, кто будет читать ваш код, и даже себе спустя какое-то время, потому что через полгода это может быть уже чужой код.

3.9.1. Подбирайте длинные имена

В начале этой главы я говорил о важности выбора имен, о том, что имя функции должно как можно точнее выражать ее назначение. Имена не должны быть двусмысленными, например `Process`, `DoWork`, `Make` и т. д., если контекст неясен. Иногда приходится создавать более длинные имена, но обычно хорошие имена удается сохранять краткими.

То же справедливо и для имен переменных. Зарезервируйте однобуквенные имена для переменных цикла (*i*, *j* и *n*) и координат (*x*, *y* и *z*), поскольку они очевидны. Выбирайте описательное имя и избегайте сокращений. Можно использовать общеизвестные аббревиатуры, такие как `HTTP` и `JSON`, `ID` и `DB`, но не произвольные сокращения слов. В любом случае вы вводите имя переменной полностью только один раз, так как в дальнейшем за вас это будет делать функция автозавершения ввода.

У описательных имен масса преимуществ. Самое главное — экономия времени. Описательное имя не нужно объяснять в комментариях. Изучите документацию о соглашениях в используемом языке. Например, соглашения Microsoft об именовании для .NET — отличная отправная точка для C#: <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines>.

3.9.2. Эффективно используйте функции

Небольшие функции проще понять. Страйтесь, чтобы функция помещалась на экране разработчика. Прокрутка затрудняет понимание работы кода. Все, что делает функция, должно быть перед глазами.

Как сократить функцию? Новички стараются для этого уместить как можно больше операторов в одну строку. Не делайте этого! Никогда не помещайте в одну строку несколько операторов. Предусмотрите для каждого оператора хотя бы одну отдельную строку. Можно даже добавить пустые строки, чтобы сгруппировать соответствующие операторы. Посмотрим на уже знакомую нам функцию в следующем листинге.

Листинг 3.17. Использование пустых строк для разделения логических частей функции

```
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
    IActionResult error() {
        Response.Cookies.Append("shipping_error", "1");
        return RedirectToAction("Index", "ShippingForm", form);
    }
    if (!ModelState.IsValid) {
        return error();
    }
    var validationResult = service.ValidateShippingForm(form);
    if (validationResult != ShippingFormValidationResult.Valid) {
        return error();
    }
    bool success = service.SaveShippingInfo(form);
    if (!success) {
        ModelState.AddModelError("", " Возникла проблема при " +
            " сохранении данных, попробуйте еще раз ");
        return error();
    }
    return RedirectToAction("Index", "BillingForm");
}
```

Часть кода, обрабатывающая ошибки

Часть, проверяющая модель MVC (модель — представление — контроллер)

Часть, проверяющая модель на стороне сервера

Часть кода, отвечающая за сохранение и обработку успешного результата

Вы спросите, как это помогает уменьшить размер записи функции? Действительно, дополнительные пустые строки делают ее больше. Но разделение на логические части упрощает понимание алгоритмов, а это основное условие создания небольших функций и описательного кода. Если логика все еще сложна для понимания, можно продолжить преобразовывать этот код. В листинге 3.18 мы определили и извлекли части логики в функции `Submit`. Мы получили блоки проверки, сохранения, обработки ошибок сохранения и сообщения об успехе. В теле функции мы оставляем только эти четыре части.

Листинг 3.18. Сохранение только описательной функциональности

```

[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
    if (!validate(form)) { ← Проверка
        return shippingFormError();
    }
    bool success = service.SaveShippingInfo(form); ← Сохранение
    if (!success) { ← Обработка ошибок
        reportSaveError();
        return shippingFormError();
    }
    return RedirectToAction("Index", "BillingForm"); ← Успешный ответ
}

private bool validate(ShipmentAddress form) {
    if (!ModelState.IsValid) {
        return false;
    }
    var validationResult = service.ValidateShippingForm(form);
    return validationResult == ShippingFormValidationResult.Valid;
}

private IActionResult shippingFormError() {
    Response.Cookies.Append("shipping_error", "1");
    return RedirectToAction("Index", "ShippingForm", form);
}
private void reportSaveError() {
    ModelState.AddModelError("", " Возникла проблема при " +
        " сохранении данных, попробуйте еще раз ");
}

```

Теперь функция настолько проста, что читается почти как фраза на родном языке — ну, может быть, как на смеси родного и турецкого, но все равно просто. Мы получили хороший описательный код, не добавив ни одного комментария. Это именно то, что требуется, если вы начинаете сомневаться, не слишком ли сложен код. И писать его быстрее, чем абзацы комментариев. Кроме того, вы скажете себе спасибо, пожав левую руку правой, когда поймете, что вам не придется синхронизировать код и комментарии, чтобы последние оставались полезными на протяжении всего жизненного цикла проекта.

Извлечение функций может показаться нудным делом, но на самом деле это очень просто в таких средах, как Visual Studio. Вы просто выбираете часть кода, которую хотите извлечь, и нажимаете **Ctrl-.** (**Ctrl** и точку) или щелкаете на значке лампочки рядом с кодом и выбираете **Extract Method** (извлечь метод). Все, что вам нужно сделать, — присвоить методу имя.

Извлеченные фрагменты можно повторно использовать в этом же файле, что сэкономит время при написании формы счета на оплату, если семантика обработки ошибок одна и та же.

Может показаться, что я противник комментирования в коде. Как раз наоборот. В отсутствие ненужных комментариев полезные начинают сверкать как бриллианты. Убрать лишнее — единственный способ сделать комментарии полезными. Думайте как Синан, когда пишете комментарии: «Понадобится ли кому-то это объяснение?». Если объяснение необходимо, сделайте его максимально ясным и подробным, даже нарисуйте схему в ASCII, если нужно. Пишите столько, сколько нужно, чтобы другим разработчикам не приходилось подходить к вам и спрашивать, что делает этот фрагмент, или чтобы ваши коллеги не вносили неверные исправления в код, потому что вы забыли объяснить, как он работает. При возникновении ошибок все будут рассчитывать на вас. Вы так же отвечаете за его исправление, как и все остальные.

Иногда писать комментарии нужно независимо от того, полезны они или нет, например, для общедоступных API, потому что у пользователей может не быть доступа к коду. Но это не значит, что код станет более понятным только потому, что в нем есть комментарии. Его по-прежнему нужно писать чисто, небольшими, легко читаемыми фрагментами.

ИТОГИ

- Избегайте жесткого кодирования, а для этого не нарушайте границы логических зависимостей.
- Не бойтесь начинать работу с нуля, потому что в следующий раз вы сделаете ее намного быстрее.
- Переделывайте код, если в нем есть зависимости, которые в будущем могут связать вам руки.
- Не копайте себе яму, придерживаясь устаревших версий, регулярно обновляйте код и устраняйте в нем ошибки.
- Повторяйте код, а не используйте его повторно, чтобы не нарушать логические связи.
- Изобретайте умные абстракции, чтобы экономить время в будущем. Считайте такие абстракции инвестициями.
- Не позволяйте внешним библиотекам, которые вы используете, диктовать вам условия.
- Используйте композицию вместо наследования, чтобы избежать привязки кода к определенной иерархии.

- Страйтесь придерживаться такого стиля написания, чтобы код легко читался сверху вниз.
- Быстро завершайте функции и страйтесь не использовать `else`.
- Используйте `goto` или, что еще лучше, локальную функцию, чтобы общий код располагался в одном месте.
- Избегайте легкомысленных, избыточных комментариев, из-за которых за лесом не видно деревьев.
- Пишите код, который описывает себя сам, продуманно подбирайте имена для переменных и функций.
- Делите функции на простые для восприятия подфункции, чтобы код был как можно более описательным.
- Пишите комментарии к коду, когда это приносит пользу.

Распробуйте тестирование

В этой главе

- ✓ Почему мы ненавидим тестирование и как нам его полюбить
- ✓ Как сделать процесс тестирования более приятным
- ✓ Избегаем TDD, BDD и других трехбуквенных сокращений
- ✓ Как определить, что тестировать
- ✓ Тестируем, чтобы меньше работать
- ✓ Тестирование в радость

Для многих разработчиков тестирование сродни писательству: оно утомительно и редко окупается. Считается, что тестирование — занятие второго сорта по сравнению с написанием кода, потому что тестировщики не делают *реальной работы*.

Причина неприязни к тестированию заключается в том, что разработчики не видят его связи с созданием программных продуктов. С точки зрения программиста, создание программы — это написание кода, а с точки зрения менеджера — выбор верного направления для команды. Тестировщик, в свою очередь, подходит к проекту с точки зрения качества продукта. Мы рассматриваем тестирование

как внешний процесс, потому что не считаем его частью разработки и стараемся избежать участия в нем.

Однако тестирование — неотъемлемая часть работы программиста, и оно помогает в создании продукта как никакой другой инструмент, обеспечивая уверенность в коде. Оно сэкономит ваше время, и вам даже не придется себя за это ненавидеть. Посмотрим, как это работает.

4.1. ТИПЫ ТЕСТОВ

Тестирование призвано повысить уверенность в поведении программы. Важно понимать, что тесты не гарантируют определенное поведение, но они на порядки повышают его вероятность. Существуют разные классификации тестов, но главный отличительный принцип — то, как запускается или реализуется тест, потому что это больше всего влияет на затраты времени.

4.1.1. Ручное тестирование

Тестирование вручную обычно проводят разработчики, запуская свой код и проверяя его поведение. Существуют разновидности ручного тестирования, например сквозное тестирование, когда каждый поддерживаемый сценарий в программе тестируется от начала до конца. Сквозное тестирование очень информативно, но отнимает много времени.

Код-ревью (рецензирование кода) можно считать одним из видов тестирования, хоть и поверхностным. Запустив выполнение кода до определенной стадии, можно понять, что он делает и что будет делать. Можно примерно представить, насколько он будет отвечать требованиям, но нельзя сказать наверняка. Тесты, в зависимости от своего типа, обеспечивают разные уровни уверенности в работе кода. В этом смысле код-ревью — тоже тестирование.

4.1.2. Автоматизированное тестирование

Вы программист и умеете писать код. Это означает, что вы можете заставить компьютер делать что-то за вас, в том числе проводить тестирование. Вы можете написать код, который проверит ваш код, так что не нужно будет делать это самому. Обычно программисты все усилия бросают на создание инструментов для разрабатываемого продукта, а не для самой разработки, но организация разработки не менее важна.

ЧТО ТАКОЕ КОД-РЕВЬЮ?

Основная задача код-ревью — просмотреть код до его размещения в репозитории и найти возможные ошибки. Это можно сделать на бумаге или онлайн, например на GitHub. К сожалению, за годы на этом ресурсе накопилось множество мусора, начиная от ритуалов посвящения, полностью разрушающих самооценку разработчика, и заканчивая нагромождением неуместных цитат, надерганных архитекторами из прочитанных ими статей.

Самое важное в код-ревью то, что это последний этап, когда можно критиковать код без необходимости исправлять его самостоятельно. После ревью код принадлежит всем, потому что все его одобрили. Когда кто-то замечает, что ваш код сортировки с производительностью $O(N^2)$ ужасен, а затем снова надевает наушники, вы всегда можете ответить: «Жаль, что ты не заметил этого при ревью, Марк». Шучу, вам должно быть стыдно писать код сортировки с $O(N^2)$, особенно после прочтения этой книги, но все равно не стесняйтесь обвинить во всем Марка! Общайтесь с коллегами, это может быть полезно.

В идеале в ходе ревью не нужно проверять стиль или форматирование кода, потому что это делают автоматические инструменты — *линтеры*, или инструменты статического анализа кода. Следует обращать внимание на ошибки и технический долг, остающийся другим разработчикам. Код-ревью сродни асинхронному парному программированию — это экономичный способ держать всех в курсе и выявлять возможные проблемы коллективно.

Автоматизированные тесты сильно отличаются по своему охвату и, что более важно, по тому, насколько они повышают уверенность в поведении программы. Самая маленькая единица такого тестирования — модульный тест. Модульные тесты проще всего писать, потому что они тестируют только одну единицу кода — общедоступную функцию. Такой тест также должен быть общедоступным, поскольку проверяет внешние видимые интерфейсы, а не внутренние подробности класса. Определения модуля в литературе различны, так называют классы, блоки и другие логические типы организации, но мне кажется, что удобнее всего принять в качестве модуля функцию.

Проблема модульных тестов состоит в том, что они позволяют проверить, нормально ли работают модули, но не могут гарантировать, что модули будут так же работать *вместе*. Следовательно, дополнительно необходимо проверять, со-вместимы ли модули между собой. Такие тесты называются *интеграционными*. Автотесты пользовательского интерфейса считаются интеграционными, если запускают код готового продукта для создания корректного пользовательского интерфейса.

4.1.3. Опасная жизнь: тестирование в рабочей среде

Однажды я купил постер с известным мемом «Я не всегда тестирую код, но если делаю это, то на *продакшен*» и повесил у одного из наших разработчиков прямо за монитором, чтобы он всегда помнил, что так делать нельзя.

ОПРЕДЕЛЕНИЕ На сленге разработчиков термин *продакшен* (production) означает живую среду реальных пользователей, в которой любое изменение влияет на фактические данные. Многие разработчики путают ее со своим компьютером, то есть *средой разработки* (development). Однако код, работающий локально на компьютере, не наносит ущерба среде продакшена. В качестве меры предосторожности иногда используется удаленная среда, симулирующая рабочую. Ее называют *подготовительной* (staging), и она не влияет на фактические данные, которые видны пользователям вашего сайта.

Тестирование на продакшен, также известное как тестирование кода в рабочей среде, считается плохой практикой; неудивительно, что ей посвятили мем. Причина в том, что к тому времени, когда вы обнаружите сбой, возможно, вы уже потеряете пользователей или клиентов. Что еще более важно, когда вы прерываете выпуск рабочей версии, есть риск нарушить рабочий процесс всей команды. Вы легко это поймете по разочарованным взглядам и вскинутым бровям коллег по офису, а также сообщениям типа: «WTF!!!!???. Количество уведомлений в Slack будет расти, как цифры на спидометре KITT¹, вместе со скоростью пара из ушей босса.

Как и любая плохая практика, тестирование в рабочей среде не всегда плохо. Оно может сойти вам с рук, если ваш сценарий не является частью часто используемого критического пути кода. В Facebook ходила мантра *Move fast and break things* (не тормози и ломай), поскольку их разработчики могли оценивать влияние производимых ими изменений на бизнес. Компания отказалась от этого лозунга после выборов в США в 2016 году, но в нем все еще есть смысл. Если сбой небольшой и произошел в редко используемой функции, его последствия вполне терпимы при условии быстрого исправления.

Можно даже обойтись без тестов, если вы уверены, что сбои не спровоцируют бегство пользователей. Мне удалось в одиночку запустить один из самых

¹ KITT, Knight Industries Two Thousand — беспилотный автомобиль, оснащенный системой распознавания голоса, герой научно-фантастического сериала 1980-х годов «Рыцарь дорог». Нормально, что вы не поняли эту ссылку, так как все, кто ее понимал, скорее всего, уже умерли, кроме, пожалуй, Дэвида Хассельhoffа. Этот парень бессмертен.

популярных веб-сайтов в Турции вообще без автотестов. Да, он работал с ошибками и длительными простоями... потому что не было автотестов!

4.1.4. Выбор правильной методологии тестирования

Решая, как тестировать сценарий, который вы пытаетесь реализовать или изменить, необходимо учесть несколько факторов. В основном это риск и стоимость. Это похоже на наши прикидки, когда родители поручали работу по дому:

- Стоимость
 - Сколько времени нужно потратить на реализацию/запуск определенного теста?
 - Сколько раз придется его повторить?
 - Если тестируемый код изменится, кто узнает, что его нужно тестировать?
 - Насколько сложно обеспечить надежность теста?
- Риск
 - Какова вероятность сбоя в этом сценарии?
 - Если возникнет сбой, насколько сильно это повлияет на бизнес? Иначе говоря: «Сколько денег будет потеряно и не уволят ли меня?»
 - Если в сценарии возникнет сбой, сколько других сценариев это затронет? Например, если перестанет работать функция отправки и получения писем, это повлияет на работу многих зависящих от нее функций.
 - Как часто меняется код и насколько он изменится в будущем? Каждое изменение увеличивает риски.

Вам нужно найти золотую середину с разумными затратами и низкими рисками. Каждый риск — следствие высоких затрат. Со временем вы составите свою мысленную карту компромиссов с оценками стоимости тестов и их рисков, подобную приведенной на рис. 4.1.

Никогда не произносите: «На моем компьютере это работает». Эта фраза должна оставаться исключительно в ваших мыслях. Нет и не будет такого кода, о котором можно было бы сказать: «На моем компьютере он не работает, но я в нем уверен!». Конечно, он должен работать на вашем компьютере! Можете ли вы представить развертывание продукта, который вы не в состоянии сами запустить? Вы можете использовать это как мантру, когда размышляете, следует ли тестировать функцию, если отсутствует цепочка ответственности. Если никто не заставляет

вас отвечать за ваши ошибки, дерзайте. Это значит, что (раздутый) бюджет компании позволяет вашему начальству терпеть эти ошибки.

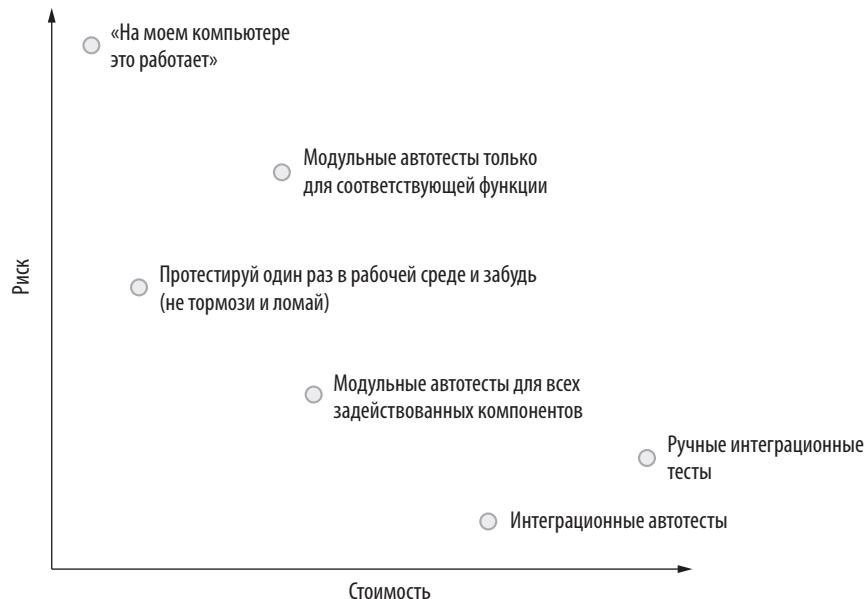


Рис. 4.1. Пример мысленной модели для оценки различных стратегий тестирования

Однако если нужно исправлять собственные ошибки, установка «на моем компьютере это работает» приводит к снижению скорости и потерям времени из-за задержки между развертыванием и циклами обратной связи. Одна из основных причин низкой продуктивности разработчиков заключается в негативном влиянии перерывов. Все дело в психологии. Я уже рассказывал, как разогрев перед серьезной задачей запускает рост продуктивности. Это психологическое состояние иногда называют *потоком* (zone). Вы в потоке, когда ваш ум начинает работать продуктивно. Перерыв выбрасывает вас из этого состояния, поэтому придется снова разминаться. Как показано на рис. 4.2, автотесты решают эту проблему, удерживая вас в потоке, пока вы не достигнете определенной уверенности в результате. Рисунок показывает, как дорого может обойтись принцип «на моем компьютере это работает» как бизнесу, так и разработчику. Каждый раз, когда вы выходите из состояния потока, требуется время, чтобы вернуться в него вновь. Иногда быстрее будет протестировать функцию вручную.

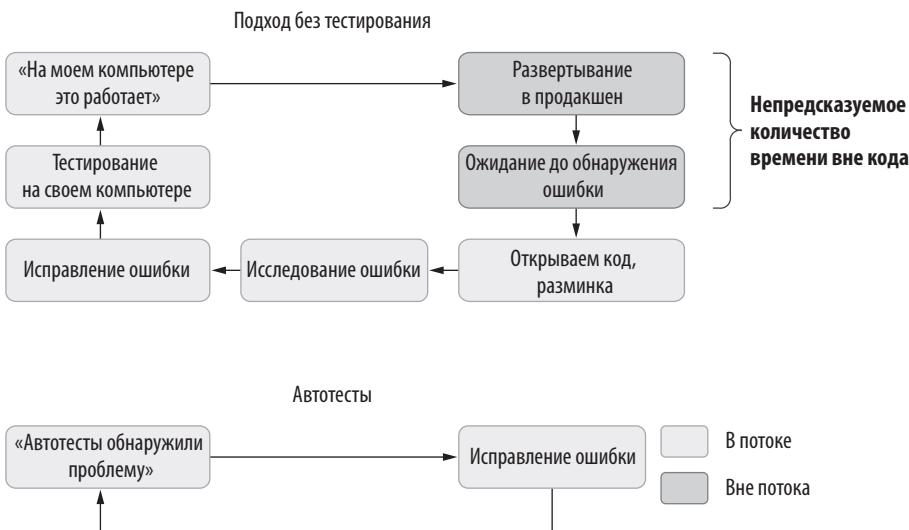


Рис. 4.2. Цикл разработки в парадигме «на моем компьютере это работает» по сравнению с автотестами

При ручных тестах можно добиться скорости итераций, аналогичной автотестам, но все равно они отнимают больше времени. Автотесты держат вас в потоке и требуют минимум времени. Можно возразить, что написание и выполнение тестов нарушает состояние потока. Тем не менее модульные тесты запускаются очень быстро и делятся считанные секунды. При разработке тестов вы действительно отвлекаетесь, но все же продолжаете думать о написанном коде. Этот процесс даже можно считать неким подведением итогов. Настоящая глава в основном посвящена обзору модульного тестирования, потому что оно оптимально с точки зрения соотношения стоимости и рисков (см. рис. 4.1).

4.2. КАК ПЕРЕСТАТЬ БЕСПОКОИТЬСЯ И ПОЛЮБИТЬ ТЕСТЫ

Модульное тестирование — это создание тестового кода, который проверяет одну условную единицу кода продукта, обычно функцию. Разработчики спорят, что считать такой единицей. По сути, это неважно, если ее можно тестировать изолированно. В любом случае невозможно протестировать весь класс одним тестом. Каждый тест проверяет только один сценарий для функции. Таким образом, обычно даже для одной функции пишут несколько тестов.

Тестовые фреймворки максимально упрощают процесс написания тестов, но использовать их не обязательно. Набор тестов может представлять собой отдельную программу, которая запускает тесты и выводит результаты. До появления тестовых фреймворков этот способ тестирования продукта был единственным. Я покажу вам простой фрагмент кода и то, как модульные тесты эволюционировали с течением времени, чтобы вам было проще писать тесты для заданной функции.

Представьте, что вам поручили изменить способ отображения дат публикаций на сайте микроблогов под названием Blabber. Раньше полностью отображались даты публикации, но после анализа трендов в социальных сетях было решено использовать сокращения, показывающие, сколько времени прошло с момента размещения публикации в секундах, минутах и часах. Вам нужно создать функцию, которая получает `DateTimeOffset` и преобразует его в строку, показывающую временной промежуток, например: «3 ч» для трех часов, «2 мин» для двух минут или «1 с» для одной секунды. Показывать необходимо только самую значимую единицу. Если сообщение размещено уже три часа, две минуты и одну секунду, выводиться должно только «3 ч».

Такая функция показана в листинге 4.1. Мы определяем *метод расширения* для класса `DateTimeOffset` в .NET, поэтому можем вызывать его где угодно как собственный метод `DateTimeOffset`.

Мы вычисляем интервал между текущим временем и временем публикации и проверяем поля, чтобы определить самую значимую единицу времени и вернуть соответствующий результат.

Листинг 4.1. Функция, преобразующая дату в строковое представление интервала

```
public static class DateTimeExtensions {
    public static string ToIntervalString(
        this DateTimeOffset postTime) { ← Определяет метод расширения
        TimeSpan interval = DateTimeOffset.Now - postTime; ← Определяет интервал
        if (interval.TotalHours >= 1.0) {
            return $"{(int)interval.TotalHours}ч";
        }
        if (interval.TotalMinutes >= 1.0) {
            return $"{(int)interval.TotalMinutes}мин";
        }
        if (interval.TotalSeconds >= 1.0) {
            return $"{(int)interval.TotalSeconds}с";
        }
        return "сейчас";
    }
}
```

← для класса `DateTimeOffset`

← Этот код можно написать короче или обеспечив лучшую производительность, но не в ущерб удобочитаемости

НЕ ЗАСОРЯЙТЕ АВТОЗАВЕРШЕНИЕ КОДА МЕТОДАМИ РАСШИРЕНИЯ

C# предоставляет удобный синтаксис для определения дополнительных методов типа, даже если источник типа недоступен. Если добавить перед первым параметром функции ключевое слово `this`, он появится в списке методов типа при автозавершении кода. Это настолько удобно, что разработчики очень любят вызывать методы расширения вместо статических. Допустим, есть простой метод:

```
static class SumHelper {
    static int Sum(int a, int b) => a + b;
}
```

Чтобы вызвать его, нужно набрать `SumHelper.Sum(amount, rate)`, но, что важнее, надо знать, что существует класс `SumHelper`. Вместо этого можно представить его как метод расширения:

```
static class SumHelper {
    static decimal Sum(this int a, int b) => a + b;
}
```

Теперь можно вызывать его, например, следующим образом:

```
int result = 5.Sum(10);
```

Выглядит хорошо, но есть проблема. Каждый метод расширения, в том числе для известного класса, такого как `string` или `int`, добавляется в автозавершение кода. Последнее представлено в Visual Studio как раскрывающийся список, который появляется при вводе точки после идентификатора. Поиск нужного метода в длинном списке совершенно не относящихся к делу методов сильно раздражает.

Не вводите специализированные методы в часто используемый класс .NET. Используйте эту возможность только для универсальных методов, которые действительно будут востребованы. Например, в класс `string` целесообразно ввести метод `Reverse`, а `MakeCdnFilename` — нет. `Reverse` применим в любом контексте, а `MakeCdnFilename` потребуется, только если необходимо изменить имя файла, чтобы оно подходило для используемой сети доставки контента. Кроме того, засорение автозавершения кода неудобно всей вашей команде. Не заставляйте других (а особенно себя) вас ненавидеть. Можно использовать статический класс и синтаксис вроде `Cdn.MakeFilename()`.

Не создавайте метод расширения, если метод можно сделать частью класса. Методы расширения имеют смысл, только когда необходимо внедрить новую функциональность за пределами границ зависимости. Например, если у вас есть веб-проект, использующий класс, определенный в библиотеке, которая не зависит от веб-компонентов. Позже вам, скорее всего, понадобится добавить в этот класс определенную функциональность, связанную с веб-функциями в веб-проекте. Лучше ввести новую зависимость только в метод расширения в веб-проекте, чем делать библиотеку зависимой от веб-компонентов. Необязательные зависимости могут связать вам руки.

У нас есть общая спецификация функции, и можно начинать писать для нее тесты. Удобно записать вместе возможные входные данные и ожидаемые выходные данные, как в табл. 4.1, чтобы убедиться, что функция работает правильно.

Таблица 4.1. Пример спецификации теста для функции преобразования

Input (Ввод)	Output (Выход)
< 1 секунды	“Сейчас”
< 1 минуты	“<seconds> с”
< 1 часа	“<minutes> мин”
=> 1 часа	“<hours> ч”

Если `DateTimeOffset` является классом, необходимо также проверить сценарий, когда мы передаем значение `null`, но поскольку это структура, она не может быть нулем. Это спасло нас от одного теста. Обычно составлять подобную таблицу нет необходимости, достаточно мысленно представить ее, но если вы в чем-то сомневаетесь, лучше составьте.

Тесты должны включать вызовы с разными значениями `DateTimeOffset` и сравнения с разными строками. На этом этапе необходимо позаботиться о надежности тестов, поскольку `Date-Time.Now` всегда меняется и нет гарантии, что тесты будут выполняться в определенное время. Если одновременно выполнялся другой тест или что-то замедляло работу компьютера, тест вывода может быть провален. Следовательно, наши тесты ненадежны и могут время от времени давать сбои.

Это указывает на проблему с дизайном. Простым решением было бы сделать функцию детерминированной, передав `TimeSpan` вместо `DateTimeOffset` и вычислив разницу в вызывающем объекте. Как видите, написание тестов для кода помогает выявлять также и проблемы проектирования, что является одним из преимуществ подхода TDD (test-driven development) — разработки через тестирование. Мы не используем TDD в этом примере, так как знаем, что можно просто изменить функцию, как в следующем листинге, и получить `TimeSpan`.

Листинг 4.2. Усовершенствованный дизайн

```
public static string ToIntervalString(
    this TimeSpan interval) { ← Теперь получаем TimeSpan
    if (interval.TotalHours >= 1.0) {
        return $"{(int)interval.TotalHours} ч";
    }
}
```

```

if (interval.TotalMinutes >= 1.0) {
    return $"{(int)interval.TotalMinutes} мин";
}
if (interval.TotalSeconds >= 1.0) {
    return $"{(int)interval.TotalSeconds} с";
}
return "сейчас";
}

```

Тестовые примеры не изменились, но сами тесты стали намного надежнее. Что еще более важно, мы разделили две разные задачи: вычисление разницы между двумя датами и преобразование интервала в строковое представление. Разделение ответственности в коде улучшает его дизайн. Вычисление разницы также может быть затратно, и для этого можно ввести отдельную функцию-обертку.

Как теперь убедиться, что функция работает? Можно просто запустить ее в профильщик и подождать пару минут, прислушиваясь, не раздаются ли крики. Если все спокойно, можно продолжать. Кстати, вы давно обновляли свое резюме? Я спросил просто из любопытства.

Мы вполне можем написать программу, которая тестирует функцию и выводит результаты. Ее пример показан в листинге 4.3. Это простое консольное приложение, которое ссылается на проект и использует метод `Debug.Assert` в пространстве имен `System.Diagnostics`, чтобы убедиться, что тест проходит. Последнее гарантирует, что функция возвращает ожидаемые значения. Поскольку `asserts` выполняются только в конфигурации `Debug`, мы также убедимся, что код не будет запускаться в другой конфигурации через директиву компилятора.

Листинг 4.3. Простое модульное тестирование

```

#if !DEBUG
#error asserts will only run in Debug configuration
#endif
using System;
using System.Diagnostics;
namespace DateUtilsTests {
    public class Program {
        public static void Main(string[] args) {
            var span = TimeSpan.FromSeconds(3);
            Debug.Assert(span.ToIntervalString() == "3 с",
"3 с провален");
            span = TimeSpan.FromMinutes(5);
            Debug.Assert(span.ToIntervalString() == "5 мин",
"5 мин провален");
            span = TimeSpan.FromHours(7);
            Debug.Assert(span.ToIntervalString() == "7 ч",
"7 ч провален");
        }
    }
}

```

Чтобы утверждения (ассерты) работали, необходим оператор препроцессора

Тестовый пример для секунд

Тестовый пример для минут

Тестовый пример для часов

```
        span = TimeSpan.FromMilliseconds(1);
        Debug.Assert(span.ToString("G") == "сейчас",
"сейчас провален");
    }
}
}
```

Тестовый пример для периода менее секунды

Зачем же нужны фреймворки модульного тестирования? Разве нельзя писать все тесты самостоятельно, как в примере выше? Можно, но это потребует больших усилий. В рассмотренном примере следует обратить внимание на следующее:

- Невозможно определить провал теста из внешней программы, например из инструмента сборки. Такие сценарии приходится обрабатывать отдельно. Тестовые фреймворки и средства запуска тестов, поставляемые с ними, легко с этим справляются.
 - Первый неудачный тест приведет к завершению программы. Если за ним последуют другие неудачи, это будет стоить нам времени. Придется запускать тесты снова и снова. Тестовые фреймворки запускают все тесты и сообщают сразу обо всех сбоях, включая ошибки компилятора.
 - Невозможно выборочно запускать определенные тесты, если, допустим, вы работаете над конкретной функцией и хотите отладить только ее. Тестовые фреймворки, напротив, позволяют проводить отладку выбранных тестов, не запуская остальные.
 - Тестовые фреймворки генерируют отчет о покрытии, который помогает выявить зоны недостаточного покрытия тестами. Для этого невозможно написать специальный тестовый код. Если вы сумели написать инструмент для анализа покрытия, вам стоит попробовать создать свой тестовый фреймворк.
 - Хотя тесты не зависят друг от друга, они выполняются последовательно, поэтому проведение всего набора занимает много времени. Если тестов немного, это не проблема, но даже в среднем проекте могут быть тысячи тестов разной продолжительности. Можно создавать потоки и запускать тесты параллельно, но это требует больших усилий. Тестовые фреймворки делают все это простым переключением.
 - Об ошибке вы узнаете только, что она произошла, но ее природа скрыта от вас. Строки не совпадают, но что это за несоответствие? Функция вернула ноль или случайно был добавлен лишний символ? Тестовые фреймворки укажут, в чем причина сбоя.

- Все, кроме использования `Debug.Assert`, предоставленного .NET, потребует написания дополнительного кода. Гораздо удобнее использовать существующий фреймворк.
- Используя тестовый фреймворк, вы сможете включиться в нескончаемые споры о том, какой из них лучше, и почувствовать свое превосходство, будучи совершенно неправым.

Теперь попробуем написать те же тесты, что в листинге 4.4, но в тестовом фреймворке. Многие фреймворки похожи, за исключением xUnit, который, вероятно, был разработан представителями инопланетной цивилизации, посетившими Землю. Но принципиальной разницы между фреймворками нет, лишь небольшие отличия в терминологии.

Для примера мы используем NUnit, но вы можете выбрать любой другой фреймворк. Вы увидите, насколько понятнее становится код. Большая часть тестового кода фактически представляет собой текстовую версию таблицы ввода/вывода (см. табл. 4.1). Предмет тестирования понятен, и что более важно, хотя у нас есть только один метод тестирования, в средстве запуска тестов мы можем запускать или отлаживать каждый тест по отдельности.

Техника, которую мы использовали в листинге 4.4 с атрибутами `TestCase`, называется *параметризованным тестом*. Если у вас есть определенный набор входных и выходных данных, вы можете просто объявить их как данные и повторно использовать в одной и той же функции, чтобы не писать каждый раз отдельные тесты. Точно так же, объединяя значения `ExpectedResult` и объявляя функцию с возвращаемым значением, вам даже не нужно явно писать `Assert`. Фреймворк делает это автоматически. Вы выполняете меньше работы!

Вы можете запустить эти тесты в окне обозревателя тестов Visual Studio: `View → Test Explorer` (Вид → Обозреватель тестов). Также можно запустить тест `dotnet` из командной строки или даже использовать стороннюю программу запуска тестов, например NCrunch. Результаты теста в обозревателе тестов Visual Studio будут выглядеть как на рис. 4.3.

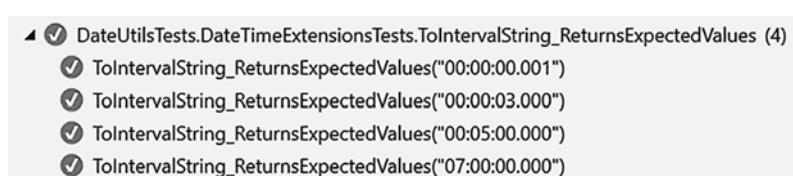
- 
- ▲ ✓ DateUtilsTests.DateTimeExtensionsTests.TolIntervalString_ReturnsExpectedValues (4)
- ✓ TolIntervalString_ReturnsExpectedValues("00:00:00.001")
 - ✓ TolIntervalString_ReturnsExpectedValues("00:00:03.000")
 - ✓ TolIntervalString_ReturnsExpectedValues("00:05:00.000")
 - ✓ TolIntervalString_ReturnsExpectedValues("07:00:00.000")

Рис. 4.3. Результаты тестирования, от которых глаз не оторвать

Листинг 4.4. Магия тестового фреймворка

```

using System;
using NUnit.Framework;
namespace DateUtilsTests {
    class DateUtilsTest {
        [TestCase("00:00:03.000", ExpectedResult = "3 с")]
        [TestCase("00:05:00.000", ExpectedResult = "5 мин")]
        [TestCase("07:00:00.000", ExpectedResult = "7 ч")]
        [TestCase("00:00:00.001", ExpectedResult = "сейчас")]
        public string ToIntervalString_ReturnsExpectedValues(
            string timeSpanText) {
            var input = TimeSpan.Parse(timeSpanText); ←
            return input.ToString(); ← Никаких ассертов!
        }
    }
}

```

Преобразование строки
в нужный тип входных
данных

На рис. 4.3 видно, как во время выполнения теста одна функция фактически делится на четыре разные функции и ее аргументы отображаются вместе с именем теста. Что еще более важно, вы можете выбрать один тест, запустить его или отладить. И если тест провалится, информативный отчет точно скажет, что не так с кодом. Допустим, вы случайно написали *сейчас* вместо *сейчас*. Ошибка будет отображаться следующим образом:

```

Message:
String lengths are both 6. Strings differ at index 5.
Expected: "сейчас"
But was: "сейчас"1
-----^

```

Вы видите не только что произошла ошибка, но и почему она произошла.

Тестовые фреймворки просты в использовании, и вы полюбите писать тесты еще больше, когда оцените, как фреймворки избавляют вас от лишней работы. Это индикаторы предполетной проверки НАСА, объявления о том, что «система работает normally» и маленькие нанороботы, выполняющие работу за вас. Любите тесты и тестовые фреймворки.

¹ Текст сообщения об ошибке: «Длина обеих строк 6. Строки различаются по индексу 5. Ожидалось: “сейчас”. Было получено: “сейчас”». — Примеч. пер.

4.3. НЕ ИСПОЛЬЗУЙТЕ TDD И ДРУГИЕ СОКРАЩЕНИЯ

У модульного тестирования, как у любой популярной религии, есть отдельные направления. В их число входят разработка через тестирование (TDD) и разработка через поведение (BDD, behavior-driven development). Я пришел к выводу, что в разработке преобладают две категории людей: те, кто любят создавать новые парадигмы и стандарты, чтобы им следовали без вопросов, и те, кто предпочитают без вопросов следовать таким парадигмам и стандартам. Мы любим инструкции и ритуалы, потому что им можно следовать не задумываясь. Это приводит к лишним затратам времени и заставляет ненавидеть тестирование.

Идея TDD заключается в том, что код будет лучше, если сначала написать его тесты. TDD предписывает сначала написать тесты для класса и только потом код этого класса. Поэтому код, который вы пишете для тестов, представляет собой руководство по реализации фактического кода. Вы пишете тесты, но они не компилируются. Затем вы пишете фактический код, и он компилируется. После вы запускаете тесты, и они проваливаются. Тогда вы исправляете ошибки в коде, чтобы тесты прошли. Концепция BDD также основана на тестах, с разницей только в названии и порядке проведения тестов.

Назвать полной ерундой принципы, лежащие в основе TDD и BDD, нельзя. Обдумывание способов тестирования улучшает понимание устройства кода. Проблема TDD заключается больше в схеме организации работы: писать тесты и из-за отсутствия реального кода получать ошибку компилирования (серъезно, Шерлок?), а после написания кода исправлять ошибки тестирования... Я ненавижу ошибки. Они заставляют меня чувствовать себя неудачником. Каждая красная волнистая линия в редакторе, каждый знак **STOP** в окне списка ошибок и каждый значок предупреждения сбивают меня с толку и отвлекают.

Если вы сосредоточиваетесь на teste, прежде чем писать код, вы начинаете думать о тестировании, а не о своей предметной области. Вы начинаете размышлять, как лучше написать тест. Ваш мозг занят элементами синтаксиса тестового фреймворка и организацией тестов, а не кодом продукта. Это не цель тестирования. Тесты не должны заставлять что-то изобретать. Тесты должны быть самым простым фрагментом кода, который вы можете написать. Если это не так, значит, вы все делаете неправильно.

Написание тестов до написания кода влечет за собой ошибку невозвратных затрат. Помните, как в главе 3 зависимости сделали код более жестким? Сюрприз! Тесты также зависят от кода. Когда у вас имеется полноценный набор тестов, вы стараетесь не менять дизайн кода, потому что это будет означать необходимость

изменения тестов. Это снижает вашу гибкость при создании прототипа кода. Возможно, тесты дают некоторое представление о том, действительно ли дизайн работает, но только в изолированных сценариях. Если вы обнаружите, что прототип плохо работает с другими компонентами, то будете вынуждены менять дизайн. Это нормально, если вы просиживаете много времени за чертежной доской, но на улицах обычно так не делают. Вы должны уметь быстро менять дизайн.

Пишите тесты, когда считете, что закончили основной объем работы над прототипом и в целом он работает нормально. Да, тесты повышают жесткость кода, но они компенсируют это, придав вам уверенности в его надежности. В итоге вы будете работать быстрее.

4.4. ПИШИТЕ ТЕСТЫ ДЛЯ СВОЕГО ЖЕ БЛАГА

Тесты не только улучшают программные продукты, но и качественно меняют ваш подход к работе. Если написание тестов перед кодом мешает изменять его дизайн, то создание тестов к готовому коду сделает его более гибким. В код будет легко вносить изменения позже, когда вы уже забудете о нем. И можно будет не беспокоиться о нарушении его поведения. Это дает свободу, работает как страховой полис и опровергает заблуждение о неокупаемых затратах. Когда вы пишете тесты после кода, то не разочаровываетесь на последующих итерациях, таких как прототипирование. Нужно переписать какой-то код? Первый шаг к этому — написать для него тесты.

Написание тестов после разработки хорошего прототипа — это своего рода подведение итогов разработки дизайна. Вы еще раз просматриваете весь код с учетом результатов тестов. Вы можете выявить проблемы, которые не обнаружили при создании прототипа.

Помните, я говорил, что внесение небольших правок в код подготовит вас к работе над большими задачами? Написание тестов тоже отлично подходит для этой цели. Добавьте к своему коду недостающие тесты. Дополнительные тесты никогда не помешают, если только они не избыточны. Они не обязательно должны быть связаны с текущей задачей. Расширьте тестовое покрытие, и кто знает, возможно, вы найдете ошибки.

Тесты могут выступать в роли документации, если они написаны ясным и понятным языком. Код каждого теста должен содержать ввод и ожидаемый вывод функции с указанием ее правильного наименования и описания. Код — не лучший способ описания, но в тысячу раз лучше, чем ничего.

Вы терпеть не можете, когда ваши коллеги ломают ваш код? Тесты в помощь. Тесты обеспечивают соблюдение соответствия между кодом и спецификацией, которое разработчики не могут нарушить. Вы не увидите, например, таких комментариев:

```
// Когда этот код был написан,  
// только Бог и я знали, что он делает.  
// Теперь это знает только Бог1.
```

Тесты гарантируют, что исправленная ошибка больше не возникнет. Добавьте тест для исправленной ошибки, чтобы спокойно забыть о ней. В противном случае кто знает, при каком изменении она появится снова? Такие тесты существенно экономят время.

Тесты улучшают и программы, и их разработчиков. Пишите тесты, чтобы стать более эффективным разработчиком.

4.5. КАК ПОНЯТЬ, ЧТО ИМЕННО ТЕСТИРОВАТЬ

Не остановится, что вечно длится,
И тесты с вечностью порою могут завершиться.

Г. Ф. Кодкрафт²

Написать один тест и проверить, как он проходит, — только половина дела. Успех теста не означает, что функция надежна. Будет ли она работать в случае ошибки в коде? Все ли возможные сценарии вы учли? Что именно вы тестируете? Если тесты не помогают найти ошибки, они провалены.

Один из моих менеджеров проверял надежность тестов следующим образом: он наугад удалял строки из кода готового продукта и запускал тесты. Если результат тестов был успешен, значит, они не работали.

¹ Этот печально известный комментарий — интерпретация шутки, первоначально приписываемой Джону Паулю Фридриху Рихтеру (John Paul Friedrich Richter), жившему в XIX веке. Он не написал ни строчки кода — только комментарии (<https://quoteinvestigator.com/2013/09/24/god-knows/>).

² Аллюзия на строки Г. Ф. Лавкрафта из рассказа «Безымянный город» («The Nameless City»): «То не мертвое, что вечность охраняет, смерть вместе с вечностью порою умирает». — Примеч. пер.

Есть и более эффективные методы определения, какие ситуации тестировать. Лучше всего начать со спецификации, но на улицах их редко встретишь. Возможно, имеет смысл создать спецификацию самому, но даже если у вас есть только код, все равно существуют способы определить, что тестировать.

4.5.1. Уважайте границы

Вы можете вызвать функцию, которая получает простое целое число из диапазона с четырьмя миллиардами различных значений. Означает ли это, что нужно проверять, работает ли функция для каждого из них? Нет. Нужно определить, какие входящие значения вызывают ветвление кода или переполнение, а затем проверить ближайшие к ним значения.

Рассмотрим функцию, которая на странице регистрации онлайн-игры проверяет по дате рождения игрока, можно ли предоставить ему доступ к игре. Примем, что возраст, с которого разрешен доступ к вашей игре, — 18 лет. Для тех, кто старше, все просто: вы вычитаете год рождения из текущего года и сравниваете с 18. Но что, если человеку исполнилось 18 лет на прошлой неделе? Неужели вы лишите его возможности получить удовольствие от посредственной графики за его же деньги? Конечно нет.

Зададим функцию `IsLegalBirthdate`. Используем класс `DateTime` вместо `DateTimeOffset` для представления даты рождения, поскольку у даты рождения нет часового пояса. Если вы родились 21 декабря в Самоа, вашим днем рождения будет 21 декабря в любой точке мира, даже в Американском Самоа, который на 24 часа опережает тезку, хотя находится всего в ста милях от него. Я уверен, что там каждый год не утихают споры о том, в какой день приглашать родню с соседнего острова на рождественский ужин. Часовые пояса — странная штука.

В любом случае сначала мы вычисляем разницу лет. Точная дата рождения нужна только в одном случае — в год 18-летия пользователя. Если это именно такой год, мы проверяем месяц и день. Если год другой, мы проверяем только то, старше ли человек 18 лет. Используем константу для обозначения допустимого возраста вместо цифры, потому что в цифре можно сделать опечатку, и если ваш босс подойдет и спросит: «Эй, можешь повысить ценз до 21 года?», достаточно будет отредактировать только одно место в функции. Вам также не придется писать комментарий `// допустимый возраст` рядом с каждой цифрой `18` в коде, чтобы объяснить ее. Она будет сама себя объяснить.

В каждом условии функции, которое включает операторы `if`, циклы `while`, операторы ветвления и т. д., только определенные входящие значения будут

следовать по внутреннему пути. Это означает, что можно разделить все входящие значения на основе условий в зависимости от входных параметров. В примере из листинга 4.5 не нужно проверять все возможные значения `DateTime` от 1 января 1 года н. э. до 31 декабря 9999 года, которых около 3,6 миллиона. Нужно проверить только семь из них.

Листинг 4.5. Алгоритм вышибалы

```
public static bool IsLegalBirthdate(DateTime birthdate) {
    const int legalAge = 18;
    var now = DateTime.Now;
    int age = now.Year - birthdate.Year;
    if (age == legalAge) {
        return now.Month > birthdate.Month
            || (now.Month == birthdate.Month
                && now.Day > birthdate.Day);
    }
    return age > legalAge;
}
```

Условия в коде

Эти семь значений перечислены в табл. 4.2.

Таблица 4.2. Разделение входных значений на основе условий

	Разница в годах	Месяц рождения	Дата рождения	Ожидаемый результат
1	=18	=Текущий месяц	<Текущего дня	Правда
2	=18	=Текущий месяц	=Текущий день	Ложь
3	=18	=Текущий месяц	>Текущего дня	Ложь
4	=18	<Текущего месяца	Любая	Правда
5	=18	>Текущего месяца	Любая	Ложь
6	>18	Любой	Любая	Правда
7	<18	Любой	Любая	Ложь

Количество сценариев сократилось с 3,6 миллиона до семи только потому, что мы определили условия. Условия, разделяющие массив входящих значений, называются *граничными*, поскольку они устанавливают границы входящих значений для возможных путей кода в функции. Далее напишем тесты для этих значений, как показано в листинге 4.6. Фактически мы создадим копию тестовой таблицы для этих значений, преобразуем ее в `DateTime` и запустим функцию. Значения `DateTime` нельзя жестко закодировать в таблице ввода/

вывода, потому что допустимость даты рождения меняется в зависимости от текущего времени.

Можно создать функцию на основе `TimeSpan`, как мы делали раньше, но разрешенный возраст основан не на точном количестве дней, а на абсолютной дате и времени. Лучше использовать табл. 4.2, поскольку она более точно отражает модель поведения. Обозначим как `-1` значения «меньше», `1` — значения «больше» и `0` — равенство. Исходя из этого определим фактические входящие значения.

Листинг 4.6. Создание тестовой функции из табл. 4.2

```
[TestCase(18, 0, -1, ExpectedResult = true)]
[TestCase(18, 0, 0, ExpectedResult = false)]
[TestCase(18, 0, 1, ExpectedResult = false)]
[TestCase(18, -1, 0, ExpectedResult = true)]
[TestCase(18, 1, 0, ExpectedResult = false)]
[TestCase(19, 0, 0, ExpectedResult = true)]
[TestCase(17, 0, 0, ExpectedResult = false)]
public bool IsLegalBirthdate_ReturnsExpectedValues(
    int yearDifference, int monthDifference, int dayDifference) {
    var now = DateTime.Now;
    var input = now.AddYears(-yearDifference)
        .AddMonths(monthDifference)
        .AddDays(dayDifference);
    return DateTimeExtensions.IsLegalBirthdate(input);
}
```

Подготовка фактического
ввода

Готово! Мы сузили диапазон возможных входящих значений и точно определили, что тестировать в функции. Теперь можно создать план тестирования.

Всякий раз, когда требуется выяснить, что тестировать в функции, начинайте со спецификации. Однако на улицах вы, скорее всего, обнаружите, что спецификация давно устарела или ее никогда и не было, поэтому второй лучший способ — начать с граничных условий. Параметризованные тесты также помогают сосредоточиться на предмете тестирования, а не написании повторяющегося тестового кода. Иногда создания новой функции для каждого теста не избежать, но в некоторых случаях, особенно таких, как этот, когда тесты имеют привязку к данным, параметризованные тесты могут сэкономить массу времени.

4.5.2. Покрытие кода

Покрытие кода — это магия, и как любая магия, это просто слова. Чтобы измерить покрытие кода, необходимо внедрить в каждую строку обратные вызовы для отслеживания того, как далеко выполняется вызываемый тестом код и какие

части тест пропускает. Так вы узнаете, какая часть кода не задействуется и, следовательно, не покрывается тестами.

Среды разработки редко поставляются с готовыми инструментами измерения покрытия кода. Они есть либо в версиях Visual Studio по астрономическим ценам, либо в других платных инструментах, таких как NCrunch, dotCover и NCover. Codecov (<https://codecov.io>) – это сервис, который работает с онлайн-репозиторием, и в нем есть бесплатная функциональность. На момент написания этой книги измерить покрытие кода локально и бесплатно в .NET было можно только при помощи библиотеки Coverlet и расширений для отчетов о покрытии кода в Visual Studio Code.

Инструменты измерения покрытия кода сообщают, какие части кода выполнялись при запуске тестов. Очень ценно понимать, какого тестового покрытия не хватает для проверки всех путей кода. Но покрытие не единственная часть истории, и уж точно не самая главная. У вас может быть полное покрытие кода, но при этом не будут проверяться отдельные тестовые случаи. Рассмотрим такую ситуацию.

Предположим, что мы закомментировали тесты, которые вызывают функцию `IsLegalBirthdate` с датой рождения ровно 18 лет назад, как показано в следующем листинге.

Листинг 4.7. Отсутствующие тесты

```
//[TestCase(18, 0, -1, expectedResult = true)]
//[TestCase(18, 0, 0, expectedResult = false)]
//[TestCase(18, 0, 1, expectedResult = false)]
//[TestCase(18, -1, 0, expectedResult = true)]
//[TestCase(18, 1, 0, expectedResult = false)]
[TestCase(19, 0, 0, expectedResult = true)]
[TestCase(17, 0, 0, expectedResult = false)]
public bool IsLegalBirthdate_ReturnsExpectedValues(
    int yearDifference, int monthDifference, int dayDifference) {
    var now = DateTime.Now;
    var input = now.AddYears(-yearDifference)
        .AddMonths(monthDifference)
        .AddDays(dayDifference);
    return DateTimeExtensions.IsLegalBirthdate(input);
}
```

Закомментированные
тестовые случаи

В этом случае такой инструмент, как, например, NCrunch, покажет отсутствующее покрытие (рис. 4.4). Кружок, обозначающий покрытие, рядом с оператором `return` внутри оператора `if` затенен, потому что мы никогда не вызываем функцию с параметром, который соответствует условию `age == legalAge`. Это значит, что мы пропускаем некоторые входящие значения.

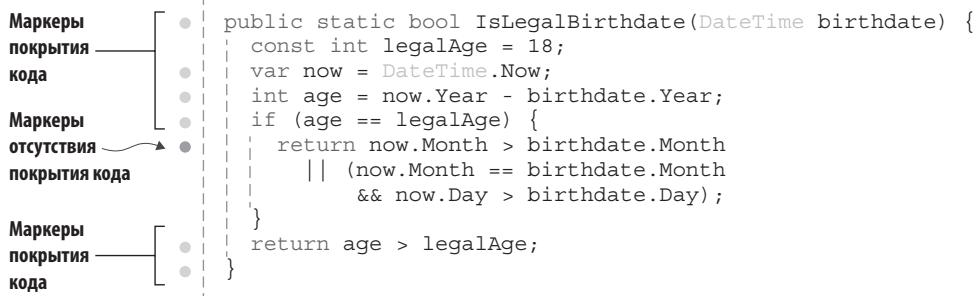


Рис. 4.4. Недостающее покрытие кода

Когда вы раскомментируете все тестовые случаи и снова запустите тесты, проверка покажет полное покрытие кода (рис. 4.5).

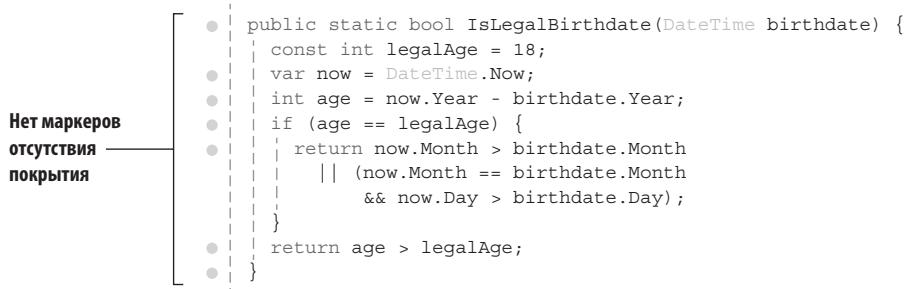


Рис. 4.5. Полное покрытие кода

Инструменты измерения покрытия кода хороши, но они отображают не все фактическое покрытие тестами. Вам все равно нужно иметь представление о диапазоне входящих значений и граничных условиях. Полное покрытие кода не означает полного покрытия тестами. Рассмотрим следующую функцию, в которой нужно вернуть элемент списка по его индексу:

```

public Tag GetTagDetails(byte numberOfItems, int index) {
    return GetTrendingTags(numberOfItems)[index];
}

```

Вызов функции `GetTagDetails(1, 0)`; будет успешным, и мы сразу же достигнем полного покрытия кода. Разве мы проверили все возможные случаи? Нет. Покрытие входящих значений и близко не равно 100%. Что, если `numberOfItems` равно нулю, а `index` не равен нулю? Что, если `index` будет отрицательным?

Нельзя сосредоточиваться исключительно на покрытии кода и пытаться заполнить все пробелы. Необходимо обеспечивать полное тестовое покрытие, учитывая все возможные входящие значения и эффективно подбирая граничные условия. Тем не менее одно не исключает другого: можно использовать оба подхода одновременно.

4.6. НЕ ПИШИТЕ ТЕСТЫ

Да, тестирование полезно, но не писать тесты — еще лучше. Как обойтись без тестов и при этом сохранить надежность кода?

4.6.1. Не пишите код

Если кода не существует, его не нужно тестировать. Удаленный код не содержит ошибок. Помните об этом. Стоит ли писать тесты, если вам вообще не нужен этот код? Может, лучше использовать существующий пакет, а не создавать его аналог с нуля? Может, использовать существующий класс, который делает то же самое, что вы пытаетесь реализовать? Например, у вас может возникнуть соблазн написать собственные регулярные выражения для проверки URL-адресов, но все, что требуется сделать, это использовать класс `System.Uri`.

Конечно, сторонний код не всегда идеален или подходит для ваших целей. Вы можете понять это не сразу, но обычно все-таки стоит рискнуть. В кодовой базе может быть код вашего коллеги, выполняющий те же функции, что вам нужны. Поищите в ней.

Если альтернатив нет, готовьтесь писать собственный код. Не бойтесь изобретать велосипед. Это может быть полезно, о чем я уже говорил в главе 3.

4.6.2. Ограничьтесь выборочными тестами

Знаменитый *принцип Парето* гласит, что 80% следствий вызываются 20% причин. По крайней мере, так говорят 80% определений. Чаще этот принцип называют *принципом 80/20*. Он применим и к тестированию. Вы можете получить 80% надежности при 20-процентном покрытии тестами, если будете выбирать тесты с умом.

Ошибки проявляются неоднородно. Вероятность их возникновения для разных строк кода различна. Вероятность обнаружить ошибку выше в часто

используемом коде. Области кода, в которых вероятность возникновения проблем высока, называют *критическими*, или *горячими путями* (hot paths).

Свой сайт я не тестировал даже после того, как он стал одним из самых популярных турецких сайтов. Но тесты пришлось добавить из-за большого количества ошибок парсера текстовой разметки. Разметка была нестандартной и мало напоминала Markdown. Поскольку логика парсинга была сложна и подвержена ошибкам, стало экономически невыгодно исправлять каждую проблему после развертывания в рабочей среде. Я разработал для нее набор тестов. Тестовых фреймворков в то время еще не существовало. По мере появления новых ошибок число тестов росло. Со временем мы разработали широкий набор тестов, который спас нас от тысяч провальных развертываний на продакшен. Тесты работают.

Даже простой просмотр домашней страницы сайта обеспечивает проверку значительной части кода, поскольку действуются многие пути, общие с другими страницами. На уличном сленге это называют *дымовым тестированием*. Метод — ровесник первых прототипов компьютера, которые пытались включить, чтобы посмотреть, не идет ли из них дым. Отсутствие дыма было хорошим знаком. Точно так же фокусировка тестирования на критических общих компонентах важнее, чем полнота покрытия кода. Не тратьте время на то, чтобы обеспечить проверку дополнительной строки, если это не имеет большого значения. Вы уже знаете, что покрытие кода — далеко не все.

4.7. ПУСТЬ ТЕСТИРОВАНИЕМ ЗАЙМЕТСЯ КОМПИЛЯТОР

В строго типизированном языке правильное применение системы типов может уменьшить количество необходимых тестовых сценариев. Я уже говорил, как ссылки, допускающие значение `null`, помогают избежать проверки значений `null` в коде, что также уменьшает потребность в тестах для сценариев с `null`. Рассмотрим простой пример. В предыдущем разделе мы убедились, что пользователю, желающему зарегистрироваться, исполнилось 18 лет. Теперь нужно проверить, допустимо ли выбранное им имя, поэтому нам требуется соответствующая функция.

4.7.1. Как исключить проверки на `null`

Допустим, что по правилу имя пользователя должно состоять из не менее чем восьми строчных буквенно-цифровых символов. Шаблон регулярного выражения для такого имени пользователя будет `^[a-zA-Z0-9]{1,8}$`. Можно написать класс

имени пользователя, как в листинге 4.8. Зададим класс `Username` для представления всех имен пользователей в коде. Нам не придется думать о том, где проверять ввод, передав функцию проверки любому коду, который требует имя пользователя.

Чтобы убедиться, что имя пользователя не принимает недействительное значение, мы проверяем параметр в конструкторе и выдаем исключение, если имя имеет неправильный формат. Остальной код вне конструктора остается шаблонным, чтобы он мог работать в сценариях сравнения. Помните, что всегда можно использовать создание базового класса `StringValue`, написав минимум кода для каждого класса значений на основе строк.

Я повторяю некоторые блоки кода в разных листингах для лучшего их понимания. Обратите внимание на использование оператора `nameof` вместо жестко закодированных строк для ссылок на параметры. Это позволяет синхронизировать имена после переименования. Кроме того, этот оператор можно использовать для полей и свойств, и он особенно полезен для сценариев, когда данные хранятся в отдельном поле и необходимо ссылаться на это поле по его имени.

Листинг 4.8. Реализация типа значения имени пользователя

```
public class Username {
    public string Value { get; private set; }
    private const string validUsernamePattern = @"^[a-zA-Z0-9]{1,8}$";

    public Username(string username) {
        if (username is null) {
            throw new ArgumentNullException(nameof(username));
        }
        if (!Regex.IsMatch(username, validUsernamePattern)) {
            throw new ArgumentException(nameof(username),
                "Invalid username");
        }
        this.Value = username;
    }

    public override string ToString() => base.ToString();
    public override int GetHashCode() => Value.GetHashCode();
    public override bool Equals(object obj) {
        return obj is Username other && other.Value == Value;
    }
    public static implicit operator string(Username username) {
        return username.Value;
    }
    public static bool operator==(Username a, Username b) {
        return a.Value == b.Value;
    }
    public static bool operator !=(Username a, Username b) {
        return !(a == b);
    }
}
```

Единственная проверка имени пользователя

Обычный шаблон, делающий класс пригодным для сравнения

МИФЫ О РЕГУЛЯРНЫХ ВЫРАЖЕНИЯХ

Регулярные выражения — одно из самых блестящих изобретений в computer science. Мы обязаны ими достопочтенному Стивену Коулу Клини (Stephen Cole Kleene). Они позволяют создать анализатор текста из пары символов. Шаблон `light` соответствует только строке `light`, а `[ln]ight` соответствует `light`, и `night`. Точно так же `li(gh){1,2}t` соответствует только словам `light` и `light`, и это не опечатка, а стихотворение Арама Сарояна (Aram Saroyan), состоящее из одного слова.

Джейми Завински (Jamie Zawinski) однажды сказал: «Некоторые люди, сталкиваясь с проблемой, думают так: «О, знаю, я использую регулярные выражения». И у них появляются две проблемы». Регулярные выражения работают с определенными задачами парсинга. Регулярные выражения не зависят от контекста, поэтому невозможно с помощью одного регулярного выражения найти самый внутренний тег в документе HTML и несовпадающие закрывающие теги. Это означает, что такие выражения не подходят для сложных задач парсинга. Тем не менее их можно использовать для анализа текста с невложенной структурой.

Регулярные выражения удивительно эффективны в ситуациях, когда их можно применять. Если вам нужна еще более высокая производительность, вы можете предварительно скомпилировать их в C#, создав объект `Regex` с параметром `RegexOptions.Compiled`. При этом пользовательский код, который анализирует строку на основе шаблона, будет создаваться по требованию. Шаблон преобразуется в C# и, в конечном итоге, в машинный код. Последовательные вызовы одного и того же объекта `Regex` будут повторно использовать скомпилированный код, повышая производительность для нескольких итераций.

Несмотря на эффективность, не используйте регулярные выражения, если существует простая альтернатива. Если вам нужно проверить, имеет ли строка определенную длину, простой `"str.Length== 5"` будет намного быстрее и проще, чем `"Regex.IsMatch (@"^.{5}$", str)"`. Точно так же класс `string` содержит множество эффективных методов для обычных операций проверки строк, например `StartsWith`, `EndsWith`, `IndexOf`, `Last-IndexOf`, `IsNullOrEmpty` и `IsNullOrWhiteSpace`. Всегда выбирайте готовые методы вместо регулярных выражений для конкретных случаев использования.

Тем не менее важно знать хотя бы базовый синтаксис регулярных выражений, потому что они могут пригодиться в среде разработки. В целях экономии времени изменять код можно довольно сложными способами. Все популярные текстовые редакторы поддерживают регулярные выражения для операций поиска и замены. Я говорю о таких операциях, как «Переместить сотни символов квадратных скобок на следующую строку, но только если они располагаются в конце строки кода». На выбор подходящего шаблона регулярного выражения достаточно пары минут, и не надо будет заниматься целый час этим перемещением вручную.

Тестирование конструктора `Username` потребует создания трех тестовых методов, как показано в листинге 4.9: для проверки допустимости значений `null`, так как вызывается другой тип исключения; для не-`null`, но недопустимых входных данных; наконец, для действительных входных данных, поскольку необходимо убедиться, что конструктор распознает их именно как действительные.

Листинг 4.9. Тесты для класса `Username`

```
class UsernameTest {
    [Test]
    public void ctor_nullUsername_ThrowsArgumentNullException() {
        Assert.Throws<ArgumentNullException>(
            () => new Username(null));
    }

    [TestCase("")]
    [TestCase("Upper")]
    [TestCase("toolongusername")]
    [TestCase("root!!")]
    [TestCase("a b")]
    public void ctor_invalidUsername_ThrowsArgumentException(string username) {
        Assert.Throws<ArgumentException>(
            () => new Username(username));
    }

    [TestCase("a")]
    [TestCase("1")]
    [TestCase("hunter2")]
    [TestCase("12345678")]
    [TestCase("abcdefgh")]
    public void ctor_validUsername_DoesNotThrow(string username) {
        Assert.DoesNotThrow(() => new Username(username));
    }
}
```

Если бы мы разрешили для класса `Username` ссылки, допускающие значение `null`, то для сценария с `null` писать тесты не пришлось бы. Единственное исключение — создание общедоступного API, который может не работать с кодом, поддерживающим ссылки, допускающие значение `null`. В этом случае пришлось бы проводить проверки на наличие `null`.

Точно так же объявление `Username` структурой, когда это уместно, сделало бы его типом значения, избавив от необходимости проверки на `null`. Использование правильных типов и структур помогает сократить количество тестов. А компилятор обеспечит правильность кода.

Использование специфических типов для соответствующих целей снижает потребность в тестах. Когда функция регистрации получает `Username` вместо

строки, не нужно тестировать, проверяет ли она свои аргументы. Точно так же когда функция получает аргумент URL-адреса в виде класса `Uri`, не нужно проверять, правильно ли она обрабатывает URL-адрес.

4.7.2. Как исключить проверки диапазона

Чтобы уменьшить диапазон возможных недопустимых входящих значений, можно использовать типы целых чисел без знака. В табл. 4.3 представлены беззнаковые версии целочисленных примитивов. В таблице вы можете видеть диапазоны значений для разных типов данных, используемых в коде. Важно помнить, совместим ли выбранный тип напрямую с `int`, потому что это основной тип для целых чисел в .NET. Вы наверняка знакомы с этими типами, но скорее всего, не думали, что они могут избавить вас от необходимости писать дополнительные тестовые сценарии. Но если функции нужны только положительные значения, то зачем использовать `int`, проверять отрицательные значения и выдавать исключения? Просто используйте `uint`.

Таблица 4.3. Целочисленные типы с диапазонами значений

Имя	Целочисленный тип	Диапазон значений	Возможность присвоения <code>int</code> без потерь
<code>int</code>	32-битный со знаком	–2147483648..2147483647	Еще бы
<code>uint</code>	32-битный без знака	0..4294967295	Нет
<code>long</code>	64-битный со знаком	–9223372036854775808..9223372036854775807	Нет
<code>ulong</code>	64-битный без знака	0..18446744073709551615	Нет
<code>ashort</code>	16-битный со знаком	–32768..32767	Да
<code>ushort</code>	16-битный без знака	0..65535	Да
<code>sbyte</code>	8-битный со знаком	–128..127	Да
<code>byte</code>	8-битный без знака	0..255	Да

При использовании беззнакового типа передача отрицательной константы в функцию вызовет ошибку компиляции. Передавать переменные с отрицательными значениями можно только при явном приведении типа, что заставляет задуматься, действительно ли имеющееся значение подходит для функции в месте вызова. Проверка отрицательных аргументов не входит в обязанности функции.

Предположим, что функция должна возвращать популярные теги на сайте микроблогов, но не более определенного их числа. Она получает ряд элементов для извлечения строк сообщений, как в листинге 4.10. Функция `GetTrendingTags` возвращает элементы с учетом их количества. Обратите внимание, что входящее значение — `byte`, а не `int`, поскольку в списке тегов не бывает больше 255 элементов. Это сразу исключает сценарии, когда входящее значение отрицательное или слишком большое. Не нужно даже проверять ввод. Таким образом, тестов становится меньше на один, а диапазон входящих значений значительно уменьшается, что сужает область ошибок.

Листинг 4.10. Получение сообщений, принадлежащих только определенной странице

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Posts {
    public class Tag {
        public Guid Id { get; set; }
        public string Title { get; set; }
    }

    public class PostService {
        public const int MaxPageSize = 100;
        private readonly IPostRepository db;

        public PostService(IPostRepository db) {
            this.db = db;
        }

        public IList<Tag> GetTrendingTags(byte numberofItems) { ←
            return db.GetTrendingTagTable()
                .Take(numberofItems) ←
                .ToList(); ←
                | byte или ushort можно передавать
                | так же безопасно, как int
        }
    }
}
```

Отметим две вещи. Во-первых, для нашего варианта использования мы выбираем меньший тип данных. Нам не нужно поддерживать миллиарды строк в поле

популярных тегов. Мы сузили пространство ввода. Во-вторых, мы выбираем беззнаковый тип `byte`, который не может быть отрицательным. Таким образом, мы избавляемся от дополнительного тестового сценария и потенциальной проблемы, которая может вызвать исключение. Функция `Take` из LINQ не выдает исключение с `List`, но может это делать, когда преобразуется в запрос для базы данных, такой как Microsoft SQL Server. Изменяя тип, мы избегаем таких сценариев и нам не нужно писать для них тесты.

Обратите внимание, что .NET использует `int` в качестве стандартного типа для многих операций, таких как индексирование и подсчет. Выбор другого типа может потребовать приведения и преобразования значений в `int` при взаимодействии со стандартными компонентами .NET.

Убедитесь, что не зарываете себя в яму из-за собственной педантичности. Чувство удовлетворенности и удовольствие от написания кода важнее, чем какой-то единичный случай, которого вы пытаетесь избежать. Например, если в будущем вам понадобится более 255 элементов, вам придется заменить все ссылки на тип `byte` ссылками на `short` или `int`, что может отнять много времени. Так что причина, по которой вы стремитесь избежать тестов, должна быть действительно веской. Возможно, окажется проще написать больше тестов, чем работать с другими типами. В конце концов, важны только ваши комфорт и время, невзирая на очевидное преимущество использования типов для определения диапазонов допустимых значений.

4.7.3. Как исключить проверки допустимых значений

Иногда значения используются для обозначения операции в функции. Типичным примером является функция `fopen` в языке программирования С. Второй принимаемый этой функцией строковый параметр указывает на режим открытия файла и может означать *открытие для чтения, открытие для добавления, открытие для записи* и т. д.

Пару десятков лет спустя команда .NET создала усовершенствованное решение — отдельные функции для каждого сценария. Появились отдельные методы `File.Create`, `File.OpenRead` и `File.OpenWrite`, позволившие избежать введения дополнительных параметров и парсинга их значений. Стало невозможно передать неверный параметр. Исчезли ошибки при парсинге параметров в функции, потому что не стало самих параметров.

Обычно значения используются для обозначения типа операции. Но лучше разделить сложные функции на более простые — это поможет передать логику и уменьшит необходимость в тестовом покрытии.

Логические параметры часто используются в C# для изменения логики выполняемой функции. Примером может служить опция сортировки в функции извлечения популярных тегов, как в листинге 4.11. Предположим, что нам нужно вывести эти теги, отсортированные по заголовку, на странице управления тегами. Вопреки законам термодинамики разработчики постоянно теряют энтропию. Они всегда стараются вносить изменения с наименьшей энтропией, не задумываясь о возможных последствиях. Часто первая мысль разработчика — покончить с задачей, добавив логический параметр.

Листинг 4.11. Логические параметры

```
public IList<Tag> GetTrendingTags(byte numberOfItems,
    bool sortByTitle) { ← Добавленный параметр
    var query = db.GetTrendingTagTable();
    if (sortByTitle) { ← Новый условный параметр
        query = query.OrderBy(p => p.Title);
    }
    return query.Take(numberOfItems).ToList();
}
```

Проблема в том, что если продолжать в том же духе, функция очень усложнится из-за роста числа комбинаций параметров. Допустим, для другой функции требуются трендовые теги, начиная со вчерашнего дня. Добавим это условие вместе с другими параметрами в следующем листинге. Теперь функция также должна поддерживать комбинации `sortByTitle` и `yesterdaysTags`.

Листинг 4.12. Дополнительные логические параметры

```
public IList<Tag> GetTrendingTags(byte numberOfItems,
    bool sortByTitle, bool yesterdaysTags) { ← Больше параметров!
    var query = yesterdaysTags
        ? db.GetTrendingTagTable()
        : db.GetYesterdaysTrendingTagTable();
    if (sortByTitle) {
        query = query.OrderBy(p => p.Title);
    }
    return query.Take(numberOfItems).ToList();
}
```

Больше условий!

Наблюдается тенденция к росту сложности функции с добавлением каждого нового логического параметра. Для трех вариантов использования уже есть четыре разновидности функции. С каждым новым параметром мы создаем версии функции, которые никто не будет использовать, хотя, возможно, когда-нибудь кто-то это все-таки сделает и окажется в тупике. Лучше всего иметь отдельную функцию для каждого клиента, как показано ниже.

Листинг 4.13. Разделение на отдельные функции

```
public IList<Tag> GetTrendingTags(byte numberOfItems) {
    return db.GetTrendingTagTable()
        .Take(numberOfItems)
        .ToList();
}

public IList<Tag> GetTrendingTagsByTitle(
    byte numberOfItems) {
    return db.GetTrendingTagTable()
        .OrderBy(p => p.Title)
        .Take(numberOfItems)
        .ToList();
}

public IList<Tag> GetYesterdaysTrendingTags(byte numberOfItems) {
    return db.GetYesterdaysTrendingTagTable()
        .Take(numberOfItems)
        .ToList();
}
```

Функциональность разделяется
по именам функций,
а не по параметрам

Теперь у нас стало на один тестовый сценарий меньше. В качестве бонусов имеем удобочитаемость и повышенную производительность. Выигрыши, конечно, ничтожны и незаметны для одной функции, но в случае масштабирования кода они могут играть важную роль, даже если вы об этом не подозреваете. Экономия увеличится в геометрической прогрессии, если избегать передачи состояния в параметрах и максимально использовать функции. Раздражающий повторяющийся код легко преобразовать в общие функции, как показано в следующем листинге.

Листинг 4.14. Рефакторинг отдельных функций с общей логикой

```
private IList<Tag> toListTrimmed(byte numberOfItems,
    IQueryable<Tag> query) {
    return query.Take(numberOfItems).ToList();
}

public IList<Tag> GetTrendingTags(byte numberOfItems) {
    return toListTrimmed(numberOfItems, db.GetTrendingTagTable());
}

public IList<Tag> GetTrendingTagsByTitle(byte numberOfItems) {
    return toListTrimmed(numberOfItems, db.GetTrendingTagTable()
        .OrderBy(p => p.Title));
}

public IList<Tag> GetYesterdaysTrendingTags(byte numberOfItems) {
    return toListTrimmed(numberOfItems,
        db.GetYesterdaysTrendingTagTable());
}
```

Общая
функциональность

В этом примере экономия незначительна, но в иных случаях подобный рефакторинг может иметь большее значение. Важный вывод: используйте рефакторинг, чтобы избежать повторения кода и комбинаторного ада.

Тот же метод можно использовать с параметрами `enum`, которые применяются для назначения функции определенной операции. Используйте отдельные функции и даже композицию функций вместо передачи списка параметров.

4.8. ИМЕНОВАНИЕ ТЕСТОВ

Имена очень важны. Поэтому важны и грамотные соглашения как для рабочего, так и для тестового кода, хотя эти виды не обязательно должны пересекаться. Тесты с хорошим покрытием могут служить спецификациями, если имеют корректное название. Из названия теста должно быть понятно:

- имя тестируемой функции;
- входные данные и начальное состояние;
- ожидаемое поведение;
- кто виноват.

Последнее, конечно, шутка. Помните? После код-ревью вашему коду дан зеленый свет. Кроме себя, винить некого. В лучшем случае вы можете разделить вину с кем-нибудь. Я обычно использую для имен тестов формат `A_B_C`, и он заметно отличается от обычного стиля именования. В предыдущих примерах мы использовали более простую схему, потому что для описания начального состояния теста служил атрибут `TestCase`. Я использую дополнительно `ReturnsExpectedValues`, но можно просто добавить `Test` к имени функции. Лучше не использовать только имя функции, потому что вас может сбить с толку ее появление в списках автозавершения кода. Точно так же если функция не принимает входные данные или не зависит от начального состояния, можно пропустить эту часть имени. Цель — тратить меньше времени на тесты, а не вводить драконовские законы именования.

Предположим, босс попросил вас написать новое правило проверки формы регистрации, чтобы код возвращал ошибку, если клиент не принял условия обслуживания. Именем такого теста будет `Register_LicenseNotAccepted_ShouldReturnFailure`, как показано на рис. 4.6.

Это не единственное возможное соглашение об именовании. Некоторые разработчики предпочитают создавать внутренние классы для каждой тестируемой

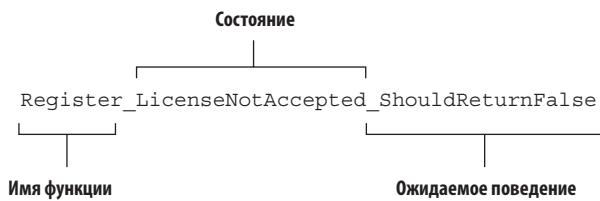


Рис. 4.6. Составляющие имени теста

функции и вносить в имена только состояние и ожидаемое поведение, но я считаю такие названия слишком громоздкими. Важно выбрать соглашение, которое лучше всего подходит для вас.

ИТОГИ

- С нежеланием писать тесты справиться легко: многие из них можно просто не писать.
- Разработка через тестирование и другие подобные парадигмы могут вызвать еще большее нежелание писать тесты. Страйтесь писать тесты, которые вас радуют.
- Фреймворки, особенно для параметризованных тестов на основе данных, значительно упрощают создание тестов.
- Количество тестовых сценариев можно заметно сократить, тщательно проанализировав граничные значения входных данных функции.
- Правильное использование типов позволит избежать множества ненужных тестов.
- Тесты не просто обеспечивают хорошее качество кода. Они помогают вам улучшить навыки разработки и повысить производительность.
- Тестирование в рабочей среде можно проводить, только если вы уже обновили свое резюме.

Вознаграждение за рефакторинг

В этой главе

- ✓ Освоение рефакторинга
- ✓ Поэтапный рефакторинг при значительных изменениях
- ✓ Использование тестов для быстрого внесения изменений в код
- ✓ Внедрение зависимостей

В главе 3 мы говорили о том, как сопротивление изменениям привело к краху французскую монархию и разработчиков программного обеспечения. Рефакторинг — это искусство изменения структуры кода. По словам Мартина Фаулер¹, Лео Броди (Leo Brodie) впервые употребил этот термин в своей книге «Thinking Forth» еще в 1984 году. Получается, что рефакторингу столько же лет, сколько «Назад в будущее» и «Парню-каратисту», моим любимым фильмам детства.

Отличный код — это, как правило, только половина успеха разработчика. Вторая половина — пригодность кода к изменениям. В идеальном мире мы должны

¹ Martin Fowler, Etymology of Refactoring (Мартин Фаулер, «Этимология рефакторинга») <https://martinfowler.com/bliki/EtymologyOfRefactoring.html>.

писать и изменять код со скоростью мысли. Необходимость нажимать на клавиши, отрабатывать синтаксис, запоминать ключевые слова и менять фильтр в кофемашине — все это препятствия на пути от идеи к продукту. Поскольку, вероятно, пройдет еще некоторое время, прежде чем искусственный интеллект научится программировать за нас, нам не помешает усовершенствовать навыки рефакторинга.

IDE играют важную роль в рефакторинге. Одним нажатием клавиши (**F2** в Visual Studio для Windows) можно переименовать класс и все ссылки на него и даже получить доступ к большинству параметров рефакторинга. Я настоятельно рекомендую изучить сочетания клавиш для функций, которые вы часто используете в своем любимом редакторе. Вы будете экономить все больше времени и круто выглядеть в глазах коллег.

5.1. ЗАЧЕМ НУЖЕН РЕФАКТОРИНГ?

Изменения неизбежны, а изменения кода неизбежны тем более. Рефакторинг — это не просто изменение кода. Его преимущества:

- *Уменьшение количества повторов и облегчение повторного использования кода.* Можно переместить и сделать общедоступным класс, чтобы он мог повторно использоваться другими компонентами. Точно так же можно извлекать методы из кода и делать их доступными для повторного использования.
- *Приближение кода к его мысленной модели.* Именование очень важно. Некоторые имена не так понятны, как другие. Переименование — часть процесса рефакторинга, которая помогает добиться дизайна, лучше соответствующего мысленной модели.
- *Упрощение кода для понимания и обслуживания.* Можно упростить код, разделив длинные функции на меньшие, которые удобнее обслуживать. Модель станет понятнее, если сложные типы данных разбиты на мелкие атомарные составляющие.
- *Предотвращение определенных классов ошибок.* Некоторые операции рефакторинга, такие как преобразование класса в структуру, могут предотвратить ошибки, связанные с `null`, что мы обсуждали в главе 2. Точно так же разрешение ссылок, допускающих значение `null`, и изменение типов данных на `non-null` могут предотвратить ошибки в результате рефакторинга.
- *Возможность подготовиться к значительным изменениям архитектуры.* Крупные изменения можно внести быстрее, если заранее подготовить код. Вы узнаете, как это сделать, в следующем разделе.

- *Исключение жестких частей кода.* Внедряя зависимости, можно избавиться от них и получить дизайн без жестких связей.

Разработчики обычно считают рефакторинг рутинной повседневной работой. Но рефакторинг — это также отдельная задача, которую вы выполняете, даже если не написали ни строчки кода. Можно даже провести рефакторинг непонятного кода, чтобы его было легче прочесть. Ричард Фейнман однажды сказал: «Если вы действительно хотите разобраться в предмете, напишите о нем книгу». Точно так же можно разобраться в коде, проведя его рефакторинг.

Для простых операций рефакторинга вообще не требуются инструкции. Вам нужно переименовать класс? Вперед. Извлечь методы или интерфейсы? Элементарно. Они есть даже в контекстном меню Visual Studio, которое в Windows можно вызвать с помощью **Ctrl-.** (**Ctrl-точка**). В большинстве случаев операции рефакторинга вообще не влияют на надежность кода. Однако когда дело доходит до значительных изменений архитектуры кодовой базы, может понадобиться помощь.

5.2. ИЗМЕНЕНИЯ АРХИТЕКТУРЫ

Попытки провести масштабные изменения архитектуры за один раз почти никогда ничем хорошим не заканчиваются. Не столько из-за технической сложности, сколько вследствие того, что большие изменения порождают большое количество ошибок и проблем с интеграцией. Тем более что работы много и выполняется она долго.

Под проблемами интеграции я подразумеваю невозможность интегрировать изменения от других разработчиков (рис. 5.1) в течение длительного времени. Это ставит вас в затруднительное положение. Как вы поступите — будете до конца работы над проектом вручную применять все изменения, внесенные в код за это время, и устранять конфликты самостоятельно или попросите других членов команды прекратить работу, пока вы не закончите свои изменения? Эта проблема возникает при рефакторинге. Ее не будет при новой разработке, потому что там вероятность конфликта с другими разработчиками намного меньше из-за отсутствия самого продукта. Таким образом, поэтапный подход лучше единовременного рефакторинга.

Чтобы создать дорожную карту, нужно знать, куда вы движетесь и где находитесь. Какой конечный результат вам необходим? Конечно, невозможно представить все сразу, потому что большой программный продукт действительно сложно уложить в голове. Но у вас может быть определенный список требований.

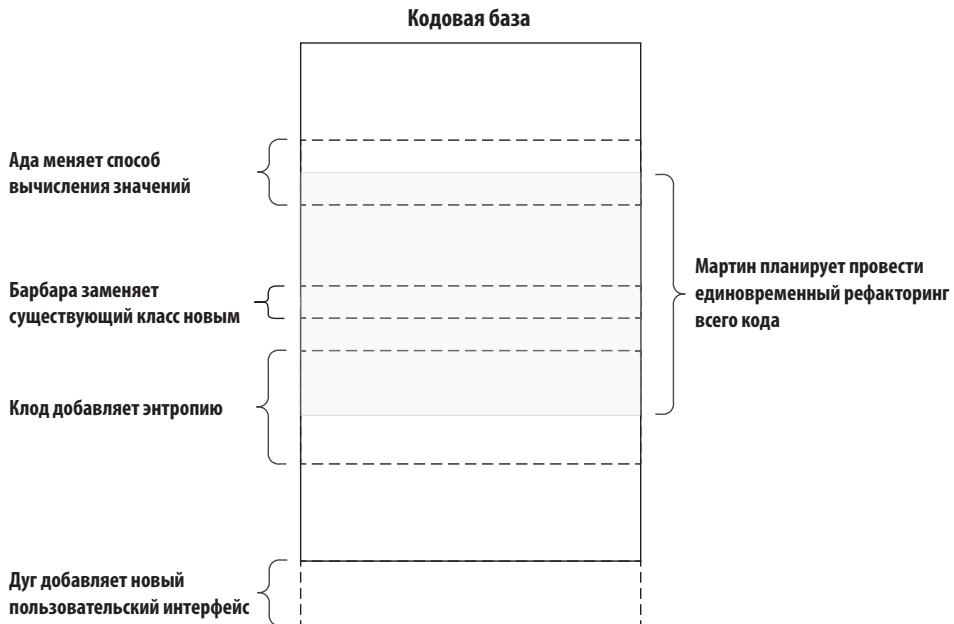


Рис. 5.1. Почему единовременный рефакторинг крупного проекта — плохая идея

Рассмотрим пример. У Microsoft есть две разновидности .NET. Первая — это .NET Framework, которому уже несколько десятков лет, а вторая — просто .NET (ранее известная как .NET Core), выпущенная в 2016 году. На момент написания этой книги обе по-прежнему поддерживаются, но очевидно, что Microsoft намерена развивать .NET и отказаться от .NET Framework. Очень вероятно, что вы столкнетесь с проектом, требующим миграции с .NET Framework на .NET.

Помимо пункта назначения, вам нужно знать, где вы находитесь. Это напоминает мне анекдот об одном генеральном директоре, который летел на вертолете и заблудился в тумане. Экипаж заметил здание и человека, стоящего на балконе. «У меня идея, — сказал директор. — Подлетим поближе». Они подлетели ближе к человеку, и директор крикнул: «Эй! Знаешь, где мы?». И получил ответ: «Да, вы в вертолете!». — «Хорошо, тогда мы должны быть в кампусе колледжа, а это, должно быть, инженерный факультет!». Человек на балконе удивился: «Как вы догадались?» — «Ваш ответ был технически верным, но совершенно бесполезным!». — «Тогда вы, должно быть, генеральный директор!» — воскликнул инженер. Теперь удивился директор: «Как вы это узнали?» — «Вы заблудились, не имели представления, где находитесь и куда направиться, но виноватым назначили меня!».

.NET FRAMEWORK МЕРТВ, ДА ЗДРАВСТВУЕТ .NET!

.NET имел большое значение в 1990-х годах, когда росла популярность интернета. Выходил даже журнал под названием *.net*, который был посвящен интернет-технологиям и служил медленной версией Google. Просмотр веб-страниц там обычно называли «серфингом в сети», «путешествием по информационной супермагистрали», «подключением к киберпространству» и другими метафорами.

.NET Framework — оригинальная программная экосистема, созданная в конце 1990-х годов, чтобы облегчить жизнь разработчиков. Она поставлялась со средой выполнения, стандартными библиотеками, компиляторами для C#, Visual Basic и затем языков F#. Java-эквивалентом в .NET Framework был JDK (Java Development Kit), включающий среду выполнения Java, компилятор языка Java, виртуальную машину Java и другие вещи, в название которых входило *Java*.

Со временем появились другие разновидности .NET, которые не были напрямую совместимы с .NET Framework, такие как .NET Compact Framework и Mono. Чтобы обеспечить совместное использование кода между различными фреймворками, Microsoft создала общую спецификацию API, определяющую совокупность общих функций .NET, под названием *.NET Standard*. У Java подобной проблемы не возникало, потому что Oracle со своей армией юристов успешно уничтожила все несовместимые альтернативы.

Позже Microsoft создала новую, кроссплатформенную версию .NET Framework. Первоначально она называлась .NET Core и, начиная с .NET 5, была переименована просто в .NET. Она несовместима напрямую с .NET Framework, но может взаимодействовать с ней с использованием общей спецификации .NET Standard.

Пока .NET Framework продолжает поддерживаться, но лет через пять мы ее можем уже не увидеть. Я настоятельно рекомендую ориентироваться на .NET, а не .NET Framework, поэтому подобрал пример сценария миграции.

Я не могу удержаться, чтобы не представить, как директор прыгает на балкон и между ним и инженером завязывается драка в стиле «Матрицы», с катанами. А все потому, что пилот не умел читать GPS, вместо этого отрабатывая на учениях маневры точного подлета к балконам.

Предположим, что у нас есть веб-сайт анонимных микроблогов под названием Blabber, написанный на .NET Framework и ASP .NET, и нам нужно перенести его на новую платформу .NET и ASP .NET Core. К сожалению, ASP .NET Core и ASP .NET несовместимы на двоичном уровне и лишь частично совместимы на уровне исходного кода. Код платформы включен в исходный код книги.

Я не буду приводить здесь полный код, потому что ASP .NET поставляется с довольно большим количеством шаблонов, но набросаю детали, которые помогут создать дорожную карту рефакторинга. Вам не нужно разбираться в архитектуре ASP .NET или общих принципах работы веб-приложений, чтобы понять процесс рефакторинга, потому что они не имеют к нему прямого отношения.

5.2.1. Выделение компонентов

Лучший способ работы с масштабным рефакторингом — выделить в коде различные по семантике компоненты. Разделим наш код на несколько частей, чтобы провести рефакторинг. Наш проект представляет собой приложение ASP .NET MVC, куда мы добавили несколько классов моделей и контроллеры. У нас может быть примерный список компонентов, как на рис. 5.2. Он не обязательно должен быть точным, это может быть предварительный список, который со временем изменится.

После того как вы составили список компонентов, оцените, сколько из них вы можете передать непосредственно в место назначения, как в примере .NET 5. Обратите внимание, что *место назначения* — это целевое состояние, обозначающее конечный результат. Можно ли перевести компоненты в целевое состояние, ничего не сломав? Будут ли они нуждаться в доработке? Ответьте на эти вопросы для каждого компонента и используйте ответы для расстановки приоритетов. На текущем этапе достаточно предположений. Можете составить таблицу оценки работ, подобную табл. 5.1.

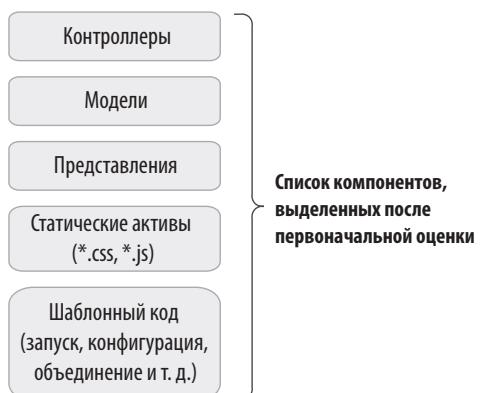


Рис. 5.2. Первоначальная оценка компонентов

Таблица 5.1. Оценка относительной стоимости и рисков операций с компонентами

Компонент	Требуемые изменения	Риск конфликта с другим разработчиком
Контроллеры	Минимальные	Высокий
Модели	Нет	Средний
Представления	Минимальные	Высокий
Статические активы	Небольшие	Низкий
Шаблонный код	Переписывание	Низкий

ЧТО ТАКОЕ MVC?

Всю историю computer science можно назвать борьбой с энтропией, или *спагетти*, как ее называют веряющие в Летающего Спагетти-монстра, создателя всей энтропии. MVC — это идея делить код на три части, чтобы избежать слишком большой взаимозависимости, иначе говоря, спагетти-кода. Одна часть определяет, как будет выглядеть пользовательский интерфейс, другая моделирует бизнес-логику, третья координирует первые две. Эти части называются соответственно «представление» (view), «модель» (model) и «контроллер» (controller). Есть много схожих способов деления кода приложения на логически независимые части, такие как MVVM (модель — model, представление — view, модель представления —.viewmodel) или MVP (модель, представление, представитель — presenter), но в их основе лежит один принцип — разделение несвязанных задач.

Такое разделение эффективно при написании кода, создании тестов и рефакторинге, потому что зависимости между выделенными слоями становятся более управляемыми. Но как отметили Дэвид Вулперт (David Wolpert) и Уильям Макреди (William Macready), сформулировавшие теорему о бесплатном обеде, бесплатных обедов не бывает. Желая получить преимущества MVC, обычно приходится писать немного больше кода, работать с большим количеством файлов, заводить больше подкаталогов и чаще ругаться в монитор. Однако в целом вы станете быстрее и эффективнее.

5.2.2. Оценка объема работы и риска

Как узнать, какой объем работы потребуется выполнить? Необходимо хотя бы в общих чертах представлять, как работают оба фреймворка, чтобы ответить на этот вопрос. Важно знать место назначения до того, как начать движение. Некоторые предположения могут оказаться ошибочными, и это нормально. Но проведение

оценки полезно, потому что вы расставите приоритеты в работе и снизите рабочую нагрузку. При этом вам довольно долго не понадобится ничего ломать.

Например, я знаю, что контроллеры и представления потребуют минимальных усилий, потому что разница в их синтаксисе в обоих фреймворках незначительна. Кроме того, я предполагаю, что придется немного повозиться с синтаксисом некоторых HTML-хелперов или элементов контроллера, но скорее всего, особых проблем здесь не возникнет. Я знаю, что статическое содержимое перемещается в папку `wwwroot/` в ASP .NET Core, что не требует больших усилий, но передать напрямую его нельзя. Наконец, я знаю, что код запуска и конфигурирования был полностью переработан в ASP .NET Core, а это значит, что мне придется писать его с нуля.

Я предполагаю, что другие члены команды будут работать над функциями, и следовательно, с контроллерами, представлениями и моделями. Существующие модели вряд ли будут меняться так же часто, как бизнес-логика или внешний вид функций, поэтому я присвою моделям средний риск. При этом риски для контроллеров и представлений должны оцениваться выше.

Помните, что другие разработчики работают над кодом, пока вы занимаетесь рефакторингом, поэтому вам необходимо как можно раньше интегрировать свои результаты в их рабочий процесс, не нарушая его. Оптимальный компонент для этого — модели в табл. 5.1. Несмотря на высокую вероятность конфликта, модель требует минимальных изменений, поэтому любые конфликты должны разрешаться легко.

Как одновременно менять существующий код и писать новый для одного и того же компонента? Перенести его в отдельный проект. Я объяснял это в главе 3, когда говорил о том, как избавиться от зависимостей, чтобы сделать структуру проекта открытой для изменений.

5.2.3. Престиж

Проводить рефакторинг втайне от коллег — все равно что менять колесо, одновременно двигаясь по шоссе. Вы как будто становитесь фокусником, незаметно меняющим старую архитектуру на новую. Самый эффективный инструмент, который вы можете использовать для этого, — извлечение разделяемых частей кода, как показано на рис. 5.3.

Конечно, разработчики не могут не заметить новый проект в репозитории, но если заранее сообщить им об изменениях, которые вы собираетесь внести, и они легко адаптируются к ним, у вас не будет проблем.

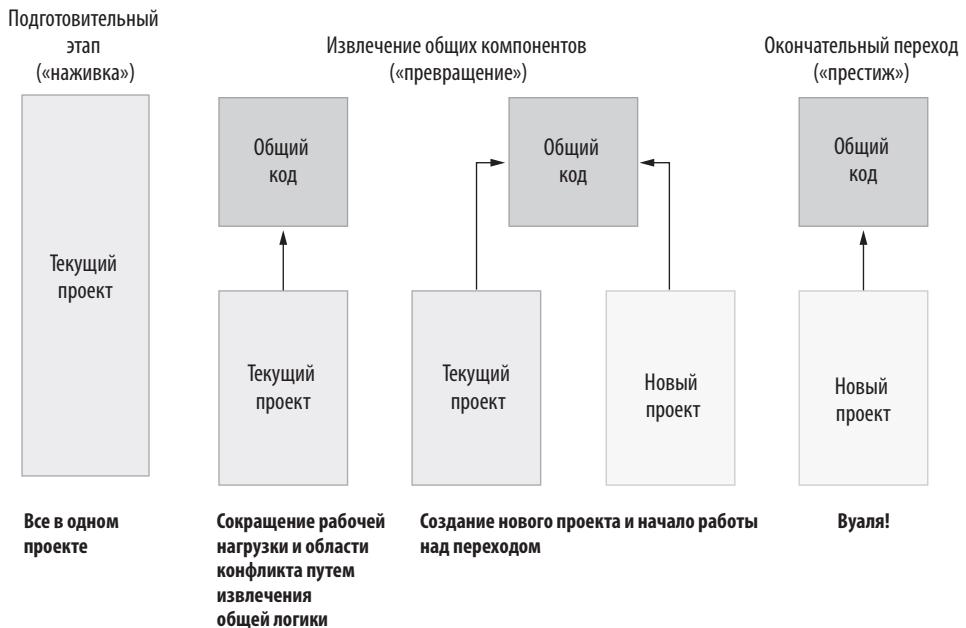


Рис. 5.3. Фокус с незаметным рефакторингом¹

Вы создаете отдельный проект, в нашем примере — Blabber.Models, перемещаете в него классы `models`, а затем добавляете ссылку в веб-проект. Код продолжит работать, как раньше, но новый код нужно будет добавлять в Blabber.Models, а не в Blabber, и ваши коллеги должны об этом знать. Затем можно будет создать новый проект и добавить в него ссылку на Blabber.Models. Получилась дорожная карта, как на рис. 5.4.

Мы проделываем все это, чтобы тратить как можно меньше усилий и как можно дольше синхронизироваться с основной веткой. Этот метод позволяет проводить рефакторинг большее время, одновременно выполняя другие, более срочные задачи. Он очень похож на точки сохранения в видеоиграх, когда в *God of War* можно в сотый раз проходить одну и той же битву с Валькирией, а не возвращаться к началу всей игры снова и снова. Все, что интегрируется в основную ветку, не нарушая сборку, считается проверенным и не требующим повторения.

¹ Названия этапов взяты из фильма Нолана «Престиж». Престижем называется третья, самая трудная часть фокуса, после наживки и превращения: «если просто что-то исчезло, этого мало. Нужно его вернуть обратно». — Примеч. ред.

Планирование нескольких этапов интеграции — самый эффективный способ провести масштабный рефакторинг.

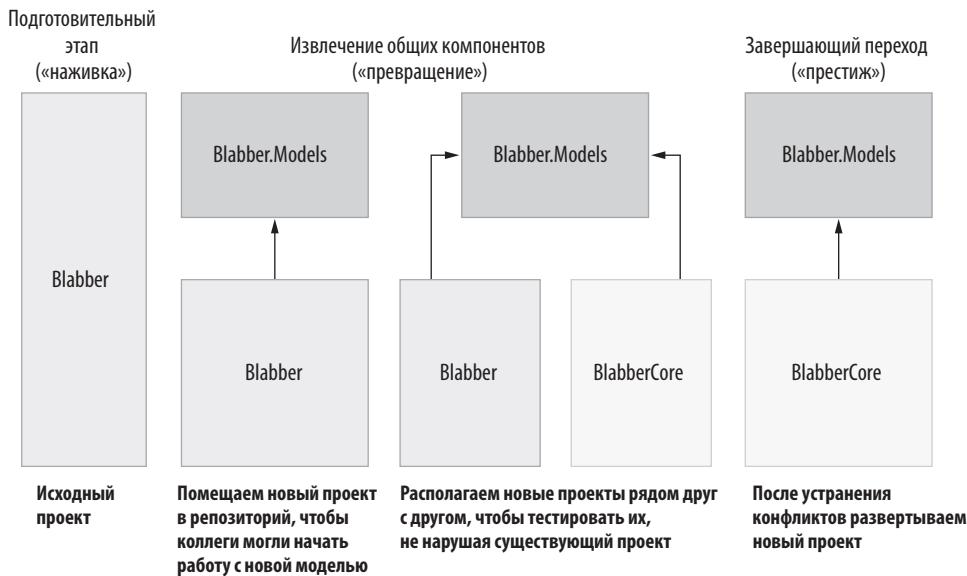


Рис. 5.4. Дорожная карта рефакторинга проекта

5.2.4. Рефакторинг, чтобы упростить рефакторинг

При перемещении кода из одного проекта в другой вы столкнетесь с зависимостями, от которых нелегко избавиться. В нашем примере часть кода может зависеть от веб-компонентов, причем переносить их в общий проект бессмысленно, поскольку новый проект BlabberCore не будет работать со старыми веб-компонентами.

В таких случаях на помощь приходит композиция. Извлечем интерфейс основного проекта и передадим его реализации вместо фактической зависимости.

Текущая реализация Blabber использует хранилище в оперативной памяти для контента, размещенного на веб-сайте. Поэтому при каждом перезапуске сайта весь контент платформы теряется. Это имеет смысл для постмодернистского художественного проекта, но обычно пользователи ожидают хоть какого-то постоянства. Предположим, что нам нужен либо Entity Framework, либо Entity Framework Core в зависимости от имеющейся инфраструктуры, но мы по-прежнему хотим использовать общий код доступа к БД двух проектов, пока

выполняется миграция, поэтому на последнем отрезке миграции фактический объем работы будет заметно меньше.

Внедрение зависимостей

Можно абстрагировать ненужную зависимость, создав для нее интерфейс и получив ее реализацию в конструкторе. Этот метод называется *внедрением зависимостей* (dependency injection, DI). Не путайте его с *инверсией зависимостей* (dependency inversion) — широко разрекламированным принципом, который на самом деле предполагает зависимость от абстракций.

Термин «внедрение зависимостей» также не совсем точен. «Внедрение» подразумевает некое вмешательство в структуру, но ничего подобного на самом деле не происходит. Возможно, *получение зависимостей* — более точное определение, так как речь идет о получении зависимостей во время инициализации, например в конструкторе. DI также называют *инверсией управления* (inversion of control, IoC), что еще больше сбивает с толку. Типичное внедрение зависимостей — это изменение дизайна продукта, как показано на рис. 5.5. Без внедрения зависимостей вы инстанцируете экземпляры зависимых классов в своем коде. При внедрении зависимостей вы получаете классы, от которых зависите, в конструкторе.



Рис. 5.5. Как внедрение зависимостей меняет дизайн класса

Рассмотрим все вышесказанное на примере простого и абстрактного кода, чтобы сосредоточиться на реальных различиях. В листинге 5.1 представлен программный код верхнего уровня C# 9.0 без метода `main` или класса программы

как такового. На самом деле можно записать этот код в файл `.cs` в папке проекта и сразу же запустить его, ничего не добавляя. Обратите внимание, как `class A` инициализирует экземпляр `class B` каждый раз, когда вызывается метод `X`.

Листинг 5.1. Код, использующий прямую зависимость

```
using System;

var a = new A(); ← Здесь экземпляр A создается в основном коде
a.X();

public class A {
    public void X() {
        Console.WriteLine("X got called");
        var b = new B(); ← Класс A создает экземпляр класса B
        b.Y();
    }
}

public class B {
    public void Y() {
        Console.WriteLine("Y got called");
    }
}
```

При внедрении зависимостей код получает свой экземпляр `class B` в своем конструкторе и через интерфейс, поэтому связь между `class A` и `class B` отсутствует (листинг 5.2). Однако существует различие в соглашениях. Поскольку мы переместили код инициализации `class B` в конструктор, то вместо создания нового экземпляра, как в листинге 5.1, всегда используется уже имеющийся экземпляр `B`. На самом деле это хорошо, потому что снижает нагрузку на сборщик мусора, но может вызвать неожиданное поведение, если состояние класса со временем изменится. Возможно, вы нарушаете поведение. Вот почему полезно обеспечить покрытие тестами.

Реализация в листинге 5.2 позволяет полностью удалить код для `B` и переместить его в совершенно другой проект, не нарушая код `A`, до тех пор, пока существует интерфейс (`IB`). Более того, мы можем переместить вместе с `B` все, что ему необходимо. Это достаточно высокая степень свободы.

Листинг 5.2. Код с внедрением зависимостей

```
using System;

var b = new B(); ← Вызывающий объект инициализирует класс B
var a = new A(b); ← В передается классу A в качестве параметра
a.X();
```

```

public interface IB {
    void Y();
}

public class A {
    private readonly IB b; ← Место хранения экземпляра B
    public A(IB b) {
        this.b = b;
    }
    public void X() {
        Console.WriteLine("X got called");
        b.Y(); ← Вызов общего экземпляра B
    }
}

public class B : IB {
    public void Y() {
        Console.WriteLine("Y got called");
    }
}

```

Теперь применим эту технику в примере с Blabber и изменим код так, чтобы он использовал хранилище базы данных вместо памяти и контент сохранялся после перезапуска. Вместо зависимости от конкретной реализации механизма БД (в данном случае – Entity Framework и EF Core) можно получить разработанный нами интерфейс, который обеспечивает требуемую функциональность компонента. Благодаря этому два проекта с разными технологиями смогут использовать одну и ту же кодовую базу, даже если общий код зависит от конкретной функциональности БД. Для этого создадим общий интерфейс IBlabDb, который указывает на функциональность базы данных, и используем его в общем коде. Две разные реализации используют один и тот же код; общий код использует разные технологии доступа к БД. Схема реализации представлена на рис. 5.6.

Вначале изменим реализацию BlabStorage в Blabber.Models, где мы провели рефакторинг, чтобы перенести операции в интерфейс. Реализация класса BlabStorage в памяти выглядит так, как показано в листинге 5.3. Класс хранит статический экземпляр списка, используемый всеми запросами, поэтому он предусматривает блокировку, чтобы избежать несогласованности. Согласованность свойства `Items` нам не важна, потому что мы только добавляем элементы в этот список, но не удаляем их (в противном случае ее необходимо было бы учитывать, чтобы избежать проблем). Обратите внимание, что мы используем `Insert` вместо `Add` в методе `Add()`, потому что это позволяет, не прибегая к сортировке, хранить публикации в порядке их создания, когда более поздние записи располагаются сверху.

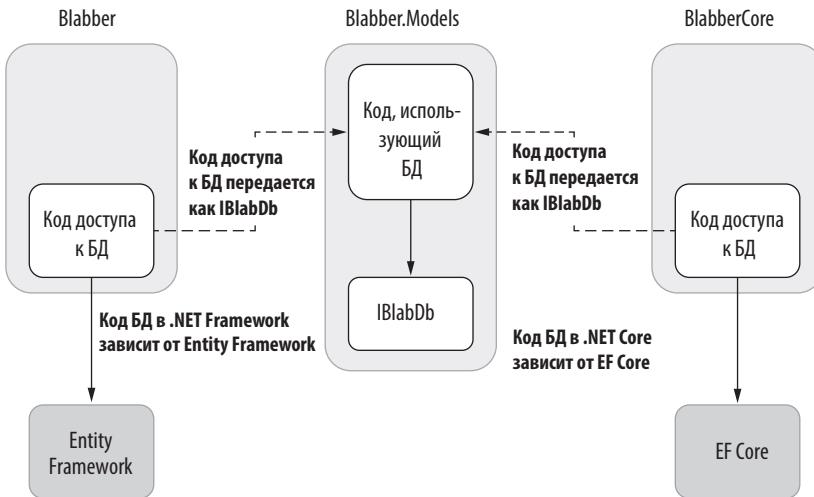


Рис. 5.6. Использование различных технологий в общем коде с внедрением зависимостей

Листинг 5.3. Первоначальная версия BlabStorage

```
using System.Collections.Generic;

namespace Blabber.Models {
    public class BlabStorage {
        public IList<Blab> items = new List<Blab>(); ← Создание пустого списка по умолчанию
        public IEnumerable<Blab> Items => items;
        public object lockObject = new object(); ← Используем объект блокировки, чтобы разрешить параллелизм
        public static readonly BlabStorage Default =
new BlabStorage(); ← Экземпляр-одиночка по умолчанию, который используется везде

        public BlabStorage() {
        }

        public void Add(Blab blab) {
            lock (lockObject) { ← Самый последний элемент находится вверху
                items.Insert(0, blab);
            }
        }
    }
}
```

При внедрении зависимостей мы удаляем все, что связано со списками в памяти, и вместо этого используем абстрактный интерфейс для всего, что связано с базой данных. Новая версия `BlabStorage` представлена в листинге 5.4. Здесь мы удаляем все, что связано с логикой хранения данных, и класс `BlabStorage`

фактически сам становится абстракцией. Кажется, что он больше ничего не делает, но по мере добавления более сложных задач мы сможем делить логику между двумя проектами. Неплохо для примера.

Мы сохраняем зависимость в закрытом и доступном только для чтения поле `db`. Полезно помечать поля ключевым словом `readonly`, если они не будут изменяться после создания объекта, чтобы компилятор мог определить, что вы или кто-то из ваших коллег случайно изменили его вне конструктора.

Листинг 5.4. Класс BlabStorage с внедрением зависимостей

```
using System.Collections.Generic;

namespace Blabber.Models {
    public interface IBlabDb {
        IEnumerable<Blab> GetAllBlabs();
        void AddBlab(Blab blab);
    }
}

public class BlabStorage {
    private readonly IBlabDb db;

    public BlabStorage(IBlabDb db) {
        this.db = db;
    }

    public I Enumerable<Blab> GetAllBlabs() {
        return db.GetAllBlabs();
    }

    public void Add(Blab blab) {
        db.AddBlab(blab);
    }
}
```

Наша фактическая реализация называется `BlabDb`, она реализует интерфейс `IBlabDb` и находится в проекте `BlabberCore`, а не `Blabber.Models`. Выбор базы данных SQLite обусловлен тем, что она не требует установки стороннего программного обеспечения и ее можно запустить сразу же. SQLite — последний дар Бога миру перед тем, как он разочаровался в человечестве. Шучу, Ричард Кипп (Richard Kipp) создал ее до того, как разочаровался в человечестве. Наш проект `BlabberCore` реализует ее в EF Core, как показано в листинге 5.5.

Возможно, вы не знакомы с EF Core, Entity Framework или ORM (объектно-реляционным отображением), но это и не обязательно. Все довольно просто, как видите. Метод `AddBlab` просто создает новую запись базы данных в памяти

и отложенную вставку в таблицу `Blabs`, после чего вызывает `SaveChanges` для записи изменений в базу данных. Точно так же метод `GetAllBlabs` просто получает все записи из базы данных, упорядоченные по дате от более поздней к более ранней. Обратите внимание, как следует преобразовать даты в UTC, чтобы убедиться, что информация о часовом поясе не потеряна, потому что SQLite не поддерживает типы `DateTimeOffset`. Независимо от того, сколько лучших практик вы изучите, всегда будут находиться сценарии, в которых они не работают. Тогда вам придется импровизировать, приспособливаться и преодолевать.

Листинг 5.5. Версия класса `BlabDb` для EF Core

```
using Blabber.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Blabber.DB {
    public class BlabDb : IBlabDb {
        private readonly BlabberContext db; ← Контекст БД EF Core

        public BlabDb(BlabberContext db) { ← Получение контекста посредством
            this.db = db;                внедрения зависимостей
        }

        public void AddBlab(Blab blab) {
            db.Blabs.Add(new BlabEntity() {
                Content = blab.Content,
                CreatedOn = blab.CreatedOn.UtcDateTime, ← Преобразование DateTimeOffset
                });                                     в тип, совместимый с БД
            db.SaveChanges();
        }

        public IEnumerable<Blab> GetAllBlabs() {
            return db.Blabs
                .OrderByDescending(b => b.CreatedOn)
                .Select(b => new Blab(b.Content,
                    new DateTimeOffset(b.CreatedOn, TimeSpan.Zero))) ← Преобразование
                .ToList();                                времени БД
            }
        }
    }
}
```

Во время рефакторинга нам удалось внедрить в проект серверную часть хранилища базы данных, не нарушая процесс разработки. Мы использовали внедрение зависимостей, чтобы избежать прямых зависимостей. Что еще более важно, наш контент теперь сохраняется после перезапуска и завершения сессии, как показано на рис. 5.7.

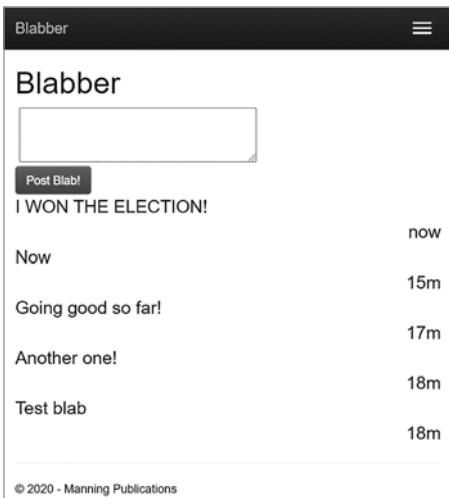


Рис. 5.7. Скриншот сайта Blabber, запущенного с базой данных SQLite

5.2.5. Финальное усилие

Вы можете извлечь все компоненты, которые могут использоваться совместно в старом и новом проектах, но рано или поздно встретите фрагмент кода, который невозможно использовать и там и там. Например, код контроллера не нужно менять при переходе от ASP .NET к ASP .NET Core, потому что синтаксис не изменится, но его невозможно сделать общим для обеих платформ, поскольку у них совершенно разные типы. Контроллеры MVC ASP .NET являются производными от `System.Web.Mvc.Controller`, а контроллеры ASP .NET Core — от `Microsoft.AspNetCore.Mvc.Controller`. В теории можно абстрагировать реализацию контроллера от интерфейса и создать пользовательские классы, которые будут использовать этот интерфейс, а не являться прямыми потомками класса контроллера, но это потребует слишком больших усилий. Прежде чем изобретать нетривиальное решение проблемы, спросите себя: «А оно того стоит?». Нетривиальная разработка всегда должна учитывать затраты.

Таким образом, в какой-то момент вам придется рискнуть, вступив в конфликт с другими разработчиками, и перенести код в новую базу. Я называю это *финальным усилием*, которое займет меньше времени, если выполнена подготовительная работа. Она позволит провести будущие операции по рефакторингу быстрее, и в конце вы получите разделенный дизайн. Это хорошая инвестиция.

В нашем примере компонент моделей составляет очень малую часть проекта, поэтому экономия незначительна. Однако, как правило, в крупных проектах объем совместно используемого кода велик, и экономия вашего времени будет заметной.

Прикладывая финальное усилие, вам необходимо перенести весь код и содержимое в новый проект, а затем сделать так, чтобы все заработало. Я добавил в примеры кода отдельный проект BlabberCore, который содержит новый код .NET, чтобы показать, как некоторые конструкции переводятся в .NET Core.

5.3. НАДЕЖНЫЙ РЕФАКТОРИНГ

IDE очень заботится, чтобы вы не сломали код, случайно выбрав не те пункты меню. Если вы отредактируете имя вручную, любой код, ссылающийся на это имя, перестанет работать. Но если вы используете функцию переименования IDE, все ссылки также будут переименованы. Однако это не панацея от ошибок. Существует много способов сослаться на имя без ведома компилятора. Например, можно создать экземпляр класса с помощью строки. В примере кода микроблога Blabber мы ссылаемся на каждую часть контента как на «блабы»¹, и у нас есть класс Blab, который определяет контент.

Листинг 5.6. Класс, представляющий контент

```
using System;

namespace Blabber
{
    public class Blab
    {
        public string Content { get; private set; } ← Конструктор гарантирует
        public DateTimeOffset CreatedOn { get; private set; } отсутствие недопустимых
        public Blab(string content, DateTimeOffset createdOn) { блаов
            if (string.IsNullOrWhiteSpace(content)) {
                throw new ArgumentException(nameof(content));
            }
            Content = content;
            CreatedOn = createdOn;
        }
    }
}
```

¹ Аналогия с «твитами» в Twitter. — Примеч. ред.

Обычно мы создаем экземпляры классов, используя оператор `new`, но можно создать экземпляр класса `Blab`, используя отражение, например, когда неизвестно, какой класс создается во время компиляции:

```
var blab = Activator.CreateInstance("Blabber.Models",
    "Blabber", "test content", DateTimeOffset.Now);
```

Всякий раз, когда мы ссылаемся на имя в строке, мы рискуем нарушить работу кода после переименования, потому что IDE не может отслеживать содержимое строк. Надеюсь, это перестанет быть проблемой, когда за проверку кода возмется всесильный искусственный интеллект. Я не знаю, почему, представляя будущее, мы воображаем, что сами будем выполнять всю работу, а ИИ будет только оценивать ее. Не проще ли ИИ занять наше место? Ведь судя по всему, он гораздо умнее, чем мы думаем.

Пока ИИ не захватил мир, IDE не гарантирует, что рефакторинг пройдет без сучка и задоринки. Да, у вас есть некоторое пространство для маневра, например использование таких конструкций, как `nameof()`, чтобы ссылаться на типы, а не жестко кодировать их в строках, о чем я говорил в главе 4. Однако это помогает не всегда.

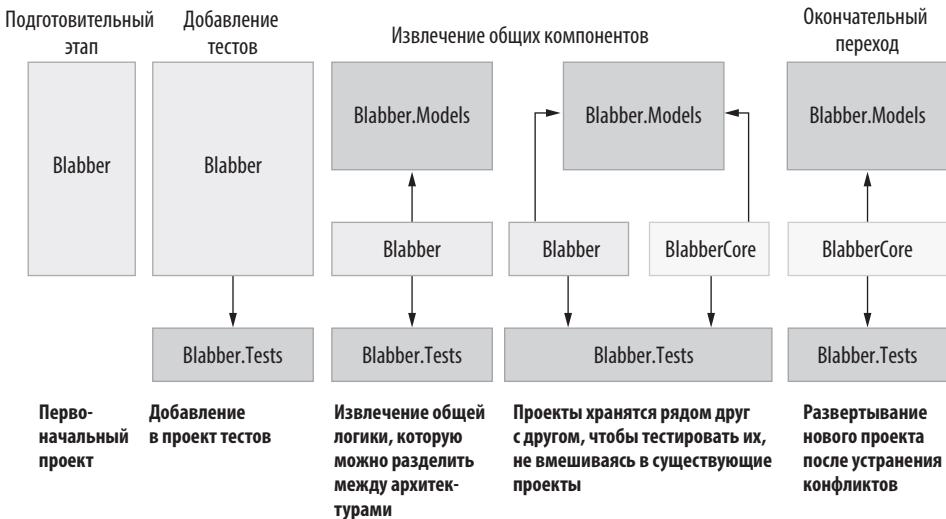


Рис. 5.8. Надежный рефакторинг с помощью тестов

Секрет надежного рефакторинга — тестирование. Хорошее тестовое покрытие кода обеспечивает гораздо больше свободы для его изменения. Поэтому обычно

разумно начинать долгосрочный проект рефакторинга с разработки недостающих тестов для соответствующего фрагмента кода. Если вернуться к примеру изменения архитектуры в главе 3, то более реалистичная дорожная карта будет включать добавление недостающих тестов ко всей архитектуре. Мы пропустили этот шаг, потому что наша кодовая база была чрезвычайно мала и проста, чтобы тестиировать ее вручную (например, запустить приложение, опубликовать блаф и посмотреть, появится ли он). На рис. 5.8 показана модифицированная версия дорожной карты проекта, которая включает этап добавления тестов, чтобы можно было провести надежный рефакторинг.

5.4. КОГДА РЕФАКТОРИНГ НЕ НУЖЕН

Преимущество рефакторинга в том, что он заставляет думать, как улучшить код. Недостаток рефакторинга в том, что в какой-то момент он может стать скорее целью, чем средством, почти как Emacs. Для тех, кто не в курсе, Emacs – это текстовый редактор, среда разработки, веб-браузер, операционная система и постапокалиптическая ролевая игра, потому что кто-то не смог вовремя сдержать лошадей. То же самое может произойти при рефакторинге, если вы будете пытаться улучшить каждый фрагмент кода. Вы впадете в зависимость и станете придумывать оправдания для изменений ради самих изменений, не пользуясь их результатами. Вы будете тратить не только свое время, но и время команды, потому что ей придется адаптироваться к каждому изменению.

На улицах надо понимать, что такое хороший код и какова его ценность. Да, код может заржаветь, если к нему не притрагиваться, но если он достаточно хороший, то легко выдержит и это испытание. Вот признаки того, что рефакторинг не требуется:

- Вы хотите сделать код «более элегантным»? Это очень красноречивый сигнал, потому что элегантность не просто субъективна, но и неопределенна и, следовательно, бессмысленна. Придется найти веские аргументы и убедительные преимущества рефакторинга, например: «Так компонент будет проще использовать, поскольку станет меньше шаблонного кода, который нужно писать каждый раз», «Это подготовит к переходу на новую библиотеку», «Это устранит зависимость от компонента X» и т. д.
- Ваш целевой компонент имеет минимальный набор зависимостей? Значит, его будет легко переместить или реорганизовать в будущем. Упражнения в рефакторинге могут оказаться бесполезными для выявления жестких частей кода. Лучше отложите их до тех пор, пока не придумаете более основательный план улучшения.

- Тестовое покрытие кода неполное? В этом случае рефакторинг однозначно лучше не проводить, особенно если у компонента много зависимостей. Если вы хотите тестировать компонент, потому что не уверены в своих действиях, то просто не предпринимайте их.
- Имеется общая зависимость? В таком случае даже при хорошем покрытии тестами и аргументированном обосновании изменений вы повлияете на работу всей команды. Лучше отложите рефакторинг, если ожидаемая выгода не компенсирует возможных затрат.

Если вы нашли какой-то из этих признаков в своем коде, не проводите рефакторинг или, по крайней мере, отложите его. Приоритеты всегда относительны, а в море рыбы всегда больше.

ИТОГИ

- Проводите рефакторинг, чтобы получать дополнительные преимущества в сравнении с теми, что дает текущая версия кода.
- Большие изменения архитектуры вносите поэтапно.
- Используйте тестирование, чтобы уменьшить количество проблем, которые могут возникнуть в ходе масштабного рефакторинга.
- Оценивайте не только затраты, но и риски.
- Когда работаете над крупными изменениями архитектуры, составляйте дорожную карту, хотя бы мысленную.
- Внедряйте зависимости, чтобы устраниТЬ барьеры. Уменьшайте жесткость кода с помощью этой же техники.
- Не проводите рефакторинг, если затраты на него больше предполагаемой выгода.

6

Все внимание безопасности

В этой главе

- ✓ Что такое безопасность
- ✓ Эффективность моделей угроз
- ✓ Как избежать распространенных ошибок безопасности, таких как внедрение SQL, CSRF, XSS и переполнение
- ✓ Как ограничить возможности злоумышленников
- ✓ Правильное хранение секретов

Вопрос безопасности остро стоит со времен злосчастного инцидента в Трое, древнем городе на территории современной Западной Турции. Троянцы считали городские стены неприступными и чувствовали себя в безопасности, но, как и современные социальные платформы, недооценили навыки социальной инженерии своих противников. Греки отступили и оставили в подарок огромного деревянного коня. Довольные троянцы взяли статую в город. Ночью греческие солдаты, прятавшиеся внутри коня, вышли и открыли ворота своим войскам. Армия греков ворвалась в город, и Троя пала. По крайней мере, таково описание событий в итоговом аналитическом отчете, составленном Гомером о, возможно, первом в истории случае *безответственного раскрытия уязвимостей*.

- *Безопасность* — это одновременно и широкий, и глубокий термин. Например, история с троянцами включает в себя психологический аспект. Первое, что вы должны понять о безопасности, — она касается не только программного обеспечения или информации, но также людей и окружающей среды. Поскольку тема безопасности очень широка, прочитав эту краткую главу, вы не станете в ней экспертом, но будете лучше понимать основы и повысите свои навыки разработки.

ИТОГОВАЯ АНАЛИТИЧЕСКАЯ ОТЧЕТНОСТЬ И ОТВЕТСТВЕННОЕ РАСКРЫТИЕ УЯЗВИМОСТЕЙ

Итоговый аналитический отчет (постмортем) — это объемный документ, который обычно составляют после значительного инцидента безопасности. Его цель — создание видимости, что руководство ничего не утаивает, предоставляя как можно больше подробностей, и скрытие факта, что оно облажалось.

Ответственное раскрытие уязвимостей — это практика публикации информации об уязвимостях системы безопасности после того, как компания, которая изначально не инвестировала в выявление проблемы, получила достаточно времени для ее решения. Компании изобрели этот термин, чтобы придать своим действиям побольше эмоциональности, а исследователь проблем безопасности чувствовал себя виноватым. Сами уязвимости безопасности называют *инцидентами*, причем никогда не *безответственными*. Я считаю, что такое раскрытие лучше было бы назвать, например, *изначально запланированным*.

6.1. ЧТО ЕЩЕ, КРОМЕ ХАКЕРОВ

Безопасность программного обеспечения обычно рассматривается с точек зрения уязвимостей, эксплойтов, атак и хакеров. Но безопасность может быть нарушена и из-за других, казалось бы, несущественных факторов. Например, вы могли случайно сохранить имена пользователей и пароли в логе сайта, который хранится на значительно менее безопасных серверах, чем ваша база данных. Подобное уже случалось с гигантами отрасли, такими как Twitter, которые обнаруживали, что хранят незашифрованные пароли в своих внутренних логах¹ и злоумышленник может их использовать, а не взламывать хешированные пароли.

¹ См. «Twitter says bug exposed user plaintext passwords» (Twitter заявил о раскрытии незашифрованных паролей пользователей вследствие ошибки безопасности), <https://www.zdnet.com/article/twitter-says-bug-exposed-passwords-in-plaintext/>.

Facebook предоставил API для разработчиков, который позволял просматривать списки друзей пользователей. Еще в 2016 году компания использовала эту информацию для составления политических профилей пользователей, чтобы влиять на выборы в США с помощью таргетированной рекламы. Эта функция работала исключительно по назначению: не было ни багов, ни дыр в безопасности, ни бэкдоров. Однако полученные данные позволяли оказывать влияние на людей против их воли, причиняя тем самым вред.

Вы будете удивлены, узнав, сколько баз данных компаний доступны в Сети без пароля. Технологии баз данных, такие как MongoDB и Redis, по умолчанию не аутентифицируют пользователей — аутентификацию необходимо активировать вручную. Очевидно, что многие разработчики этого не делают, что приводит к массовым утечкам данных.

Среди разработчиков и DevOps-инженеров бытует негласное правило: «Не проводите развертывание по пятницам». Логика проста. Если что-то пойдет не так, в выходные некому будет это исправлять, поэтому рискованные мероприятия следует проводить в начале недели. Иначе могут пострадать и сотрудники, и компания. Наличие выходных не является уязвимостью системы безопасности, но все же может привести к катастрофическим последствиям.

Это подводит нас к взаимосвязи безопасности и надежности. Безопасность, как и тестирование, является составной частью надежности ваших услуг, данных и бизнеса. Решения, связанные с безопасностью, становится легче принимать, рассматривая безопасность с точки зрения надежности. Разбираясь в других аспектах надежности, таких как тестирование, которое обсуждалось в предыдущих главах, вы начинаете лучше разбираться и в безопасности.

Даже если вы не отвечаете за безопасность продуктов, которые разрабатываете, позаботьтесь о надежности своего кода, чтобы избежать головной боли в будущем. Кодеры с улиц думают не только о настоящем, но и о будущем. Ваша цель — добиться превосходного результата минимальными усилиями. Если вы будете относиться к решениям, связанным с безопасностью, как к необходимым условиям надежности, то значительно облегчите свою работу. Я рекомендую применять для каждого продукта лучшие практики безопасной разработки, такие как параметризованные запросы для выполнения операторов SQL, о чем мы подробно поговорим позже. Возможно, потребуются дополнительные усилия, но они помогут выработать навык, полезный в долгосрочной перспективе. Кратчайший путь не является таковым, если он мешает вам совершенствоваться.

Как всем людям, разработчикам присущи человеческие слабости, в первую очередь ошибки в просчете вероятностей. Я знаю это как человек, который в начале

2000-х несколько лет использовал пароль *password* почти на всех платформах. Я считал, что никому не придет в голову, что я такой тупой. И оказался прав — никто не заметил мою тупость. К счастью, меня ни разу не взломали, по крайней мере пароль не был скомпрометирован, но я и не был мишенью в то время. Это означает, что я верно (или случайно) угадал модель угроз.

6.2. МОДЕЛИРОВАНИЕ УГРОЗ

Модель угроз — это четкое описание того, что может пойти не так в контексте безопасности. Оценка модели угроз обычно происходит так: «Нет, тут проблем не будет» или «Эй, подождите...». Модель угроз создается, чтобы приоритизировать необходимые меры безопасности, оптимизировать расходы и повысить эффективность. Термин звучит очень технологично, потому что процесс оценки может быть запутанным, но описание модели угроз должно быть понятным.

Модель угроз помогает определить, что не представляет угрозу безопасности и от чего нет смысла защищаться, подобно тому как не стоит бояться, что в Сиэтле наступит небывалая засуха или в Сан-Франциско внезапно появится доступное жилье, хотя, конечно, в теории это возможно.

На самом деле мы разрабатываем модели угроз бессознательно. Например, одна из самых распространенных моделей — «Мне нечего скрывать!». Она применяется против таких угроз, как взлом, государственная слежка или бывший партнер, который должен был достичь совершеннолетия десять лет назад. Это означает, что нам безразлично, что наши данные будут скомпрометированы и использованы для каких-либо целей. Это объясняется в основном тем, что мы даже не можем себе представить, как наши данные могут быть использованы. Конфиденциальность в этом смысле похожа на ремень безопасности: 99% времени он не нужен, но в остальных случаях спасает вам жизнь. Когда хакеры, узнав ваш SSN¹, оформляют на вас кредит и оставляют с огромными долгами, вы постепенно понимаете, что стоило бы кое-что скрыть. Когда данные с вашего мобильного телефона указывают, что вы находились на месте убийства в то же время, когда оно было совершено, вы становитесь самым убежденным сторонником конфиденциальности.

Настоящее моделирование угроз немного сложнее. Оно включает в себя анализ действующих лиц, потоков данных и границ доверия. Были разработаны специальные формальные методы для создания моделей угроз. Но если вы не

¹ Social Security Number — номер карточки социального страхования. — Примеч. пер.

эксперт, отвечающий за безопасность организации, то вам достаточно базового понимания проблем для определения приоритетов безопасности.

Прежде всего, примите как данность: рано или поздно ваше приложение или платформа столкнутся с проблемами безопасности. От них никуда не деться. Аргументы «это же всего лишь внутренний сайт», «мы же используем VPN», «это всего лишь мобильное приложение на зашифрованном устройстве», «все равно никто не знает о моем сайте» и «мы же используем PHP» тут не помогут, особенно последний.

Неизбежность проблем с безопасностью обусловлена также тем, что все относительно. Не существует абсолютно безопасной системы. Банки, больницы, кредитные организации, ядерные реакторы, государственные учреждения, криптовалютные биржи и почти все остальные организации сталкивались с проблемами безопасности различной степени серьезности. Затронут ли они ваш сайт с рейтингом лучших фотографий котиков? Дело в том, что ваш сайт может быть использован как платформа для серьезных атак. Один из паролей пользователей, которые вы храните, может совпадать с паролем для входа на сайт ядерного исследовательского центра, где работает этот же пользователь, потому что мы не очень хорошо запоминаем пароли. Эта ситуация представлена на рис. 6.1.

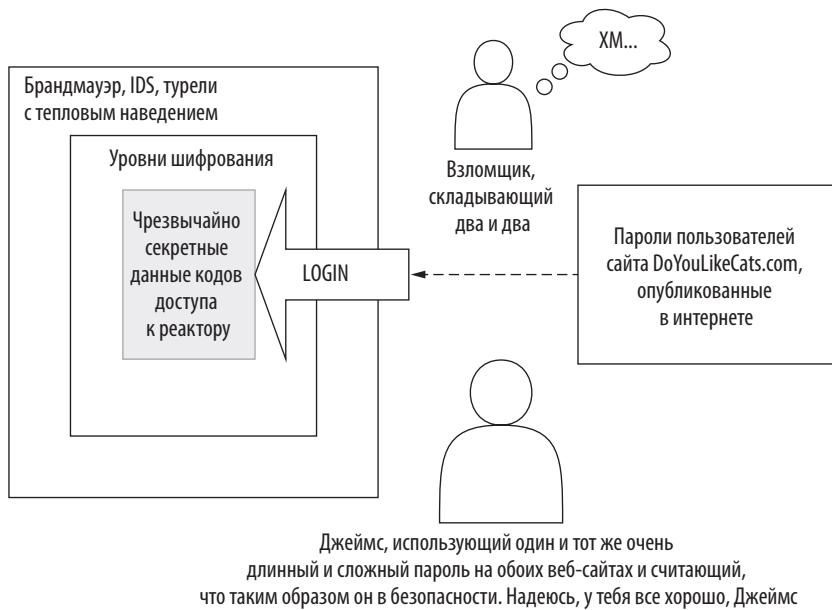


Рис. 6.1. Безопасность не всегда касается только программного обеспечения

Но чаще всего хакеры даже не знают, что проникли на ваш сайт, потому что они не заходят сами. Они используют ботов, которые делают всю тяжелую работу по сканированию сайта на наличие уязвимостей, а затем просто собирают данные. Что ж, роботы действительно постепенно занимают наши рабочие места.

6.2.1. Модели угроз карманного формата

Скорее всего, вам не потребуется моделировать все угрозы для приложения. Вас могут вообще не затронуть инциденты, связанные с безопасностью. Но вы должны писать максимально безопасный код, и это несложно, если следовать определенным принципам. Основное, что вам нужно, — мини-модель угроз для приложения. Она включает в себя следующее:

- *Активы приложения.* По сути, это все, что вы не хотите потерять, включая ваш исходный код, проектную документацию, базу данных, частные ключи, токены API, конфигурации сервера и ваше избранное в Netflix.
- *Серверы, на которых хранятся активы.* К каждому серверу кто-то имеет доступ, и каждый сервер обращается к другим серверам. Вам важно знать эти отношения, чтобы предвидеть возможные проблемы.
- *Чувствительность информации.* Можете ее оценить, задав себе вопросы: «Сколько людей и организаций пострадает, если эта информация станет достоянием общественности?», «Насколько серьезным может быть вред?» и «Бывал ли я в турецкой тюрьме?».
- *Пути доступа к активам.* У вашего приложения есть доступ к базе данных. Есть ли к ней другой доступ и у кого? Насколько это безопасно? Что произойдет, если кто-то обманом получит доступ к БД? Сможет ли он удалить рабочую базу данных, выполнив простой  ? Получит ли он доступ только к исходному коду? Хотя любой, кто имеет доступ к исходному коду, имеет и доступ к рабочей БД.

На основе этой информации вы можете нарисовать базовую модель угроз на листе бумаги. Для любого, кто использует ваше приложение или сайт, она может выглядеть как рис. 6.2. На нем видно, что все пользователи имеют доступ только к мобильному приложению и веб-серверам. С другой стороны, веб-серверы имеют доступ к наиболее важным ресурсам, таким как база данных, а также выход в интернет. Это означает, что веб-серверы — самый высокорисковый ресурс, открытый для внешнего мира.

¹ Информация скрыта. Конфиденциально. Поэтому наши базы данных в безопасности.

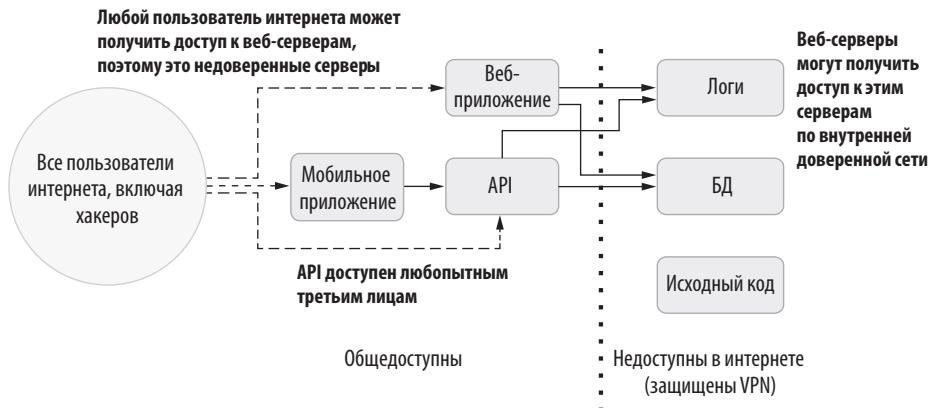


Рис. 6.2. Доступность серверов в сети

Помимо обычных пользователей, есть и другие с разными правами доступа к серверам и хранящимся на них ресурсам. На рис. 6.3 показаны уровни доступа для разных ролей. Поскольку генеральный директор любит все контролировать и имеет неограниченный доступ, самый простой способ проникнуть на сервер — отправить ему электронное письмо. Вы ожидаете, что другие роли будут иметь доступ только к тем ресурсам, которые им необходимы, но обычно это не так.

Если взглянуть на эту модель с высоты птичьего полета, становится очевидным, что если генеральный директор получит письмо с просьбой войти в VPN, чтобы что-то проверить, и из этого письма он будет перенаправлен на фишинговый сайт, злоумышленник получит доступ ко всем данным компании. Именно модель угроз делает такие вещи очевидными и помогает понять факторы риска.

Если ваш директор с манией контроля занимает первое место в списке потенциальных вредителей, то код, запущенный на веб-серверах, — на втором, и не только ваш код, кстати. Инцидент может возникнуть из-за отложенного обновления безопасности на сервере. Но нет ничего хуже возможности просто набрать текст в форме на сайте, чтобы получить доступ ко всем данным в базе или уничтожить их.

Итак, веб-приложение или API — одни из самых простых точек входа для взломщика или бота. Потому что приложение уникально, оно существует только на ваших серверах, и вы — единственный, кто его тестировал. Все сторонние компоненты на ваших серверах прошли миллионы итераций тестирования, исправления ошибок и проверок безопасности. Даже если бы у вас был бюджет, чтобы проделать аналогичные проверки, времени все равно не хватило бы.

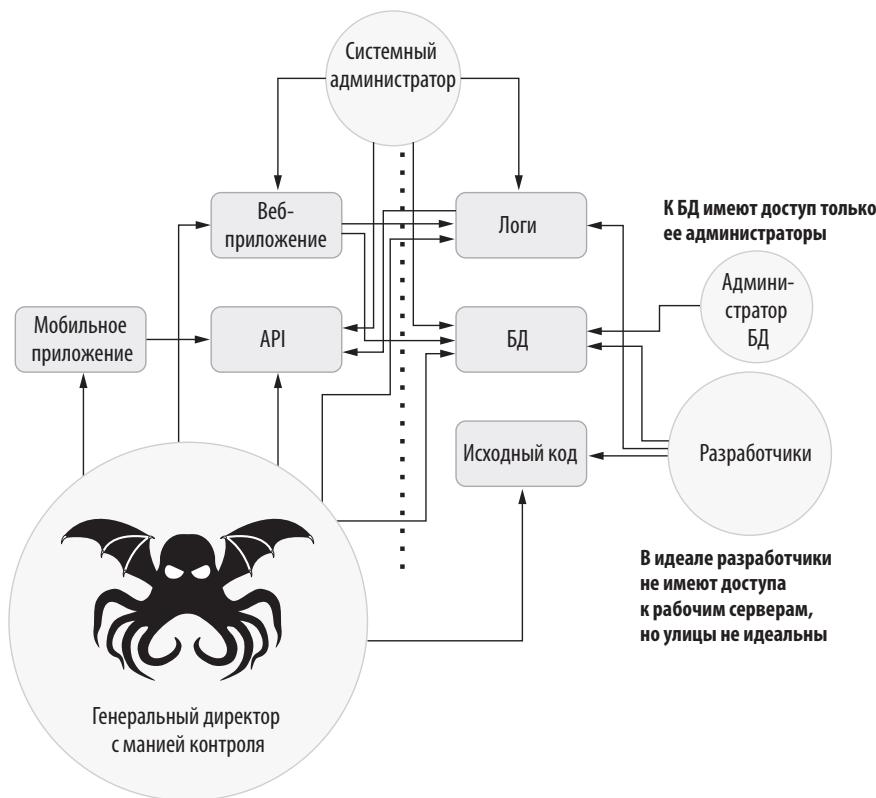


Рис. 6.3. Доступность сервера в зависимости от прав пользователей

Цель хакера или бота может варьироваться от простой остановки сервиса, если это атака Rent-a-DoS (отказ в обслуживании), которую заказал конкурент, до извлечения ценных пользовательских данных, таких как совпадающие пароли к разным ресурсам, чтобы получить доступ к конфиденциальной информации на сервере.

Когда у вас есть список возможных угроз, вы можете устраниить их, закрывая лазейки. Поскольку веб-приложение или API — одни из самых вероятных источников угроз, важно уметь писать для них безопасный код.

6.3. НАПИСАНИЕ БЕЗОПАСНЫХ ВЕБ-ПРИЛОЖЕНИЙ

Каждое приложение уникально, но есть несколько простых практических способов, чтобы сделать его более безопасным. Для уличных кодеров важно знать,

когда эти методы действительно оптимальны, а когда от них лучше отказаться. Рассмотрим варианты атак, которые можно предотвратить, изменив подход к проектированию веб-приложений.

6.3.1. Проектирование с учетом требований безопасности

Параметры безопасности может быть трудно модернизировать из-за решений, которые привели к написанию небезопасного кода. В некоторых случаях повысить безопасность приложения можно только путем переработки его дизайна. Поэтому важно учитывать вопросы безопасности еще на ранних стадиях разработки. Обратите внимание на следующие рекомендации:

1. Изучите модель угроз. Оцените риски, а также текущие и будущие затраты на обеспечение безопасности.
2. Решите, где хранить секреты приложения (пароли БД, ключи API). Задайте для них жесткие правила. Исходите из того, что ваш исходный код находится в общем доступе. Я рассмотрю лучшие практики хранения секретов далее в этой главе.
3. Ограничивайте права доступа. В идеале код не должен требовать больших прав, чем необходимо для выполнения его задачи. К примеру, не стоит предусматривать в приложении предоставление прав администратора БД, если оно не требует периодического восстановления данных. Если расширенные права требуются только для нескольких задач, выделите эти задачи в отдельную изолированную сущность, например в отдельное приложение. Веб-приложения должны запускаться под учетными записями с минимальными правами.
4. Примените принцип ограничения прав ко всей организации. Сотрудники не должны иметь доступа к ресурсам, которые не требуются им для выполнения повседневных задач. У генерального директора не должно быть доступа к БД и вообще к серверам. Не потому, что никому нельзя доверять, а чтобы снизить риски в случаях компрометации извне.

Если эта работа будет проделана до начала написания кода для нового приложения или даже новой функции, в будущем вы будете чувствовать себя намного спокойнее.

Некоторые дальнейшие темы этой главы актуальны только для веб-разработки и разработки API, а примеры относятся к специфическим библиотекам. В частности, если вы не планируете разработку инструментов, требующих удаленного доступа, то можете пропустить раздел о хранении пользовательских секретов.

6.3.2. Повышение безопасности через неясность

Безопасность программного обеспечения — это вопрос гонки со временем. Независимо от того, насколько безопасным вы считаете свой продукт, все сводится к тому, насколько защищены люди и все, что их окружает. Любую защиту можно нарушить. Раньше считалось, что для взлома 4096-битного ключа RSA не хватит срока жизни Вселенной, но оказалось, что достаточно было всего-навсего создать квантовый компьютер. Так что единственная цель каждой защитной меры — выиграть время, заставляя злоумышленников потрудиться.

Эксперты по информационной безопасности ненавидят концепцию обеспечения безопасности через неясность (*security by obscurity*). Как сказал Бенджамин Франклин, «тот, кто пытается добиться безопасности за счет неясности, не достоин ни безопасности, ни неясности». Ладно, он сказал немного иначе, но близко к этому.¹ Причина, по которой безопасность противопоставляется неясности, в том, что вы не выигрываете время, а если и выигрываете, то лишь на границах. Эксперты категорически не согласны с тем, что одной неясности достаточно. Это так, неясность никогда не эффективна сама по себе. Не делайте на нее ставку и используйте только при наличии ресурсов. Но все-таки она может обеспечить пограничную безопасность.

Давайте проясним: пограничная безопасность — это временное решение, которое может поддерживать проект, пока он не достигнет определенного уровня. Я помню, как в первый год работы Ekşi Sözlük административный интерфейс скрывался за малоизвестным URL-адресом без аутентификации. Здесь необходимо пояснить контекст: это был 1999 год, на сайте было максимум 1000 пользователей, и я никому не сообщал этот адрес. Я не стал вкладывать большие средства в сложный механизм аутентификации и авторизации, а сосредоточился на том, что было важно для пользователей. Я знал, что вопрос безопасности можно откладывать только до определенного времени, поэтому ввел систему аутентификации, как только смог.

Точно так же интернет долгое время работал по протоколу HTTP, и широко использовалась схема аутентификации Basic, которая не шифровала пароли, а просто кодировала их в Base64². Это была наглядная иллюстрация безопасности через неясность. Да, ни один здравомыслящий эксперт по безопасности не рекомендовал эту схему, но многие веб-сайты использовали ее, неважно, знали разработчики о рисках или нет. Если вы находились в одной сети с другим

¹ На самом деле Франклин сказал: «Те, кто способны отказаться от свободы ради обретения безопасности, не заслуживают ни свободы, ни безопасности». — *Примеч. ред.*

² Base64 — это метод двоичного кодирования, который преобразует непечатаемые символы в нечитаемые.

пользователем, например подключались к одной открытой точке доступа Wi-Fi, то могли легко получить его пароли и данные трафика. В конце концов атаки MITM (Man in the middle — человек посередине) и приложения для скимминга паролей стали так распространены, что в последнее десятилетие произошел качественный переход на HTTPS, HTTP/2, TLS 1.3 и более безопасные протоколы аутентификации, такие как OAuth2. Но до этого безопасность через неясность десятилетиями работала у нас на глазах.

Мы подошли к сути: расставьте приоритеты безопасности на основе вашей модели угроз, и если модель допускает, безопасность через неясность будет работать на вас точно так же, как табличка «Осторожно, злая собака» на заборе снижает риск ограбления, даже если собаки за забором нет.

Идеальная безопасность недостижима, и вы всегда будете сталкиваться с компромиссами между удобством пользователя и безопасностью, например, Telegram выбрал худшую модель безопасности, чем WhatsApp, но его удобнее использовать, поэтому люди переходят на него, даже если знают о рисках. Очень важно, чтобы вы тоже знали о рисках компромиссных решений, которые принимаете. Простой отказ от всех мер под предлогом «Да ну, безопасность через неясность все равно не работает» — худший вариант.

Тем временем настоящие технологии безопасности становятся доступнее. Раньше приходилось покупать SSL-сертификаты за 500 долларов, чтобы сайт работал с HTTPS, но теперь доступны бесплатные сертификаты инициативы Let's Encrypt ([Let's Encrypt: `https://letsencrypt.org`](https://letsencrypt.org)). Для обеспечения безопасной системы аутентификации теперь достаточно подключить такую библиотеку к проекту. Убедитесь, что вы не преувеличиваете требования к безопасности и не придумываете оправдания, чтобы использовать неясность. Если разница в усилиях незначительна, а риски высоки, реальная технология всегда предпочтительнее безопасности через неясность. Неизвестность не обеспечит настоящей безопасности, но иногда она помогает выиграть время, чтобы успеть разобраться в вопросе.

6.3.3. Не используйте собственные механизмы безопасности

Безопасность — чрезвычайно сложная проблема. Поэтому не разрабатывайте собственные механизмы безопасности, будь то хеширование, шифрование или троттлинг¹. Эксперименты с кодом — это нормально, но не используйте

¹ Троттлинг (throttling) — управление частотой вызова функции, чтобы предотвратить слишком частые запуски и/или обеспечить ее выполнение с заданной периодичностью. — *Примеч. ред.*

собственный защитный код в рабочей среде. Этот совет часто формулируют как «не создавайте свою криптовалюту». Спецификации, связанные с безопасностью, предполагают, что их читатель обладает экспертными знаниями, но обычный разработчик может упустить важные детали. В результате созданные им инструменты не будут соответствовать даже минимальным требованиям безопасности.

Возьмем, к примеру, хеширование. Почти любой его алгоритм до SHA2 имеет серьезные бреши в безопасности. Даже группе экспертов по криптографии трудно создать безопасный алгоритм хеширования, не имеющий слабых мест.

Не думаю, что вы сейчас броситесь изобретать собственный алгоритм хеширования, но хватит ли вам благоразумия не придумывать также и свою функцию сравнения строк? Ниже в этой главе я подробно расскажу, как хранить секреты.

Вы можете повысить степень защиты от уязвимостей, просто изменив организацию своего рабочего процесса. Ниже мы рассмотрим общие векторы атак (но не исчерпывающий их список) и примеры расстановки приоритетов. При этом мы увидим, что достижение приемлемого уровня безопасности не всегда требует больших усилий. Можно работать, не снижая эффективности, и при этом создавать безопасные продукты.

6.3.4. Атаки путем внедрения SQL-кода

С атаками путем внедрения SQL-кода уже давно научились справляться, но это по-прежнему популярный способ компрометирования сайтов. Этот тип атак должен был исчезнуть с лица земли почти одновременно с окончанием режиссерской карьеры Джорджа Лукаса, но почему-то выжил, в отличие от лукасовской режиссуры.

Суть атаки довольно проста. Рассмотрим распространенный сценарий: на сайте запущен SQL-запрос для поиска идентификатора пользователя по имени пользователя, указанного для просмотра профиля. Запрос выглядит так:

```
SELECT id FROM users WHERE username='<username here>'
```

Простой подход к построению этого запроса при заданном в качестве входных данных имени пользователя — встроить последнее в запрос, используя операции со строками. В листинге 6.1 показана простая функция `GetUserId`, которая принимает имя пользователя в качестве параметра и формирует текст запроса путем объединения строк. Обычно так делают новички, но на первый взгляд это нормальный подход. Код создает команду с запросом на основе нашего, в который

уже подставлено имя пользователя, и выполняет ее. Результат возвращается как целое число, для которого допускается нулевое значение, поскольку подходящей записи может не существовать. Обратите внимание, что мы объединяя строки, но не делаем этого в цикле, как было показано в главе 2. Этот метод позволяет избежать избыточного выделения памяти.

НЕОБЯЗАТЕЛЬНЫЕ ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

В листинге 6.1 функции `GetUserId` мы специально используем допускающий нулевое значение (`nullable`) возвращаемый тип вместо псевдоидентификатора, обозначающего отсутствие значения, такого как `-1` или `0`. Компилятор может выявлять в коде вызывающей стороны непроверенные `nullable` возвращаемые значения и выдавать ошибку. То есть если бы возвращалось обычное целочисленное значение, такое как `0` или `-1`, компилятор не знал бы, является ли оно допустимым. В нашем коде использована возможность, недоступная в версиях C# до 8.0. Будущее — сейчас!

Листинг 6.1. Извлечение идентификатора пользователя из базы данных — уязвимый вариант

```
public int? GetUserId(string username) {
    var cmd = db.CreateCommand();
    cmd.CommandText = @"  

        SELECT id  

        FROM users  

        WHERE name=' " + username + " ';  

    return cmd.ExecuteScalar() as int?;
}
```



Формируем запрос к БД

Получаем результат или null, если запись не существует

Допустим, что мы запускаем нашу функцию со значением `placid_turn`. Если убрать лишние пробелы, SQL-запрос будет выглядеть так:

```
SELECT id FROM users WHERE username='placid_turn'
```

Теперь представим, что значение имени пользователя содержит апостроф, например `hackin'`. В этом случае запрос будет выглядеть так:

```
SELECT id FROM users WHERE username='hackin'
```

Заметили, что произошло? Мы допустили синтаксическую ошибку. Класс `SqlCommand` вызовет исключение `SqlException`, и пользователь увидит страницу ошибки. Пока выглядит не страшно, взломщик добьется только ошибки, которая не повлияет на надежность услуг или безопасность данных.

А теперь рассмотрим такое имя пользователя: 'OR username='one_lame''. Запрос для него будет выглядеть так:

```
SELECT id FROM users WHERE username=' OR username='one_lame''
```

Первый апостроф закрыл цитату, и появилась возможность добавить в запрос другие выражения. Хотя в ответ снова вернется ошибка синтаксиса, уже становится страшнее. Как видите, можно изменять запрос, чтобы выводить записи, которые не должны были быть выведены. Теперь устраним синтаксическую ошибку, добавив двойной дефис после имени пользователя:

```
SELECT id FROM users WHERE username=' OR username='one_lame' --'
```

Двойной дефис в SQL похож на двойной слеш (//) во всех языках стиля C, кроме ранних версий самого C, и означает встраивание комментария. Таким образом, остальная часть строки считается комментарием, поэтому запрос выполняется и возвращает информацию для one_lame.

Также мы не ограничены одним оператором SQL, можно запустить несколько операторов. В большинствеialectов языка они должны разделяться точкой с запятой. Если допустимая длина имени пользователя позволяет, можно сделать так:

```
SELECT id FROM users WHERE username='';DROP TABLE users --'
```

Этот запрос удалит пользователей вместе со всеми записями в таблице, если отсутствует конфликт блокировок или активная транзакция, вызывающая тайм-аут. И все это можно сделать удаленно, только введя специально придуманное имя пользователя. Результат — утечка или потеря данных. Конечно, потерянные данные можно восстановить из резервной копии, но то, что утекло из бутылки, вернуть назад не получится.

Неверное решение проблемы внедрения SQL-кода

Как бы вы решали проблему описанной уязвимости в своем приложении? Первое, что приходит на ум, это экранирование — замена каждого символа апострофа (') двойным апострофом (''). Тогда злоумышленник не сможет закрыть кавычку, которую открывает ваш SQL-запрос, так как двойные апострофы считаются обычными символами, а не синтаксическими элементами.

Проблема в том, что в Unicode имеется несколько вариантов символа апострофа. Например, помимо обычного символа с кодом U+0027, есть еще U+02BC.

К сожалению, вы не можете быть уверены, что ПО, обслуживающее БД, не воспримет альтернативный апостроф в качестве обычного или не заменит полученный вариант на символ, который принимает БД. Таким образом, не может быть гарантий, что экранирование сработает без ошибок.

РЕЗЕРВНЫЕ КОПИИ И ПРАВИЛО РЕЗЕРВНОГО КОПИРОВАНИЯ 3-2-1

Помните, в предыдущих главах я говорил, что регрессионные ошибки — худшие, из-за них мы теряем время, как если бы разрушали идеально построенное здание только для того, чтобы построить его заново? Отсутствие резервных копий еще хуже. Ошибки можно исправить, а потеря данных вынуждает создавать код с нуля. Если это не ваши данные, пользователи никогда не будут создавать их снова. Это один из первых уроков, которые я усвоил как разработчик. В начале карьеры я был очень рисковым (иначе говоря, недалеким) человеком. Еще в 1992 году я написал инструмент сжатия и опробовал его на оригинале своего исходного кода. Инструмент преобразовал весь код в один байт информации. Я все еще жду, что появится алгоритм для извлечения плотно упакованных битов. Системы контроля версий в то время были известны немногим, поэтому я сделал вывод о важности резервных копий.

Второй урок я получил в начале 2000-х годов. Прошел год с тех пор, как я создал *Eksi Sözlük*, к счастью, без проблем с Y2K. Я не сомневался в важности резервных копий, но хранил почасовые копии на том же сервере и копировал их на удаленный сервер только раз в неделю. Однажды диски на сервере сгорели — буквально самовоспламенились, и данные на них было невозможно восстановить. Тогда я понял, как важно хранить резервные копии на разных серверах. Позже я узнал, что в разработке существует негласное правило резервного копирования 3-2-1, которое гласит: «Создавайте три резервные копии, две из них храните на отдельных носителях, причем одна копия должна находиться вне вашего офиса». Очевидно, что стратегии резервного копирования требуют тщательного планирования, и возможно, вы никогда не будете заниматься ими профессионально, но «правило 3-2-1» — необходимый минимум, которым лучше не пренебрегать.

Идеальное решение проблемы внедрения SQL-кода

Самый безопасный способ решить проблему внедрения SQL — использовать *параметризованные запросы*. Вместо изменения самой строки запроса вы передаете список параметров, который обрабатывается на стороне БД. Код в листингах 6.2 и 6.1 очень похож. Разница в том, что теперь параметр запроса имеет синтаксис `@parameterName`, а значение этого параметра указывается в отдельном объекте `Parameters`, связанном с этой командой.

Листинг 6.2. Использование параметризованных запросов

```
public int? GetUserId(string username) {  
    var cmd = db.CreateCommand();  
    cmd.CommandText = @"  
        SELECT id  
        FROM users  
        WHERE username=@username"; ← Имя параметра  
    cmd.Parameters.AddWithValue("username", username); ← Передает фактическое  
    return cmd.ExecuteScalar() as int?;  
}
```

Вуаля! Теперь вы можете отправить любой символ в имени пользователя, но не сможете изменить запрос. Экранирование не требуется, потому что запрос и значения параметров отправляются в отдельных структурах данных.

Еще одним преимуществом использования параметризованных запросов является уменьшение засорения кэша планов запросов. Такой план создается в БД при первом запуске запроса и хранится в кэше для быстрого выполнения повторяющихся запросов. Используемая структура подобна словарю, поэтому поиск выполняется со скоростью $O(1)$. Но как и все во Вселенной, кэш планов запросов имеет ограниченную емкость. По мере того как вы отправляете запросы к БД, в кэше плана накапливаются записи, например:

```
SELECT id FROM users WHERE username='oracle'
SELECT id FROM users WHERE username='neo'
SELECT id FROM users WHERE username='trinity'
SELECT id FROM users WHERE username='morpheus'
SELECT id FROM users WHERE username='apoc'
SELECT id FROM users WHERE username='cypher'
SELECT id FROM users WHERE username='tank'
SELECT id FROM users WHERE username='dozer'
SELECT id FROM users WHERE username='mouse'
```

Записи о свежих запросах с отличными друг от друга именами пользователей вытесняют из кэша более ранние записи, которые могут быть более полезными. Это и есть засорение кэша плана запросов.

При использовании параметризованных запросов все записи в плане будут выглядеть одинаково:

```
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
```

Поскольку текст всех запросов одинаков, БД будет использовать только одну запись кэша плана для всех таких запросов. Поэтому у других запросов будет больше шансов найти себе свободное место. Таким образом, в дополнение к полной защите от внедрения SQL-кода вы получите повышение скорости обработки запросов. Бесплатно!

Как и любая рекомендация в этой книге, параметризованный запрос не является универсальным средством. Помните об этом. У вас может возникнуть соблазн сделать все запросы параметризованными. Но не стоит без необходимости использовать параметризацию, например, констант, потому что оптимизатор плана запроса может найти лучшие планы для определенных значений. Так, вы можете составить следующий запрос, хотя параметр `status` всегда имеет значение `active`:

```
SELECT id FROM users WHERE username=@username AND status=@status
```

Оптимизатор плана запроса посчитает, что `status` может получить любое значение, и выберет соответствующий план. Это может повлечь за собой выбор неверного индекса для `active` и снижение скорости обработки запросов. Хм, может, пора написать главу о базах данных?

Когда нельзя использовать параметризованные запросы

Параметризованные запросы универсальны. Можно даже использовать различное количество параметров с названиями вида `@p0`, `@p1` и `@p2`, добавляя значения в цикле. Тем не менее иногда параметризованные запросы лучше не применять. Например, чтобы избежать повторного загрязнения кэша плана запроса. Или, если необходим определенный синтаксис SQL вроде сопоставления с образцом (подумайте об операторах `LIKE`, а также символах `%` и `_`), который может не поддерживаться в параметризованных запросах. Тогда лучше агрессивно дезинфицировать текст, а не экранировать его.

Если параметр является числом, преобразуйте его в правильный числовой тип (`int`, `float`, `double`, `decimal` и т. д.) и используйте в запросе, а не помещайте непосредственно в строку, даже если это повлечет лишнее преобразование целого числа в строку.

Если это строка, но вам не нужны специальные символы или требуется только подмножество специальных символов, удалите из строки все, кроме допустимых

символов. Сейчас это называют *списком разрешений* (allow-listing). Так вы защищите SQL-запросы от случайного проникновения в них вредоносного символа.

Некоторые абстракции БД могут не поддерживать обычные параметризованные запросы. Тогда такие запросы могут передаваться альтернативными способами. Например, Entity Framework Core использует интерфейс `FormattableString` для выполнения повторяющейся операции. В листинге 6.3 представлен запрос, аналогичный приведенному в листинге 6.2, но сформированный в EF Core. Функция `FromSqlInterpolated` умно использует `FormattableString` и синтаксис интерполяции строк C#. Таким образом, библиотека может применять строковый шаблон, заменять аргументы параметрами и строить параметризованный запрос без вашего участия.

ИНТЕРПОЛИРУЙ, УСЛОЖНЯЙ, ВОЗВЫШАЙ (ЛЮБЕЗНО ПРЕДОСТАВЛЕНО ГРУППОЙ RUSH)

В начале был `String.Format()`. Им можно заменить строки, не прибегая к запутанному синтаксису объединения строк. Например, вместо `a.ToString() + " " + b.ToString() + "=" + c.ToString()` можно просто написать `String.Format("{0}+{1}={2 }" a, b, a+b)`. Используя `String.Format`, легче понять, как будет выглядеть результирующая строка. Но понять, какой параметр какому выражению соответствует, на самом деле не так просто.

Начиная с C# 6.0, появился синтаксис интерполяции строк, который позволяет писать выражения наподобие `$(a)+{b}={a+b}`. Он великолепен: при его использовании понятно, как будет выглядеть результирующая строка, и в то же время ясно, чему в шаблоне соответствуют переменные.

Но `$. . .` — это в основном синтаксический сахар для `String.Format(..., ...)`, который обрабатывает строку перед вызовом функции. Если же нам требовались аргументы интерполяции в самой функции, то приходилось писать новые сигнатуры функций, аналогичные `String.Format`, и самим вызывать форматирование, что усложняло работу.

К счастью, новый синтаксис интерполяции строк позволяет выполнять автоматическое приведение к классу `FormattableString`, который содержит как строковый шаблон, так и его аргументы. Функция может получать строку и аргументы по отдельности, если изменить тип строкового параметра на `FormattableString`. Благодаря этому возможны интересные варианты применения, такие как задержка обработки текста в библиотеках журналирования или, как в примере в листинге 6.3, параметризованные запросы без обработки строки. `FormattableString` почти не отличается от литералов шаблонов JavaScript, которые служат той же цели.

Листинг 6.3. Параметризованный запрос с EF Core

```
public int? GetUserId(string username) {
    return dataContext.Users
        .FromSqlInterpolated(
            $"SELECT * FROM users WHERE username={username}")
        .Select(u => (int?)u.Id)
        .FirstOrDefault();
}
```

Использует интерполяцию строк для создания параметризованного запроса

Приводит к `FormattableString` при передаче к `FromSqlInterpolated`

Делает значение по умолчанию `null` вместо нуля для целых чисел путем приведения типов к `nullable`

Возвращает первое значение из запроса, если оно имеется

ВЫВОДЫ

Не увлекайтесь параметризованными запросами, используйте их в основном для пользовательского ввода. Параметризация — мощный инструмент, который идеален для обеспечения безопасности приложения и одновременного кэширования плана запроса большого размера. Тем не менее помните о недостатках параметризации, таких как плохая оптимизация запросов, и не используйте ее для постоянных значений.

6.3.5. Межсайтовый скрипting

Мне кажется, что для пущего драматизма межсайтовый скрипting (мне больше нравится аббревиатура XSS, потому что ее альтернатива CSS¹ совпадает с названием популярного языка таблиц стилей в вебе) стоило назвать *внедрением JavaScript*. Термин «межсайтовый скрипting» похож на название соревнования вроде лыжных гонок. Если бы я не знал, что это такое, то подумал бы: «Ого, межсайтовый скрипting. Звучит неплохо. Я бы хотел, чтобы мои скрипты работали на разных сайтах».

XSS — двухэтапная атака. Первый этап — это вставка кода JavaScript на страницу, а второй — загрузка более крупного кода JavaScript по сети и его выполнение на веб-странице. У этой атаки масса возможностей. Можно узнать историю активности пользователей и даже данные сеанса, заполучив куки в результате перехвата сеанса.

Извини, Дейв, я не могу это внедрить

XSS становится возможным в основном вследствие плохого кода HTML. В этом смысле он похож на внедрение SQL. Вместо ввода апострофа для управления

¹ Cross-site scripting

HTML-кодом используются угловые скобки. Если оказывается, что можно изменить код HTML, в него вводятся теги `<script>` с заключенным в них кодом JavaScript.

Простой пример – функция поиска на веб-сайтах. Результаты поиска выводятся на отдельной странице, но если ничего не найдено, обычно появляется сообщение об ошибке вроде следующего: «По запросу “потоковые конденсаторы для продажи” ничего не найдено». Итак, что произойдет, если ввести поисковый запрос "`<script>alert('hello!');</script>`"? Если вывод должным образом не закодирован, есть вероятность получить что-то близкое к изображенному на рис. 6.4.

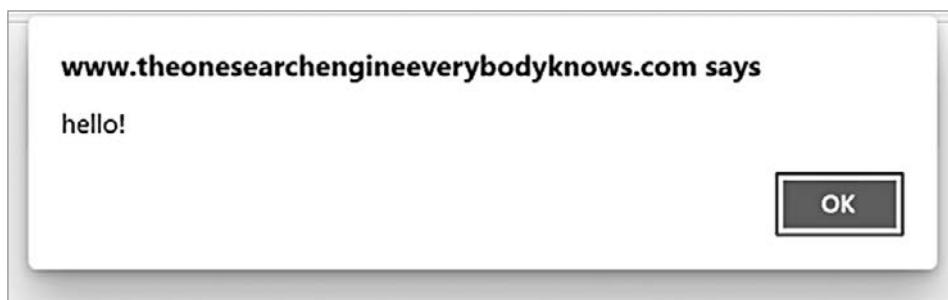


Рис. 6.4. Ваш код запущен на чужом сайте, что же может пойти не так?

Если можно внедрить простую команду `alert`, то безусловно, можно внедрить и что-то еще. Можно прочесть файлы cookie и отправить их на другую веб-страницу. Можно загрузить весь код JavaScript с удаленного URL-адреса и запустить его на этой странице. Вот о чем говорит определение «межсайтовый». Разрешение коду JavaScript отправлять запросы на сторонние веб-сайты считается межсайтовым запросом.

Как предотвратить XSS

Самый простой способ защититься от XSS – закодировать текст с экранированием специальных символов HTML. Они будут представлены своим HTML-аналогом, а не собственным символом, как показано в табл. 6.1. Обычно можно обойтись и без подобных таблиц, выполняя кодирование с использованием проверенных функций. Таблица приведена здесь для справки, чтобы вы могли распознать объекты, когда увидите их в HTML. При экранировании пользовательский ввод не будет считаться HTML и отобразится как обычный текст (рис. 6.5).

Таблица 6.1. Эквиваленты специальных символов HTML

Символ	Экранированный объект HTML	Альтернатива
&	&	&
<	<	<
>	>	>
”	"	"
’	'	'

Your search - "<script>alert("hello!");</script>" - did not match any documents.

Suggestions:

- Make sure all words are spelled correctly.
- Try different keywords.
- Try more general keywords.

Рис. 6.5. При правильном экранировании HTML может не представлять никакой опасности¹

Многие современные фреймворки по умолчанию кодируют обычный текст в HTML. В листинге ниже показан код шаблона Razor для собственной поисковой системы Fooble. В нем мы используем синтаксис @, чтобы напрямую добавить значение на HTML-страницу результатов поиска без кодирования.

Листинг 6.4. Фрагмент страницы результатов поиска поисковой системы

```
<p>
  Поиск по запросу <em>"@Model.Query"</em> ← Кодирование не используется
  не дал результатов.
</p>
```

Хотя строка запроса выводится напрямую, риска XSS не возникает, как показано на рис. 6.6. Если вы просмотрите исходный код сгенерированной веб-страницы, то увидите, что это полная цитата, как в следующем листинге.

¹ На скриншоте: Поиск «...» не дал результатов. Убедитесь, что все слова написаны правильно; попробуйте другие ключевые слова; попробуйте более общие ключевые слова. — *Примеч. пер.*



Рис. 6.6. Мы прекрасно избежали атаки XSS¹

Листинг 6.5. Сгенерированный исходный HTML-код

```
<p>
    Поиск по запросу
    => <em>"&lt;script&gt;alert("hello!");&lt;/script&gt;"</em> ←
    не дал результатов.
</p>
```

Отличный пример экранирования

Тогда зачем вообще переживать об XSS? Потому что, как мы помним, программисты тоже люди. Несмотря на появление продвинутых технологий создания шаблонов, они иногда думают, что при выводе необработанного HTML ничего плохого не произойдет.

Распространенные ловушки XSS

Одна из популярных ловушек — несоблюдение принципа разделения ответственности, например, хранение HTML в модели. У вас может возникнуть соблазн вернуть строку со встроенным HTML-кодом, потому что проще интегрировать логику в код страницы. Например, вернуть простой текст или ссылку в методе get в зависимости от того, кликабельный ли текст. Используя ASP.NET MVC, проще всего записать:

```
return View(isUserActive
    ? $"<a href='/profile/{username}'>{username}</a>"
    : username);
```

¹ На скриншоте: Добро пожаловать в Fooble! Fooble — это абсолютно бесполезная поисковая система, которая не возвращает ничего. Поиск «...» не дал результатов. На кнопках: «Ведите поисковый запрос», «Искать!», «Чувствую себя немного странно». — Примеч. пер.

а затем в представлении:

```
@Html.Raw(Model)
```

И больших усилий требует создание нового класса для совместного хранения `active` и `username`, например, так:

```
public class UserViewModel {
    public bool IsActive { get; set; }
    public string Username { get; set; }
}
```

А затем модели в контроллере:

```
return View(new UserViewModel()
{
    IsActive = isActive,
    Username = username,
});
```

И условной логики в шаблоне для корректного отображения имени пользователя:

```
@model UserViewModel
. . . other code here
@if (Model.IsActive) {
    <a href="/profile/@Model.IsActive">
        @Model.Username
    </a>
} else {
    @Model.Username
}
```

Может показаться, что правильный способ предполагает очень много работы, при том что наша цель — писать меньше кода. Однако можно избежать больших накладных расходов и упростить работу, перейдя на Razor Pages с ASP.NET MVC. Если это невозможно, есть варианты и для существующего кода. Например, исключить отдельную модель, используя вместо нее кортеж:

```
return View((Active: isActive, Username: username));
```

Таким образом можно сохранить код шаблона без изменений. Это убережет вас от создания нового класса, хотя у создания нового класса есть свои преимущества, такие как повторное использование. Такое же преимущество можно получить и от новых записей C#, объявив модель представления неизменяемой в одной строке кода!

```
public record UserViewModel(bool IsActive, string Username);
```

Приложение Razor Pages уже помогает сократить код, потому что больше не нужен отдельный класс модели. Логика контроллера инкапсулирована в класс `View-Model`, созданный на странице.

Если невозможно избежать включения HTML-кода в контроллер MVC или Razor Pages ViewModel, используйте типы `HtmlString` или `IHtmlContent`, которые позволяют явно объявлять хорошо закодированные строки HTML. Подобный сценарий с использованием `HtmlString` выглядел бы так, как показано в листинге 6.6. Поскольку ASP.NET не кодирует `HtmlString`, его даже не нужно оберачивать оператором `Html.Raw`.

В листинге 6.6 показана реализация XSS-устойчивого вывода HTML. `Username` определяется как `IHtmlContent`, а не `string`. Таким образом, Razor будет напрямую использовать содержимое строки без кодирования. Кодирование будет проводить `HtmlContent-Builder` только для тех частей, которые вы явно укажете.

Листинг 6.6. XSS-устойчивые конструкции в HTML

```
public class UserModel : PageModel {
    public IHtmlContent? Username { get; set; }

    public void OnGet(string username) {
        bool isActive = isActive(username);
        var content = new HtmlContentBuilder();
        if (isActive) {
            content.AppendFormat("<a href='/user/{0}'>", username); ← Этот HTML кодирует
            ← только имя
            ← пользователя
        }
        content.Append(username); ← Тоже кодирует имя пользователя
        if (isActive) {
            content.AppendHtml("</a>"); ← Кодирование вообще не применяется
        }
        Username = content;
    }
}
```

Политика защиты контента (CSP)

CSP (Content Security Policy, политика защиты контента) — еще одно оружие борьбы с XSS-атаками. Это заголовок HTTP, ограничивающий ресурсы, которые можно запросить со сторонних серверов. Я считаю CSP сложным в использовании, поскольку современные сайты включают связи со множеством внешних ресурсов, будь то шрифты, файлы сценариев, код аналитики или контент CDN. Все эти ресурсы и доверенные домены могут меняться в любое время. Трудно поддерживать список доверенных доменов в актуальном состоянии. Трудно разобраться в его запутанном синтаксисе. Проверить его правильность тоже трудно.

Правилен ли CSP, если предупреждения не выводятся, или политика слишком гибкая? CSP может быть могущественным союзником, но я не осмелюсь толкать вас в его объятия, обосновывая это таким кратким обзором очень сложной темы. Запомните главное: используете вы CSP или нет, всегда тщательно кодируйте вывод HTML.

Выводы

Атак XSS легко избежать, если не искать легких путей вроде внедрения HTML или игнорирования кодирования. Если внедрить HTML необходимо, позаботьтесь о правильном кодировании значений. Если вы боитесь, что попытки обезопаситься от XSS приведут к увеличению размеров кода, то существуют способы уменьшить эти накладные расходы.

6.3.6. Межсайтовая подделка запроса (CSRF)

Операции изменения веб-содержимого выполняются в протоколе HTTP с помощью команды `POST`, а не `GET`, так как невозможно создать активную ссылку на `POST`-адрес. Его можно опубликовать только один раз. Если этого не удастся сделать, браузер выдаст предупреждение о необходимости повторной отправки. Поэтому операции публикации на форумах, входа в систему и значимых изменений обычно выполняются с помощью `POST`. Существуют также похожие команды `DELETE` и `PUT`, но они используются не так часто и их нельзя вызвать из HTML-формы.

Из-за указанной особенности `POST` мы доверяем ему больше, чем следовало бы. Слабое место `POST` в том, что исходная форма не обязательно должна находиться в том же домене, из которого сделан запрос. Она может располагаться на любой веб-странице. Это позволяет злоумышленникам отправлять `POST`-запросы, обманом побуждая вас перейти по ссылке на их веб-страницу. Предположим, что операция удаления в Твиттере работает как операция `POST` по URL-адресу https://twitter.com/delete/{tweet_id}.

Что произойдет, если я размешу веб-сайт на своем домене `streetcoder.org/about` и добавлю форму, как в следующем листинге, даже не написав ни строки на JavaScript?

Листинг 6.7. Совершенно невинная веб-форма

```
<h1>Добро пожаловать на сверхсекретный сайт!</h1>
<p>Нажмите кнопку, чтобы продолжить</p>
<form method="POST">
```

```

action="https://twitter.com/i/api/1.1/statuses/destroy.json">
<input type="hidden" name="id" value="123" />
<button type="submit">Continue</button>
</form>

```

К счастью, твита с идентификатором 123 не существует, но если бы он существовал, а Твиттер был всего лишь простым стартапом, не умеющим защищаться от CSRF, можно было бы удалить твит, направив его автору запрос на переход по ссылке на теневой сайт. Если же вы используете JavaScript, то можете отправлять запросы POST, даже не требующие пользовательских действий с элементами веб-формы.

Избежать таких проблем можно, используя случайно сгенерированное число для каждой сгенерированной формы, которое реплицируется как в самой форме, так и в заголовках ответов веб-сайта. Поскольку теневой сайт не знает эти числа и не может манипулировать заголовками ответа веб-сервера, он не сможет выдать свой запрос за запрос пользователя. К счастью, фреймворки обычно делают это сами, и вам просто нужно разрешить генерацию токенов и их проверку на стороне клиента. Так, ASP.NET Core 2.0 автоматически включает токены в формы, поэтому вам останется только убедиться, что они проверяются в случае, если вы создаете формы иначе, например в собственном HTML-помощнике. Тогда необходимо явно создать токены подделки запроса в шаблоне, используя такой помощник:

```

<form method="post">
    @Html.AntiForgeryToken()
    ...
</form>

```

Также следует убедиться, что токен проверен на стороне сервера. Это тоже происходит автоматически, но если вы отключили все проверки, эту проверку можно выборочно включить для определенных действий контроллера или Razor Pages, используя атрибут `ValidateAntiForgeryToken`:

```

[ValidateAntiForgeryToken]
public class LoginModel: PageModel {
    ...
}

```

Поскольку в современные фреймворки, такие как ASP.NET Core, уже встроен автоматический механизм защиты от CSRF, вам достаточно знать лишь основы, чтобы понять его преимущества. Но если вы реализуете этот механизм самостоятельно, важно понимать, как и почему он работает.

6.4. ФЛУД

Сбой в работе сервиса называется отказом в обслуживании (DoS, denial of service). Обычно он представляет собой остановку сервера, зависание или аварийное завершение работы, резкое увеличение загрузки ЦП или перенасыщение его пропускной способности. Иногда последний тип атак называют *флудом* (flood). Рассмотрим его подробнее и разберемся, как ему противостоять.

От флуда невозможно защититься полностью, потому что повышенная активность обычных пользователей тоже может послужить причиной сбоя на сайте. Законопослушного пользователя трудно отличить от злоумышленника. Существуют способы смягчить последствия DoS-атак, ограничив возможности злоумышленника. Один из популярных — капча.

6.4.1. Не используйте капчу

Капча — бич Сети. Это популярный способ отделить зерна от плевел, но большая проблема для людей. Ее идея в том, чтобы задать вопрос, на который легко ответит человек, но не программа, используемая для атак, например: «Что будет на обед?».

Проблема капчи в том, что она сложна и для людей. Рассмотрим пример «Выберите все изображения, на которых есть светофоры». Выбирать ли только те, на которых видны сами табло светофора, или также и те, где виден только корпус? А что насчет столба?

Другой пример: что делать с граффити, которое, как считается, легко прочесть? Это буквы *rn* или просто *m*? *5* считается буквой? Почему я должен страдать? Полюбуйтесь на рис. 6.7.

Введите буквы, которые вы видите ниже:



Рис. 6.7. Вы человек или робот?

Как мера борьбы с DoS-атаками капча полезна и вредна одновременно. На начальном этапе жизни вашего приложения вы не хотите думать о проблемах пользовательского опыта (UX). Когда я выпустил Ekşi Sözlük в 1999 году, на нем не нужно было даже вводить имя пользователя. Любой желающий мог сразу написать что угодно под любым никнеймом. Вскоре это создало проблемы, потому что люди начали писать под чужими никами, но это случилось уже после того, как им действительно понравился ресурс. Не заставляйте пользователей страдать, пока не станете достаточно популярным. Именно тогда боты обнаружат ваш сайт и атакуют его, но пользователи будут терпеливы и лояльны, потому что он им нравится.

Это справедливо для всех случаев решения технических проблем, затрагивающих UX. Например, пока на экране отображается страница Cloudflare «Пожалуйста, подождите пять секунд, пока мы проверим, что вы не злоумышленник», 53% посетителей покидают ресурс, если им приходится ждать загрузки более трех секунд. Вы теряете пользователей только из-за того, что кто-то посчитает прибыльным делом постоянные атаки на ваш сайт. Что вы предпочитаете, терять 53% посетителей постоянно или один раз в месяц всех посетителей, пришедших в течение одного часа?

6.4.2. Альтернативы капче

Пишите производительный код, используйте агрессивное кэширование и при необходимости троттлинг. Мы уже обсуждали, какие преимущества в плане производительности предоставляют некоторые методы программирования, а впереди нас ждет целая глава, посвященная исключительно оптимизации производительности.

Но есть и подвох. Троттлинг IP-адреса затронет всех, у кого адрес одинаковый, например всех сотрудников одной компании. Выход за границы определенной зоны может помешать быстро обслуживать запросы значительной части пользователей.

Альтернатива троттлингу – *доказательство работы* (proof of work). Возможно, вы слышали об этом, когда пытались разобраться с майнингом криптовалют. В этом случае, чтобы сделать запрос, компьютер или мобильное устройство должны решить сложную задачу, например выполнить целочисленную факторизацию, что гарантированно займет определенное время. Или можно спросить их о смысле жизни, существования Вселенной и всего остального. Доказано, что поиски ответа на эти вопросы занимают некоторое время.

Доказательство работы требует большого объема ресурсов и может повлиять на время работы от аккумулятора и производительность устройств. А еще — на взаимодействие с пользователем, причем даже хуже, чем капча.

Существуют и более дружественные пользователю требования, такие как необходимость входа в систему после того, как сайт достигнет определенного порога посещаемости. Проверка подлинности стоит дешево, но регистрация и подтверждение адреса электронной почты определенно требуют времени. И снова придется заставлять пользователей совершать какие-то действия. Увидев, что прежде чем получить доступ к контенту сайта, нужно зарегистрироваться или установить мобильное приложение, пользователи, скорее всего, просто вырутятся и уйдут. Выбирая способ ограничить возможности злоумышленника, учитывайте эти плюсы и минусы.

6.4.3. Не применяйте кэш

Словарь, возможно, — самая популярная структура, используемая в веб-фреймворках. Заголовки HTTP-запросов и ответов, файлы cookie и записи кэша хранятся в словарях. Как я уже говорил в главе 2, словари имеют сложность $O(1)$, поэтому поиск в них выполняется мгновенно.

Словари настолько практичны, что можно не удержаться просто запускать один из них, чтобы сохранять кэш чего-либо. В .NET существует даже потокобезопасный `ConcurrentDictionary`, который подходит для ручного кэша.

Обычные словари, включенные в структуру, как правило, не предназначены для ключей, основанных на пользовательском вводе. Если злоумышленник знает, какую среду выполнения вы используете, он может запустить атаку с коллизией хешей. Отправляя запросы с разными ключами, соответствующими одному и тому же хеш-коду, можно вызывать коллизии, о чем я говорил в главе 2. Это приводит к снижению производительности поиска до $O(N)$ и падению приложения.

Пользовательские словари, разработанные для веб-компонентов, таких как `SipHash`, обычно используют другой алгоритм хеширования с лучшими свойствами распределения и, следовательно, с меньшей вероятностью коллизий. Такие алгоритмы в среднем медленнее, чем обычные хеш-функции, но благодаря устойчивости к коллизиям более эффективны при атаках.

По умолчанию в словарях не предусмотрен механизм вытеснения — они бесконечно увеличиваются. Это ни на что не влияет при локальном тестировании,

но в рабочей среде может вызвать ошибку. В идеале структура данных кэша должна иметь возможность исключать старые записи, чтобы контролировать использование памяти.

Учитывая эти факторы, старайтесь по максимуму использовать предоставляемую фреймворком инфраструктуру кэширования всякий раз, когда у вас возникает желание «просто кэшировать это в словаре».

6.5. ХРАНЕНИЕ СЕКРЕТОВ

Секреты (пароли, закрытые ключи и токены API) — это ключи от королевства. Они представляют собой небольшие фрагменты данных, обеспечивающие несопоставимо больший объем доступа. У вас есть пароль к рабочей БД? Тогда у вас есть доступ ко всему. У вас есть токен API? Если да, то вы можете делать все, что этот API разрешает. Вот почему секреты должны быть частью модели угроз.

Разделение системы на части — одно из лучших средств защиты от угроз безопасности. Его можно добиться, в том числе, с помощью безопасного хранения секретов.

6.5.1. Хранение секретов в исходном коде

Программисты отлично умеют находить кратчайшие пути, включая использование ярлыков и легких решений. Вот почему стало таким популярным хранение паролей в исходном коде. Мы любим быстрое прототипирование, потому что ненавидим все, что создает помехи рабочему потоку.

Кажется, что в хранении секретов в исходном коде нет ничего плохого, потому что доступ к коду есть только у вас, а разработчики имеют доступ только к паролям рабочей БД. Проблема в том, что эта логика не принимает во внимание фактор времени. В итоге весь исходный код размещается на GitHub, причем с ним обращаются не так аккуратно, как с рабочей БД, хотя он содержит ключи к ней. Клиенты могут запросить исходный код в рамках договора. Разработчики могут хранить локальные копии исходного кода для его проверки, а их компьютеры могут быть скомпрометированы. А вот локальную копию рабочей БД обычно не делают, потому что последняя слишком велика и, по мнению разработчиков, более чувствительна.

Правильное хранение

Если вы не храните секреты в исходном коде, как исходный код их узнает? Вы можете хранить их в самой БД, но это ведет к противоречию. Где тогда хранить

пароль к БД? Использование БД для хранения секретов — плохая идея еще и потому, что тогда все защищенные ресурсы без необходимости помещаются в одну группу доверия с БД. То есть если хакер получает пароль к базе данных, он получает сразу все. Допустим, вы руководитель ИТ-службы Пентагона и храните коды ядерных пусковых установок в базе данных сотрудников, поскольку она хорошо защищена. Может возникнуть неловкая ситуация, если бухгалтер случайно откроет не ту таблицу. Точно так же приложение может иметь API-доступ к более ценным ресурсам, чем ваша база данных. Подобное несоответствие необходимо учитывать в модели угроз.

Идеальный способ хранения секретов — использование специально предназначенногодляэтойцели хранилища, например диспетчер паролей как холодное хранилище и облачное хранилище ключей (Azure Key Vault, AWS KMS). Если веб-серверы и БД находятся на одной границе доверия в модели угроз, можно добавить секреты в переменные среды на сервере. Облачные сервисы позволяют настраивать переменные среды в интерфейсе администратора.

Современные веб-фреймворки предоставляют разные варианты хранения секретов, поддерживаемые средствами безопасного хранения операционной системы или облачного провайдера, в дополнение к переменным среды, которые напрямую отображаются в конфигурации. Допустим, приложение имеет следующую конфигурацию:

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information"  
    }  
  },  
  "MyAPICode": "somesecretvalue"  
}
```

Вы не хотите оставлять `MyAPICode` в своей конфигурации, потому что любой, у кого есть доступ к исходникам, будет иметь и доступ к ключу API. Тогда вы удаляете ключ и передаете его как переменную в рабочей среде. На компьютере разработчика вместо переменной среды можно использовать пользовательские секреты. В среде .NET можно инициализировать и настроить пользовательские секреты с помощью команды `dotnet`:

```
dotnet user-secrets init -id myproject
```

Она инициализирует проект для использования `myproject` `id` в качестве идентификатора доступа к соответствующим секретам пользователей. Затем можно

добавить пользовательские секреты к учетной записи разработчика с помощью этой команды:

```
dotnet user-secrets set MyAPIkey somesecretvalue
```

После настройки загрузки пользовательских секретов в конфигурацию секреты будут загружаться из специального файла и конфигурация будет переопределена. Доступ к секретному ключу API получаем так же, как к конфигурации:

```
string apiKey = Configuration["MyAPIKey"];
```

Облачные сервисы, такие как Azure или AWS, позволяют настраивать одни и те же секреты через их переменные среды или конфигурации хранилища ключей.

Утечки данных неизбежны

Популярный сайт Have I Been Pwned¹? (<https://haveibeenpwned.com>) — сервис сбора данных об утечках паролей, связанных с адресами электронной почты. На момент написания этой книги меня взламывали 16 раз. Данные утекали и будут утекать. Вы всегда должны учитывать подобный риск и проектировать дизайн с защитой от утечек.

Не собираите данные, которые вам не нужны

Данные невозможно потерять, если их вообще не существует. Не собираите данные, кроме тех, без которых ваш сервис не сможет функционировать. Дополнительными преимуществами станут меньшие требования к хранилищу, более высокая производительность, сокращение работы по управлению данными и меньшие неудобства для пользователя. Например, на многих сайтах при регистрации требуется ввести имя и фамилию. Вам действительно нужны эти данные?

Без некоторых данных, например паролей, обойтись нельзя. Однако ответственность за хранение паролей велика, потому что люди часто используют один и тот же пароль для разных сервисов. Поэтому в случае утечки паролей с вашего ресурса могут быть скомпрометированы и банковские счета пользователей. Можно переложить вину на пользователя, который не использует менеджер

¹ pwned — это видоизмененный глагол own (владеть), обозначающий, что предмет разговора пострадал от хакеров. Пример: I got pwned because I chose my birth date as my PIN — Меня взломали, потому что я использовал в PIN-коде дату своего рождения.

паролей, но вы имеете дело с людьми. Вы можете сделать несколько простых вещей, чтобы предотвратить утечку паролей.

Как правильно хешировать пароли

Самый распространенный способ предотвратить утечку паролей — использовать хеширование. Вместо паролей вы храните их криптографически безопасные хеши. Для этого не подойдет произвольный алгоритм хеширования, например `GetHashCode()` из главы 2, потому что обычные алгоритмы легко сломать или вызвать их коллизии. Криптографически безопасные алгоритмы намеренно замедлены и устойчивы к различным видам атак.

Криптографически безопасные хеш-алгоритмы различаются по своим характеристикам. Для хеширования паролей лучше всего подходит алгоритм, в котором используются многократные итерации. Современные алгоритмы могут требовать сравнительно большого объема памяти, чтобы предотвратить атаки со специальных чипов, предназначенных для взлома определенного алгоритма.

Никогда не используйте однократные хеш-функции, даже если они криптографически безопасны, такие как SHA2, SHA3 или, не дай бог, MD5 или SHA1, потому что они давно взломаны. Криптографическая безопасность означает лишь, что вероятность коллизий для алгоритма исключительно низка, но нет гарантий устойчивости к атакам грубой силы (брутфорсу). Для устойчивости к брутфорсу требуется медленная работа алгоритма.

Широко используемой хеш-функцией для замедления работы является PBKDF2, что звучит как название русской секретной службы, но расшифровывается как Password-Based Key Derivation Function Two — *функция создания ключа на основе пароля 2*. Она может работать с любой хеш-функцией, потому что запускает их в цикле и объединяет результаты. PBKDF2 использует хеш-алгоритм SHA1, который считается слабым и не рекомендован к применению, поскольку с ним легко создать коллизии.

К сожалению, PBKDF2 можно взломать относительно быстро, потому что она может обрабатываться параллельно на графическом процессоре, и для взлома созданы специализированные интегральные схемы ASIC (специальная схема) и FPGA (программируемая схема). Если ваши данные недавно утекли в Сеть, вы наверняка не захотите, чтобы злоумышленник слишком быстро подбирал комбинации, пытаясь взломать их. Новые алгоритмы хеширования, такие как bcrypt, scrypt и Argon2, устойчивы к атакам с использованием существующих графических процессоров и интегральных схем.

Все современные хеш-алгоритмы, устойчивые к брутфорсу, принимают в качестве параметра либо коэффициент сложности, либо количество итераций. Но ваши настройки сложности не должны быть настолько жесткими, чтобы попытка входа на веб-сайт превращалась в DoS-атаку. Не стоит стремиться к сложности, требующей времени обработки более 100 мс. Я настоятельно рекомендую провести бенчмарктест сложности хеширования вашего пароля, чтобы убедиться, что она не создаст проблем, потому что нежелательно изменять алгоритмы хеширования на ходу.

Современные фреймворки, такие как ASP.NET Core, предоставляют готовую функциональность хеширования паролей, но ее текущая реализация основана на PBKDF2. Как было сказано выше, эта технология отстает в плане безопасности. Принимая решения, касающиеся хеширования, важно действовать осознанно.

Я рекомендую выбрать алгоритм, который поддерживается вашим фреймворком. Если его по какой-то причине не получается использовать, выбирайте самый проверенный. Новые алгоритмы обычно еще не протестированы и не проверены так же тщательно, как старые.

Сравнивайте строки безопасно

Итак, вы выбрали алгоритм и храните хеши паролей вместо самих паролей. Теперь все, что вам нужно, — прочитать пароль, введенный пользователем, хешировать его и сравнить с паролем в БД. Звучит просто, правда? Для этого вполне подойдет простое циклическое сравнение, как в листинге 6.8. В нем реализовано прямое сравнение массивов. Сначала мы проверяем длину, а затем запускаем цикл, чтобы убедиться, что все элементы одинаковы. Если мы находим несоответствие, то немедленно возвращаемся, поэтому не утруждаем себя проверкой остальных значений.

Листинг 6.8. Наивная функция сравнения двух хеш-значений

```
private static bool compareBytes(byte[] a, byte[] b) {
    if (a.Length != b.Length) {
        return false; ← На всякий случай проверим соответствие длины
    }
    for (int n = 0; n < a.Length; n++) {
        if (a[n] != b[n]) {
            return false; ← Несоответствие значений
        }
    }
    return true; ← Успех!
}
```

Как этот код может быть небезопасным? Проблема возникает из-за раннего возврата, если найдено несоответствующее значение. Это означает, что можно

узнать длину совпадения, измерив время возврата функции, как показано на рис. 6.8, и если алгоритм хеширования известен, найти правильный хеш, создавая пароли, соответствующие определенному первому значению хеш-функции, затем первым двум значениям, и т. д. Да, различия во времени составляют милли- или даже наносекунды, но их все же можно измерить относительно контрольного показателя. Если с первого раза не получилось, измерения можно повторить, чтобы получить более точные результаты. Это намного быстрее, чем перебирать все возможные комбинации.

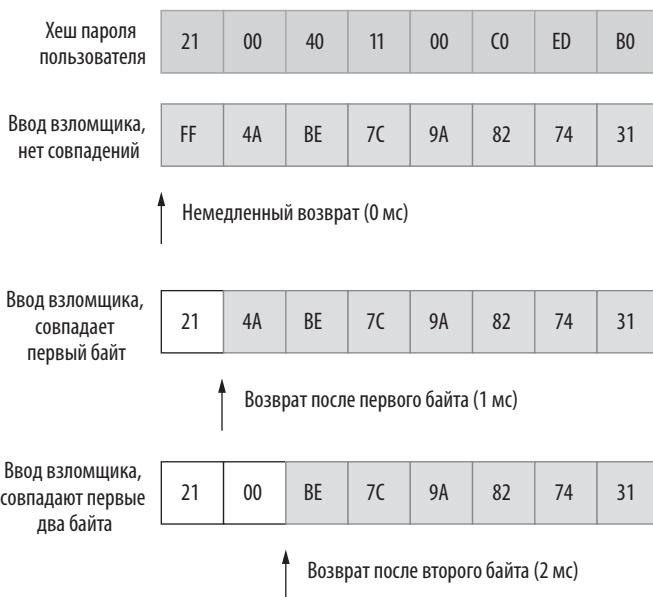


Рис. 6.8. Как быстрое сравнение помогает злоумышленникам определить хеш

Чтобы устранить проблему, необходима функция, выполняющая сравнение за постоянное время, как в листинге 6.9. Вместо раннего возврата значение сохраняется и сравнение продолжается, даже если оно изначально не удаётся. Таким образом, все сравнения занимают одинаковое время, что позволяет избежать утечки значений хешей пользователей.

Листинг 6.9. Безопасное сравнение хешей

```
private static bool compareBytesSafe(byte[] a, byte[] b) {
    if (a.Length != b.Length) {
        return false; ← Это исключительный случай. Сохраняем его
    }
}
```

```

bool success = true;
for (int n = 0; n < a.Length; n++) {
    success = success && (a[n] == b[n]);
}
return success;
}

```

Постоянно обновляем результирующую переменную, не завершая сравнение досрочно

Возвращаем окончательный результат

Не используйте фиксированные соли

Соли — это строки, добавляемые в алгоритмы хеширования паролей. Благодаря этому пароли отклоняются как неподходящие, даже если они относятся к одним и тем же хеш-значениям, чтобы злоумышленник не мог вычислить все одинаковые пароли, подобрав хеш-значение только для одного из них. Таким образом, даже если все пользователи установили пароль `hunter2`, значения хеш-функции у них будут разные, что усложнит жизнь злоумышленнику.

Разработчики могут использовать для солей простые значения, например хеш имени или идентификатор пользователя, потому что их обычно легче генерировать, чем массив случайных значений. Однако это совершенно ненужное упрощение снижает безопасность. Для солей всегда следует использовать случайные значения, но не только обычные псевдослучайные, а значения, генерированные CSPRNG (cryptographically secure pseudorandom number generator) — криптографически безопасным генератором псевдослучайных чисел.

О случай, о рандом!

Регулярные случайные значения генерируются с помощью простых и предсказуемых алгоритмов. Они лишь имитируют случайность и отлично подходят для задания поведения непредсказуемого врага в игре или для определения сегодняшней избранной публикации на сайте. Эти алгоритмы быстрые, но не безопасные. Подобные случайные значения можно предсказать либо сузить область их поиска, поскольку они имеют тенденцию повторяться через относительно короткие интервалы. Люди успешно разгадали алгоритмы генератора случайных чисел игровых автоматов в Лас-Вегасе, когда у разработчиков этих автоматов не было ничего лучше.

Вам нужны криптографически безопасные псевдослучайные числа. Их чрезвычайно трудно предсказать, поскольку на генерацию влияют несколько сильных источников энтропии, таких как аппаратные компоненты компьютера, а также сложные алгоритмы. Работают они медленно, поэтому их обычно используют только в контексте безопасности.

Во многих библиотеках криптографически безопасных хешей доступна функция генерации хеша, которая получает только длину соли, а не саму соль. Библиотека позаботится о создании этой случайной соли, и ее можно получить из результатов, как в листинге 6.10, где в качестве примера используется PBKDF2. Мы создаем реализацию функции получения ключа RFC2898. Это PBKDF2 с алгоритмом HMAC-SHA1. Используем оператор `using`, потому что примитивы безопасности могут задействовать неуправляемые ресурсы операционной системы, и их стоит очищать, когда они выходят за границы области действия. Мы используем простую запись, чтобы вернуть и хеш, и сгенерированную соль в одном пакете.

Листинг 6.10. Генерация криптографически безопасных случайных значений

```
public record PasswordHash(byte[] Hash, byte[] Salt); ← Запись, которая содержит
private PasswordHash hashPassword(string password) { значения хеша и соли
    using var pbkdf2 = new Rfc2898DeriveBytes(password,
        saltSizeInBytes, iterations); ← Создание экземпляра генератора хешей
        var hash = pbkdf2.GetBytes(keySizeInBytes); ← Генерирование хеш-значения
        return new PasswordHash(hash, pbkdf2.Salt);
}
```

UUID не случайны

Универсальные уникальные идентификаторы (UUID, *universally*), или *глобальные уникальные идентификаторы* (GUID, *globally unique identifiers*), как их называют во вселенной Microsoft, представляют собой случайные числа, такие как 14e87830-bf4c-4bf3-8dc3-57b97488ed0a. Раньше они генерировались на основе малоизвестных данных, например MAC-адреса сетевого адаптера или системных даты/времени. Сейчас они в основном подбираются случайным образом. В первую очередь они должны быть уникальными, а не безопасными.

Эти идентификаторы до сих пор можно подобрать, потому что они не всегда создаются с использованием CSPRNG. Нельзя полагаться на случайность GUID, например, для генерации токена активации, отправляемого в электронном письме новому зарегистрированному пользователю. Для создания маркеров безопасности всегда используйте CSPRNG. UUID могут быть не полностью случайными, но они более безопасны в качестве идентификаторов, чем простые монотонные (увеличивающиеся на единицу) целые числа. По такому UUID злоумышленник может вычислить номера предыдущих заказов или общее количество заказов магазина на текущий момент. Однако это невозможно, если UUID полностью случайный.

С другой стороны, у полностью случайных UUID плохое распределение. Даже две последовательные записи попадут в совершенно разные места в индексе базы данных, и чтение будет происходить медленнее. Чтобы избежать этого, были разработаны новые стандарты: UUIDv6, UUIDv7 и UUIDv8. Эти UUID по-прежнему только относительно случайны, но они содержат метки времени для значительно более равномерного распределения.

ИТОГИ

- Используйте мысленные или зарисованные на бумаге модели угроз, чтобы приоритизировать меры безопасности и выявить слабые места.
- Проектируйте приложения в первую очередь с учетом безопасности, потому что модернизировать систему безопасности может быть очень сложно.
- Безопасность через неясность не создает реальной защиты, но может нанести реальный ущерб. Относитесь к ней соответственно.
- Не внедряйте собственные примитивы безопасности, даже когда речь идет о сравнении двух хеш-значений. Доверяйте проверенным решениям с хорошей реализацией.
- Пользовательский ввод — это зло.
- Используйте параметризованные запросы для защиты от внедрения SQL. Если по какой-то причине это невозможно, агрессивно проверяйте и деперсонифицируйте пользовательский ввод.
- Убедитесь, что при добавлении на страницу пользовательский ввод правильно кодируется в HTML, чтобы избежать XSS-уязвимостей.
- Страйтесь не использовать капчу для предотвращения DoS-атак, особенно на начальном этапе жизни продукта. Попробуйте другие методы, такие как троттлинг и агрессивное кэширование.
- Храните секреты отдельно от исходного кода.
- Храните хеши паролей в базе данных с помощью надежных специализированных алгоритмов.
- В операциях, связанных с безопасностью, используйте криптографически безопасные псевдослучайные числа, но не GUID.

Самостоятельная оптимизация

В этой главе

- ✓ Преждевременная оптимизация
- ✓ Нисходящий подход к проблемам производительности
- ✓ Оптимизация узких мест ЦП и ввода/вывода
- ✓ Делаем безопасный код быстрее, а небезопасный — безопаснее

Книги, посвященные оптимизации в программировании, всегда начинаются с известной цитаты знаменитого ученого в области computer science Дональда Кнута: «Преждевременная оптимизация — корень всех зол». Мало того что это утверждение ложно, так оно еще и неверно цитируется. Ложно оно, во-первых, потому, что настоящий корень всех зол — объектно-ориентированное программирование, поскольку оно, как известно, ведет к плохой наследственности и борьбе классов. Во-вторых, цитату выдернули из контекста, почти как текст для *lorem ipsum*. На самом деле Кнут написал следующее: «Не следует переживать из-за низкой эффективности, например, в 97% случаев преждевременная

оптимизация — корень всех зол. Но мы не должны отказываться от своих возможностей в этих критических 3%»¹.

Я утверждаю, что преждевременная оптимизация — это корень всего обучения. Не отказывайтесь от того, что вам нравится. Оптимизация — это решение проблем, а преждевременная оптимизация касается несуществующих гипотетических проблем, которые нужно решить подобно шахматистам, которые расставляют фигуры, чтобы бросить вызов самим себе. Это хорошее упражнение. Вы всегда можете уничтожить результаты своего труда, как я говорил в главе 3, сохранив приобретенную мудрость. Исследовательское программирование — это законный способ улучшить свои навыки, если вы контролируете риски и время. Не лишайте себя возможности учиться.

Тем не менее вас не зря отговаривают от преждевременной оптимизации. Оптимизация может сделать код жестче, что усложнит его обслуживание. Оптимизация — это инвестиция, и ее окупаемость сильно зависит от размеров затрат, то есть потраченного времени. Если спецификации изменятся, выполненная оптимизация может стать ловушкой, из которой будет очень трудно выбраться. Что еще важнее, можно оптимизировать проблему, которой изначально и не существовало, снизив надежность кода.

Например, вы работаете с операцией копирования файлов и знаете, что чем больше размер буфера чтения и записи, тем быстрее совершается операция. У вас может возникнуть соблазн считывать и записывать в память весь объем файла сразу, максимизируя размер буфера. Это может привести или к тому, что приложение будет потреблять огромное количество памяти, или к сбою при попытке чтения очень больших файлов. Вы должны четко понимать, на какие компромиссы вы идете при оптимизации, и верно определять проблему, которую необходимо решить.

7.1. РЕШАЕМ ПРАВИЛЬНУЮ ПРОБЛЕМУ

Проблему низкой производительности можно решить разными способами, причем в зависимости от ее характера эффективность решения и затраты времени на него могут сильно различаться. Первый шаг к пониманию истинной

¹ Дональд Кнут сообщил мне, что цитата из исходной статьи была исправлена и перепечатана в его книге *Literate Programming* (Грамотное программирование). Получение личного ответа от него стало для меня одним из самых ярких моментов в процессе написания этой книги.

природы проблемы производительности — выяснить, существует ли эта проблема вообще.

7.1.1. Простой бенчмаркинг

Бенчмаркинг — это сравнение показателей производительности. Он не поможет выявить основную причину проблемы с производительностью, но поможет определить, что она действительно существует. Такие библиотеки, как BenchmarkDotNet (<https://github.com/dotnet/BenchmarkDotNet>), упрощают реализацию бенчмарк-тестов, поскольку предусматривают меры безопасности, позволяющие избежать статистических ошибок. Но даже если вы не работаете с библиотеками, определить, сколько времени занимает выполнение фрагментов кода, можно с помощью таймера.

Меня всегда интересовало, насколько функция `Math.DivRem()` быстрее обычной операции деления с остатком. `DivRem` рекомендуется использовать, если требуется узнать одновременно результат деления и остаток, но у меня до сих пор не было повода проверить, соответствует ли это действительности:

```
int division = a / b;
int remainder = a % b;
```

Этот код примитивен, поэтому легко понять, что компилятор прекрасно его оптимизирует, в то время как применение `Math.DivRem()` выглядит как сложный вызов функции:

```
int division = Math.DivRem(a, b, out int remainder);
```

СОВЕТ У вас может возникнуть соблазн назвать `%` оператором модуля, но это не так. Это оператор остатка в C или C#. Для положительных значений разницы нет, но для отрицательных значений результаты отличаются. Например, $-7 \% 3$ равно -1 в C# и 2 в Python.

Можно быстро создать набор тестов с помощью BenchmarkDotNet, поскольку он отлично подходит для микробенчмаркинга — тестирования небольших и быстрых функций, которое проводится, если нет других вариантов или если босс в отпуске. BenchmarkDotNet исключает ошибки измерения, связанные с временными отклонениями или накладными расходами на вызовы функций.

В листинге 7.1 представлен код, который использует BenchmarkDotNet для тестирования скорости `DivRem` по сравнению с ручными операциями деления/

остатка. По сути, мы создаем новый класс, описывающий набор бенчмарк-тестов с тестируемыми операциями, которые помечены атрибутами `[Benchmark]`. `BenchmarkDotNet` самостоятельно вычисляет, сколько раз необходимо вызывать эти функции для получения точных результатов, потому что однократное измерение или запуск небольшого количества итераций тестов чреваты ошибками. Операционные системы многозадачны, и задачи, работающие в фоновом режиме, могут повлиять на производительность тестируемого кода. Переменные, используемые в расчетах, помечаются `[Params]`, чтобы компилятор не удалял операции, которые он считает ненужными. Компиляторы легко отвлекаются, но неплохо соображают.

Листинг 7.1. Пример кода `BenchmarkDotNet`

```
public class SampleBenchmarkSuite {
    [Params(1000)] ←
    public int A; ←
    Избегаем оптимизации
    компилятора
    [Params(35)] ←
    public int B; ←

    [Benchmark]
    public int Manual() {
        int division = A / B;
        int remainder = A % B;
        return division + remainder;
    }

    [Benchmark]
    public int DivRem() {
        int division = Math.DivRem(A, B, out int remainder);
        return division + remainder;
    }
}
```

← Атрибутами помечены операции, для которых необходимо провести бенчмаркинг

← Возвращаем значения, поэтому компилятор не удаляет этапы вычислений

Эти тесты можно запустить, просто создав консольное приложение и добавив строку `using` и вызов `Run` в метод `Main`:

```
using System;
using System.Diagnostics;
using BenchmarkDotNet.Running;

namespace SimpleBenchmarkRunner {
    public class Program {
        public static void Main(string[] args) {
            BenchmarkRunner.Run<SampleBenchmarkSuite>();
        }
    }
}
```

Результаты теста будут показаны через минуту работы приложения:

Method	a	b	Mean	Error	StdDev
Manual	1000	35	2.575 ns	0.0353 ns	0.0330 ns
DivRem	1000	35	1.163 ns	0.0105 ns	0.0093 ns

Оказывается, `Math.DivRem()` в два раза быстрее, чем выполнение операций деления и остатка по отдельности. Не пугайтесь столбца `Error` (ошибка), потому что это всего лишь статистический параметр, помогающий оценить точность, когда результаты BenchmarkDotNet недостаточно показательны. Это не стандартная ошибка, а скорее уменьшение доверительного интервала 99,9% до половины.

Хотя BenchmarkDotNet чрезвычайно прост и в нем предусмотрены инструменты для уменьшения количества статистических ошибок, возможно, вы не захотите подключать внешнюю библиотеку только ради бенчмаркинга. Тогда просто напишите свой собственный бенчмарк-тест с помощью `Stopwatch`, как в листинге 7.2. Просто повторяйте цикл достаточное время, чтобы понять различия в производительности функций. Используем тот же класс `suite`, который мы создали для BenchmarkDotNet, но задействуем собственные циклы и способы измерения результатов.

Листинг 7.2. Бенчмаркинг своими руками

```
private const int iterations = 1_000_000_000;

private static void runBenchmarks() {
    var suite = new SampleBenchmarkSuite {
        A = 1000,
        B = 35
    };

    long manualTime = runBenchmark(() => suite.Manual());
    long divRemTime = runBenchmark(() => suite.DivRem());
    reportResult("Manual", manualTime);
    reportResult("DivRem", divRemTime);
}

private static long runBenchmark(Func<int> action) {
    var watch = Stopwatch.StartNew();
    for (int n = 0; n < iterations; n++) {
        action(); ←
    }
    watch.Stop(); | Код, для которого необходимо провести
    return watch.ElapsedMilliseconds; | бенчмаркинг, вызывается здесь
}
```

```
private static void reportResult(string name, long milliseconds) {
    double nanoseconds = milliseconds * 1_000_000;
    Console.WriteLine("{0} = {1}ns / operation",
        name,
        nanoseconds / iterations);
}
```

Результат теста будет примерно таким же:

```
Manual = 4.611ns / operation
DivRem = 2.896ns / operation
```

Обратите внимание, что тесты не пытаются устраниить накладные расходы на вызов функций или на сам цикл `for`, поэтому кажется, что они занимают больше времени, но результат остается неизменным: `DivRem` в два раза быстрее, чем ручные операции деления и остатка.

7.1.2. Производительность и время отклика

Результаты бенчмарк-тестов всегда относительны. Они не скажут, быстро работает код или медленно, но скажут, медленнее он или быстрее, чем другой код. Пользователи воспринимают любое действие, которое занимает более 100 мс, как задержку, а действие, занимающее более 300 мс, как медленное. О целой секунде даже не думайте. Большинство пользователей закроют веб-страницу или приложение, если им придется ждать более трех секунд. Ожидание более пяти секунд сродни расчету времени существования Вселенной и уже не имеет значения. Рисунок 7.1 иллюстрирует это.

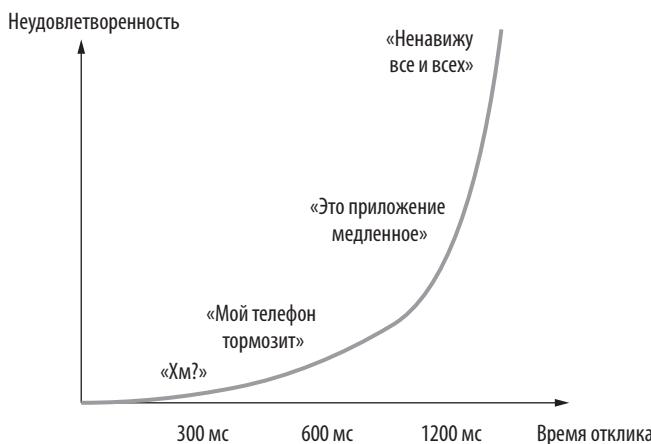


Рис. 7.1. Время задержки отклика и неудовлетворенность пользователя

Очевидно, что производительность не всегда связана с временем отклика. В отзывчивых приложениях некоторые операции могут выполняться медленнее. Рассмотрим, например, приложение, которое использует технологии искусственного интеллекта, чтобы подставлять на видео заданное лицо вместо оригинального. Поскольку такая задача требует больших вычислительных ресурсов, самый быстрый способ ее выполнения — не предпринимать других действий, пока она не завершится. Но это будет означать заморозку пользовательского интерфейса, из-за которой пользователь подумает, что ситуация вышла из-под контроля, и закроет приложение. Таким образом, вместо того чтобы выполнять вычисления как можно быстрее, вы выделяете часть вычислительных ресурсов на отображение индикатора выполнения или, возможно, на расчет приблизительного оставшегося времени и вывод красивой анимации, чтобы развлечь пользователей, пока те ждут. В итоге код работает медленно, но результат более успешен.

Даже если бенчмарк-тесты относительны, все равно можно составить представление о скорости выполнения операций. Питер Норвиг (Peter Norvig) в своем блоге¹ предложил отслеживать значения задержки, чтобы составить представление о разнице в скоростях процессов. Мои предварительные расчеты представлены в табл. 7.1. Вы можете предложить свои оценки скорости.

Таблица 7.1. Задержка в различных контекстах

Хранилище информации	Время чтения одного байта
Регистр ЦП	1 нс
Кэш L1 процессора	2 нс
ОЗУ	50 нс
Твердотельный накопитель, подключенный по шине PCI Express	250 000 нс
Локальная сеть	1 000 000 нс
Сервер на другом конце света	150 000 000 нс

Задержка влияет не только на взаимодействие с пользователем, но и на производительность. Ваша база данных находится на диске, а сервер базы данных — в Сети. Это означает, что даже если вы пишете самые быстрые SQL-запросы и определяете самые быстрые индексы, вы все равно ограничены законами

¹ «Teach Yourself Programming in Ten Years» («Научитесь программировать за десять лет»), <http://norvig.com/21-days.html#answers>.

физики и не можете получить результат быстрее чем за миллисекунду. Каждая миллисекунда, которую вы тратите, съедает часть общего бюджета, который в идеале составляет менее 300 мс.

7.2. АНАТОМИЯ МЕДЛИТЕЛЬНОСТИ

Чтобы понять, как улучшить производительность, сначала нужно разобраться, почему она низкая. Как мы видели, не все проблемы с производительностью связаны с низкой скоростью, некоторые из них обусловлены вопросами отзывчивости. Тем не менее именно скорость в наибольшей степени связана с общими принципами работы компьютеров, поэтому полезно ознакомиться с некоторыми низкоуровневыми концепциями. Они помогут понять методы оптимизации, которые я буду обсуждать ниже в этой главе.

ЦП – это микросхемы, обрабатывающие инструкции, которые оничитывают из ОЗУ и выполняют в бесконечном цикле. Этот процесс можно сравнить с вращением колеса, когда на каждом обороте запускается следующая команда, как показано на рис. 7.2. Некоторые операции могут занимать несколько оборотов, но основной единицей является один оборот, известный как *такт* или *цикл*.

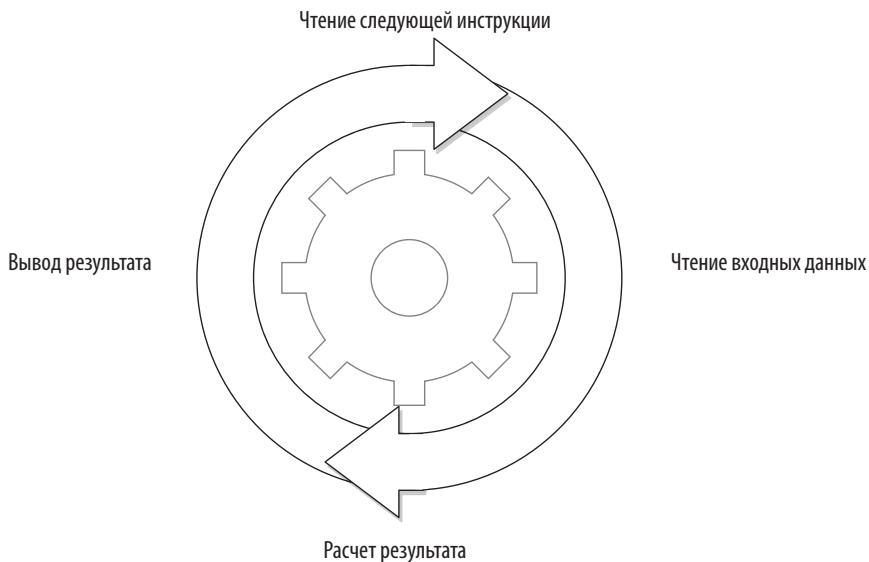


Рис. 7.2. Структура одного цикла процессора

Скорость процессора, обычно выражаемая в герцах, показывает, сколько тактов он может обрабатывать в секунду. Первый электронный компьютер ENIAC мог обрабатывать 100 000 циклов в секунду, или 100 кГц. Древний Z80 с тактовой частотой 4 МГц в моем 8-битном домашнем компьютере в 1980-х годах мог обрабатывать только 4 миллиона циклов в секунду. Современный процессор AMD Ryzen 5950X с частотой 3,4 ГГц может обрабатывать 3,4 *миллиарда* циклов в секунду на каждом из своих ядер. Это не значит, что процессоры обрабатывают такое количество инструкций, потому что, во-первых, некоторые инструкции выполняются более чем за один такт, а во-вторых, современные процессоры могут обрабатывать несколько инструкций параллельно на одном ядре. Таким образом, иногда процессоры могут выполнять даже больше инструкций, чем позволяет их тактовая частота.

Выполнение некоторых инструкций может занимать разное время, зависящее от их аргументов. Например, таковы инструкции копирования блочной памяти. Эта операция занимает $O(N)$ времени в зависимости от размера блока.

По сути, каждая проблема производительности, связанная со скоростью кода, сводится к тому, сколько инструкций и сколько раз выполняются. Когда вы оптимизируете код, то пытаетесь либо уменьшить количество выполняемых инструкций, либо использовать более быструю версию инструкции. Функция `DivRem` работает быстрее, чем выполняется обычная операция деления с остатком, потому что она преобразуется в инструкции, которые занимают меньше тактов.

7.3. НАЧНИТЕ СВЕРХУ

Второй по эффективности способ уменьшить количество выполняемых инструкций — выбрать более быстрый алгоритм. Очевидно, что самый эффективный способ — полностью удалить код. Я серьезно: удалите код, который вам не нужен. Не храните ненужный код в базе. Даже если его хранение не снижает скорость выполнения кода, оно снижает производительность разработчиков, что в конечном итоге снизит и производительность кода. Не нужно пользоваться возможностью закомментировать ненужный код. Для восстановления используйте функцию сохранения истории в своей любимой системе управления версиями, такой как `Git` или `Mercurial`. Если функция требуется вам очень редко, лучше внедрить ее в конфигурацию, а не переводить в комментарии. Так она останется актуальной и рабочей. И для вас не станет неожиданностью, что код, с которого вы только что сдули пыль, вообще не компилируется, потому что все изменилось.

Как я говорил в главе 2, более быстрый алгоритм может дать преимущества, даже если он плохо реализован. Поэтому сначала спросите себя: «Это действительно лучший способ?». Можно сделать плохой код быстрее, но самое разумное — решать проблему *сверху*, в самом широком смысле, погружаясь глубже в сценарий, пока вы не определите, где на самом деле находится проблема. Этот способ обычно быстрее, и результат его применения в конечном итоге легче в обслуживании.

Рассмотрим пример, когда пользователи жалуются на медленную загрузку их профиля в приложении, и вы можете воспроизвести проблему самостоятельно. Проблема может быть как на стороне клиента, так и на стороне сервера. Вы начинаете сверху: сначала определяете, на каком из двух основных уровней возникает проблема, исключая уровень клиента или уровень сервера. Если проблема не возникает при прямом вызове API, значит, она на стороне клиента, в противном случае — на стороне сервера. Вы продолжаете анализ, пока не находите реальную причину проблемы. В некотором смысле вы выполняете бинарный поиск, как показано на рис. 7.3.

Если следовать по нисходящему пути, вы гарантированно найдете первопричину, не строя догадок. Вы выполните бинарный поиск вручную и тем самым используете алгоритмы в реальной жизни, чтобы облегчить ее, — так держать! Определив место, где возникает проблема, проверьте наиболее сложные блоки кода. Ненужное повышение сложности кода является следствием использования определенных шаблонов. Рассмотрим некоторые из них.



Рис. 7.3. Нисходящий подход к определению причины проблемы

7.3.1. Вложенные циклы

Один из самых простых способов замедлить код — поместить его в другой цикл. Когда мы пишем вложенные циклы, то недооцениваем эффект умножения количества вычислений. К тому же вложенные циклы не всегда заметны. Вернемся к нашему примеру с медленной загрузкой профиля пользователя и предположим, что вы обнаружили проблему в серверном коде, который генерирует профили. Имеется функция, которая возвращает бейджи пользователя и выводит их в его профиле. Пример кода может выглядеть так:

```
public IEnumerable<string> GetBadgeNames() {
    var badges = db.GetBadges();
    foreach (var badge in badges) {
        if (badge.Visible) {
            yield return badge.Name;
        }
    }
}
```

Здесь нет очевидных вложенных циклов. На самом деле эту же функцию в LINQ можно написать вообще без циклов, хотя проблема сохранится:

```
public IEnumerable<string> GetBadgesNames() {
    var badges = db.GetBadges();
    return badges
        .Where(b => b.Visible)
        .Select(b => b.Name);
}
```

Где внутренний цикл? Этот вопрос вам придется задавать себе на протяжении всей карьеры. Виновник медленной работы — свойство `Visible`, причем мы просто не знаем, что у него внутри.

Свойства в C# появились, потому что разработчики языка устали добавлять `get` перед каждым именем функции, каким бы простым оно ни было. На самом деле код свойств преобразуется в функции и к их именам при компиляции добавляются префиксы `get_` и `set_`. Преимущество свойств в том, что они позволяют изменить функции члена поля в классе, не нарушая совместимости. Однако свойства скрывают потенциальные проблемы. Они выглядят как простые поля, базовые операции доступа к памяти, поэтому легко предположить, что вызов свойства обходится недорого. В идеале не следовало бы помещать в свойства код, требующий больших вычислительных ресурсов. К сожалению, понять, не сделал ли этого кто-то другой, невозможно, по крайней мере, не видя кода.

Изучив код свойства `IsVisible` класса `Badge`, видим, что он более затратен, чем кажется:

```
public bool IsVisible {
    get {
        var visibleBadgeNames = db.GetVisibleBadgeNames();
        foreach (var name in visibleBadgeNames) {
            if (this.Name == name) {
                return true;
            }
        }
        return false;
    }
}
```

Это свойство не стесняясь вызывает базу данных, чтобы получить список имен отображаемых бейджей, и сравнивает их в цикле, проверяя, является ли рассматриваемый бейдж одним из видимых. Этот код изобилует сложно объяснимыми недостатками, но главный урок, который необходимо извлечь, — следует остерегаться использования свойств. Они содержат скрытую логику, которая не всегда проста.

Есть много возможностей для оптимизации `IsVisible`, первая и главная из которых — не запрашивать список имен видимых бейджей при каждом вызове свойства. Их можно хранить в статическом списке, который извлекается только один раз, при условии, что список меняется редко и можно выполнить перезагрузку, когда произойдет изменение. Также можно использовать кэширование — этот вариант я объясню позже. Таким образом, код свойства можно сократить до следующего вида:

```
private static List<string> visibleBadgeNames = getVisibleBadgeNames();

public bool IsVisible {
    get {
        foreach (var name in visibleBadgeNames) {
            if (this.Name == name) {
                return true;
            }
        }
        return false;
    }
}
```

Список хорош тем, что у него есть метод `Contains`, поэтому можно избавиться от цикла:

```
public bool IsVisible {  
    get => visibleBadgeNames.Contains(this.Name);  
}
```

Внутренний цикл наконец исчез, но не уничтожен до конца. Необходимо посолить и сжечь его кости. Списки в C#, по сути, являются массивами и имеют сложность поиска $O(N)$. Это означает, что цикл всего лишь переместился внутрь другой функции, в данном случае `List<T>.Contains()`. Чтобы уменьшить сложность, надо не просто устраниить цикл, а изменить алгоритм поиска.

Можно отсортировать список и выполнить бинарный поиск, чтобы уменьшить сложность до $O(\log N)$. К счастью, мы читали главу 2 и знаем, что структура данных `HashSet<T>` обеспечивает значительно лучшую производительность поиска $O(1)$, определяя местоположение элемента с использованием его хеша. Код свойства наконец приобретает нормальный вид:

```
private static HashSet<string> visibleBadgeNames = getVisibleBadgeNames();  
  
public bool IsVisible {  
    get => visibleBadgeNames.Contains(this.Name);  
}
```

Мы не проводили бенчмаркинг этого кода, но как видно из рассмотренного примера, анализ болевых точек алгоритма может оказаться очень полезным. Исправив код, всегда стоит проверить, стал ли он работать эффективнее. Чужой код полон сюрпризов и белых пятен, которые могут преподносить неожиданности.

Возможности оптимизации метода `GetBadgeNames()` этим не исчерпываются. Стоит задать и другие вопросы его разработчику, например, зачем хранить отдельный список имен видимых бейджей, а не однобитовый флаг в записи `Badge` в базе данных? Можно использовать и отдельные таблицы, объединяя их при запросе к БД. Однако ограничимся тем, что без внутреннего цикла поиск уже должен стать на порядки быстрее.

7.3.2. Строко-ориентированное программирование

Строки чрезвычайно практичны. Их удобно читать, они могут содержать любой текст, и ими легко манипулировать. Я уже рассказывал, как использование других типов вместо строк делает производительность выше, но строки тоже могут использоваться в коде.

Зачастую разработчики любую коллекцию стремятся представить в виде коллекции строк. Например, если требуется сохранить флаг в контейнере `HttpContext.Items` или `ViewData`, кто-нибудь обязательно напишет:

```
HttpContext.Items["Bozo"] = "true";
```

Позже он проверит этот же флаг следующим образом:

```
if ((string)HttpContext.Items["Bozo"] == "true") {  
    ...  
}
```

Приведение типа к строке обычно выполняется только после того, как компилятор начнет возмущаться: «Эй, ты уверен, что хочешь это сделать? Это не коллекция строк». И действительно, никто не замечал, что это коллекция объектов. На самом деле этот код можно исправить, просто введя логическую переменную:

```
HttpContext.Items["Bozo"] = true;
```

А проверить значение можно так:

```
if ((bool?)HttpContext.Items["Bozo"] == true) {  
    ...  
}
```

Все это позволяет избежать накладных расходов на хранение и парсинг, а также случайных опечаток, таких как ввод `True` вместо `true`. Накладные расходы, связанные с этими ошибками, ничтожны, но если допускать их постоянно, расходы могут значительно возрасти. Когда корабль тонет, гвозди уже бесполезны, но если забивать их правильно при постройке, судно останется на плаву.

7.3.3. Вычисление 2b || !2b

Булевые выражения в операторах `if` вычисляются в том порядке, в котором записаны. Компилятор C# генерирует умный код, чтобы избежать ненужных расчетов. Вспомним, к примеру, суперзатратное свойство `IsVisible`. Рассмотрим такую проверку:

```
if (badge.Visible && credits > 150_000) {
```

Дорогостоящее свойство вычисляется перед простой проверкой значения. Если вы вызываете эту функцию в основном со значениями аргумента меньше

150 000, `isVisible` не будет вызываться большую часть времени. Можно просто поменять местами выражения:

```
if (credits > 150_000 && badge.isVisible) {
```

Таким образом, дорогостоящая операция не будет выполняться без необходимости.

Этот способ применим и к логическим операциям ИЛИ (`||`). Тогда после первого возвращения значения `true` вычисления не будут проводиться в оставшейся части выражения. Очевидно, что в реальной жизни свойство `isVisible` используется редко, тем не менее я рекомендую строить выражения в следующем порядке типов операторов:

1. Переменные.
2. Поля.
3. Свойства.
4. Вызовы методов.

Конечно, не для каждого логического выражения можно свободно менять порядок операторов. Рассмотрим такой пример:

```
if (badge.isVisible && credits > 150_000 || isAdmin) {
```

Нельзя просто переместить `isAdmin` в начало, потому что это повлияет на расчеты. Убедитесь, что вы случайно не нарушили логику оператора `if`, оптимизируя вычисления для логических переменных.

7.4. РАЗБИВАЕМ БУТЫЛКУ ПО ГОРЛЫШКУ

Задержки в разработке могут возникать на трех уровнях: ЦП, ввода/вывода и человека. Каждый из них можно оптимизировать, находя более быструю альтернативу, распараллеливая задачи или удаляя соответствующую переменную из общего уравнения.

Если вы уверены, что используете правильный алгоритм или метод, в конечном итоге все сводится к тому, как оптимизировать сам код. Чтобы оценить варианты оптимизации, стоит знать о возможностях процессора.

7.4.1. Не упаковывайте данные

Чтение из адреса памяти, скажем, 1023, может занять больше времени, чем из адреса 1024 из-за накладных расходов при чтении из невыровненных адресов памяти. Выравнивание в этом смысле означает расположение в памяти ЦП, кратной 4, 8, 16 и т. д. до размера машинного слова, как показано на рис. 7.4. В некоторых старых процессорах доступ к невыровненной памяти наказывается тысячью ударов электрическим током. Это правда: некоторые процессоры вообще не позволяют получить доступ к невыровненной памяти, например Motorola 68000, который используется в Amiga, и некоторые процессоры на базе ARM.



Рис. 7.4. Выравнивание адресов памяти

РАЗМЕР МАШИННОГО СЛОВА

Размер слова обычно определяется тем, сколько бит данных ЦП может обрабатывать за раз. Он различен, например, для 32-битного и 64-битного процессоров. Размер слова в основном отражает размер аккумулятора регистра процессора. Регистры выступают в качестве переменных уровня ЦП, а аккумулятор — наиболее часто используемый регистр. Возьмем, к примеру, процессор Z80. Он имеет 16-битные регистры и может адресовать 16-битную память, но считается 8-битным процессором, поскольку имеет 8-битный аккумулятор.

К счастью, у нас есть компиляторы, которые обычно заботятся о выравнивании. Но можно переопределить поведение компилятора, и он не заметит ничего подозрительного в идеи хранить больший объем данных компактно в небольшом пространстве памяти. Рассмотрим структуру данных в листинге 7.4. Поскольку это структура, C# будет применять выравнивание только на основе некоторых эвристик, а это может означать отсутствие выравнивания вообще. Тогда может возникнуть соблазн сохранять значения в байтах как небольшой пакет для передачи.

Листинг 7.3. Структура упакованных данных

```
struct UserPreferences {
    public byte ItemsPerPage;
    public byte NumberOfItemsOnTheHomepage;
    public byte NumberOfAdClicksICanStomach;
    public byte MaxNumberOfTrollsInADay;
    public byte NumberOfCookiesIAmWillingToAccept;
    public byte NumberOfSpamEmailILoveToGetPerDay;
}
```

Но поскольку доступ к невыровненным адресам памяти происходит медленнее, экономия памяти нивелируется штрафом за доступ к каждому члену в структуре. Если вы измените типы данных в структуре с `byte` на `int` и проведете бенчмарктест для определения разницы, то увидите, что тип `byte` обрабатывается почти в два раза медленнее, даже несмотря на то что они занимают только четверть объема памяти, как показано в табл. 7.2.

Таблица 7.2. Разница между временем доступа к выровненным и невыровненным элементам

Метод	Среднее время
ByteMemberAccess	0.2475 нс
IntMemberAccess	0.1359 нс

Мораль сей басни в том, что не следует увлекаться оптимизацией использования памяти. Иногда этот параметр критичен, например, при создании массива из миллиарда чисел разница между `byte` и `int` может достигать 3 Гбайт. Рациональное использование памяти предпочтительнее для операций ввода/вывода, но в остальных случаях доверяйте выравниванию. Непреложный закон бенчмаркинга гласит: «Семь раз отмерь, один раз отрежь, потом измерь еще раз и больше не режь».

7.4.2. Производите вычисления локально

Кэширование — это хранение часто используемых данных в месте, к которому можно получить доступ быстрее, чем к обычному месту их нахождения. Процессор имеет собственную кэш-память с разной скоростью, и она быстрее, чем оперативная память. Я не буду вдаваться в технические подробности того, как устроен кэш, но суть в том, что процессоры считывают данные из памяти кэша намного быстрее, чем из оперативной. Поэтому последовательное чтение выполняется быстрее, чем случайное. Например, последовательное чтение массива

быстрее, чем связанного списка, хотя в обоих случаях требуется время $O(N)$. Так происходит благодаря высокой вероятности нахождения следующего элемента массива в кэшированной области памяти. Элементы же связанных списков разбросаны по памяти, потому что выделены отдельно.

Предположим, у вас есть ЦП с кэшем 16 байт, а также массив и связанный список из трех целых чисел. На рис. 7.5 показано, что чтение первого элемента массива также вызовет загрузку остальных элементов в кэш ЦП, а обход связанного списка приведет к промаху кэша и принудительной загрузке новой области в него.



Рис. 7.5. Расположение кэшей массива и связанного списка

Процессоры обычно делают ставку на то, что данныечитываются последовательно. Это не значит, что связанные списки бесполезны. Они имеют отличную производительность вставки/удаления и меньшие накладные расходы памяти при увеличении размера. Списки на основе массивов должны перераспределять и копировать буферы по мере роста, а это очень медленные процессы, требующие выделения дополнительной памяти. Последнее может привести к непропорциональному использованию памяти в больших списках. Однако в большинстве случаев список сослужит вам хорошую службу и его быстрее читать.

7.4.3. Разделяйте зависимые процессы

Одна инструкция ЦП обрабатывается разными дискретными блоками процессора. Например, один блок отвечает за декодирование инструкции, а другой — за доступ к памяти. Но поскольку блоку декодера необходимо дождаться завершения инструкции, он может выполнять работу по декодированию следующей инструкции во время доступа к памяти.

Такая обработка называется *конвейерной* и означает, что ЦП может выполнять несколько инструкций параллельно на одном ядре, если следующая инструкция не зависит от результата предыдущей.

Рассмотрим пример: необходимо вычислить контрольную сумму для значений массива байтов, как в листинге 7.4. Обычно контрольные суммы используются для обнаружения ошибок, и суммирование чисел — не самый лучший вариант, но будем считать, что это государственный заказ.

Посмотрите на код, и вы увидите, что значение результата `result` постоянно обновляется. Следовательно, расчеты зависят от `i` и `result`. Это означает, что ЦП не может распараллелить процесс, потому что он зависит от операции.

Листинг 7.4. Простая контрольная сумма

```
public int CalculateChecksum(byte[] array) {
    int result = 0;
    for (int i = 0; i < array.Length; i++) {
        result = result + array[i]; ← Зависит как от i, так и от предыдущего
    }                                         значения результата
    return result;
}
```

Существуют способы уменьшить количество зависимостей или по крайней мере снизить блокирующее влияние потока инструкций. Один из них — изменить порядок инструкций, увеличив расстояние между зависимыми блоками кода, чтобы первая инструкция не блокировала следующую в конвейере, поскольку следующая инструкция зависит от результата первой операции.

Поскольку сложение выполняется в любом порядке, можно разделить эту операцию на четыре части в одном коде, чтобы ЦП распараллелил работу. Возможная реализация показана в следующем листинге. В этом коде содержится больше инструкций, но слагаемые контрольной суммы теперь вычисляются четырьмя разными аккумуляторами, а затем складываются. В конце в отдельном цикле суммируются оставшиеся байты.

Листинг 7.5. Распараллеливание процесса на одном ядре

```
public static int CalculateChecksumParallel(byte[] array) {
    int r0 = 0, r1 = 0, r2 = 0, r3 = 0; ← Четыре аккумулятора!
    int len = array.Length;
    int i = 0;
    for (; i < len - 4; i += 4) {
        r0 += array[i + 0];
        r1 += array[i + 1];
        r2 += array[i + 2];
        r3 += array[i + 3]; ← Эти расчеты не зависят
                             друг от друга
    }
    int remainingSum = 0;
    for (; i < len; i++) { ← Вычисление суммы оставшихся байтов
        remainingSum += array[i];
    }
    return r0 + r1 + r2 + r3 + remainingSum;
}
```

```

        remainingSum += i;
    }
    return r0 + r1 + r2 + r3 + remainingSum; ← Итоговое сложение
}

```

Мы проделали гораздо больше работы, чем для более простого кода из листинга 7.4, и тем не менее этот процесс на моей машине выполняется на 15 % быстрее. Не ждите волшебства от такой микрооптимизации, но вы оцените ее эффективность при работе с кодом, потребляющим много ресурсов ЦП. Главный вывод из этого состоит в том, что изменение порядка и даже удаление зависимостей в коде может повысить скорость выполнения, поскольку зависимый код засоряет пайплайн.

7.4.4. Будьте предсказуемы

Самый популярный вопрос на Stack Overflow за всю его историю: «Почему отсортированный массив обрабатывается быстрее, чем несортированный?»¹. Чтобы оптимизировать время выполнения, ЦП старается действовать на опережение и подготовиться к выполнению следующих блоков кода. Для этого используется, в частности, *прогнозирование ветвлений*. Код ниже — просто засахаренная версия сравнений и ветвлений:

```

if (x == 5) {
    Console.WriteLine("X is five!");
} else {
    Console.WriteLine("X is something else");
}

```

Оператор `if` и фигурные скобки — это элементы структурного программирования. Они позволяют понять, какой код обрабатывает ЦП. За кулисами, на этапе компиляции, код преобразуется в низкоуровневый, подобный этому:

```

compare x with 5
branch to ELSE if not equal
write "X is five"
branch to SKIP_ELSE
ELSE:
    write "X is something else"
SKIP_ELSE:

```

¹ Этот вопрос можно найти по адресу <http://mng.bz/Exxd>.

ПЕРЕСТАНЬТЕ БЕСПОКОИТЬСЯ И НАУЧИТЕСЬ ЛЮБИТЬ СБОРКУ

Машинный код, родной язык ЦП — это просто последовательность чисел. Ассемблер — синтаксис, созданный для удобочитаемости машинного кода. Синтаксис ассемблера различается в зависимости от архитектуры ЦП, поэтому я рекомендую вам ознакомиться хотя бы с одним из его видов. Это поучительно и поможет избавиться от страха перед процессами, происходящими под капотом. Ассемблер может показаться сложным, но он проще, чем языки, на которых мы пишем программы, даже самые примитивные из них. Листинг на ассемблере представляет собой ряд меток и инструкций, например таких:

```
let a, 42
some_label:
    decrement a
    compare a, 0
    jump_if_not_equal some_label
```

Это базовый цикл уменьшения от 42 до 0, написанный на псевдоассемблере. Реальные инструкции короче, чтобы их было легче писать и сложнее читать. Например, этот же цикл в архитектуре x86 будет выглядеть так:

```
mov a1, 42
some_label:
    dec a1
    cmp a1, 0
    jne some_label
```

А так в архитектуре ARM:

```
mov r0, #42
some_label:
    sub r0, r0, #1
    cmp r0, #0
    bne some_label
```

Его можно записать и короче, используя другие инструкции, но если вы разбираетесь в структуре сборки, то можете посмотреть, какой машинный код генерирует JIT-компилятор, и понять его фактическое поведение. Это особенно полезно, когда вы анализируете задачи, потребляющие много ресурсов процессора.

Я немного изменил его, поскольку фактический машинный код менее понятен, но суть ясна. Неважно, насколько элегантен создаваемый дизайн, — в конечном итоге он становится набором операций сравнения, сложения и ветвления. В листинге 7.6 показан фактический результат сборки этого фрагмента кода для архитектуры x86. После того как вы увидели псевдокод, вы его, наверное, узнаете.

На сайте Sharplab.io есть отличный онлайн-инструмент, который позволяет увидеть результаты сборки программ на C#. Надеюсь, он переживет эту книгу.

Листинг 7.6. Реальный сборочный код нашего сравнения

```

Указатель на cmp ecx, 5      ← Инструкция сравнения
строку «Х— jne ELSE      ← Инструкция ветвления (перейти, если не равно)
это что-то mov ecx, [0xf59d8cc] ← Указатель на строку «Х равно 5»
другое» ret                ←
→ ELSE: mov ecx, [0xf59d8d0] ←
                           call System.Console.WriteLine(System.String)
                           ret
                           ←
                           ←
                           ← Инструкция возврата

```

Пока сравнение не выполнено, ЦП не знает, будет ли оно успешным, но благодаря прогнозированию ветвлений он делает надежное предположение на основе наблюдений. Процессор заранее начинает обрабатывать инструкции из ветки, по которой предполагается дальнейший путь, и если предсказание успешное, производительность повышается. Вот почему обработка массива случайных значений может происходить медленнее, если она включает сравнение значений: в этом случае предсказание ветвлений не работает. Отсортированный массив эффективнее, потому что ЦП в нем может правильно предсказать порядок обработки и выполняемые ветви.

Помните об этом, когда будете обрабатывать данные. Чем меньше сюрпризов вы заготовите для процессора, тем быстрее он будет работать.

7.4.5. SIMD

ЦП также поддерживают выполнение вычислений с множественными данными одновременно в одной инструкции. Этот метод называется *одиночным потоком команд, множественным потоком данных* (single instruction, multiple data, SIMD). Он может значительно повысить производительность на совместимых архитектурах, если требуется выполнять одни и те же вычисления для нескольких переменных.

Принцип SIMD аналогичен рисованию несколькими ручками, скрепленными вместе. Все ручки будут выполнять одну и ту же операцию, но в разных местах бумаги. Инструкция SIMD будет производить арифметические вычисления с несколькими значениями, но в рамках одной операции.

C# обеспечивает функциональность SIMD через типы `Vector` в пространстве имен `System.Numerics`. Так как поддержка SIMD для разных процессоров

различается, а некоторые ЦП вообще не поддерживают SIMD, сначала нужно проверить, доступен ли он для вашего ЦП:

```
if (!Vector.IsHardwareAccelerated) {
    . . . non-vector implementation here . . .
}
```

Затем нужно выяснить, сколько элементов заданного типа ЦП может обрабатывать одновременно:

```
int chunkSize = Vector< int >.Count;
```

В этом случае мы ищем значения `int`. Количество элементов, которые может обрабатывать ЦП, меняется в зависимости от типа данных. Зная это количество, вы можете обрабатывать буфер по частям.

Допустим, необходимо умножить значения в массиве. Умножение серии значений — распространенная задача обработки данных, будь то изменение громкости звукозаписи или регулировка яркости изображения. Например, при увеличении значений яркости пикселей вдвое изображение становится в два раза ярче. Точно так же при увеличении звука вдвое громкость повышается в два раза. Наивная реализация будет выглядеть так, как показано в следующем листинге. Мы просто перебираем элементы и заменяем имеющееся значение результатом умножения.

Листинг 7.7. Типичное умножение на месте

```
public static void MultiplyEachClassic(int[] buffer, int value) {
    for (int n = 0; n < buffer.Length; n++) {
        buffer[n] *= value;
    }
}
```

Использование типа `Vector` для этих вычислений усложняет код и вроде бы должно его замедлять (листинг 7.8). Мы проверяем поддержку SIMD и запрашиваем размер фрагмента для целочисленных значений. Затем мы просматриваем буфер заданного размера и копируем значения в векторные регистры, создавая экземпляры `Vector<T>`. Этот тип поддерживает стандартные арифметические операторы, поэтому мы просто умножаем векторный тип на заданное число. При этом автоматически будут перемножены сразу все элементы фрагмента. Обратите внимание, что мы объявляем переменную `n` вне цикла `for`, поскольку начинаем с ее последнего значения во втором цикле.

Листинг 7.8. Умножение в стиле «Мы больше не в Канзасе»¹

Вызовы классической реализации,
если SIMD не поддерживается

```
public static void MultiplyEachSIMD(int[] buffer, int value) {
    if (!Vector.IsHardwareAccelerated) {
        MultiplyEachClassic(buffer, value);
    }
    int chunkSize = Vector<int>.Count; ← Запрос количества значений,
    int n = 0; ← которое SIMD может
    for (; n < buffer.Length - chunkSize; n += chunkSize) { ← обрабатывать за один раз
        var vector = new Vector<int>(buffer, n); ← Копирование сегмента
        vector *= value; ← массива в регистры SIMD
        vector.CopyTo(buffer, n); ← Подстановка
        } ← результата
    for (; n < buffer.Length; n++) { ← Обработка оставшихся байтов
        buffer[n] *= value; ← обычным способом
    }
}
```

Перемножение всех значений
за один раз

Слишком много работы, не так ли? Тем не менее показатели бенчмарк-теста впечатляют (табл. 7.3). Код SIMD в два раза быстрее обычного кода. И он может быть еще быстрее в зависимости от обрабатываемых типов данных и выполняемых операций.

Таблица 7.3. Эффективность SIMD

Метод	Среднее значение
MultiplyEachClassic	5.641 мс
MultiplyEachSIMD	2.648 мс

Используйте SIMD для задач, которые подразумевают интенсивные вычисления и одновременное выполнение одной и той же операции с несколькими элементами.

¹ «Мы больше не в Канзасе» — pilotный эпизод американского телесериала «90210: Новое поколение», установивший рекорд канала The CW по количеству зрителей. — *Примеч. ред.*

7.5. ВВОД И ВЫВОД

Ввод/вывод охватывает все взаимодействие ЦП с периферийным оборудованием — диском, сетевым адаптером или даже графическим процессором. И именно ввод/вывод обычно является самым медленным звеном в цепочке. Просто представьте: жесткий диск на самом деле представляет собой вращающийся диск со шпинделем, перебирающий данные. По сути, это роботизированная рука, которая постоянно движется. Пакет в Сети может передаваться со скоростью света, и тем не менее, чтобы обогнать Землю, ему понадобится более 100 миллисекунд. Принтеры намеренно создаются медленными, неэффективными и раздражающими.

В большинстве случаев ускорить сам ввод/вывод невозможно, поскольку его низкая скорость обусловлена законами физики. Но аппаратное обеспечение независимо от ЦП, поэтому оно может выполнять одни задачи, пока ЦП занимается другими. Следовательно, можно совмещать работу ЦП и ввода/вывода, сокращая общее время выполнения операции.

7.5.1. Ускоряйте ввод/вывод

Да, ввод/вывод выполняется медленно из-за присущих аппаратным средствам ограничений, но его можно ускорить. Например, каждое чтение с диска вызывает накладные расходы операционной системы. Рассмотрим код копирования файлов, приведенный в следующем листинге. Он довольно прост — копирует каждый байт, прочитанный из исходного файла, и записывает эти байты в файл назначения.

Листинг 7.9. Простое копирование файла

```
public static void Copy(string sourceFileName,
    string destinationFileName) {

    using var inputStream = File.OpenRead(sourceFileName);
    using var outputStream = File.Create(destinationFileName);
    while (true) {
        int b = inputStream.ReadByte();    ← Чтение байта
        if (b < 0) {
            break;
        }
        outputStream.WriteByte((byte)b);  ← Запись байта
    }
}
```

Проблема в том, что каждый системный вызов подразумевает сложную сопутствующую обработку. Здесь функция `ReadByte()` вызывает функцию чтения

операционной системы. Операционная система вызывает переключение на ядро, то есть ЦП меняет режим работы. Подпрограмма операционной системы ищет дескриптор файла и необходимые структуры данных. Она проверяет, помещен ли уже результат ввода/вывода в кэш, и если нет, то вызывает необходимые драйверы устройств для выполнения фактической операции ввода/вывода на диске. Прочитанная часть памяти копируется в буфер в адресном пространстве процесса. Эти операции происходят почти молниеносно, но все-таки их временем не следует пренебрегать.

Многие устройства ввода/вывода читают/записывают в так называемые *блочные устройства*. Сетевые устройства и устройства хранения обычно являются блочными. Клавиатура — это символьное устройство, потому что она отправляет по одному символу за раз. Блочные устройства не могут считывать данные размером меньше блока, поэтому не имеет смысла считывать что-то меньшее. Например, если размер сектора жесткого диска составляет 512 байт, то эта величина — размер блока для такого диска. Конечно, в современных дисках размер блока может быть и больше, но давайте возьмем в качестве примера именно эту величину и посмотрим, насколько можно повысить производительность при буфере такого размера.

В листинге 7.10 показана рассмотренная выше операция копирования, которая принимает размер буфера в качестве параметра и выполняет чтение и запись с его учетом.

Листинг 7.10. Копирование файла с использованием буферов

```
public static void CopyBuffered(string sourceFileName,
    string destinationFileName, int bufferSize) {
    using var inputStream = File.OpenRead(sourceFileName);
    using var outputStream = File.Create(destinationFileName);
    var buffer = new byte[bufferSize];
    while (true) {
        int readBytes = inputStream.Read(buffer, 0, bufferSize); ← Чтение байтов
        if (readBytes == 0) {                                bufferSize за раз
            break;
        }
        outputStream.Write(buffer, 0, readBytes);           ← Запись байтов bufferSize за раз
    }
}
```

Если мы напишем бенчмарк-тест для функции копирования по одному байту и варианты с разными размерами буфера, то увидим разницу в производительности и выигрыш, который обеспечивает чтение больших фрагментов за раз. Результаты сравнения двух методов представлены в табл. 7.4.

Таблица 7.4. Влияние размера буфера на производительность ввода/вывода

Метод	Размер буфера	Среднее значение
Copy	1	1 351.27 мс
CopyBuffered	512	217.80 мс
CopyBuffered	1024	214.93 мс
CopyBuffered	16384	84.53 мс
CopyBuffered	262144	45.56 мс
CopyBuffered	1048576	43.81 мс
CopyBuffered	2097152	44.10 мс

Даже при размере буфера 512 байт разница очень заметна — операция копирования становится быстрее в шесть раз. Наибольшая же выгода достигается при увеличении буфера до 256 Кбайт, хотя в дальнейшем ускорение операции остается незначительным при любом увеличении размеров буфера. Я проводил эти тесты на компьютере с Windows, где для операций ввода/вывода и управления кэшем по умолчанию используется размер буфера 256 Кбайт. Вот почему после 256 Кбайт выгода становится незначительной. Подобно тому как «фактическое содержимое упаковки может отличаться», может быть другим и реальный опыт работы с операционной системой. При работе с вводом/выводом ищите идеальный размер буфера и не выделяйте больше памяти, чем действительно необходимо.

7.5.2. Делайте ввод/вывод неблокирующим

Многие не понимают, что такое асинхронный ввод/вывод. Его часто путают с многопоточностью, которая представляет собой модель распараллеливания, ускоряющую любую операцию, поскольку процессы выполняются на отдельных ядрах. Асинхронный ввод/вывод — это распараллеливание только операций с большим объемом ввода/вывода, работающее на одном ядре. Многопоточность и асинхронный ввод/вывод можно использовать вместе.

Ввод/вывод асинхронен по своей природе, поскольку внешнее оборудование почти всегда медленнее, чем ЦП, а ЦП не любит ждать и ничего не делать. Такие механизмы, как прерывания и прямой доступ к памяти (DMA, direct memory access), были созданы, чтобы оборудование могло сигнализировать ЦП о завершении операции ввода/вывода и ЦП передал результат. Таким образом, пока аппаратное обеспечение занято обработкой операции ввода/вывода, ЦП может

выполнять другие действия, а затем проверить завершение этой операции. Этот механизм лежит в основе асинхронного ввода/вывода.

На рис. 7.6 показано, как работают оба типа параллелизации. На обеих иллюстрациях вычисление второго кода (CPU Op #2) зависит от результата выполнения первого кода ввода/вывода (I/O Op #1). Поскольку вычисление второго кода нельзя распараллелить в одном потоке, он выполняется последовательно и за большее время, чем при многопоточности на четырехъядерной машине. С другой стороны, при этом нет необходимости использовать потоки или занимать ядра, что дает большие преимущества.

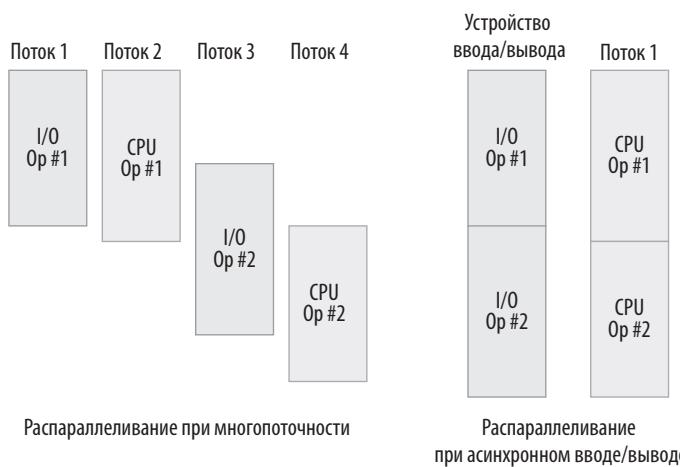


Рис. 7.6. Разница между многопоточностью и асинхронным вводом/выводом

Преимущество в производительности возникает благодаря тому, что асинхронный ввод/вывод обеспечивает естественное распараллеливание кода без выполнения дополнительной работы. Вам даже не нужно создавать еще один поток. Можно выполнять несколько операций ввода/вывода параллельно и собирать результаты без проблем, свойственных многопоточности, таких как гонка. Асинхронный ввод/вывод практичен и масштабируем.

Асинхронный код повышает скорость отклика в механизмах, управляемых событиями, особенно в пользовательских интерфейсах (UI), без использования потоков. На первый взгляд пользовательский интерфейс не имеет ничего общего с вводом/выводом, но пользовательский ввод поступает с устройств ввода/вывода, таких как сенсорный экран, клавиатура или мышь, и пользовательский

интерфейс вызывается событием пользовательского ввода. UI идеальны для асинхронного ввода/вывода и асинхронного программирования в целом. Даже анимации на основе таймера управляются аппаратно, поэтому прекрасно подходят для асинхронного ввода/вывода.

7.5.3. Архаичные способы

До начала 2010-х годов асинхронный ввод/вывод управлялся с помощью обратного вызова. Асинхронные функции операционной системы требовали передачи им функции обратного вызова, чтобы выполнять ее после завершения ввода/вывода. До тех пор можно было заниматься другими задачами. Рассмотренная выше операция копирования файла в устаревшей асинхронной семантике выглядела бы почти так же, как в листинге 7.11. Этот код — сложный и непрезентабельный, и, видимо, именно поэтому бумеры не очень любят асинхронный ввод/вывод. На самом деле я так намучился с написанием этого кода, что пришлось все-таки использовать современные конструкции вроде `Task`, чтобы закончить его. Я приведу этот код, просто чтобы вы оценили современные достижения и поняли, сколько времени они нам экономят.

Самое интересное в этом древнем коде, что результат чудесным образом возвращается немедленно. Это означает, что ввод/вывод работает в фоновом режиме, операция выполняется, и в это время можно заниматься другим делом в том же потоке. Многопоточности нет. На самом деле это одно из огромных преимуществ асинхронного ввода/вывода, поскольку он экономит потоки и может масштабироваться, о чем я расскажу в главе 8. А если заняться нечем, то можно просто ждать завершения.

В листинге 7.11 мы определяем две функции обработки. Первая — асинхронная функция `Task`, называемая `onComplete()`, которая запускается после завершения выполнения других функций. Вторая — локальная функция `onRead()`, которая вызывается каждый раз после завершения операции чтения. Мы передаем этот обработчик в функцию `BeginRead` потока, и она инициирует операцию асинхронного ввода/вывода и регистрирует `onRead` в качестве обратного вызова, который будет вызываться при чтении блока. В `onRead` начинается операция записи буфера, который был только что полностью прочитан, и проверяется, что для следующей операции чтения в качестве обратного вызова установлен тот же обработчик `onRead`. Когда весь код выполнен, запускается функция `onComplete`. Это очень запутанный способ организации асинхронного процесса.

Листинг 7.11. Устаревший код операции копирования файлов с использованием асинхронного ввода/вывода

```
public static Task CopyAsyncOld(string sourceFilename,
    string destinationFilename, int bufferSize) {
    var inputStream = File.OpenRead(sourceFilename);
    var outputStream = File.Create(destinationFilename);

    var buffer = new byte[bufferSize];
    var onComplete = new Task(() => { ←
        inputStream.Dispose();
        outputStream.Dispose();
    });
    ← Вызывается после
    ← завершения функции

    void onRead(IAsyncResult readResult) { ←
        int bytesRead = inputStream.EndRead(readResult); ←
        if (bytesRead == 0) { ←
            onComplete.Start(); ←
            return;
        }
        outputStream.BeginWrite(buffer, 0, bytesRead, ←
            writeResult => { ←
                outputStream.EndWrite(writeResult); ←
                inputStream.BeginRead(buffer, 0, bufferSize, onRead, ←
                    null); ←
                }, null);
        ← Запускает следующую
        ← операцию чтения
    }
    ← Запускает операцию записи
    ← Запускает первую
    ← операцию чтения
    var result = inputStream.BeginRead(buffer, 0, bufferSize, ←
        onRead, null);
    return Task.WhenAll(onComplete); ←
    ← Возвращает ожидаемую
    ← Task для onComplete
}
```

← Вызывается каждый раз после
← завершения операции чтения

← Получает количество
← прочитанных байтов

← Запускает последнюю функцию Task

← Запускает операцию записи

← Подтверждает
← завершение
← операции
записи

← Запускает следующую
← операцию чтения

← Запускает первую
← операцию чтения

← Возвращает ожидаемую
← Task для onComplete

Недостаток такого подхода состоит в том, что чем больше асинхронных операций запускается, тем сложнее за ними следить. Все может легко превратиться в *ад обратных вызовов* (callback hell) — термин, придуманный разработчиками Node.js.

7.5.4. Современные операторы `async/await`

К счастью, потрясающие дизайнеры из Microsoft нашли отличный способ писать код асинхронного ввода/вывода с использованием семантики `async/await`. Этот механизм, впервые представленный в C#, стал настолько популярным и зарекомендовал себя настолько практическим, что вошел и в другие языки программирования, такие как C++, Rust, JavaScript и Python.

Приведенный выше код можно переписать с использованием операторов `async/await` (листинг 7.12). Воистину глоток свежего воздуха! Мы объявляем функцию

с ключевым словом `async` и можем использовать в ней `await`. Операторы `await` задают якорь, но на самом деле не ждут выполнения следующего за ними выражения. Они просто обозначают будущие точки возврата, поэтому уже не нужно задавать новый обратный вызов для каждого продолжения. Код можно писать как обычный синхронный. Благодаря этому функция по-прежнему возвращает результат немедленно, как в листинге 7.11. Обе функции — `ReadAsync` и `WriteAsync` — возвращают объект `Task`, такой как `CopyAsync`. Кстати, в классе `Stream` уже есть функция `CopyToAsync`, упрощающая сценарии копирования, но здесь мы разделили операции чтения и записи, чтобы привести исходный код в соответствие с оригинальным.

Листинг 7.12. Современный код копирования файлов с асинхронным вводом/выводом

```
public async static Task CopyAsync(string sourceFilename, ←
    string destinationFilename, int bufferSize) { ←
    ←
    using var inputStream = File.OpenRead(sourceFilename);
    using var outputStream = File.Create(destinationFilename);
    var buffer = new byte[bufferSize];
    while (true) { ←
        int readBytes = await inputStream.ReadAsync( ←
        buffer, 0, bufferSize);
        if (readBytes == 0) { ←
            break;
        }
        await outputStream.WriteAsync(buffer, 0, readBytes); ←
    }
}
```

Функция объявляется с ключевым словом `async` и возвращает `Task`

Любая операция, следующая за `await`, скрыто преобразуется в обратный вызов

Код с ключевыми словами `async/await` во время компиляции скрыто преобразуется в код, похожий на листинг 7.11, включающий обратные вызовы и т. д. `Async/await` сэкономят вам много времени.

7.5.5. Подводные камни асинхронного ввода/вывода

Не обязательно использовать асинхронные механизмы только для ввода/вывода. Асинхронную функцию можно объявить, не вызывая операций ввода/вывода, и использовать только обработку ЦП. Однако в этом случае ресурсы расходуются без какой-либо выгоды. Компилятор обычно предупреждает о такой ситуации, но я видел много примеров, когда предупреждения намеренно игнорировались, потому что никто не хотел отвечать за последствия исправлений. В результате проблемы с производительностью накапливались, а затем их требовалось исправить все сразу, что приводило к еще большим проблемам.

Сообщайте о таких ситуациях в ходе код-ревью и старайтесь, чтобы ваш голос был услышен.

Работая с `async/await`, всегда имейте в виду, что `await` — это не ожидание. Да, `await` гарантирует, что следующая строка будет запущена после завершения выполнения предыдущей, но это реализуется без ожидания или блокировки, путем скрытых асинхронных обратных вызовов. Если асинхронный код ожидает завершения какой-либо операции, значит, вы что-то делаете не так.

7.6. ЕСЛИ НИЧЕГО НЕ ПОМОГАЕТ, КЭШИРУЙТЕ

Кэширование — один из самых надежных способов быстрого повышения производительности. Инвалидация кэша может быть серьезной проблемой, но ее не возникнет, если кэшировать то, что не требует инвалидации. Вам также не нужен сервис кэширования на отдельном сервере, такой как Redis или Memcached. Вы можете использовать кэш в памяти, подобный предоставляемому Microsoft в классе `MemoryCache` пакета `System.Runtime.Caching`. Да, он не может масштабироваться за определенные пределы, но это вряд ли потребуется, когда проект еще на старте. Ekşi Sözlük обрабатывает 10 миллионов запросов в день на одном сервере БД и на четырех веб-серверах, но по-прежнему использует кэш в памяти.

Не используйте структуры данных, не подходящие для кэширования. В них, как правило, нет механизма вытеснения или истечения срока действия, поэтому они становятся источником утечек памяти и, в конечном итоге, сбоев. Используйте объекты, предназначенные для кэширования. База данных также может быть отличным постоянным кэшем.

Не бойтесь бесконечного срока жизни в кэше, поскольку либо вытеснение кэша, либо перезапуск приложения обязательно произойдут, прежде чем Вселенная закончит свое существование.

ИТОГИ

- Используйте преждевременную оптимизацию для тренировки навыков.
- Не загоняйте себя в угол ненужными оптимизациями.
- Всегда проверяйте результаты оптимизации бенчмарк-тестами.
- Обеспечивайте баланс оптимизации и отзывчивости.

- Возьмите за правило выявлять проблемный код, такой как вложенные циклы, большое количество типов `string` и неэффективные логические выражения.
- При построении структур данных учитывайте преимущества выравнивания памяти для повышения производительности.
- Если вы хотите провести микрооптимизацию, изучите поведение ЦП и используйте такие инструменты, как локальный кэш, конвейерная обработка и SIMD.
- Повышайте производительность ввода/вывода с помощью механизмов буферизации.
- Применяйте асинхронное программирование для параллельного выполнения кода и операций ввода/вывода без потери потоков.
- При крайней необходимости используйте кэш.



Приятная масштабируемость

В этой главе

- ✓ Масштабируемость и производительность
- ✓ Прогрессивная масштабируемость
- ✓ Нарушение правил баз данных
- ✓ Плавная параллелизация
- ✓ Истина в монолите

«Это было самое прекрасное время, это было самое злосчастное время, это был век мудрости и век безумия».

Чарльз Диккенс о масштабируемости

О том, что такое масштабируемость, я узнал еще в 1999 году в связи с техническими решениями, которые использовал для сайта *Ekşi Sözlük*. Вся база данных этого сайта изначально представляла собой один текстовый файл. Операции записи блокировали его, в результате чего все зависало для всех посетителей сайта. Операции чтения также были не очень эффективны — извлечение одной

записи занимало $O(N)$ времени, требуя сканирования всей базы данных. Это был худший из худших дизайнов.

Код зависал не из-за того, что аппаратное обеспечение сервера было медленным. На скорость влияли структуры данных и решения в части параллелизма. В этом суть масштабируемости. Одна только хорошая производительность не сделает систему масштабируемой. Чтобы удовлетворять потребности растущего количества пользователей, потребуется учесть все аспекты дизайна.

Но главное в этой истории то, что ужасный дизайн не имел сколько-нибудь серьезной ценности, поскольку я запустил сайт всего за несколько часов. Первоначальные технические решения не имели значения в долгосрочной перспективе. К тому же я погасил большую часть технического долга. Как только база данных стала создавать слишком много проблем, я изменил ее технологию. Я написал код сайта заново, когда прежняя технология перестала работать. Турецкая пословица гласит: «Караван готовят в дороге», что означает «Решай проблемы на ходу».

Уже не раз в этой книге я советовал семь раз отмерить и один раз отрезать, что, очевидно, противоречит девизу «*Que será, será*»¹. Дело в том, что не существует универсального решения для всех наших проблем. Необходимо владеть всеми возможными инструментами и применять соответствующий конкретному случаю.

С точки зрения системы масштабируемость означает ускорение работы за счет добавления большего количества единиц аппаратного обеспечения. С точки зрения программирования это способность кода поддерживать постоянную скорость отклика в условиях растущего количества запросов. Очевидно, что существует верхний предел допустимой нагрузки для кода, и цель написания масштабируемого кода — максимально увеличить этот предел.

Как и в случае с рефакторингом, к масштабируемости лучше всего подходить инкрементно, небольшими шажками. Можно спроектировать полностью масштабируемую систему с нуля, но количество необходимых для этого усилий и времени, а также возможная выгода плохо согласуются с задачей выпустить продукт как можно скорее.

¹ Популярная песня 1950-х годов, которую исполняла Дорис Дэй, любимая певица моего отца. «*Que Será, Será*» с итальянского переводится как «Что будет, то будет». Это официальный лозунг развертываний по пятницам, за которым обычно следует хит 4 Non Blondes «*What's Up?*» в субботу, и все заканчивается «*Calling It Quits*» Эйми Манн в понедельник.

Некоторые вещи вообще невозможно масштабировать. Как красноречиво сказал Фред Брукс в своей замечательной книге «Мифический человеко-месяц», «девять женщин за месяц ребенка не рожают». Брукс имел в виду, что привлечение большего количества исполнителей к работе над проектом, у которого уже сорваны сроки, только увеличит задержку, но это также справедливо и для определенных аспектов масштабируемости. Например, ядро ЦП никогда не обработает больше инструкций в секунду, чем позволяет его тактовая частота. Да, я говорил, что это количество можно немного превзойти, используя SIMD, прогнозирование ветвлений и т. д., но все-таки для ядра существует верхний предел производительности.

Первый шаг на пути к масштабируемому коду — удаление плохого кода, который препятствует масштабированию. Плохой код создает узкие места, из-за чего производительность не увеличивается даже после добавления аппаратных ресурсов. Удаление части кода может показаться нелогичным. Поэтому рассмотрим эти потенциально узкие места и способы избавиться от них более подробно.

8.1. НЕ ИСПОЛЬЗУЙТЕ БЛОКИРОВКИ

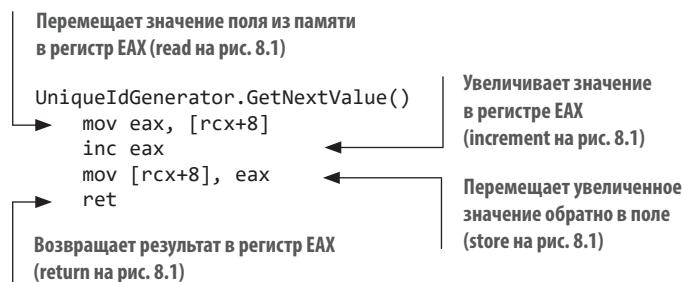
В программировании блокировка — это особенность, позволяющая писать потокобезопасный код. *Потокобезопасный* означает, что фрагмент кода может исправно выполняться, даже если он вызывается двумя или более потоками одновременно.

Рассмотрим класс, отвечающий за создание уникальных идентификаторов для сущностей приложения, и предположим, что ему необходимо генерировать последовательные числовые идентификаторы. Обычно это не очень хорошая идея, как я уже говорил в главе 6. Постепенно увеличивающиеся идентификаторы могут вызвать утечку информации о том, сколько заказов вы получаете в день, сколько у приложения пользователей и т. д. Но предположим существование веской причины использования именно таких идентификаторов, например, как гарантии, что ничего не будет пропущено. Простая реализация будет выглядеть так:

```
class UniqueIdGenerator {
    private int value;
    public int GetNextValue() => ++value;
}
```

Когда несколько потоков используют один и тот же экземпляр класса, они могут получить одно и то же значение или значения, которые не соответствуют

порядку. Это возможно, так как на ЦП выражение `++value` преобразуется в несколько операций: одна считывает значение `value`, другая увеличивает его, третья сохраняет увеличенное значение в поле и, наконец, последняя возвращает результат, как это видно на примере вывода для компилятора JIT на ассемблере в архитектуре x86:¹



Каждая строка представляет собой инструкцию, и ЦП выполняет их последовательно. Если визуализировать несколько ядер ЦП, одновременно выполняющих одни и те же инструкции, будет легче понять, как это вызывает конфликты в классе. На рис. 8.1 видно, что три потока возвращают одно и то же значение 1, хотя функция вызывалась три раза.

Значение поля	Поток № 1	Поток № 2	Поток № 3
0	read	read	
0	increment	increment	read
1	store	store	increment
1	return	return	store
1			return

Рис. 8.1. Одновременное выполнение нескольких потоков приводит к нарушению состояния

Приведенный выше код, использующий регистр EAX, не является потокобезопасным. Состояние, когда все потоки стремятся сами выполнять операции

¹ Компилятор JIT (just in time — точно в срок) преобразует либо исходный код, либо промежуточный код (называемый байт-кодом, IL, IR и т. д.) к собственному набору инструкций архитектуры ЦП, на которой он работает.

с данными, не обращая внимания на другие потоки, называется *состоянием гонки*. ЦП, языки программирования и операционные системы предоставляют множество функций, решающих эту проблему. Обычно все они сводятся к *блокировке* других ядер ЦП, чтобы они не могли выполнять одновременное чтение или запись в ту же область памяти.

В следующем примере используется атомарное приращение, которое напрямую увеличивает значение в ячейке памяти и предотвращает доступ других ядер ЦП к той же области памяти. Благодаря этому предотвращаются проблемы, когда несколько потоков считывают одно и то же значение или пропускают значения:

```
using System.Threading;
class UniqueIdGeneratorAtomic {
    private int value;
    public int GetNextValue() => Interlocked.Increment(ref value);
}
```

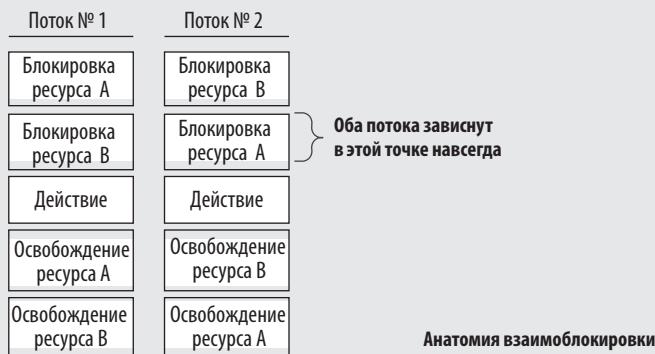
В этом случае блокировка реализуется самим ЦП, и он ведет себя так, как показано на рис. 8.2. Инструкция блокировки ЦП работает, только пока выполняется следующая за ней инструкция, поэтому блокировка автоматически снимается после завершения каждой операции атомарного приращения в памяти. Обратите внимание, что инструкции `return` возвращают не текущее значение поля, а результат операции приращения. В любом случае значение поля увеличивается последовательно.

Значение поля	Поток № 1	Поток № 2	Поток № 3	
0	lock	lock		
1	mem add	Инструкция <code>lock</code> ожидает снятия блокировки		
2	return 1	mem add	Инструкция <code>lock</code> ожидает снятия блокировки	
2		return 2	mem add	
3			return 3	

Рис. 8.2. При атомарном приращении одно ядро ЦП ожидает выполнения операции на другом ядре

БРР-Р, ВЗАИМОБЛОКИРОВКА

Взаимоблокировка возникает, когда одному потоку нужны ресурсы, полученные другим. Попасть в нее довольно легко: поток 1 получает ресурс A и ждет освобождения ресурса B, а поток 2 получает ресурс B и ждет освобождения ресурса A, как показано на следующем рисунке.



Результат похож на бесконечный цикл, ожидающий условия, которое никогда не будет выполнено. Вот почему важно четко указать, какая блокировка и для какой цели используется в коде. Хороший вариант – предусмотреть для блокировок отдельный объект, чтобы отслеживать код, который использует определенные блокировки, и проверять, что они не используются другим кодом. Но с `lock(this)` это невозможно.

Взаимоблокировки приводят к зависанию приложений, и вопреки распространенному мнению, их невозможно исправить, колотя мышкой по столу, крича на монитор или в ярости хлопая дверьми.

От взаимоблокировок не существует волшебной пилюли. Необходимо четко понимать механизмы блокировки в коде и соблюдать правило: всегда начинать с самой последней блокировки. При этом старайтесь снимать блокировки как можно скорее. Некоторые конструкции упрощают использование блокировок, например каналы в языке программирования Go, но и в этом случае взаимоблокировки возможны, хотя и менее вероятны.

Очень часто операции атомарного приращения недостаточно, чтобы сделать код потокобезопасным. Что, если необходимо синхронно обновить два разных счетчика? Если атомарными операциями невозможно обеспечить согласованность, используйте оператор `lock` в C#, как показано в листинге 8.1. Мы рассмотрим

исходный простой пример, но блокировку можно использовать для сериализации любого изменения состояния в том же процессе. Выделим новый объект-пустышку, поскольку .NET хранит информацию о блокировке в заголовке объекта.

Листинг 8.1. Потокобезопасный счетчик с оператором lock в C#

```
class UniqueIdGeneratorLock {
    private int value;
    private object valueLock = new object(); ← Наш объект блокировки
    public int GetNextValue() {
        lock (valueLock) { ← Другие потоки ожидают, пока мы закончим
            return ++value; ← Выход из области видимости автоматически
        }
    }
}
```

Почему мы выделяем новый объект? Разве нельзя было просто использовать `this`, чтобы собственный экземпляр также действовал как блок? Это избавило бы нас от необходимости набирать лишний код. Проблема в том, что наш экземпляр может быть заблокирован кодом, который мы не контролируем. Это может привести к ненужным задержкам или даже *взаимоблокировкам*, потому что наш код может ожидать выполнения другого кода.

Наш собственный код блокировки будет вести себя, как показано на рис. 8.3. Как видите, он менее эффективен, чем атомарное приращение, но по-прежнему совершенно потокобезопасен.

Итак, блокировки вызывают остановку других потоков в ожидании определенного условия. Это может стать одним из самых серьезных препятствий для масштабируемости. Нет ничего хуже, чем тратить драгоценное время процессора на ожидание. Необходимо максимально сокращать время ожидания. Как это сделать?

Во-первых, убедитесь, что блокировки действительно нужны. Я видел код, написанный очень умными программистами, который работает вообще без блокировок, но без необходимости ожидает выполнения определенного условия. Если экземпляра объекта не будет управляться другими потоками, блокировок вообще можно избежать. Но я не утверждаю, что тогда они гарантированно не понадобятся, потому что трудно оценить побочные эффекты кода. Даже объект с локальной областью действия может использовать общие объекты и, следовательно, может потребовать блокировки. Вы должны четко понимать логику и побочные эффекты своего кода. Не используйте блокировки только потому, что они волшебным образом делают код потокобезопасным. Разберитесь, как работают блокировки, и четко определите свои действия.

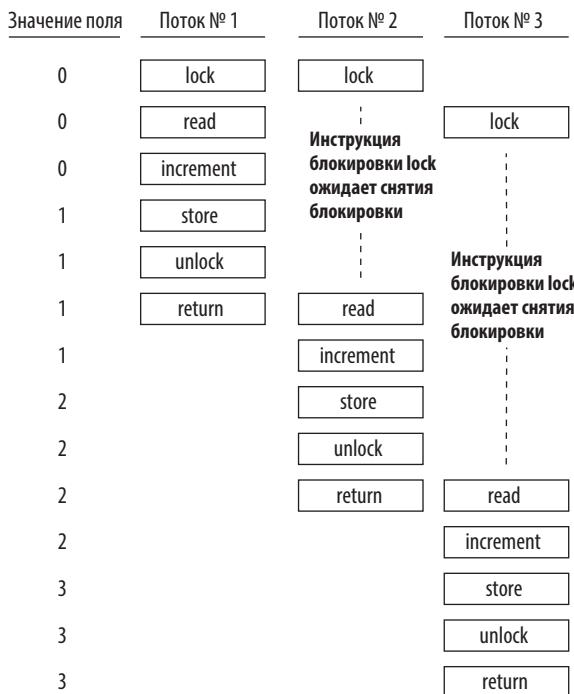


Рис. 8.3. Использование оператора lock в C#, чтобы избежать состояния гонки

Во-вторых, выясните, есть ли у используемой разделяемой структуры данных альтернатива без блокировок. Некоторые структуры данных позволяют организовать доступ к ним напрямую для нескольких потоков без блокировок. Тем не менее реализация структур без блокировок может быть сложной. Они могут быть даже более медленными, чем их блокируемые аналоги, но лучше масштабируемыми. Обычный сценарий, в котором структура без блокировок может быть полезной, — это общие словари, или карты, как их называют на некоторых платформах. Словарь может понадобиться для сущностей, общих для всех потоков, например определенных ключей и значений, и лучший способ его организовать — использовать блокировки.

Рассмотрим пример, в котором требуется хранить токены API в памяти, чтобы не проверять их подлинность в базе данных при каждом обращении. Подходящая структура данных для этой цели — кэш, который также может иметь реализации без блокировок, но разработчики стараются использовать наиболее близкие инструменты, в данном случае это словарь:

```
public Dictionary<string, Token> Tokens { get; } = new();
```

Обратите внимание на замечательный синтаксис `new()` в C# 9.0. Наконец-то закончились темные времена, когда приходилось записывать один и тот же тип дважды при объявлении членов класса. Теперь компилятор определяет тип на основе объявления.

Мы знаем, что словари не являются потокобезопасными, но это становится проблемой, только если имеется несколько потоков, изменяющих структуру данных. Это важно: если у вас есть структура данных, которую вы инициализируете при запуске приложения и никогда не меняете, не обязательно делать ее блокируемой или потокобезопасной. Все структуры только для чтения без побочных эффектов потокобезопасны.

ПОБОЧНЫЕ ЭФФЕКТЫ

Что означает код с побочными эффектами, кроме головной боли и тошноты, возникающих во время ревью? Этот термин родом из функционального программирования. Если функция меняет что-то за пределами своей области действия, это считается побочным эффектом и может касаться не только переменных или полей, но всего чего угодно. Например, если функция записывает сообщение в лог, это вызывает необратимое изменение в выводе лога, что также считается побочным эффектом.

Функцию без побочных эффектов можно запускать сколько угодно, и в среде ничего не изменится. Они называются *чистыми функциями*. Например, функция, которая вычисляет площадь круга и возвращает результат, является чистой функцией:

```
class Circle {  
    public static double Area(double radius) => Math.PI *  
                                              Math.Pow(radius, 2);  
}
```

Она считается чистой не только из-за отсутствия побочных эффектов, но и поскольку члены и функции, к которым она обращается, также являются чистыми. В противном случае они также могут вызвать побочные эффекты, которые появятся в нашей функции. Одно из преимуществ чистых функций состоит в том, что они гарантированно являются потокобезопасными, поэтому их можно без проблем запускать параллельно с другими чистыми функциями.

Поскольку в этом примере нам нужно совершать действия со структурой данных, требуется интерфейс-обертка для обеспечения блокировки, как показано в листинге 8.2. В методе `get` видно, что если токен не найден в словаре, он создается

заново путем чтения соответствующих данных из базы данных. Чтение из базы данных может занять много времени, поэтому выполнение всех запросов будет приостановлено до завершения этой операции.

Листинг 8.2. Потокобезопасный словарь на основе блокировки

```
class ApiTokens {
    private Dictionary<string, Token> tokens { get; } = new(); ← Это общий
    public void Set(string key, Token value) {
        lock (tokens) { ←
            tokens[key] = value; ← Блокировка здесь по-прежнему нужна, потому что
        } ←
    } ←
    public Token Get(string key) {
        lock (tokens) {
            if (!tokens.TryGetValue(key, out Token value)) {
                value = getTokenFromDb(key); ← Этот вызов может занять много времени,
                tokens[key] = value; ← блокируя остальные вызовы
                return tokens[key];
            }
            return value;
        }
    }
    private Token getTokenFromDb(string key) {
        . . . a time-consuming task . . .
    }
}
```

Приведенный вариант вообще не масштабируется, и здесь отлично подойдет альтернатива без блокировок. .NET предоставляет два набора потокобезопасных структур данных. В одном из них имена начинаются с `Concurrent*` и используются кратковременные блокировки. Они оптимизированы так, чтобы удерживаться в течение короткого времени, что делает их довольно быстрыми и, возможно, более простыми, чем альтернатива без блокировок. Другой набор — `Immutable*`, в котором исходные данные никогда не изменяются, но каждая операция модификации создает новую копию данных с изменениями. Он действительно работает медленно, но иногда может быть предпочтительнее, чем `Concurrent`.

Если использовать `ConcurrentDictionary`, код станет более масштабируемым, как показано в следующем листинге. Как видите, операторы `lock` больше не нужны, а длительный запрос выполняется параллельно с другими запросами и почти ничего не блокирует.

Листинг 8.3. Потокобезопасный словарь без блокировок

```

class ApiTokensLockFree {
    private ConcurrentDictionary<string, Token> tokens { get; } = new();

    public void Set(string key, Token value) {
        tokens[key] = value;
    }

    public Token Get(string key) {
        if (!tokens.TryGetValue(key, out Token value)) {
            value = getTokenFromDb(key); ← Теперь это работает параллельно!
            tokens[key] = value;
            return tokens[key];
        }
        return value;
    }

    private Token getTokenFromDb(string key) {
        . . . a time-consuming task . . .
    }
}

```

Небольшой минус заключается в том, что несколько запросов могут выполнить дорогостоящую операцию, такую как `getTokenFromDb`, параллельно для одного и того же токена, поскольку блокировки больше этому не препятствуют. В худшем случае одна и та же трудоемкая операция будет без необходимости выполняться параллельно для одного и того же токена, но даже тогда вы не будете блокировать другие запросы, поэтому, скорее всего, производительность окажется выше, чем при альтернативном сценарии. Иногда есть смысл отказаться от блокировок.

8.1.1. Блокировка с двойной проверкой

Сложно обеспечить создание только одного экземпляра объекта, когда его запрашивают несколько потоков. А если два потока делают одновременно один и тот же запрос? Предположим, что у нас есть объект кэша. Если мы случайно предоставим два разных экземпляра, разные части кода будут иметь разный кэш, что приведет к несоответствиям или потерям. Чтобы избежать этого, вы защищаете код инициализации блокировкой, как показано в следующем листинге. Статическое свойство `Instance` удерживает блокировку перед созданием объекта, поэтому другие экземпляры не создадут один и тот же экземпляр дважды.

Листинг 8.4. Как обеспечить условие, что будет создан только один экземпляр

```
class Cache {
    private static object instanceLock = new object();
    private static Cache instance;
    public static Cache Instance {
        get {
            lock(instanceLock) {
                if (instance is null) {
                    instance = new Cache();
                }
            }
            return instance;
        }
    }
}
```

Объект, используемый для блокировки

Значение кэшированного экземпляра

Прочие инициаторы вызова ожидают выполнения другого потока

Объект создается только один раз!

Код работает normally, но каждый доступ к свойству `Instance` приводит к активации блокировки. Это чревато ненужным ожиданием. Наша цель — сократить блокировку. Можно добавить вторичную проверку значения экземпляра: вернуть его значение перед получением блокировки, если он уже инициализирован, и активировать блокировку только в случае, если он не был инициализирован, как показано в листинге 8.5. Это простое дополнение, но оно устраняет 99,9 % конфликтов блокировок в коде, делая его более масштабируемым. Вторичная проверка внутри оператора `lock` необходима, поскольку существует небольшая вероятность, что другой поток уже инициализировал значение и снял блокировку непосредственно перед тем, как мы ее установили.

Листинг 8.5. Блокировка с двойной проверкой

```
public static Cache Instance {
    get {
        if (instance is not null) {
            return instance;
        }
        lock (instanceLock) {
            if (instance is null) {
                instance = new Cache();
            }
            return instance;
        }
    }
}
```

Обратите внимание на проверку на `null` в C# 9.0, основанную на сопоставлении с шаблоном

Возвращает экземпляр, ничего не блокируя

Блокировку с двойной проверкой можно провести не для всех структур данных. Например, она невозможна для элементов словаря, потому что во время операций со словарем невозможно осуществлять чтение из него потокобезопасным способом вне блокировки.

С# прошел долгий путь и значительно упростил безопасную одноэлементную инициализацию с помощью вспомогательных классов, таких как `LazyInitializer`. Тот же код свойства можно записать более простым способом. `LazyInitializer` выполняет скрытую блокировку с двойной проверкой, избавляя вас от дополнительных усилий.

Листинг 8.6. Безопасная инициализация с помощью `LazyInitializer`

```
public static Cache Instance {
    get {
        return LazyInitializer.EnsureInitialized(ref instance);
    }
}
```

Существуют и другие сценарии, в которых блокировка с двойной проверкой может оказаться полезной. Например, если необходимо убедиться, что количество элементов в списке не превышает заданное значение, можно безопасно проверить свойство `Count`, поскольку во время проверки не нужно обращаться ни к одному из элементов списка. `Count` обычно предоставляет простой доступ к полю и является потокобезопасным, если только читаемое число не используется в итерациях элементов. Пример показан в следующем листинге, и он полностью потокобезопасен.

Листинг 8.7. Альтернативные сценарии блокировки с двойной проверкой

```
class LimitedList<T> {
    private List<T> items = new();

    public LimitedList(int limit) {
        Limit = limit;
    }

    public bool Add(T item) {
        if (items.Count >= Limit) { ← Первая проверка за пределами блокировки
            return false;
        }
        lock (items) {
            if (items.Count >= Limit) { ← Вторая проверка внутри блокировки
                return false;
            }
        }
    }
}
```

```
        items.Add(item);
        return true;
    }
}

public bool Remove(T item) {
    lock (items) {
        return items.Remove(item);
    }
}

public int Count => items.Count;
public int Limit { get; }
}
```

Вы, наверное, заметили, что код в листинге 8.7 не содержит свойства индексатора `indexer` для доступа к элементам списка по их индексам. Это связано с невозможностью обеспечить потокобезопасную нумерацию при прямом доступе к индексу без полной блокировки списка перед его нумерацией. Приведенный класс полезен только для подсчета элементов, но не для доступа к ним. Однако доступ к самому свойству счетчика вполне безопасен, поэтому его можно использовать в блокировке с двойной проверкой, чтобы улучшить масштабируемость.

8.2. СМИРИТЕСЬ С НЕСООТВЕТСТВИЯМИ

Базы данных предоставляют множество функций, позволяющих избежать несоответствий: блокировки, транзакции, атомарные счетчики, журналы транзакций, контрольные суммы, моментальные снимки и т. д. Все потому, что БД предназначены для систем, в которых хранение недостоверных данных недопустимо: банков, ядерных реакторов и приложений для знакомств.

Надежность относительна. Чтобы повысить производительность и масштабируемость, допустимо пожертвовать некоторой частью надежности. NoSQL – это философия отказа от определенных преимуществ согласованности традиционных реляционных баз данных, таких как внешние ключи и транзакции, во имя повышенной производительности, масштабируемости и неясности.

Чтобы воспользоваться преимуществами такого подхода, не обязательно использовать только NoSQL. Аналогичные результаты достижимы и при работе с обычной базой данных, такой как MySQL или SQL Server.

8.2.1. Страшный NOLOCK

NOLOCK дает подсказку, что механизм SQL, который его читает, может быть не согласован и содерхать данные еще не подтвержденных транзакций. Звучит устрашающе, но так ли это на самом деле? Рассмотрим Blabber, платформу микроблогов, которую мы обсуждали в главе 4. При публикации нового поста необходимо обновлять таблицу-счетчик постов. Если пост не опубликован, счетчик не должен увеличиваться. Пример кода будет выглядеть, как показано в листинге 8.8. Мы оборачиваем все в транзакцию, поэтому если операция завершится ошибкой на каком-то этапе, количество постов останется согласованным.

Листинг 8.8. Сказ о двух таблицах

```
public void AddPost(PostContent content) {
    using (var transaction = db.BeginTransaction()) {
        db.InsertPost(content);           ← Вставить пост в его таблицу
        int postCount = db.GetPostCount(userId); ← Извлечь счетчик постов
        postCount++;
        db.UpdatePostCount(userId, postCount); ← Обновить счетчик постов
    }
}
```

Инкапсулировать
все в транзакции

← Вставить пост в его таблицу

← Извлечь счетчик постов

← Обновить счетчик постов

Код напоминает пример генератора уникальных идентификаторов из предыдущего раздела: помните, как потоки параллельно выполняли чтение, увеличение и сохранение, и мы использовали блокировку, чтобы гарантировать согласованность значений? Здесь происходит то же самое, и мы жертвуем масштабируемостью. Но нужна ли нам такая согласованность? Представлю вам другую идею достижения согласованности.

Согласованность в конечном счете (eventual consistency) означает, что согласованность обеспечивается с задержкой. В нашем примере неверные счетчики сообщений будут обновляться через определенные временные интервалы. Основное преимущество такого подхода состоит в том, что блокировки становятся не нужны. Пользователи вряд ли обратят внимание на неверные данные счетчика сообщений, прежде чем он будет исправлен. При этом обеспечивается масштабируемость, поскольку чем меньше блокировок, тем больше можно выполнять параллельных запросов к базе данных.

Периодический запрос, обновляющий таблицу, будет удерживать в ней блокировки, но это будут блокировки отдельных элементов, возможно, определенной строки или, в худшем случае, одной страницы. Можно также ввести блокировки

с двойной проверкой: сначала запустить запрос только для чтения о том, какие строки необходимо обновить, а затем запрос на обновление. Это гарантирует, что в базе данных не придется ничего блокировать из-за простого обновления.

Такой запрос будет выглядеть, как показано в листинге 8.9. Сначала мы выполняем запрос `SELECT` для выявления несовпадающих счетчиков, который не сохраняет блокировки. Затем обновляем счетчик сообщений, основываясь на несоответствиях записей. Можно пакетировать эти обновления, но индивидуальное выполнение будет сохранять детализированные блокировки, возможно, на уровне строк. Это обеспечит возможность большего числа запросов к одной таблице, не сохраняя блокировку дольше необходимого. Недостаток этого метода состоит в том, что обновление каждой отдельной строки будет занимать больше времени, но в конце концов и оно завершится.

Листинг 8.9. Периодически выполняемый код для достижения конечной согласованности

```
public void UpdateAllPostCounts() {
    var inconsistentCounts = db.GetMismatchedPostCounts();
    foreach (var entry in inconsistentCounts) {
        db.UpdatePostCount(entry.UserId, entry.ActualCount);
    }
}
```

При выполнении этого запроса блокировки не сохраняются

Блокировка сохраняется только для одной строки

Запрос `SELECT` в SQL не блокирует таблицу, но может быть задержан другой транзакцией. Вот тут-то и появляется подсказка `NOLOCK`. Она позволяет запросу считывать *грязные данные*, но взамен ему уже не нужно соблюдать блокировки, сохраняемые другими запросами или транзакциями. Реализовать это просто. Например, в SQL Server вместо `SELECT * FROM customers` записываем `SELECT * FROM customers (NOLOCK)`, и `NOLOCK` применяется к таблице клиентов `customers`.

Что такое грязные данные? Если транзакция начинает добавлять записи в базу данных, но еще не завершена, эти записи считаются грязными. То есть запрос с подсказкой `NOLOCK` может возвращать строки, которые еще не существуют в базе данных или никогда не будут в ней существовать. Во многих сценариях такой уровень несогласованности может быть допустим. Например, не стоит использовать `NOLOCK` при аутентификации пользователя, потому что это может стать проблемой безопасности, но он вполне подойдет для показа постов. В худшем случае вы увидите пост, который существует очень недолго и исчезнет при следующем обновлении.

Возможно, вы уже сталкивались с подобным на социальных платформах. Пользователи удаляют свой контент, но удаленные посты продолжают появляться в вашей ленте, хотя вы получаете сообщение об ошибке, если пытаетесь с ними взаимодействовать. Такая ситуация возникает, поскольку платформа допускает некоторый уровень несогласованности ради сохранения масштабируемости.

NOLOCK можно применить ко всему содержимому, запустив сначала инструкцию SQL, которая называется очень концептуально: `SET TRANSACTION ISOLATION LEVEL READ_UNCOMMITTED`. Мне кажется, у Pink Floyd есть песня с похожим названием. В любом случае это утверждение вполне осмысленно и хорошо передает на-мерения.

Не бойтесь несогласованности, если знаете о ее последствиях. Если вы ясно пред-ставляете результаты компромиссов, то можете предпочесть преднамеренную несогласованность, чтобы сохранить масштабируемость.

8.3. НЕ КЭШИРУЙТЕ ПОДКЛЮЧЕНИЯ К БАЗЕ ДАННЫХ

Достаточно частая ошибка в коде — установление одного соединения с базой дан-ных и его совместное использование. На бумаге идея выглядит привлекательно: она позволяет избежать накладных расходов на подключение и аутентификацию для каждого запроса, поэтому запросы выполняются быстрее. Кроме того, код с обилием команд открытия и закрытия соединения выглядит громоздко. Но дело в том, что при единственном соединении с базой данных невозможно вы-полнять параллельные запросы к ней. Эффективно выполняется только один запрос за раз. Это серьезный барьер для масштабируемости, как видно из рис. 8.4.



Рис. 8.4. Узкое место, возникающее при совместном использовании одного соединения в приложении

Одно-единственное соединение имеет и другие недостатки. Для выполнения запросов могут требоваться разные области транзакций, и при попытке одновременно использовать одно соединение для нескольких запросов может возникнуть конфликт.

Приходится согласиться, что частично эта проблема возникает из-за того, что соединения называются соединениями, хотя на самом деле они не являются таковыми. На самом деле большинство клиентских библиотек соединения с базами данных не открывают новое соединение при создании объекта подключения. Вместо этого они поддерживают определенное количество уже открытых соединений и просто используют одно из них. Когда вы думаете, что устанавливаете соединение, вы на самом деле извлекаете уже открытое из так называемого *пула соединений*, или *пула подключений* (connection pool). Когда вы закрываете соединение, оно помещается обратно в пул, и его состояние сбрасывается, поэтому ранее выполнявшиеся запросы не повлияют на новые.

Я уже слышу, как вы говорите: «Я знаю, что делать! Я просто буду поддерживать активное соединение для каждого запроса и закрою соединение, когда выполнение запроса завершится!». Действительно, это позволит избежать блокировок параллельно выполняемых запросов. На рис. 8.5 мы видим, что для каждого запроса устанавливается отдельное соединение и благодаря этому они выполняются параллельно.

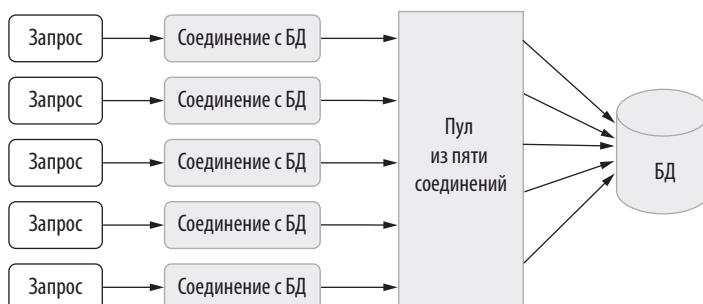


Рис. 8.5. Одно соединение на HTTP-запрос

Проблема в том, что если число запросов в нашем примере будет больше пяти, то клиенту придется ждать, пока пул соединений выделит для него доступное соединение. Запросы помещаются в очередь, и их поток не может масштабироваться, даже если они не выполняются в данный момент, поскольку пул соединений не знает, используется ли запрошенное соединение, если только оно не закрыто явным образом. Эта ситуация изображена на рис. 8.6.

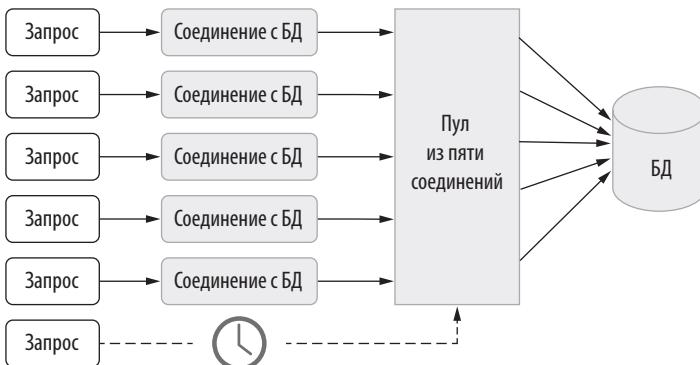


Рис. 8.6. Объекты, блокирующие новые запросы

А если я скажу, что существует метод, полностью противоречащий здравому смыслу, но обеспечивающий максимальную масштабируемость кода? Секрет в том, чтобы поддерживать соединения только на время жизни запросов. Это позволит вернуть соединение в пул максимально быстро, делая его доступным для ожидающих запросов и обеспечивая наилучшую масштабируемость. На рис. 8.7 показано, как это работает. Пул соединений обслуживает не более трех запросов одновременно, оставляя место для еще одного или двух.

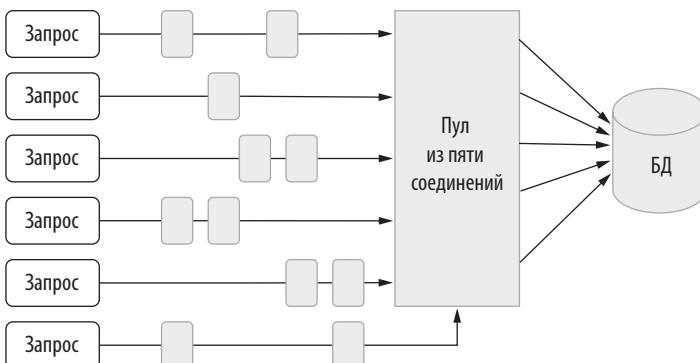


Рис. 8.7. Позапросное соединение с базой данных

Причина, по которой этот метод работает, заключается в том, что запрос никогда не сводится только к выполнению, происходит еще и некоторая обработка. Это

означает, что время, в течение которого вы удерживаете подключение, пока выполняются другие действия, тратится впустую. Оставляя соединения открытыми как можно дольше, вы делаете доступными для запросов максимальное их количество.

Однако это требует дополнительных усилий. Рассмотрим пример обновления предпочтений пользователя на основе его имени. Обычно такой запрос выглядит, как показано в следующем листинге. Он запускается сразу, без учета времени жизни соединения.

Листинг 8.10. Типичное выполнение запроса с экземпляром общего соединения

```
public void UpdateCustomerPreferences(string name, string prefs) {
    int? result = MySqlHelper.ExecuteScalar(customerConnection, ←
        "SELECT id FROM customers WHERE name=@name", ←
        new MySqlParameter("name", name)) as int?; ←
    if (result.HasValue) {
        MySqlHelper.ExecuteNonQuery(customerConnection, ←
            "UPDATE customer_prefs SET pref=@prefs", ←
            new MySqlParameter("prefs", prefs));
    }
}
```

Использование общего соединения

Это объясняется существованием открытого соединения, которое можно использовать повторно. Если предусмотреть открытие и закрытие соединения, код станет немного сложнее, как в листинге 8.11.

Листинг 8.11. Установление соединений для каждого запроса

```
public void UpdateCustomerPreferences(string name, string prefs) {
    using var connection = new MySqlConnection(connectionString); ←
    connection.Open(); ←
    int? result = MySqlHelper.ExecuteScalar(customerConnection, ←
        "SELECT id FROM customers WHERE name=@name", ←
        new MySqlParameter("name", name)) as int?; ←
    //connection.Close(); | Это просто глупо
    //connection.Open(); | Церемония
    if (result.HasValue) { | соединения
        MySqlHelper.ExecuteNonQuery(customerConnection, ←
            "UPDATE customer_prefs SET pref=@prefs", ←
            new MySqlParameter("prefs", prefs));
    }
}
```

Вы можете подумать, что необходимо закрывать и открывать соединение между двумя запросами, чтобы можно было вернуть его в пул и сделать доступным для других запросов, но это совершенно не обязательно. Экономия времени при этом будет минимальной, а вот накладные расходы заметно вырастут. Также обратите

внимание, что мы не закрываем соединение явно при завершении функции, поскольку оператор `using` в начале гарантирует, что все ресурсы, относящиеся к объекту подключения, освобождаются сразу после выхода из функции, что, в свою очередь, приводит к закрытию соединения. Можно обернуть процедуру установления соединения во вспомогательную функцию, чтобы не писать ее каждый раз:

```
using var connection = ConnectionHelper.Open();
```

Это сэкономит несколько нажатий клавиш, но может привести к ошибкам. Если забыть поместить оператор `using` перед вызовом, а компилятор не напомнит об этом, то соединение не будет закрыто.

8.3.1. В виде ORM

К счастью, современные инструменты объектно-реляционного отображения (ORM, object relational mapping) — это библиотеки, которые скрывают тонкости базы данных, предоставляя совершенно другой набор сложных абстракций. Например, Entity Framework делает все автоматически, поэтому вам не нужно думать об открытии или закрытии соединения. Она устанавливает соединение, когда это необходимо, и разрывает его, когда оно больше не требуется. В Entity Framework можно использовать один общий экземпляр `DbContext` на протяжении всего времени существования запроса. Однако вы вряд ли захотите использовать один экземпляр для всего приложения, потому что `DbContext` не является потокобезопасным.

Запрос из листинга 8.11 можно написать с помощью Entity Framework так, как показано в листинге 8.12. В таких запросах можно использовать и синтаксис LINQ, но я считаю функциональный синтаксис более удобным для чтения и лучше компонуемым.

Листинг 8.12. Множественные запросы с Entity Framework

```
public void UpdateCustomerPreferences(string name, string prefs) {
    int? result = context.Customers
        .Where(c => c.Name == name)
        .Select(c => c.Id)
        .Cast<int?>()
        .SingleOrDefault();
    if (result.HasValue) {
        var pref = context.CustomerPrefs
            .Where(p => p.CustomerId == result)
            .Single();
        pref.Prefs = prefs;
        context.SaveChanges();
    }
}
```

Соединение будет открываться до и автоматически закрываться после каждой из этих строк

У вас может быть больше пространства для масштабирования приложения, если вы будете разбираться в семантике времени жизни классов `Connection`, пулов соединений и фактических сетевых соединений с базой данных.

8.4. НЕ ИСПОЛЬЗУЙТЕ ПОТОКИ

Масштабируемость — это не только высокая степень параллелизации, но и экономия ресурсов. Невозможно масштабироваться за пределы имеющейся памяти или ресурсов ЦП. ASP.NET Core использует структуру пула потоков, чтобы поддерживать определенное количество потоков для параллельного обслуживания веб-запросов. Принцип очень похож на пул соединений: наличие набора уже инициализированных потоков позволяет избежать накладных расходов на их создание. Количество потоков в пуле обычно превышает число ядер ЦП, потому что потоки часто ожидают завершения чего-либо, чаще всего ввода/вывода. Таким образом, выполнение каких-то потоков может быть запланировано на одном ядре ЦП, в то время как другие потоки ожидают завершения ввода/вывода. На рис. 8.8 видно, как количество потоков, превышающее число ядер ЦП, помогает использовать их более эффективно. Пока один поток ожидает завершения операции, на том же ядре запускается другой, таким образом обслуживается больше потоков, чем доступно ядер ЦП.

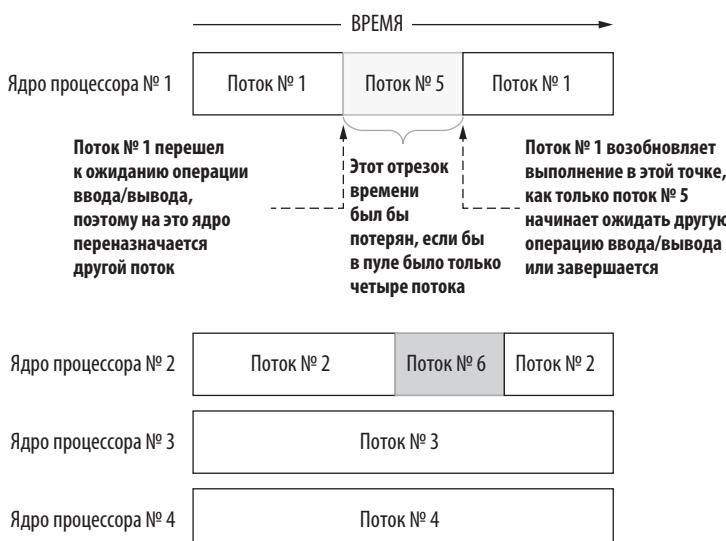


Рис. 8.8. Оптимизация использования ЦП, когда количество потоков больше, чем число ядер процессора

Этот вариант лучше, чем если бы количество потоков было равно числу ядер ЦП, но драгоценное время ЦП используется все еще не максимально эффективно. Операционная система дает потокам время на выполнение, а затем уступает ядро другим потокам, чтобы каждый из них получал возможность запускаться в разумные сроки. Этот метод называется *вытеснением*, и именно так раньше работала многозадачность на одноядерных процессорах. Операционная система жонглировала всеми потоками на одном ядре, создавая иллюзию многозадачности. К счастью, поскольку большинство потоков ожидают ввода/вывода, пользователи не замечают, что потоки запускаются на одном доступном им ЦП по очереди. Только если будет запущено ресурсоемкое приложение, недостаток возможностей ЦП станет заметен.

Из-за специфики организации работы потоков в операционных системах количество потоков в пуле, превышающее число ядер ЦП, позволяет лишь частично повысить эффективность использования ЦП, но фактически может даже навредить масштабируемости. Если потоков слишком много, все они начинают получать меньшую долю времени процессора, поэтому им требуется работать дольше, и в результате сайт или API начинает тормозить.

Лучший способ эффективно использовать время, затрачиваемое на ожидание ввода/вывода, — это асинхронный ввод/вывод, который мы обсуждали в главе 7. Асинхронный ввод/вывод выполняется явно: ключевое слово `await` означает, что поток будет ожидать результата обратного вызова, поэтому может использоваться другими запросами, в то время как аппаратное обеспечение обрабатывает сам запрос ввода/вывода. Таким образом, несколько запросов могут обслуживаться в одном и том же потоке параллельно, как показано на рис. 8.9.

Асинхронный ввод/вывод предоставляет множество преимуществ. Обновить имеющийся код для его использования просто, если у вас есть фреймворк, поддерживающий асинхронные вызовы. Например, в ASP.NET Core действия контроллера или обработчики Razor Page можно описать и как обычные, и как асинхронные методы, поскольку фреймворк создает для них необходимые надстройки. Все, что вам необходимо сделать, — переписать функцию с использованием асинхронных вызовов и пометить метод как `async`. Да, проверять, что код работает правильно, все равно придется, но это по-прежнему просто.

Вернемся к примеру из листинга 8.6 и сделаем его асинхронный вариант. Различия выделены в листинге 8.13 жирным начертанием. Ниже я разберу их подробно.

Листинг 8.13. Преобразование блокирующего кода в асинхронный

```
public async Task UpdateCustomerPreferencesAsync(string name,
    string prefs) {
    int? result = await MySqlHelper.ExecuteScalarAsync(
        customerConnection,
        "SELECT id FROM customers WHERE name=@name",
        new MySqlParameter("name", name)) as int?;
    if (result.HasValue) {
        await MySqlHelper.ExecuteNonQueryAsync(customerConnection,
            "UPDATE customer_prefs SET pref=@prefs",
            new MySqlParameter("prefs", prefs));
    }
}
```

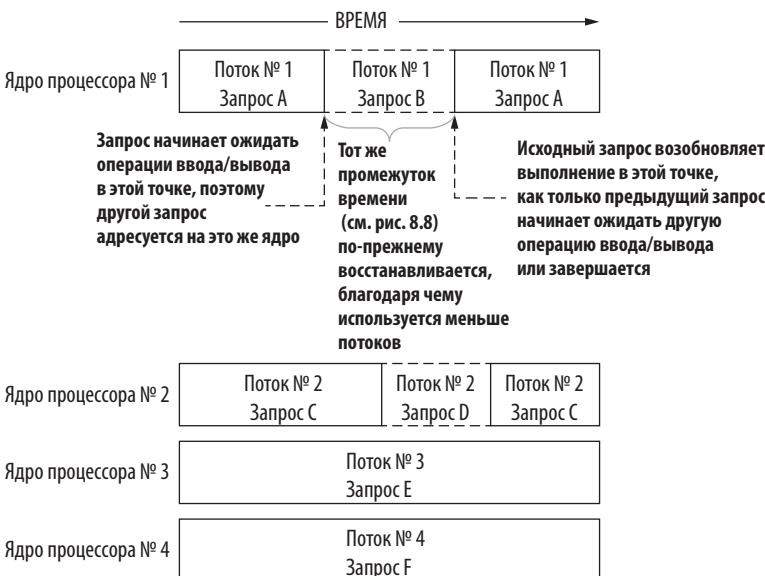


Рис. 8.9. Улучшение параллелизма с меньшим количеством потоков и асинхронным вводом-выводом

Важно понимать цель изменений, чтобы делать их осознанно и правильно.

- На самом деле в названии асинхронных функций не обязательно должен присутствовать суффикс `Async`, но это соглашение помогает понять, чего ожидать. `Async` не является частью сигнатуры функции. Необходимо просмотреть исходный код, чтобы понять, действительно ли функция, содержащая его,

является асинхронной. Асинхронная функция возвращается немедленно, даже если вы не ожидаете этого и ошибочно предполагаете, что она завершила работу. Страйтесь придерживаться соглашений, если только вам не требуется задавать конкретные имена, например, для действий контроллера, потому что они также могут назначать маршруты URL. Соглашения об именовании помогают также, если необходимо обеспечить две перегрузки одной и той же функции, поскольку возвращаемые типы не являются отличительным признаком перегрузки. Вот почему в .NET имена почти всех асинхронных методов имеют суффикс `Async`.

- Ключевое слово `async` в начале объявления функции означает только, что в функции можно использовать `await`. Компилятор скрыто принимает эти асинхронные операторы, генерирует необходимый код обработки и преобразует их в серию обратных вызовов.
- Все асинхронные функции должны возвращать `Task` или `Task<T>`. Асинхронная функция без возвращаемого значения также может иметь возвращаемый тип `void`, но это создает проблемы. Например, меняется семантика обработки исключений, и вы теряете компонуемость. Компонуемость в асинхронных функциях позволяет определить действие, которое происходит, когда функция завершается программным способом с использованием методов `Task`, таких как `ContinueWith`. Поэтому асинхронные функции, не имеющие возвращаемого значения, всегда должны использовать `Task`. При добавлении ключевого слова `async` значения после операторов `return` автоматически обрабатываются в `Task<T>`, поэтому не нужно создавать `Task<T>` самостоятельно.
- Ключевое слово `await` гарантирует, что следующая строка будет выполнена только после завершения выполнения предыдущей. Если не добавить `await` перед множественными асинхронными вызовами, они начнут выполняться параллельно, и иногда это даже хорошо, но нужно убедиться, что ожидается их завершение, потому что в противном случае выполнение задач может быть прервано. С другой стороны, в параллельных операциях часто возникают ошибки; например, в Entity Framework Core невозможно выполнять несколько запросов параллельно, используя один и тот же `DbContext`, поскольку `DbContext` не является потокобезопасным. Однако таким образом можно распараллелить другие операции ввода-вывода, например чтение файла. Предположим, необходимо запустить два веб-запроса одновременно так, чтобы они не ждали друг друга. Можно выполнять два веб-запроса одновременно и дождаться их завершения, как показано в листинге 8.14. Мы определяем функцию, которая получает список URL-адресов и запускает задачу загрузки для каждого URL-адреса, не дожидаясь завершения загрузки предыдущего, то есть параллельно в одном потоке. Можно использовать один экземпляр

объекта `HttpClient`, потому что он потокобезопасен. Функция ожидает завершения всех задач и формирует окончательный ответ из их результатов.

Листинг 8.14. Параллельная загрузка нескольких веб-страниц в одном потоке

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace Connections {
    public static class ParallelWeb {
        public static async Task<Dictionary<Uri, string>> DownloadAll(IEnumerable<Uri> uris) {
            var runningTasks = new Dictionary<Uri, Task<string>>(); ← Полученный тип
            var client = new HttpClient(); ← Достаточно одного экземпляра
            foreach (var uri in uris) {
                var task = client.GetStringAsync(uri); ← Запускает задачу,
                runningTasks.Add(uri, task); ← Сохраняет задачу в каком-то
                                              расположении
            }
            await Task.WhenAll(runningTasks.Values); ← Ждет, пока все задачи
            return runningTasks.ToDictionary(kp => kp.Key, ← будут выполнены
                                              kp => kp.Value.Result); ← Создает новый словарь
                                              из результатов
                                              выполненных задач
        }
    }
}
```

Временное хранилище для отслеживания запущенных задач

Полученный тип

Достаточно одного экземпляра

Сохраняет задачу в каком-то расположении

Запускает задачу, но не ждет ее завершения

Ждет, пока все задачи будут выполнены

Создает новый словарь из результатов выполненных задач

8.4.1. Подводные камни асинхронного кода

Преобразуя код в асинхронный, необходимо помнить об ограничениях. Попытка сделать весь код асинхронным может только все ухудшить. Рассмотрим некоторые из подводных камней.

Если нет ввода/вывода, не должно быть и асинхронности

Если функция не вызывает асинхронную функцию, то ее не обязательно делать асинхронной. Асинхронное программирование помогает увеличить масштабируемость, только если оно используется для операций, связанных с вводом/выводом. Асинхронность операций, связанных с ЦП, не поможет добиться лучшей масштабируемости, поскольку для их выполнения потребуются отдельные потоки, в отличие от операций ввода/вывода, которые могут выполняться параллельно в одном потоке. Компилятор также может выдать предупреждение, если вы попытаетесь использовать ключевое слово `async` в функции, которая не выполняет другие асинхронные операции. Если вы проигнорируете эти

предупреждения, то просто получите излишне раздутый и, возможно, более медленный код из-за асинхронных лесов, добавленных в функцию. Вот пример ненужного использования ключевого слова `async`:

```
public async Task<int> Sum(int a, int b) {
    return a + b;
}
```

Такое действительно бывает, я сам видел, как разработчики без уважительной причины задавали свои функции как асинхронные. Для введения асинхронности всегда должно быть четкое обоснование.

Не смешивайте синхронность и асинхронность

Очень сложно безопасно вызвать асинхронную функцию в синхронном контексте. Кто-то скажет: «Да просто вызови `Task.Wait()` или `Task.Result`, и все будет в порядке». Нет, не будет. Этот код станет преследовать вас во сне, он будет вызывать проблемы в самый неподходящий момент, и в конце концов вам захочется спать спокойно, без кошмаров.

Самая большая проблема состоит в том, что асинхронные функции могут вызвать взаимоблокировки из-за других функций, которые зависят от завершения кода вызывающей стороны. Обработка исключений также может быть нелогичной, поскольку она будет заключена в отдельный `AggregateException`.

Не добавляйте асинхронный код в синхронный контекст. Это сложно и обычно под силу только фреймворкам. В C# 7.1 добавлена поддержка асинхронных функций `Main`, поэтому можно сразу же запустить асинхронный код, но нельзя вызвать асинхронную функцию из синхронной веб-операции. Хотя наоборот сделать можно. Можно и нужно добавлять синхронный код в асинхронные функции, потому что не каждую функцию можно сделать полностью асинхронной.

8.4.2. Многопоточность и асинхронность

Асинхронный ввод/вывод обеспечивает лучшую масштабируемость, поскольку потребляет меньше ресурсов, чем многопоточность в тяжелом коде ввода/вывода. Но многопоточность и асинхронность не исключают друг друга. Можно использовать оба метода. Можно даже использовать асинхронное программирование для описания многопоточности, например асинхронной обработки длительных операций ЦП:

```
await Task.Run(() => computeMeaningOfLifeUniverseAndEverything());
```

Код по-прежнему будет запускаться в отдельном потоке, но механизм `await` упрощает синхронизацию завершения операции. Этот же код для обычных потоков будет выглядеть сложнее, поскольку требуется примитив синхронизации, такой как событие:

```
ManualResetEvent completionEvent = new(initialState: false);
```

ВИДИТЕ NEW?

Долгое время программистам для инициализации объекта приходилось писать `SomeLongTypeName something = new SomeLongTypeName();`. Вводить один и тот же тип дважды утомительно, невзирая на IDE. Эта проблема была частично решена после добавления в язык ключевого слова `var`, но оно не работает с объявлениями членов класса.

C# 9.0 сделал значительный шаг вперед в этом отношении: теперь не нужно прописывать тип класса после `new`, если тип объявлен до этого. Можно просто написать `SomeLongTypeName something = new();`. И все это благодаря усилиям потрясающей команды разработчиков C#!

Объект события, который вы объявляете, должен быть доступен из точки синхронизации, что создает дополнительную сложность. Код также становится более сложным:

```
ThreadPool.QueueUserWorkItem(state => {
    computeMeaningOfLifeUniverseAndEverything();
    completionEvent.Set();
});
```

Таким образом, асинхронное программирование может несколько упростить написание многопоточных операций, но не является полноценной альтернативой и не улучшает масштабируемость. Многопоточный код, написанный с использованием асинхронного синтаксиса, остается обычным многопоточным кодом, который не экономит ресурсы, как асинхронный код.

8.5. УВАЖАЙТЕ МОНОЛИТ

Прилепите на свой монитор стикер, который вы удалите только тогда, когда скотите состояние на стартапе. Напишите на стикере: «Ни каких микросервисов».

Идея микросервисов проста: если разделить код на отдельные самостоятельные проекты, в будущем станет проще развернуть эти проекты на отдельных

серверах — бесплатное масштабирование! Проблема здесь, как и во многих других вопросах, которые я затрагивал в этой книге, заключается в кратном росте сложности. Разделите ли вы весь общий код? Неужели проекты ничего не используют совместно? А как же их зависимости? Сколько проектов вам нужно будет обновить, если просто изменить базу данных? Как вы делитесь контекстом при аутентификации и авторизации? А безопасность? А что насчет увеличения времени ожидания в результате складывания межсерверных задержек? Как сохранить совместимость? Что, если вы развернете один сервис, а другой перестанет работать из-за изменений? Вы способны справиться с этим уровнем сложности?

Я использую термин *монолит* в качестве антонима микросервисам. Монолит — это когда компоненты ПО находятся в одном проекте или, по крайней мере, в нескольких тесно связанных проектах, развернутых на одном сервере. Но если компоненты взаимозависимы, можно ли переместить часть из них на другой сервер, чтобы масштабировать приложение?

В этой главе мы увидели, как добиться лучшей масштабируемости даже на одном ядре ЦП, не говоря уже об одном сервере. Монолит может масштабироваться. Он может работать нормально долгое время, пока приложение не потребуется разделить. К этому моменту у вашего старта папа будет достаточно денег, чтобы нанять больше разработчиков, которые этим займутся. Не усложняйте новый проект микросервисами, когда аутентификация, координация и синхронизация создадут проблемы на самом раннем этапе жизненного цикла продукта. Ekşi Sözlük более 20 лет ежемесячно обслуживает 40 миллионов пользователей на монолитной архитектуре. Монолит — это также естественный следующий этап развития локального прототипа. Плытите по течению и начинайте внедрять микросервисную архитектуру только тогда, когда ее преимущества перевешивают недостатки.

ИТОГИ

- Подходите к масштабируемости как к поэтапной программе похудения. Небольшие улучшения в конечном итоге приведут к лучшей, масштабируемой системе.
- Одно из основных препятствий масштабируемости — блокировки. С ними никак, но и без них тоже. И все-таки иногда они неизбежны.
- Чтобы сделать код более масштабируемым, выбирайте неблокирующие или параллельные структуры данных, не устанавливайте блокировки вручную.

- Используйте блокировку с двойной проверкой, когда это безопасно.
- Чтобы добиться лучшей масштабируемости, научитесь справляться с несоответствиями. Определите, какие виды несоответствия допустимы для вашего проекта, и используйте эту возможность, чтобы создавать более масштабируемый код.
- Хотя ORM обычно считаются рутиной, они также могут помочь создавать лучше масштабируемые приложения с применением оптимизаций, о которых вы, возможно, и не подозревали.
- Используйте асинхронный ввод/вывод во всем коде, связанном с вводом/выводом. Он должен хорошо масштабироваться, чтобы сохранять потоки доступными и оптимизировать использование ЦП.
- Применяйте многопоточность для распараллеливания работы, связанной с ЦП, но не ждите, что даже при использовании синтаксиса асинхронного программирования вы добьетесь масштабируемости асинхронного ввода/вывода.
- Монолитная архитектура успеет пройти кругосветку, пока завершится обсуждение дизайна микросервисной архитектуры.

9

Жизнь с ошибками

В этой главе

- ✓ Лучшие практики обработки ошибок
- ✓ Жизнь с ошибками
- ✓ Преднамеренная обработка ошибок
- ✓ Избегайте отладки
- ✓ Продвинутая отладка с помощью резиновой уточки

Самое глубокое литературное произведение о жуках¹ — «Превращение» Франца Кафки. В нем рассказывается история разработчика Грегора Замзы, который однажды проснулся и обнаружил, что стал жуком. Ладно, на самом деле он не был разработчиком, потому что все результаты программирования в 1915 году представляли собой пару страниц кода, написанного Адой Лавлейс за 70 лет до создания этой книги. Но он был близок к разработке, так как работал коммивояжером.

¹ Игра слов. Bug — в буквальном переводе «жук», а также баг — ошибка в компьютерной программе. — *Примеч. пер.*

Ошибки (баги) — это основные единицы измерения качества программного обеспечения. Поскольку разработчики считают каждый баг пятном на своей репутации, они обычно либо стараются вообще не допускать ошибок, либо активно отрицают их существование, утверждая, что на их компьютере все работает или что это такая особенность ПО, а не ошибка («это не баг, а фича»).

ЗАДАЧА О КОММИВОЯЖЕРЕ

Задача о коммивояжере — краеугольный камень computer science, поскольку задача вычисления оптимального маршрута для коммивояжера является *NP-полной (NP-complete)*. Совершенно нелогичное сокращение для «*nondeterministic polynomial-time complete*» (решаемая за полиномиальное время на недетерминированной машине Тьюринга задача). Поскольку в этом сокращении пропущено много слов, я долгое время считал, что оно означает *неполиномиальное полное*, и меня это очень смущало.

Задачи с полиномиальным временем (P) можно решить быстрее, чем путем перебора всех возможных комбинаций, которые в противном случае имеют факториальную сложность, вторую наихудшую сложность из всех возможных. NP — это надмножество полиномиальных задач, которые можно решить только полным перебором (брутфорсом). Полиномиальные задачи всегда предпочтительнее, чем NP-задачи. Для недетерминированных задач с полиномиальным временем нет известного полиномиального алгоритма решения, но их решение может быть проверено за полиномиальное время. В этом смысле NP-полная задача означает: «С этим сложно справиться, но решение можно довольно быстро проверить».

Разработка чрезвычайно сложна из-за того, что программные продукты не-предсказуемы. Такова природа *машины Тьюринга*, теоретической конструкции, лежащей в основе работы всех компьютеров и большинства языков программирования и названной по имени своего создателя, Алана Тьюринга. Язык программирования, основанный на машине Тьюринга, называется *полным по Тьюрингу*. Машины Тьюринга допускают бесконечную свободу творчества, которая реализуется в программных продуктах, но правильность продукта невозможно проверить, не запустив его. Некоторые языки, например HTML, XML или регулярные выражения, значительно менее функциональны, чем полные по Тьюрингу языки, и зависят от машин, не являющихся полными по Тьюрингу. Из-за природы машины Тьюринга баги неизбежны. Не бывает программ без багов. Принятие этого факта до того, как вы приступите к разработке, значительно облегчит вам жизнь.

9.1. НЕ ИСПРАВЛЯЙТЕ ОШИБКИ

Команда разработчиков любого крупного проекта должна наладить процесс сортировки, чтобы решать, какие ошибки исправлять. Термин *сортировка* (triaging) возник во время Первой мировой войны, когда медикам приходилось решать, каких пациентов лечить, а каких нет, чтобы направлять ограниченные ресурсы на тех, у кого еще был шанс выжить. Это единственный способ эффективно использовать ограниченные ресурсы. Сортировка помогает решить, что исправлять в первую очередь и нужно ли это исправлять вообще.

Как определить приоритет ошибки? Если вы не принимаете все бизнес-решения в одиночку, то для определения приоритета ошибки ваша команда должна руководствоваться четкими критериями. В нашей команде в Microsoft был принят сложный набор критерииев, согласно которому группа технических специалистов решала, какие ошибки следует исправлять. Мы ежедневно собирались в комнате под названием War Room (командный пункт) для определения приоритета ошибок и обсуждали, какие из них стоит исправлять. Это целесообразно для продукта таких масштабов, как Windows, но может быть лишним для большинства проектов.

Мне пришлось отстаивать высокий приоритет ошибки, когда после обновления перестала работать автоматизированная система в главном дворце бракосочетания в Стамбуле и все свадьбы пришлось остановить. Я должен был обосновать свою позицию, разделив невозможность вступить в брак на такие приземленные составляющие, как *применимость, воздействие и серьезность*. Вопрос «Сколько пар женятся в день в Стамбуле?» вдруг прозвучал как значимая часть интервью.

Упрощает определение приоритета использование косвенного измерения, называемого *серьезностью*. Хотя наша цель — однозначная оценка приоритета, наличие дополнительного измерения может облегчить выбор, когда две проблемы очевидно имеют одинаковый приоритет. Я считаю измерение приоритета с учетом серьезности разумным балансом между бизнесом и технологиями. *Приоритет* — это оценка влияния ошибки на бизнес, а *серьезность* — это оценка влияния на клиента. Например, если веб-страница на вашей платформе не работает, это проблема высокой серьезности, поскольку клиент не может ее использовать. Но оценка приоритета может быть совершенно другой, поскольку зависит от того, главная ли это страница или второстепенная, которую посещают лишь несколько человек в день. Точно так же проблема с отображением логотипа компании на домашней странице может считаться не слишком серьезной, но при этом иметь высший бизнес-приоритет. Оценка серьезности снимает некоторую

нагрузку с бизнес-приоритизации, потому что невозможно разработать точные метрики для определения приоритета ошибок.

Можно ли достичь того же уровня детализации только с одним параметром? Например, иметь шесть уровней приоритета вместо трех уровней приоритета и серьезности. Проблема в том, что чем больше уровней, тем сложнее их различать. Обычно дополнительный критерий помогает более точно оценить важность проблемы.

Также необходимо определить порог приоритета и серьезности, ниже которого любые ошибки классифицируются как *не требующие исправления*. Например, любая ошибка, которая имеет как низкий приоритет, так и низкую серьезность, может считаться не требующей исправления и исключаться из списка.

В табл. 9.1 показан пример системы уровней приоритета и серьезности.

Таблица 9.1. Значения приоритета и серьезности

Приоритет	Серьезность	Реальный смысл
Высокий	Высокая	Исправить немедленно
Высокий	Низкая	Босс хочет, чтобы это было исправлено
Низкий	Высокая	Пусть это исправляет стажер
Низкий	Низкая	Не исправлять. Исправить, только если в офисе больше нечего заниматься. Но и тогда пусть это делает стажер

Отслеживание ошибок тоже требует затрат. В Microsoft у нас уходило не менее часа в день только на то, чтобы оценить приоритет ошибок. Крайне важно не рассматривать повторно ошибки, которые вряд ли когда-нибудь будут исправлены. Постарайтесь определиться с такими ошибками как можно раньше. Это сэкономит вам время и поможет поддерживать высокое качество продукта.

9.2. УЖАС ОШИБОК

Не каждый баг вызван ошибкой в коде, и не каждая ошибка приложения подразумевает наличие бага в коде. Эта истина наиболее очевидна, когда вы видите всплывающее диалоговое окно «unknown error» («неизвестная ошибка»). Если это неизвестная ошибка, как можно быть уверенными, что это вообще ошибка? Может быть, это небывалый успех!

Разработчики инстинктивно воспринимают все ошибки приложения как результат багов в коде и стараются последовательно и настойчиво их устранять. Рассуждения такого рода обычно приводят к ситуации с неизвестной ошибкой, поскольку что-то пошло не так, а разработчик даже не озабочился подумать, ошибка ли это. Из-за такого подхода разработчики относятся ко всем ошибкам одинаково, обычно либо сообщая о каждой из них, независимо от того, нужно ли это пользователю, либо скрывая их все и пряча на сервере в логе, который никто никогда не прочитает.

С навязчивой идеей одинаково относиться ко всем ошибкам можно справиться, если считать их частью состояния системы. Возможно, не следует называть их ошибками. Лучше было бы назвать их *необычными и неожиданными изменениями состояния или исключениями*. Хотя подождите, такое уже есть!

9.2.1. Неприятная правда об исключениях

Исключения, возможно, самая неверно понимаемая конструкция в истории программирования. Я сбился со счета, сколько раз видел неудачный код, помещенный в блок `try`, за которым следовал пустой блок `catch`. Это все равно что закрыть дверь в горящую комнату и предположить, что пожар потухнет сам собой. Причем такое предположение законно, но стоит оно довольно дорого.

Листинг 9.1. Решение всех жизненных проблем

```
try {
    doSomethingMysterious();
}
catch {
    // Все хорошо
}
```

Программистов я не виню. Как сказал Абрахам Маслоу в 1966 году, «если единственный инструмент, который у вас есть, это молоток, то любая проблема становится гвоздем». Я уверен, что молоток в свое время был прорывным изобретением и каждый старался использовать его при решении любых задач. Люди эпохи неолита, вероятно, писали в своих наскальных блогах о том, насколько революционным был молоток и как он хорош в решении проблем, не зная, что в будущем появятся гораздо более удобные приспособления для намазывания масла на хлеб.

Я видел, как общий обработчик исключений добавлялся для всего приложения. Но если он игнорирует все исключения, предотвращая сбои, тогда почему

остаются баги? Их бы давно уже не было, если бы добавление пустого обработчика помогало от них избавиться.

Исключения — это решение проблемы неопределенного состояния. В те времена, когда обработка ошибок касалась только возвращаемых значений, можно было игнорировать ошибку, как будто она успешно обработана, и продолжить работу. Однако это приводило приложение в непрогнозируемое состояние. Проблема с неизвестным состоянием в том, что невозможно заранее узнать ни его последствия, ни степень их серьезности. Это почти единственная причина появления экранов критических ошибок операционной системы, таких как паника ядра в системах UNIX или печально известный синий экран смерти в Windows. Они останавливают систему, чтобы предотвратить возможные дальнейшие повреждения.

Неизвестное состояние означает, что невозможно предсказать, что произойдет дальше. Может быть, ЦП просто сойдет с ума и войдет в бесконечный цикл, или жесткий диск решит записывать нули в каждый сектор, или ваш аккаунт в Twitter будет выводить заглавными буквами случайные политические лозунги.

Коды ошибок отличаются от исключений тем, что для последних можно определить, будут ли они обрабатываться. Необработанное исключение обычно ведет к завершению работы приложения, поскольку возникает состояние, которое не ожидалось. Операционные системы завершают работу приложения, если оно не может обработать исключение. Это не применяется к драйверам устройств и компонентам уровня ядра, так как они не работают в изолированных областях памяти в отличие от пользовательских процессов. Вот почему приходится полностью останавливать систему. В микроядерных ОС это некритично, потому что количество компонентов уровня ядра минимально, и даже драйверы устройств работают в пользовательском пространстве, но неизбежно снижение производительности, с чем мы пока не смирились.

Главная особенность исключений — их исключительность. Они не предназначены для общего управления потоком, для этого используются результаты и специальные конструкции. Исключения применяются для случаев, когда что-то происходит вне условий выполнения функции. Например, если функция $(a, b) \Rightarrow a/b$ гарантирует выполнение операции деления, но не может этого сделать при $b=0$.

Предположим, вы загружаете обновление для своего десктопного приложения, сохраняете загруженную копию на диск и при следующем запуске переключаете приложение на только что загруженную версию. Это популярный способ самостоятельного обновления приложений, и соответствующая операция будет выглядеть, как показано на рис. 9.1. Он немного упрощен, поскольку не учитывает незавершенные обновления, но передает суть процесса.

Если во время обновления возникнет исключение, вы получите неполную папку `app2`. При этом файлы приложения будут заменены поврежденными версиями, что приведет к катастрофическому состоянию, из которого невозможно восстановиться.

На любом этапе может возникнуть исключение, ведущее к сбою системы, если его не обработать или обработать неправильно.



Рис. 9.1. Примитивная логика самостоятельного обновления приложений

Рисунок 9.1 показывает важность устойчивости процессов к исключениям. Любой сбой на каждом этапе может вызвать повреждения, которые невозможно будет устраниТЬ. Приложение не должно оставаться в таком состоянии, даже если возникает исключение.

9.2.2. Не перехватывайте исключения

В качестве быстрого и простого исправления кода, в котором возник сбой из-за исключения, применяются блоки `try/catch`. Если игнорировать исключение, сбой исчезнет, но это не устраниТ его причину.

Считается, что исключения вызывают сбои, потому что это самый простой способ идентифицировать проблему, не создавая других проблем. Не бойтесь сбоев с удобной трассировкой стека, которая поможет точно определить место, где он произошел. Бойтесь ошибок, не вызывающих очевидного сбоя: проблем, которые скрываются за пустыми операторами `catch`, прячутся в коде и маскируются под выглядящее правильным состояние, накапливая ошибки в течение долгого времени и в итоге вызывая либо заметное замедление операций, либо неожиданную ошибку, например `OutOfMemoryException`. Необязательные блоки `catch` предотвращают некоторые сбои, но могут стоить не одного часа чтения логов. Исключения хороши тем, что позволяют обнаружить проблему до того, как ее станет трудно найти.

Первое правило обработки исключений: не перехватывайте их. Вызов второго правила обработки исключений приводит к ошибке `IndexOutOfRangeException`.

Видите, что происходит, когда у вас есть только одно правило? Не перехватывайте исключение из-за того, что оно вызывает сбой. Если причина в неправильном поведении, исправьте ошибку. Если причина в известном состоянии, добавьте в код явные операторы обработки для этого конкретного случая.

Всякий раз, когда есть возможность получить исключение в каком-то месте кода, спрашивайте себя: «Буду ли я предусматривать специальные меры или хочу предотвратить сбой?» Если верно второе, обработка этого исключения может быть ненужной и даже вредной, поскольку может скрыть более глубокую и серьезную проблему с кодом.

Рассмотрим самообновляемое приложение, о котором шла речь в разделе 9.2.1. В нем может быть предусмотрена функция загрузки группы файлов приложения в папку, как показано в листинге 9.2. Необходимо загрузить два файла с сервера обновлений, если они являются последними версиями. Очевидно, что при таком подходе возникает много проблемных мест, например неиспользование центрального реестра для контроля версий. Что произойдет, если я начну загрузку в то время, как разработчики дистанционно обновляют файлы? Половина файлов у меня будет из предыдущей версии, а вторая половина — из следующей, что приведет к ошибке установки. Для примера предположим, что разработчики отключили веб-сервер перед обновлением, обновили файлы и затем включили его, чтобы предотвратить такую ошибку.

Листинг 9.2. Код для одновременной загрузки нескольких файлов

```
private const string updateServerUriPrefix =
    "https://streetcoder.org/selfupdate/";

private static readonly string[] updateFiles =
    new[] { "Exceptions.exe", "Exceptions.app.config" }; ← Список файлов для загрузки
```

```

private static bool downloadFiles(string directory,
    IEnumerable<string> files) {
    foreach (var filename in updateFiles) {
        string path = Path.Combine(directory, filename);
        var uri = new Uri(updateServerUriPrefix + filename);
        if (!downloadFile(uri, path)) {
            return false; ← Выявляем проблему с загрузкой и очисткой
        }
    }
    return true;
}

private static bool downloadFile(Uri uri, string path) {
    using var client = new WebClient();
    client.DownloadFile(uri, path); ← Загружает отдельный файл
    return true;
}

```

Мы знаем, что `DownloadFile` может вызывать исключения по разным причинам. На самом деле у Microsoft есть отличная документация о поведении функций .NET, в том числе о том, какие исключения они могут генерировать. Метод `DownloadFile` в `WebClient` может генерировать три исключения:

- `ArgumentNullException`, когда аргумент имеет значение `null`;
- `WebException`, когда что-то неожиданное происходит во время загрузки, например потеря интернет-соединения;
- `NotSupportedException`, когда один и тот же экземпляр `WebClient` вызывается из нескольких потоков, чтобы показать, что класс не является потокобезопасным.

Для предотвращения сбоя можно обернуть вызов `DownloadFile` в `try/catch`, чтобы загрузка продолжалась. Поскольку многим разработчикам все равно, какие типы исключений перехватывать, они просто делают это с помощью неназначенного блока `catch`. Мы вводим код результата, чтобы определять, произошла ли ошибка.

Листинг 9.3. Предотвращение сбоев путем создания новых ошибок

```

private static bool downloadFile(Uri uri, string path) {
    using var client = new WebClient();
    try {
        client.DownloadFile(uri, path);
        return true;
    }
    catch {
        return false;
    }
}

```

Недостаток такого подхода в том, что вы перехватываете все три возможных исключения, два из которых на самом деле указывают на конкретные ошибки программиста. `ArgumentNullException` выдается при передаче вызывающей стороной недопустимого аргумента. Она означает, что где-то в стеке вызовов есть либо неверные данные, либо неверная проверка ввода. Точно так же `NotSupportedException` выдается при неправильном использовании клиента. Это означает, что вы скрываете множество легко исправимых ошибок, которые могут привести к более серьезным последствиям в случае перехвата всех исключений. Нет, перехватывать их все не нужно. При отсутствии возвращаемого значения простая ошибка аргумента привела бы к пропуску файлов, и мы бы даже не знали, существуют ли они.

Перехватывайте конкретные исключения, не являющиеся ошибкой программиста, как в листинге 9.4. Мы перехватываем только `WebException`, так как знаем, что загрузка может завершиться неудачей в любое время по множеству причин. Поэтому необходимо сделать эту ошибку частью состояния. Перехватывайте исключение, только если оно ожидается. Мы позволяем другим типам исключений вызывать сбой, потому что сглупили и заслуживаем того, чтобы жить с его последствиями, прежде чем он вызовет более серьезную проблему.

Листинг 9.4. Точная обработка исключений

```
private static bool downloadFile(Uri uri, string path) {
    using var client = new WebClient();
    try {
        client.DownloadFile(uri, path);
        return true;
    }
    catch (WebException) { ← Не перехватывайте их все
        return false;
    }
}
```

Анализаторы кода советуют избегать использования нетипизированных блоков `catch`, поскольку они слишком широки и вызывают перехват нерелевантных исключений. Блоки `Catchall` следует использовать, только если вам действительно требуется перехват всех исключений в мире, например, для такой глобальной цели, как ведение лога.

9.2.3. Устойчивость к исключениям

Код должен работать правильно и без обработки исключений, в том числе при сбое. Нормально работающий поток возможен, даже если вы постоянно получаете

исключения. Дизайн продукта должен допускать исключения, поскольку они неизбежны. Можно поместить универсальный `try/catch` в метод `Main`, но приложение все равно неожиданно завершит работу, когда обновления вызовут перезапуск. Нельзя позволять исключениям нарушать состояние приложения.

Если происходит сбой Visual Studio, то изменяемый в этот момент файл не теряется. При повторном запуске приложения выводится предложение восстановить файл. Для этого Visual Studio сохраняет копии несохраненных файлов во временном хранилище и удаляет их при успешном сохранении. При запуске Visual Studio проверяет наличие временных файлов и спрашивает, хотите ли вы их восстановить. Код необходимо проектировать так, чтобы предвидеть подобные проблемы.

В нашем примере с автоматически обновляющимся приложением процесс должен допускать исключения и восстанавливаться после перезапуска приложения. Устойчивая к исключениям конструкция инструмента самообновления будет выглядеть, как показано на рис. 9.2. На нем представлена схема загрузки атомарного пакета, чтобы набор файлов оставался последовательным. Также мы создаем резервные копии исходных файлов перед заменой их новыми, чтобы иметь возможность восстановления, если что-то пойдет не так.

Продолжительность установки обновления намекает на сложность процесса и высокую вероятность сбоев. Однако описанные выше методы предотвратят возникновение некорректного состояния.

Устойчивость к исключениям начинается с идемпотентности. Функция или URL-адрес являются идемпотентными, если возвращают один и тот же результат независимо от того, сколько раз они вызывались. С чистыми функциями, такими как `Sum()`, все просто, но с функциями, которые изменяют внешнее состояние, могут возникать сложности. Примером может служить процесс оформления заказа в онлайн-магазинах. Если случайно дважды нажать кнопку «Заказать», списутся ли деньги с кредитной карты дважды? Некоторые сайты пытаются бороться с такими ошибками, размещая предупреждения типа «Не нажимайте эту кнопку дважды!». Но как известно, кошки, гуляющие по клавиатуре, не умеют читать.

Идемпотентность в упрощенном виде обычно характерна для веб-запросов, например: «запросы HTTP `GET` должны быть идемпотентными, а все, что неидемпотентно, должно быть запросами `POST`». Но запросы `GET` могут не быть идемпотентными, скажем, для динамически изменяющегося содержимого, а запрос `POST` может быть идемпотентным, как лайк в соцсетях: многократная повторная отправка лайка не должна влиять на счетчик.

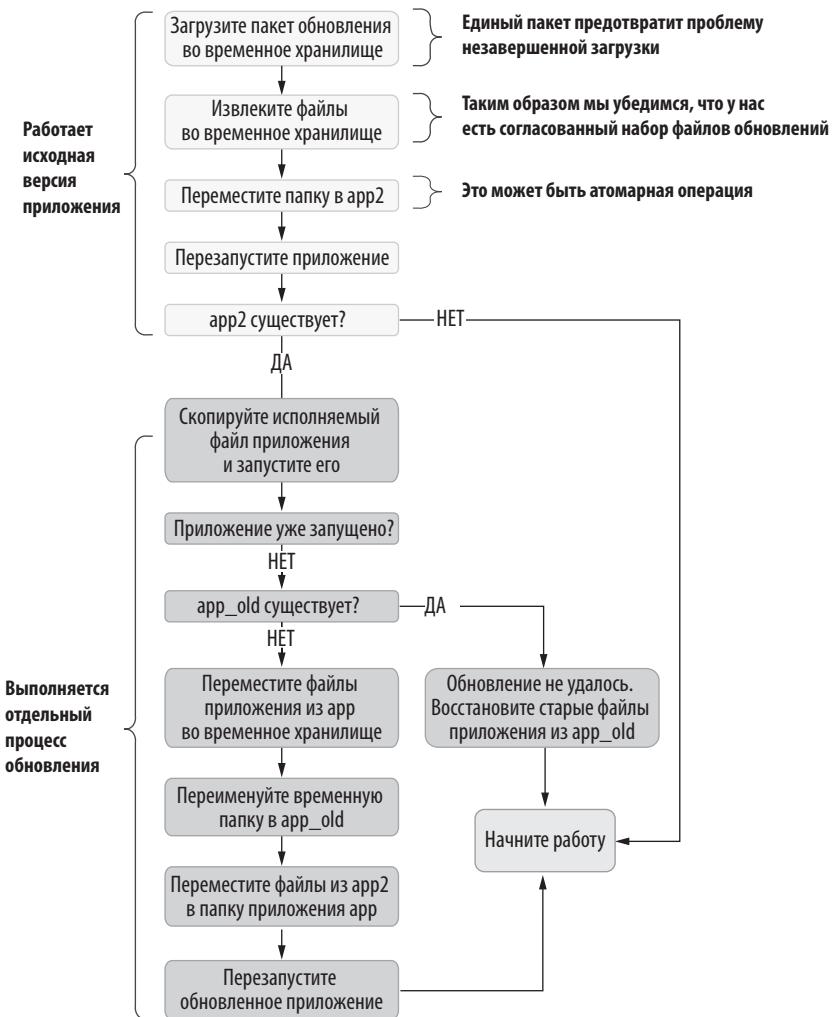


Рис. 9.2. Более устойчивая к исключениям версия автоматически обновляемого приложения

Как идемпотентность способствует устойчивости к исключениям? Если функция разработана так, чтобы ее побочные эффекты оставались одинаковыми независимо от того, сколько раз она вызывается, то при неожиданном прерывании выполнения функции подобная согласованность предоставляет определенные преимущества. Благодаря им код можно вызывать неоднократно и безопасно.

Как добиться идемпотентности? В нашем примере заказам может быть присвоен уникальный номер обработки. После начала обработки заказа следует создать запись в БД и в начале функции обработки проверить его существование, как показано на рис. 9.3. Код должен быть потокобезопасным, потому что некоторые кошки бегают очень быстро.

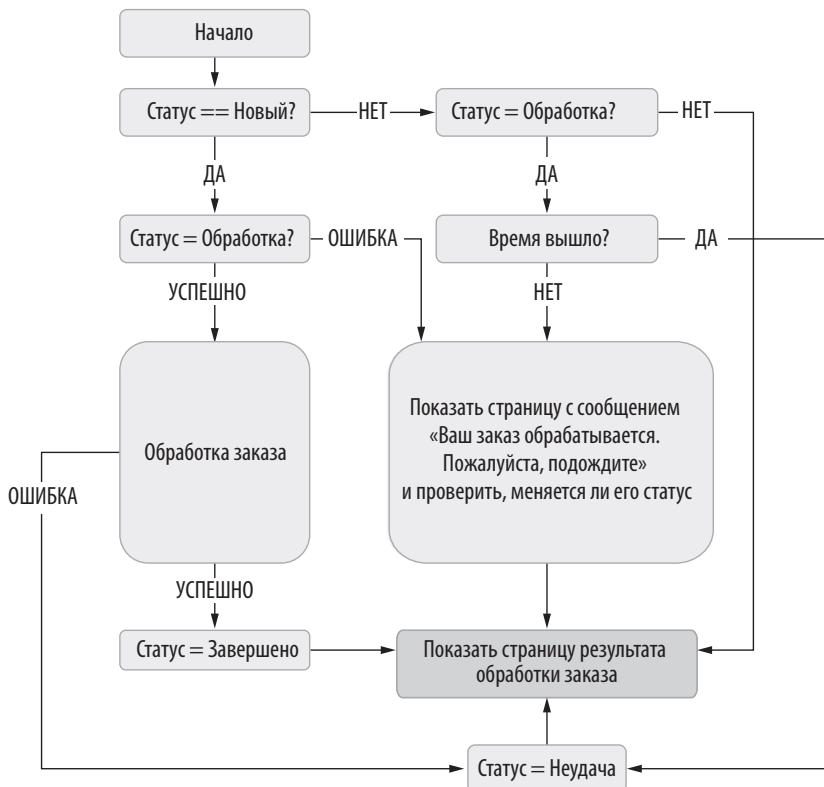


Рис. 9.3. Пример идемпотентного размещения заказа

Транзакции БД помогают избежать некорректного состояния, поскольку возвращаются к исходному состоянию, если перестают работать из-за исключения. Но во многих сценариях они не требуются.

В нашем примере (рис. 9.3) имеется операция изменения статуса заказа, но как гарантировать ее атомарность? Что, если кто-то изменит статус до того, как мы прочитаем результат? Секрет в том, чтобы использовать операцию условного

обновления базы данных. Она гарантирует, что статус будет соответствовать ожидаемому и может выглядеть так:

```
UPDATE orders SET status=@NewState WHERE id=@OrderID status=@CurrentState
```

UPDATE возвращает количество затронутых строк. Если состояние изменится во время операции UPDATE, она завершится ошибкой и вернет 0. Если изменение состояния прошло успешно, возвращается 1. Это свойство можно использовать для атомарного обновления статуса записи, как показано на рис. 9.3.

Пример реализации представлен в листинге 9.5. Мы задаем каждый отдельный статус, в котором может находиться заказ во время обработки, и разрешаем обработку на разных уровнях. Если заказ уже обрабатывается, мы выводим страницу обработки, и аннулируем заказ, если срок его обработки истек.

Листинг 9.5. Идемпотентная обработка заказа

```
public enum OrderStatus {
    New,
    Processing,
    Complete,
    Failed,
}

[HttpPost]
public IActionResult Submit(Guid orderId) {
    Order order = db.GetOrder(orderId);

    if (!db.TryChangeOrderStatus(order, from: OrderStatus.New, ← Стремится изменить
        to: OrderStatus.Processing)) { ← статус атомарно
        if (order.Status != OrderStatus.Processing) {
            return redirectToResultPage(order);
        }
        if (DateTimeOffset.Now - order.LastUpdate > orderTimeout) { ← Проверяет
            db.ChangeOrderStatus(order, OrderStatus.Failed); ← тайм-аут
            return redirectToResultPage(order);
        }
        return orderStatusView(order); ← Выводит страницу обработки
    }
    if (!processOrder(order)) {
        db.ChangeOrderStatus(order, OrderStatus.Failed);
    } else {
        db.TryChangeOrderStatus(order,
            from: OrderStatus.Processing,
            to: OrderStatus.Complete); ← В случае ошибки на странице результатов
    } ← будет выведен верный результат
    return redirectToResultPage(order);
}
```

Хотя это запрос HTTP POST, отправка заказа может вызываться несколько раз без побочных эффектов, следовательно, она идемпотентна. Если веб-приложение аварийно завершает работу и вы перезапускаете его, оно все равно может восстановиться из многих недопустимых состояний, таких как состояние обработки. Обработка заказов может быть более сложной, и иногда может требоваться периодическая внешняя очистка, но устойчивость к исключениям все равно может быть высокой даже без операторов `catch`.

9.2.4. Устойчивость без транзакций

Только идемпотентности может быть недостаточно для обеспечения устойчивости к исключениям, но она служит отличной основой, поскольку заставляет нас анализировать, как функция будет вести себя в разных состояниях. В нашем примере на стадии обработки заказа могут вызываться исключения, оставляя обработку незавершенной и делая невозможным повторный вызов того же ее этапа. Обычно транзакции защищают от такой ситуации, потому что откатывают все изменения, не оставляя после себя грязных данных. Но не каждое хранилище имеет поддержку транзакций. Например, ее не имеют файловые системы.

Даже если транзакции использовать не получится, кое-что сделать все же можно. Предположим, вы создали приложение для обмена изображениями, в котором пользователи могут загружать альбомы и делиться ими с друзьями. В сети доставки контента (кстати, *content delivery network, CDN* – отличное имя для файловых серверов) для каждого альбома может иметься папка, в которой лежат файлы изображений, а в базе данных будут храниться записи альбомов. Непрактично оборачивать операцию создания такой структуры в транзакцию, поскольку она требует использования нескольких технологий.

Альбомы обычно создаются следующим образом: сначала создается запись альбома, затем папка и, наконец, в папку загружаются изображения. Но если в этом процессе возникнет исключение, в записи альбома будут отсутствовать какие-то изображения. Эта проблема характерна практически для всех типов взаимозависимых данных.

Есть несколько способов ее избежать. В нашем примере с альбомом можно сначала создавать временную папку для изображений, затем перемещать ее в UUID альбома и только потом создавать запись альбома. Таким образом исключается доступ пользователей к неполным альбомам.

Другой вариант – сначала создавать запись альбома с неактивным состоянием, а затем добавлять остальные данные. Статус записи альбома можно изменять

на *активный* только после завершения операции загрузки. При этом вы будете избавлены от дубликатов записей альбомов, если процесс загрузки изображений будет прерван исключением.

В обоих случаях можно периодически проводить очистку БД с удалением заброшенных записей. Если альбом создается обычным способом, трудно отличить нормальные ресурсы от результатов прерванных операций.

9.2.5. Исключения и ошибки

Считается, что исключения означают ошибки, и иногда это соответствует действительности, но не все ошибки классифицируются как исключения. Не используйте исключения, если ожидаете, что вызывающая сторона будет обрабатывать их большую часть времени.

Показательный пример – сравнение методов `Parse` и `TryParse` в .NET, из которых первый при недопустимом вводе генерирует исключение, а второй просто возвращает `false`. Сначала был только `Parse`. Затем в .NET Framework 2.0 появился `TryParse`, поскольку неверный ввод оказался обычным делом в большинстве сценариев, а исключения работают медленно, увеличивая накладные расходы. Последнее обусловлено тем, что исключения несут в себе трассировку стека, а для сбора информации требуется обход стека. Это может стоить очень дорого по сравнению с простым возвратом логического значения. Исключения также труднее обрабатывать, потому что требуется выполнять весь алгоритм `try/catch`, а простое значение `result` можно проверить только с помощью `if`, как показано в следующем листинге. Очевидно, что вариант с `try/catch` предполагает больше ввода кода, его сложнее правильно реализовать, потому что разработчик просто может забыть сохранить специфичный для `FormatException` обработчик исключений, и код сложнее для понимания.

Листинг 9.6. Сказ о двух парсерах

```
public static int ParseDefault(string input,      ← Реализация с Parse
    int defaultValue) {
    try {
        return int.Parse(input);
    }
    catch (FormatException) {      ← Искушение опустить тип исключения
        return defaultValue;
    }
}

public static int ParseDefault(string input,      ← Реализация с TryParse
```

```
int defaultValue) {
    if (!int.TryParse(input, out int result)) {
        return defaultValue;
    }
    return result;
}
```

`Parse` следует использовать, когда вы ожидаете, что ввод всегда будет правильным. Если вы уверены, что входящие значения всегда будут иметь верное форматирование, а любое недопустимое состояние на самом деле является ошибкой, то необходимо обеспечить выдачу исключений. В некотором смысле это риск, потому что вы объявляете любое недопустимое входящее значение ошибкой: «Ломайся, если можешь!».

В большинстве случаев для возврата подходят стандартные значения ошибок. Можно даже ничего не возвращать, если значение ошибки не требуется. Например, если вы ожидаете, что операция «поставить лайк» всегда будет успешной, не возвращайте значение. Возврат функции уже означает успех.

Типы результатов ошибок могут быть разными в зависимости от того, сколько, по вашему мнению, информации нужно вызывающей стороне. Если вызывающей стороне важно только, успешна операция или нет, а детали не важны, достаточно вернуть логическое значение, у которого `true` будет означать успех, а `false` — неудачу. Если имеется третье состояние или вы уже используете тип `bool` для чего-то еще, может потребоваться другой подход.

Например, в Reddit можно ставить лайки или дизлайки только достаточно новому контенту. Нельзя оценивать комментарии или сообщения старше шести месяцев. Также нельзя оценивать удаленные сообщения. Это означает, что операция оценивания иногда может заканчиваться неудачей, и об этом требуется сообщить пользователю. Но нельзя просто вывести предупреждение «Неизвестная ошибка операции», потому что пользователь может подумать, что это временный сбой, и продолжать попытки поставить оценку. Необходимо ответить: «Это сообщение создано слишком давно» или «Это сообщение удалено». Логично было бы просто скрывать кнопки оценивания для таких сценариев, чтобы пользователь сразу понимал невозможность действия, но Reddit настаивает на их отображении.

В сценарии Reddit можно использовать `enum`, чтобы различать виды отказа. Возможный вариант кода представлен в листинге 9.7. Он написан только для части функциональности, но нам сейчас не нужны дополнительные значения для других возможностей. Например, неудачное оценивание из-за ошибки БД должно быть исключением, указывающим либо на отказ инфраструктуры, либо на ошибку. Также необходим стек вызовов, и надо организовать регистрацию.

Листинг 9.7. Результаты оценивания на Reddit

```
public enum VotingResult {
    Success,
    ContentTooOld,
    ContentDeleted,
}
```

`enum` хорош тем, что при использовании выражений-переключателей компилятор предупреждает вас о случаях, которые вы не рассмотрели. Компилятор C# не делает этого для операторов `switch`, а только для выражений, потому что последние недавно добавлены в язык для разработки таких сценариев. Пример исчерпывающей обработки `enum` для операции `upvote` представлен в листинге ниже. Но все равно возможно предупреждение о том, что описание `switch` недостаточно полное, так как теоретически можно присвоить `enum` недопустимые значения вследствие особенностей языка C#.

Листинг 9.8. Исчерпывающая обработка `enum`

```
[HttpPost]
public IActionResult Upvote(Guid contentId) {
    var result = db.Upvote(contentId);
    return result switch {
        VotingResult.Success => success(),
        VotingResult.ContentTooOld
            => warning("Content is too old. It can't be voted"),
        VotingResult.ContentDeleted
            => warning("Content is deleted. It can't be voted"),
    };
}
```

9.3. НЕ ЗАНИМАЙТЕСЬ ОТЛАДКОЙ

Отладка — стариинный термин; он появился даже раньше, чем программирование, еще до того, как Грейс Хоппер сделала его популярным в 1940-х годах, найдя мотылька в реле компьютера Mark II¹. Первоначально этот термин использовался в аэронавтике для обозначения процесса обнаружения неисправностей самолетов. Менее популярным в последнее время его сделала продвинутая практика Кремниевой долины — увольнять генерального директора всякий раз, когда проблема обнаруживается постфактум.

¹ Отладка — debugging. Буквальное значение — «избавление от жуков» (или прочих насекомых). — Примеч. ред.

Современная отладка в основном подразумевает запуск отладчика, определение точек останова, пошаговый анализ кода и проверку состояния программы. Отладчики очень удобны, но не всегда они являются лучшим инструментом. Выявление первопричины проблемы может занять очень много времени. А иногда отладить программу вообще невозможно. У вас может даже не быть доступа к среде, в которой работает код.

9.3.1. Отладка `printf()`

Вставка в программу команд консольного вывода, чтобы найти проблему, — очень старый метод. Хотя современные разработчики обзавелись модными отладчиками с функциями пошаговой отладки, они не всегда являются самыми эффективными инструментами для определения первопричины проблемы. Иногда лучше подходит более простой метод. Отладка `printf()` получила свое название по функции `printf()` в языке программирования С. Ее название расшифровывается как *форматированная печать* (`print formatted`). Она очень похожа на `Console.WriteLine()`, хотя и с другим синтаксисом форматирования.

Непрерывная проверка состояния приложения, вероятно, старейший способ отладки программ. Он даже старше, чем мониторы. В старых компьютерах на передней панели были световые индикаторы, которые отображали битовые состояния регистров ЦП, чтобы программисты понимали, почему что-то не работает. К счастью для меня, мониторы были изобретены еще до моего рождения.

Отладка `printf()` — это способ периодического отображения состояния работающей программы, чтобы программист мог понять, где возникает проблема. Считается, что она не подходит для новичков, но это не совсем так, к тому же такая отладка может быть эффективнее пошаговой. Например, программист может выбрать частоту генерации отчетов о состоянии. При пошаговой отладке вы можете задавать точки останова, но на самом деле не можете пропустить больше одной строки. Вам придется либо выполнять сложную настройку таких точек, либо без устали жать `Step Over`. Это отнимает довольно много времени и утомляет.

Что еще более важно, `printf()` и `Console.WriteLine()` записывают состояние в историю консольного терминала. Благодаря этому, просматривая выходные данные терминала, можно построить логическую цепочку переходов между состояниями, что невозможно сделать с помощью пошагового отладчика.

Не все программы имеют функции вывода консоли, веб-приложений или служб. В .NET есть альтернативы для этих сред, в первую очередь `Debug.WriteLine()` и `Trace.WriteLine()`. `Debug.WriteLine()` записывает выходные данные в консоль вывода отладчика, которая отображается в окне вывода отладчика в Visual Studio вместо собственного вывода консоли приложения. Самое большое преимущество `Debug.WriteLine` заключается в том, что вызовы к нему полностью удаляются из оптимизированных (релизных) двоичных файлов, поэтому они не влияют на производительность выпущенного кода. Однако это создает проблемы для отладки рабочего кода. Даже если операторы вывода сохранились в коде, не существует практического способа их использовать. В этом смысле `Trace.WriteLine()` лучше, поскольку трассировка .NET, помимо обычного вывода, имеет настраиваемые во время выполнения слушатели. Вы можете записывать выходные данные трассировки в текстовый файл, журнал событий, файл XML и куда угодно, если установлен правильный компонент. Благодаря магии .NET можно даже перенастроить трассировку во время работы приложения.

Трассировку легко настроить, например, ее можно включить во время выполнения кода. Ниже мы рассмотрим пример трассировки в работающем веб-приложении.

9.3.2. Дамп-дайвинг

Еще одна альтернатива пошаговой отладке — анализ аварийных дампов. Хотя аварийные дампы не обязательно создаются после сбоя, они представляют собой файлы, содержащие моментальный снимок области памяти программы. В системах UNIX их также называют *дампами ядра*. Создать аварийный дамп можно вручную, щелкнув правой кнопкой мыши на имени процесса в диспетчере задач Windows, а затем выбрав *Создать файл дампа (Create dump file)*, как показано на рис. 9.4. Это неинвазивная операция, которая приостанавливает процесс только до своего завершения. Затем процесс продолжит работу.

Такой же плавный дамп ядра в вариантах UNIX можно выполнить, не убивая приложение, но это немного сложнее. Для этого требуется инструмент `dotnet dump`:

```
dotnet tool install --global dotnet-dump
```

Он отлично подходит для анализа аварийных дампов, поэтому рекомендуется установить его даже в Windows. Команда установки в Windows та же.

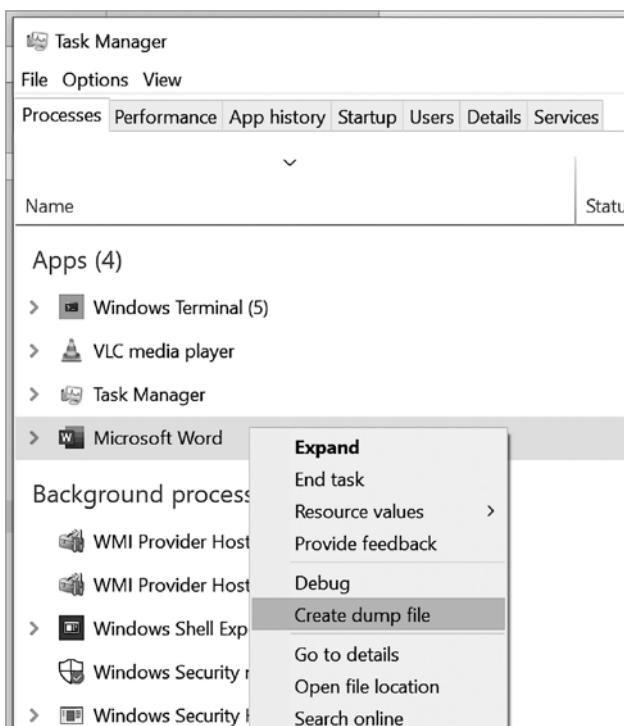


Рис. 9.4. Создание аварийного дампа работающего приложения вручную

В примерах для этой главы на GitHub есть проект под названием `InfiniteLoop`, который непрерывно потребляет ресурсы ЦП. Это может быть как веб-приложение, так и служба, запущенная на рабочем сервере, поэтому можно потренироваться в определении проблемы, что очень похоже на тренировку вскрытия замков на макетах. Вы думаете, что эти навыки вам не нужны, только пока не узнаете, сколько зарабатывают слесари. Весь код приложения показан в листинге 9.9. По сути, мы непрерывно выполняем операцию умножения в цикле без пользы для мира во всем мире. Вероятно, этот цикл по-прежнему значительно менее энергозатратен, чем майнинг. Мы используем случайные значения, определенные во время выполнения, чтобы предотвратить случайную оптимизацию цикла компилятором.

Листинг 9.9. Приложение `InfiniteLoop` с нерациональным потреблением ресурсов ЦП

```
using System;

namespace InfiniteLoop {
```

```

class Program {
    public static void Main(string[] args) {
        Console.WriteLine("This app runs in an infinite loop");
        Console.WriteLine("It consumes a lot of CPU too!");
        Console.WriteLine("Press Ctrl-C to quit");
        var rnd = new Random();
        infiniteLoopAggressive(rnd.NextDouble());
    }

    private static void infiniteLoopAggressive(double x) {
        while (true) {
            x *= 13;
        }
    }
}

```

Скомпилируйте приложение `InfiniteLoop` и оставьте его работать в отдельном окне. Предположим, что это наш сервис на продакшнене и нам нужно выяснить, где он завис или почему потребляет так много ресурсов процессора. Нам бы очень помог стек вызовов, и мы можем найти его с помощью аварийных дампов, ничего не ломая.

У каждого процесса есть идентификатор (process identifier, PID) — уникальное числовое значение. Найдите PID процесса после запуска приложения. Можно либо использовать диспетчер задач Windows, либо просто запустить эту команду в командной строке PowerShell:

```
Get-Process InfiniteLoop | Select -ExpandProperty Id
```

В системе UNIX можно ввести:

```
pgrep InfiniteLoop
```

Получив в выводе PID процесса, можно создать файл дампа командой `dotnet dump`:

```
dotnet dump collect -p PID
```

Если ваш PID, скажем, 26190, введите

```
dotnet dump collect -p 26190
```

Команда покажет путь к аварийному дампу:

```
Writing full to C:\Users\ssg\Downloads\dump_20210613_223334.dmp
Complete
```

Затем можно проанализировать сгенерированный файл дампа с помощью `dotnet dump`:

```
dotnet dump analyze .\dump_20210613_223334.dmp
Loading core dump: .\dump_20210613_223334.dmp ...
Ready to process analysis commands. Type 'help' to list available commands or
  'help [command]' to get detailed help on a command.
Type 'quit' or 'exit' to exit the session.
> _
```

В UNIX для обозначения пути к файлу используется прямой слеш вместо обратного слеша в Windows. У этого различия интересная история, которая сводится к тому, что Microsoft добавила каталоги в MS-DOS 2.0 вместо 1.0.

Приглашение `analyze` принимает множество команд, список которых можно вывести, вызвав справку. Но чтобы понять, что делает процесс, достаточно знать только несколько из них. В частности, команда `threads` показывает потоки, запущенные в рамках процесса:

```
> threads
*0 0x2118 (8472)
 1 0x7348 (29512)
 2 0x5FF4 (24564)
 3 0x40F4 (16628)
 4 0x5DC4 (24004)
```

Текущий поток отмечен звездочкой, и его можно изменить с помощью команды `setthread`, например, так:

```
> setthread 1
> threads
 0 0x2118 (8472)
*1 0x7348 (29512)
 2 0x5FF4 (24564)
 3 0x40F4 (16628)
 4 0x5DC4 (24004)
```

Как видите, активный поток изменился. Однако `dotnet dump` может анализировать только управляемые потоки. Если вы попытаетесь просмотреть стек вызовов неуправляемого потока, то получите сообщение об ошибке:

```
> clrstack
OS Thread Id: 0x7348 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
```

Для полного анализа вам потребуется встроенный отладчик, такой как WinDbg, LLDB или GDB. В принципе, их работа будет аналогична анализу аварийных дампов. Но сейчас нас не интересует неуправляемый стек. Обычно приложению принадлежит поток 0, поэтому вернемся к нему и снова запустим команду `clrstack`:

```
> setthread 0
> clrstack
OS Thread Id: 0x2118 (0)
      Child SP          IP Call Site
000000D850D7E678 00007FFB7E05B2EB
    InfiniteLoop.Program.infiniteLoopAggressive(Double)
    [C:\Users\ssg\src\book\CH09\InfiniteLoop\Program.cs @ 15]
000000D850D7E680 00007FFB7E055F49 InfiniteLoop.Program.Main(System.String[])
    [C:\Users\ssg\src\book\CH09\InfiniteLoop\Program.cs @ 10]
```

За исключением пары неудобочитаемых адресов памяти, стек вызовов достаточно информативен. Он показывает, что делал этот поток, когда мы получили дамп до номера строки (числа после знака @), которой он соответствует, даже не прерывая выполнение процесса! Он получает эти данные из файлов отладочной информации с расширением .pdb в .NET и сопоставляет адреса памяти с символами и номерами строк. Вот почему важно развернуть отладочные символы на рабочем сервере на случай, если потребуется точно определить место возникновения ошибки.

Отладка аварийных дампов — это обширная тема, которая охватывает множество сценариев, таких как выявление утечек памяти и условий гонки. Ее логика во всех операционных системах, языках программирования и средствах отладки достаточно универсальна. Вы получаете снимок памяти и можете изучить содержимое файла, стек вызовов и данные. Считайте это отправной точкой и альтернативой традиционной пошаговой отладке.

9.3.3. Продвинутая отладка с помощью резиновой уточки

Как я упоминал в начале этой книги, единственный способ решить проблему — обсудить ее с резиновой уточкой, сидящей на вашем столе. Идея состоит в том, что, проговаривая проблему, вы формулируете ее более четко и решение находится волшебным образом.

В качестве альтернативы птичке я использую черновики сообщений на Stack Overflow. Вместо того чтобы задавать, возможно, глупый вопрос и заставлять кого-то тратить на него время, я просто описываю проблему, не публикуя сообщение. Почему на Stack Overflow? Потому что моральное давление со стороны

коллег на этом ресурсе заставляет меня возвращаться к важнейшему для решения проблемы вопросу: «Что я пробовал сделать?».

Спрашивать себя об этом очень полезно, но самое главное, это помогает осознать, что вы еще не испробовали все возможные решения. Размыслия только над этим вопросом, я открыл множество вариантов решений, которые ранее не рассматривал.

Важно, что модераторы Stack Overflow просят вас формулировать вопросы кратко и точно. Слишком общие вопросы сочтут не соответствующими теме, что стимулирует сужать проблему до конкретной задачи, которую можно проанализировать. Практикуясь в этом на Stack Overflow, вы выработаете привычку и позже сможете ограничиваться собственным мысленным анализом.

ИТОГИ

- Используйте приоритизацию, чтобы не тратить ресурсы на исправление незначимых ошибок.
- Перехватывайте исключения, только если знаете, что с ними делать.
- Пишите изначально устойчивый к исключениям код, который выдерживает сбои, а не пытайтесь избежать их постфактум.
- Когда ошибки распространены или ожидаемы, используйте вместо исключений код результата, или `enum`.
- Используйте предоставляемые фреймворком возможности отслеживания, чтобы выявлять проблемы быстрее, чем при пошаговой отладке.
- Используйте анализ аварийного дампа для выявления проблем с кодом, выполняющимся в рабочей среде, если другие методы недоступны.
- Используйте черновики как инструмент для отладки резиновой уточкой, чтобы проверить, какие варианты вы пробовали.

Седат Капаноглу

Кодер с улицы. Правила нарушать рекомендуется

Перевела с английского М. Трусковская

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Е. Строганова</i>
Литературный редактор	<i>Д. Гудилин</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, Н. Викторова</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 31.05.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 25,800. Тираж 1000. Заказ 0000.

Эл Свейгарт

РЕКУРСИВНАЯ КНИГА О РЕКУРСИИ



Книга «Рекурсивная книга о рекурсии» содержит примеры кода на языке Python и JavaScript, которые иллюстрируют основы рекурсии и проясняют фундаментальные принципы всех рекурсивных алгоритмов. Из книги вы узнаете о том, когда стоит использовать рекурсивные функции (и, главное, когда этого не нужно делать), как реализовывать классические рекурсивные алгоритмы, часто обсуждаемые на собеседованиях, а также о том, как рекурсивные методы помогают решать задачи, связанные с обходом дерева, комбинаторикой и другими сложными темами.

[КУПИТЬ](#)