Код, который работает всегда и везде

Стильный Java

Марко Фаэлла

предисловие Кея Хорстманна

MANNING

Seriously Good Software

Code that Works, Survives, and Wins

MARCO FAELLA

Foreword by Cay Horstmann

MANNING

SHELTER ISLAND

Марко Фаэлла
предисловие Кея Хорстманна

Стильный Java

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2021

Марко Фаэлла

Стильный Java. Код, который работает всегда и везде

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Художественный редактор	*В. Мостипан*
Корректоры	*Н. Викторова, М. Молчанова*
Верстка	*Л. Егорова*

ББК 32.973.2-018-02
УДК 004.42
Фаэлла Марко

Ф30 Стильный Java. Код, который работает всегда и везде. — СПб.: Питер, 2021. — 352 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1739-0

В современном мире разработки успешность приложения уже не определяется параметром «просто работает». Хороший программист должен знать возможности языка, практические приемы проектирования и платформенные средства для достижения максимальной производительности и жизнеспособности программ. Эта книга написана для разработчиков, которые хотят создавать качественный софт. Затронуты все ключевые показатели ПО: скорость, затраты памяти, надежность, удобочитаемость, потоковая безопасность, универсальность и элегантность. Реальные задачи и прикладные примеры кода на Java помогут надежно усвоить концепции.

Пройдя все этапы создания центрального проекта книги, вы сможете уверенно выбрать правильный путь оптимизации собственного приложения.

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

ISBN 978-1617296291 англ.
ISBN 978-5-4461-1739-0

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: 01.2021. 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 23.12.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 700. Заказ 0000.

Оглавление

Предисловие ... 13
Введение ... 15
Благодарности ... 17
О книге ... 18
Об авторе ... 22
Об обложке ... 23
От издательства ... 24

Часть I. Отправная точка

Глава 1. Свойства кода и постановка задачи ... 26
1.1. Свойства кода ... 26
1.1.1. Внутренние и внешние свойства ... 27
1.1.2. Функциональные и нефункциональные свойства ... 28
1.2. Преимущественно внешние свойства ... 29
1.2.1. Правильность ... 30
1.2.2. Стабильность ... 30
1.2.3. Удобство использования ... 31
1.2.4. Эффективность ... 31
1.3. Преимущественно внутренние свойства кода ... 32
1.3.1. Удобочитаемость ... 32
1.3.2. Возможность повторного использования ... 32

1.3.3. Удобство тестирования ... 33
1.3.4. Удобство сопровождения ... 33
1.4. Взаимодействие свойств кода ... 34
1.5. Специфичные свойства ... 36
1.5.1. Потокобезопасность ... 36
1.5.2. Лаконичность ... 37
1.6. Сквозной пример: система резервуаров для воды ... 37
1.6.1. API ... 38
1.6.2. Сценарий использования ... 39
1.7. Модель данных и представления ... 41
1.7.1. Хранение объема воды ... 42
1.7.2. Хранение информации о соединениях ... 43
1.8. Hello, резервуары! [Novice] ... 45
1.8.1. Поля и конструктор ... 45
1.8.2. Методы getAmount и addWater ... 47
1.8.3. Метод connectTo ... 47
Итоги ... 50
Дополнительная литература ... 50

Глава 2. Эталонная реализация ... 51
2.1. Код [Reference] ... 53
2.1.1. Диаграммы распределения памяти ... 54
2.1.2. Методы ... 57
2.2. Требования к памяти ... 59
2.2.1. Требования к памяти в Reference ... 61
2.3. Временна́я сложность ... 63
2.3.1. Временная сложность в Reference ... 67
2.4. Применим изученное ... 67
Итоги ... 68
Ответы на вопросы и упражнения ... 69
Дополнительная литература ... 72

Часть II. Свойства программного кода

Глава 3. Жажда скорости: эффективность по времени ... 74
3.1. Добавление воды с постоянным временем [Speed1] ... 76
3.1.1. Временная сложность ... 78

3.2. Добавление соединений за постоянное время [Speed2] ... 79
3.2.1. Представление групп в виде циклических списков ... 79
3.2.2. Отложенные обновления ... 82
3.3. Лучший баланс: алгоритмы поиска объединений [Speed3] ... 85
3.3.1. Поиск представителя группы ... 87
3.3.2. Соединение деревьев ... 90
3.3.3. Временная сложность в наихудшем случае ... 91
3.3.4. Амортизированная временная сложность ... 93
3.3.5. Амортизированный анализ массивов с изменяемым размером ... 95
3.4. Сравнение реализаций ... 98
3.4.1. Эксперименты ... 99
3.4.2. Теория и практика ... 101
3.5. А теперь совсем другое ... 102
3.5.1. Быстрая вставка ... 103
3.5.2. Быстрые запросы ... 104
3.5.3. Все операции ... 104
3.6. Реальные сценарии использования ... 105
3.7. Применим изученное ... 106
Итоги ... 107
Ответы на вопросы и упражнения ... 108
Дополнительная литература ... 111

Глава 4. Эффективность по затратам памяти ... 113
4.1. Первые шаги [Memory1] ... 114
4.1.1. Временная сложность и затраты памяти ... 117
4.2. Простые массивы ... 119
4.2.1. Временная сложность и затраты памяти ... 121
4.3. Отказ от объектов [Memory3] ... 124
4.3.1. API без объектов ... 124
4.3.2. Поля и метод getAmount ... 127
4.3.3. Создание резервуаров фабричным методом ... 128
4.3.4. Соединение резервуаров по идентификаторам ... 131
4.3.5. Временная сложность и затраты памяти ... 135
4.4. Черная дыра [Memory4] ... 135
4.4.1. Временная сложность и затраты памяти ... 139
4.5. Баланс затрат памяти и времени ... 139

4.6. А теперь совсем другое ... 141
4.6.1. Малое количество дубликатов ... 142
4.6.2. Большое количество дубликатов ... 143
4.7. Реальные сценарии использования ... 144
4.8. Применим изученное ... 145
Итоги ... 147
Ответы на вопросы и упражнения ... 147
Дополнительная литература ... 152

Глава 5. Надежность за счет мониторинга ... 154
5.1. Контрактное проектирование ... 154
5.1.1. Предусловия и постусловия ... 155
5.1.2. Инварианты ... 158
5.1.3. Правильность и стабильность ... 158
5.1.4. Проверка контрактов ... 160
5.1.5. Более широкая картина ... 162
5.2. Контрактное проектирование резервуаров ... 164
5.3. Резервуары, проверяющие свои контракты [Contracts] ... 167
5.3.1. Проверка контракта addWater ... 167
5.3.2. Проверка контракта connectTo ... 171
5.4. Резервуары, проверяющие свои инварианты [Invariants] ... 174
5.4.1. Проверка инвариантов в connectTo ... 176
5.4.2. Проверка инвариантов в addWater ... 178
5.5. А теперь совсем другое ... 179
5.5.1. Контракты ... 180
5.5.2. Эталонная реализация ... 180
5.5.3. Проверка контрактов ... 181
5.5.4. Проверка инвариантов ... 182
5.6. Реальные сценарии использования ... 184
5.7. Применим изученное ... 185
Итоги ... 187
Ответы на вопросы и упражнения ... 187
Дополнительная литература ... 193

Глава 6. Надежность за счет тестирования ... 194
6.1. Основные понятия тестирования ... 195
6.1.1. Покрытие в тестировании ... 195

6.1.2. Тестирование и контрактное проектирование ... 196
6.1.3. JUnit ... 198
6.2. Тестирование резервуаров [UnitTests] ... 200
6.2.1. Инициализация тестов ... 200
6.2.2. Тестирование addWater ... 202
6.2.3. Тестирование connectTo ... 209
6.2.4. Выполнение тестов ... 211
6.2.5. Оценка покрытия кода ... 212
6.3. Тестируемость [Testable] ... 214
6.3.1. Управляемость ... 214
6.3.2. Наблюдаемость ... 215
6.3.3. Isolation: разделение зависимостей ... 216
6.4. А теперь совсем другое ... 218
6.4.1. Повышение удобства тестирования ... 219
6.4.2. Набор тестов ... 221
6.5. Реальные сценарии использования ... 224
6.6. Применим изученное ... 225
Итоги ... 226
Ответы на вопросы и упражнения ... 226
Дополнительная литература ... 233

Глава 7. Удобочитаемость ... 234
7.1. Разный взгляд на удобочитаемость ... 234
7.1.1. Корпоративные руководства по стилю программирования ... 236
7.1.2. Составляющие удобочитаемости ... 237
7.2. Структурные факторы удобочитаемости ... 238
7.2.1. Управляющие команды ... 239
7.2.2. Выражения и локальные переменные ... 241
7.3. Внешние факторы удобочитаемости ... 242
7.3.1. Комментарии ... 243
7.3.2. Выбор имен ... 244
7.3.3. Пропуски и отступы ... 245
7.4. Код [Readable] ... 246
7.4.1. Документирование заголовка класса комментариями Javadoc ... 247
7.4.2. Чистка метода connectTo ... 251
7.4.3. Чистка addWater ... 255

7.5. Напоследок об удобочитаемости ... 257
7.6. А теперь совсем другое ... 258
7.7. Реальные сценарии использования ... 260
7.8. Применим изученное ... 261
Итоги ... 263
Ответы на вопросы и упражнения ... 263
Дополнительная литература ... 267

Глава 8. Потокобезопасность ... 269
8.1. Проблемы потокобезопасности ... 270
8.1.1. Уровни конкурентности ... 272
8.1.2. Политика конкурентности для резервуаров ... 275
8.2. Взаимоблокировки ... 276
8.2.1. Атомарные последовательности блокировок ... 277
8.2.2. Упорядоченные последовательности блокировок ... 279
8.2.3. Скрытое состояние гонки ... 281
8.3. Потокобезопасные резервуары [ThreadSafe] ... 282
8.3.1. Синхронизация connectTo ... 283
8.3.2. Синхронизация addWater и getAmount ... 284
8.4. Неизменяемость ... 287
8.4.1. API ... 289
8.4.2. Реализация ... 290
8.5. А теперь совсем другое ... 294
8.6. Реальные сценарии использования ... 296
8.7. Применим изученное ... 297
Итоги ... 299
Ответы на вопросы и упражнения ... 299
Дополнительная литература ... 302

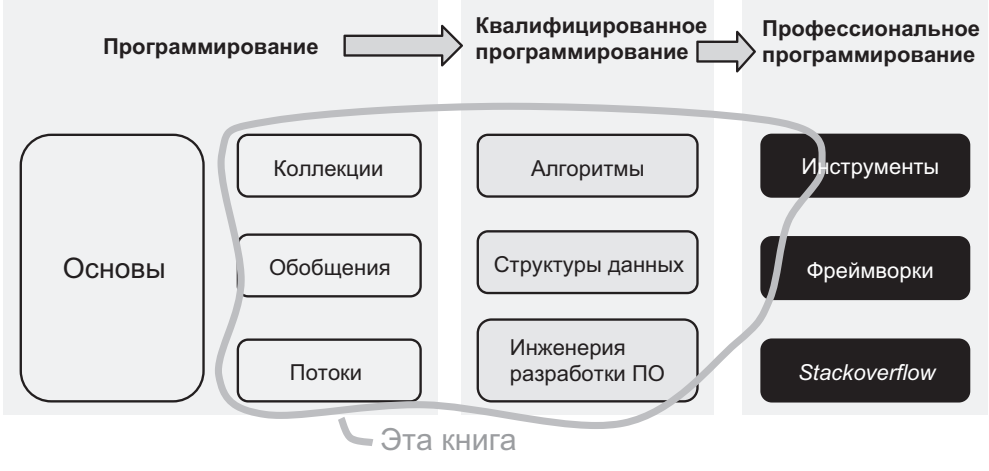
Глава 9. Повторное использование ... 304
9.1. Определение границ ... 304
9.2. Общая структура ... 306
9.2.1. API атрибутов ... 310
9.2.2. Изменяемые коллекторы ... 311
9.2.3. Адаптация Attribute для функциональных интерфейсов ... 317
9.3. Обобщенная реализация резервуара ... 318
9.4. Общие соображения ... 320

9.5. Использование готового кода [Generic] ... 321
9.5.1. Обновленный сценарий использования ... 321
9.5.2. Проектирование конкретного атрибута ... 322
9.5.3. Определение класса для резервуара ... 324
9.6. Посты в соцсетях ... 324
9.7. А теперь совсем другое ... 325
9.7.1. Интерфейс для параметрических функций ... 327
9.7.2. Схема коммуникаций ... 329
9.8. Реальные сценарии использования ... 332
9.9. Применим изученное ... 334
Итоги ... 336
Ответы на вопросы и упражнения ... 336
Дополнительная литература ... 342

Приложения

Приложение А. Программный гольф: компактность ... 344
А.1. Самая короткая программа, которая у меня получилась [Golfing] ... 344
Дополнительная литература ... 347

Приложение Б. Финальная версия класса для резервуара с водой ... 348
Б.1. Улучшения удобочитаемости ... 349
Б.2. Улучшения надежности ... 350


Программирование
Квалифицированное программирование
Профессиональное программирование
Основы
Коллекции
Обобщения
Потоки
Алгоритмы
Структуры данных
Инженерия разработки ПО
Инструменты
Фреймворки
Stackoverflow
Эта книга

Предисловие

За прошедшие тридцать лет я написал немало книг по программированию, и неудивительно, что время от времени кто-нибудь обращается ко мне за советом о том, как это сделать. Я всегда прошу прислать образец главы. В большинстве случаев на этом все заканчивается, и меня это нисколько не огорчает. Ведь если кто-то не может написать даже главу, то дальше и обсуждать нечего.

В январе 2018 года я получил сообщение от Марко Фаэллы — профессора Неапольского университета, с которым я уже встречался в Калифорнийском университете в Санта-Крус. Он попросил совета относительно проекта книги. Более того, он уже написал несколько глав! То, что я увидел, мне понравилось. Я похвалил работу автора и внес несколько предложений. На этом общение прервалось. Меня это не удивило. Один из редакторов сказал мне, что он знал великое множество людей, которые начинали писать книгу... и лишь немногих, кто ее дописал.

В апреле 2019 года я получил новое сообщение — книга готовилась к печати в издательстве Manning и выглядела просто отлично. В августе Марко попросил меня написать предисловие, что я и делаю с огромным удовольствием.

Когда я пишу книгу о языке программирования, то уделяю основное внимание специфичным для него конструкциям и API. И предполагаю, что читатель разбирается в структурах данных, алгоритмах и основных принципах разработки, таких как тестирование, рефакторинг и паттерны проектирования. Конечно, я как профессор отлично знаю, что университет не учит применять перечисленное.

Именно этот пробел восполняет данная книга. Если вы знакомы с основами программирования на языке Java, книга поможет вам писать более качественные программы и представит классические темы, такие как проектирование алгоритмов и API, тестирование и конкурентность, в новом свете. Из одного

примера Марко выводит множество идей и реализаций. Сам я не люблю метод сквозного примера, вынуждающий читать книгу последовательно, контролируя текущее состояние кода. Но пример Марко (природу которого я не хочу заранее раскрывать) очень умно спроектирован. От вас потребуется лишь усвоить пару интересных базовых концепций, а дальше в каждой главе код будет развиваться независимо. В общем, это высший пилотаж.

В основных главах вы найдете раздел «А теперь совсем другое», где вам будет предложено применить материал главы в иной ситуации. Рекомендую относиться к этому как к неожиданным вопросам во врезках и упражнениям в конце глав.

Создание качественной программы — непростое дело. Всегда полезно возвращаться к принципам проектирования и методологии разработки. В этой книге вы взглянете на этот процесс под новым углом, и, надеюсь, вам это понравится так же, как и мне.

Кей Хорстманн,
Автор «Java»[1]*, «Java / Scala / JavaScript for the Impatient»*
и многих других книг для начинающих и опытных программистов

[1] *Хорстманн К. С.* Java / Пер. с англ. И. В. Берштейна. 11-е изд. — М.; СПб.: Диалектика, 2020. — (Серия «Библиотека профессионала»)

Введение

Я хотел назвать книгу «Java: упражнения в стиле». Но умные люди из издательства Manning помогли мне выбрать более яркое и запоминающееся название[1]. Дело в том, что в работе «Упражнения в стиле» французский писатель Раймон Кено записал одну простую историю в девяносто девяти стилистических вариантах.

Конечно, программирование не литература, как бы известные личности вроде Дональда Кнута ни пытались их породнить. Однако только новички верят, что у любой задачи программирования есть один лучший способ решения — как единственное решение простой математической задачи. Выходит, что у программирования больше сходства с литературой, чем с математикой. И правда, в процессе эволюции в языках программирования появляются более абстрактные конструкции, добавляющие новые способы достижения тех же целей. А самые популярные языки, такие как Java, развиваются ускоренными темпами, чтобы не отставать от молодых языков.

В этой книге я освещаю широкий спектр проблем и решений, которые необходимо учитывать (или по крайней мере знать об их существовании) при написании кода. Предложенная задача тривиальна: создать класс резервуаров, которые можно соединять трубами и наполнять водой. С резервуарами взаимодействуют клиенты: добавляют или удаляют воду, устанавливают новые трубы. В восемнадцати реализациях этой задачи я постараюсь довести до максимума разные показатели кода: скорость выполнения, удобочитаемость и др. Это не будет сухой последовательностью фрагментов кода. Каждый раз, когда того потребует контекст, я затрону специализированные темы computer science (структуры данных, теорию сложности, амортизированную сложность),

[1] Издательство «Питер» не смогло в полной мере пойти навстречу автору, но постаралось приблизить название русского издания к изначальной задумке. — *Примеч. ред.*

Java-программирования (синхронизацию программных потоков и модель памяти Java), методологии разработки (методологию контрактного проектирования и приемы тестирования). Я покажу, что даже простой пример при глубоком анализе оказывается связанным с огромной сетью тем, знание которых помогает писать более качественный код.

Дополнительная литература

Одна из целей этой книги — пробудить в вас интерес к различным дисциплинам, связанным с разработкой ПО. Именно по этой причине каждая глава завершается разделом «Дополнительная литература», где кратко перечислены лучшие, по моему мнению, источники информации по теме главы. Предисловие не станет исключением, поэтому держите:

- *Raymond Queneau*. Exercises in Style (New Directions, 2013).

  Образец «упражнений в стиле», написанный на французском языке в 1947 году.

- *Cristina Videira Lopes*. Exercises in Programming Style (CRC, 2014).

  Автор решает простую задачу программирования в тридцати трех стилях на языке Python. Вместо оптимизаций разных аспектов кода вы найдете ограничения, добавленные каждому стилю, и узнаете много интересного об истории языков программирования.

- *Matt Madden*. 99 Ways to Tell a Story: Exercises in Style (Jonathan Cape, 2006).

  Когда захотите отвлечься от программирования, полистайте этот комикс с простой историей, нарисованной в девяносто девяти графических стилях.

Благодарности

Я знал, что мне придется упорно работать над книгой, но был удивлен, что не только мне. Я говорю о сотрудниках издательства Manning, которые (каждый на своем месте) поднимали качество книги до предела. Рецензенты не жалели времени на написание подробных отзывов для улучшения содержания. Надеюсь, результат им понравится.

Я изучал Java по книгам Кея и много лет рекомендовал их студентам. Для меня большая честь, что он любезно согласился написать предисловие.

Благодарю всех рецензентов: Адитья Каушик (Aditya Kaushik), Александрос Кофудакис (Alexandros Koufoudakis), Бонни Бэйли (Bonnie Bailey), Элио Сальваторе (Elio Salvatore), Флавио Диез (Flavio Diez), Фостер Хейнс (Foster Haines), Грегори Решетняк (Gregory Reshetniak), Хессам Шафий Мокаддам (Hessam Shafiei Moqaddam), Ирфан Улла (Irfan Ullah), Джейкоб Ромеро (Jacob Romero), Джон Гатри (John Guthrie), Хуан Дж. Дурилло (Juan J. Durillo), Кимберли Уинстон-Джексон (Kimberly Winston-Jackson), Михал Амброзиевич (Michał Ambroziewicz), Серж Саймон (Serge Simon), Стивен Парр (Steven Parr), Торстен Вебер (Thorsten Weber) и Трэвис Нелсо (Travis Nelso) — ваши предложения помогли улучшить книгу.

Моя страсть к программированию началась в период, когда отец учил меня — любознательного восьмилетку — рисовать круг при помощи цикла `for` и двух таинственных заклинаний, которые назывались `sin` и `cos`. *Grazie*!

О книге

Главная идея этой книги — показать мышление опытного разработчика с помощью сравнения свойств программного кода (или нефункциональных требований).

Схема на с. 12 связывает содержимое книги с широким спектром знаний, необходимых профессиональному разработчику. Изучение Java требует знакомства с классами, методами, полями и т. д. (здесь база не рассматривается). Далее освоение языка идет по трем путям:

- *Путь языка программирования* — изучение расширенных возможностей языка (таких, как обобщения и многопоточность).
- *Путь алгоритмов* — изучение базовых теоретических принципов, стандартных алгоритмов и структур данных.
- *Путь технологии программирования* — изучение принципов проектирования и передовых методов, упрощающих управление сложностью, особенно в крупных проектах.

В книге рассмотрены все эти области, но в несколько нетрадиционном виде. Вместо того чтобы освещать разные аспекты по отдельности, я смешал их в соответствии с потребностями определенной главы.

Каждая глава посвящена одному свойству программного кода, такому как скорость выполнения или удобочитаемость. Они не только важны и распространены, но и могут осмысленно применяться к малым кодовым единицам (например, к отдельному классу). Я постарался сосредоточиться на общих принципах и приемах программирования, а не на конкретных инструментах. В ряде случаев я ссылался на инструменты и библиотеки, которые помогут оценить и оптимизировать рассматриваемые свойства.

Для кого написана эта книга

Книга станет идеальной отправной точкой для начинающего разработчика и поможет расширить его представления о кодинге. Точнее, книга ориентирована на две основные аудитории:

- Разработчики с пробелами в теоретической подготовке или подготовкой в другой области (отличной от computer science) найдут здесь обзор теоретических и практических методов и компромиссов, связанных с нетривиальными задачами программирования.
- Студенты и специалисты computer science найдут в книге объединяющий практический сценарий для разных учебных дисциплин. Таким образом, книга дополнит учебники и другие материалы по программированию.

В обоих случаях извлечь максимальную пользу из книги помогут знания:

- основных концепций программирования (итераций, рекурсии и т. д.);
- основных концепций объектно-ориентированного программирования (инкапсуляции, наследования и т. д.);
- средств языка Java среднего уровня (обобщений, стандартных коллекций и базовой многопоточности: создания потоков, применения ключевого слова `synchronized` и т. д.).

Структура книги

Ниже приведен краткий список глав и свойств кода, которые в них рассматриваются. Не пренебрегайте упражнениями в конце каждой главы. Они сопровождаются подробными решениями и завершают материал главы применением описанных методов в разных контекстах.

Глава 1. В первой главе описана задача программирования, которую мы будем решать (класс для представления резервуаров с водой). Здесь приведена наивная реализация, которая демонстрирует типичные заблуждения неопытных программистов.

Глава 2. Подробное описание эталонной реализации, обеспечивающей хороший баланс разных свойств.

Глава 3. Сосредоточившись на эффективности по времени, мы улучшим время выполнения эталонной реализации более чем на два порядка (в 500 раз) и увидим, что разные сценарии практического использования вынуждают нас идти на разные компромиссы.

Глава 4. Проведем эксперименты с эффективностью по затратам памяти и увидим, что по сравнению с эталонной реализацией затраты памяти сокращаются более чем на 50 % при использовании объектов и на 90 % — при отказе от использования отдельного объекта для каждого резервуара.

Глава 5. Постараемся достичь надежности за счет *контрактного проектирования* и усиления эталонного класса проверками во время выполнения, а также с помощью тестовых условий, основанных на контрактах методов и инвариантах классов.

Глава 6. Постараемся достичь надежности за счет модульного тестирования с помощью методов проектирования и выполнения набора тестов, а также рассмотрим метрики и средства тестового покрытия кода.

Глава 7. Произведем рефакторинг эталонной реализации для применения рекомендуемых методов создания чистого самодокументируемого кода.

Глава 8. В контексте конкурентности и потокобезопасности вспомним основные понятия синхронизации потоков и выясним, почему в нашем текущем примере необходимо применять нетривиальные механизмы для предотвращения взаимных блокировок и состояния гонки.

Глава 9. Рассмотрим возможность повторного использования: обобщим эталонный класс, чтобы он мог применяться в других приложениях с аналогичной общей структурой.

Приложение А. При обсуждении лаконичности кода я представлю компактную реализацию примера, объем исходного кода которого составит всего 15 % от эталонной версии. Конечно, получится заумный код, за который вас запинают на любом сеансе рецензирования кода.

Приложение Б. Наконец, мы соберем воедино все свойства и построим финальную версию класса, представляющего резервуары.

Сетевой репозиторий

Весь код, представленный в книге, доступен в открытом сетевом репозитории (https://bitbucket.org/mfaella/exercisesinstyle). Основная часть кода состоит из разных версий класса `Container`. Каждой версии назначено условное имя, соответствующее имени пакета. Например, первая версия представлена в разделе 1.8 под именем `Novice`. В репозитории соответствующий класс называется `eis.chapter1.novice.Container`. В таблице в приложении Б перечислены основные классы и их характеристики.

Код примеров этой книги также доступен для загрузки на веб-сайте Manning по адресу: https://www.manning.com/books/seriously-good-software.

Почему Java? Какая версия Java?

Как вы отлично знаете, язык Java стремительно развивается: новые версии выходят каждые полгода. На момент написания книги текущая версия — Java 12.

Но эта книга посвящена не освоению Java, а приобретению привычки оценивать и искать баланс между разными свойствами программного продукта независимо

от языка, на котором он создан. Примеры написаны на Java, потому что он популярен и у меня есть большой опыт работы с ним.

Принципы, которые я излагаю в книге, так же хорошо работают на других языках. Чем ближе ваш язык программирования к Java (например, к нему близок C#), тем больше материала из книги вы сможете позаимствовать без изменений. Более того, в книге часто встречаются примечания для C#, в которых подчеркиваются различия в языках, связанные с материалом текущей главы.

В сетевом репозитории книги хранится код, 99 % которого написано на Java 8. В нескольких его частях использованы дополнения из Java 9 (например, возможность конструирования списка статическим методом `List.of`).

Форум для обсуждения книги

Приобретая книгу, вы получаете бесплатный доступ к закрытому веб-форуму Manning, на котором можно публиковать комментарии к книге, задавать технические вопросы и получать помощь от автора и других пользователей. Для доступа к форуму откройте страницу https://livebook.manning.com/#!/book/seriously-good-software/discussion. За информацией о форумах Manning и правилах поведения обращайтесь по адресу https://livebook.manning.com/#!/discussion.

В рамках обязательств перед читателями издательство Manning предоставляет ресурс для общения читателей с автором. Эти обязательства не подразумевают конкретную степень вовлеченности автора, участие которого в работе форума остается добровольным (и неоплачиваемым). Задавайте автору интересные вопросы, чтобы он не терял интереса к происходящему! Форум и архивы предшествующих обсуждений доступны на веб-сайте издательства, пока книга находится в печати.

Об авторе

Марко Фаэлла — преподаватель computer science в Неаполитанском университете имени Фридриха II (Италия). Помимо академических исследований в области computer science Марко увлеченно занимается преподаванием и программированием. Последние 13 лет он ведет курсы программирования повышенной сложности, а также является автором учебника для желающих получить сертификат Java-разработчика и видеокурса по потокам в языке Java.

Об обложке

Иллюстрация, помещенная на обложку второго издания книги, называется Homme Tscheremiss («черемис» — представитель народа, живущего неподалеку от современной Финляндии), была позаимствована из изданного в 1797 г. каталога национальных костюмов Жака Грассе де Сен-Совера (1757–1810) «Costumes de Différents Pays». Каждая иллюстрация красиво нарисована и раскрашена от руки. Иллюстрации из каталога Грассе де Сен-Совера напоминают о культурных различиях между городами и весями мира, имевших место почти двести лет назад. Люди, проживавшие в изолированных друг от друга регионах, говорили на разных языках и диалектах. По одежде человека можно было определить, в каком городе, поселке или поселении он проживает.

С тех пор дресс-код сильно изменился, да и различия между разными регионами стали не столь выраженными. В наше время довольно трудно узнать жителей разных континентов, не говоря уже о жителях разных городов или регионов. Возможно, мы отказались от культурных различий в пользу более разнообразной личной жизни — и конечно, в пользу более разнообразной и стремительной технологической жизни.

Сейчас, когда все компьютерные книги похожи друг на друга, издательство Manning стремится к разнообразию и помещает на обложки книг иллюстрации, показывающие особенности жизни в разных странах мира два века назад.

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.

Часть I

Отправная точка

Главная идея этой книги — раздельная оптимизация разных свойств программного кода с помощью одного сквозного примера. В части I книги я представлю понятие свойств кода и опишу ту простую задачу программирования, которую мы будем снова и снова решать в этой книге.

Далее мы рассмотрим две предварительные реализации: наивная версия, которую может написать неопытный программист, и эталонная версия, обеспечивающая разумные компромиссы между свойствами кода.

1 Свойства кода и постановка задачи

В этой главе:

- ✓ Оценка кода с разных точек зрения и в контексте разных целей.
- ✓ Внутренние и внешние свойства кода.
- ✓ Функциональные и нефункциональные свойства кода.
- ✓ Взаимодействия и компромиссы между свойствами кода.

Основная идея этой книги — передать образ мышления опытного разработчика с помощью сравнения различных качества кода (также известных как нефункциональные требования). Многие из этих свойств — например, скорость выполнения или удобочитаемость — универсальны для любого кода. Чтобы более наглядно выделить этот факт, мы будем в каждой главе возвращаться к одному сквозному примеру: простому классу, представляющему систему резервуаров для воды.

В этой главе будут представлены свойства кода, рассматриваемые в книге. Кроме того, я приведу спецификации для примера с резервуарами и его предварительную реализацию.

1.1. СВОЙСТВА КОДА

Под термином *свойство* мы будем подразумевать характеристику, которой может обладать (или не обладать) фрагмент кода, а не некую черту всего кода. Вот

почему я говорю о свойствах *во множественном числе*. Не все характеристики могут считаться свойствами; например, язык программирования, на котором написан блок кода, безусловно является его характеристикой, но к свойствам он не относится. Свойства — характеристики, которые могут оцениваться по некоторой шкале (по крайней мере теоретически).

Разумеется, из всех свойств кода наибольший интерес вызывают те, с помощью которых можно оценить, насколько система соответствует поставленным требованиям. К сожалению, даже простое формулирование (не говоря уже о выполнении) требований к блоку кода — задача не из простых. Этой теме посвящена целая область анализа требований. Как такое возможно? Разве недостаточно, чтобы система надежно и последовательно предоставляла необходимые возможности?

Во-первых, пользователи далеко не всегда точно знают, какие возможности им нужны: чтобы разобраться в этом, им нужны время и помощь. Во-вторых, выполнением требований дело не заканчивается. Возможности могут предоставляться быстро или медленно, с большей или меньшей точностью, после долгого обучения пользователя или после беглого взгляда на хорошо спроектированный пользовательский интерфейс и т. д. Кроме того, со временем любую систему приходится изменять, исправлять или совершенствовать, что приводит к появлению новых переменных качества. Насколько легко разобраться во внутреннем устройстве системы? Насколько легко она изменяется и расширяется без нарушения работоспособности частей? Список вопросов только растет.

Чтобы внести некое подобие порядка в это многообразие свойств, эксперты рекомендуют разделить их на категории по двум характеристикам: внутренние и внешние, а также функциональные и нефункциональные.

1.1.1. Внутренние и внешние свойства

Конечный пользователь воспринимает внешние свойства, взаимодействуя с системой, при этом внутренние свойства он может оценить, только просмотрев исходный код. Между двумя категориями нет четкой границы: конечный пользователь может косвенно воспринимать некоторые внутренние свойства, а все внешние свойства в итоге зависят от исходного кода.

СТАНДАРТЫ СВОЙСТВА КОДА

Организации — разработчики стандартов ISO и IEC определили требования к свойству кода еще в 1991 году в стандарте 9126, который был заменен стандартом 25010 в 2011 году.

Например, удобство сопровождения (простота изменения, исправления или расширения системы) является внутренним свойством кода, но конечные пользователи поздно узнают о дефекте, и его исправление требует много времени. И наоборот, устойчивость к некорректным входным данным обычно относится к внешним свойствам кода, но она становится внутренним атрибутом, если рассматриваемый фрагмент кода — возможно, библиотека — недоступен для конечного пользователя и взаимодействует только с другими системными модулями.

1.1.2. Функциональные и нефункциональные свойства

Второе различие проявляется между свойствами, относящимися к тому, что *делает* код (функциональные свойства) и *как* он работает (нефункциональные свойства) (рис. 1.1). Отделение внутренних свойств от внешних также проявляется и в этом различии: если программный код что-то делает, то результат так или иначе виден конечному пользователю. Поэтому все функциональные свойства являются внешними. С другой стороны, нефункциональные могут быть как внутренними, так и внешними в зависимости от того, относятся ли они в большей степени к самому коду или к проявляемым им характеристикам.

Рис. 1.1. Функциональные и нефункциональные требования воздействуют на код в разных направлениях. Ваша задача — найти баланс между ними

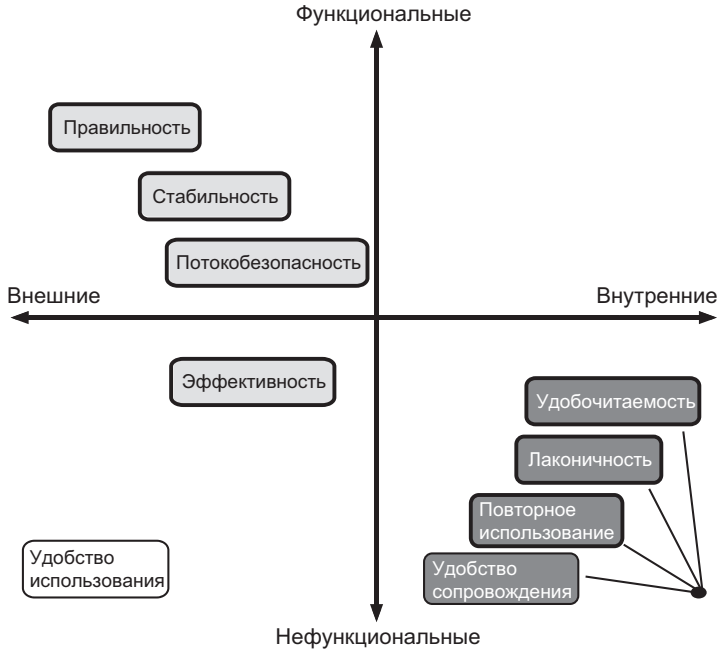


Рис. 1.2. Классификация свойств программного кода в контексте двух разделений: внутренние и внешние (горизонтальная ось) и функциональные и нефункциональные (вертикальная ось). Свойства, рассматриваемые в книге, выделены жирной рамкой

В следующих разделах приводятся примеры обоих видов. А пока взгляните на рис. 1.2: все свойства, упомянутые в этой главе, размещены на плоскости. Горизонтальная ось представляет различия между внутренними и внешними свойствами, а вертикальная — различия между функциональными и нефункциональными. В следующем разделе представлены основные свойства кода, непосредственно видимые конечному пользователю.

1.2. ПРЕИМУЩЕСТВЕННО ВНЕШНИЕ СВОЙСТВА

Внешние свойства кода относятся к наблюдаемому поведению программы, и естественно, основное внимание в процессе разработки уделяется именно им. Кроме обсуждения этих свойств как атрибутов программного кода, я также рассмотрю их на примере обычного тостера, чтобы дать представление о них в самом общем смысле. В следующих подразделах приводятся описания важнейших внешних свойств.

1.2.1. Правильность

Соблюдение заявленных целей (требований, спецификаций).

Правильно работающий тостер должен поджаривать нарезанный хлеб, пока он не подрумянится и не станет хрустящим. В то же время код тостера должен обеспечивать функциональность, согласованную с заказчиком. Правильность *является* функциональным свойством по определению.

Никакого секретного рецепта правильности нет, но разработчики применяют различные приемы и технологии для повышения вероятности того, что написанный код будет правильным и все дефекты будут выявлены. В этой книге мы сосредоточимся на универсальных практических приемах.

Прежде всего, правильность может быть достигнута только в том случае, когда разработчик хорошо понимает спецификации, на которые нужно ориентироваться. Рассмотрение спецификаций в категориях *контрактов* и реализация защитных мер для контроля за их выполнением — полезные концепции, которые будут рассмотрены в главе 5. Основной способ выявления неизбежных дефектов заключается в имитации ряда взаимодействий с кодом, то есть в *тестировании*. В главе 6 описаны систематические методы планирования тестовых сценариев и оценки их эффективности. Наконец, передовые методы написания удобочитаемого кода способствуют достижению правильности — они помогают как автору кода, так и его коллегам выявить проблемы до и после тестирования. В главе 7 приведена подборка таких методов.

1.2.2. Стабильность

Устойчивость перед некорректными входными данными или аномальными и непредвиденными внешними условиями (например, отсутствием некоторого ресурса).

Правильность и стабильность иногда объединяют под термином *надежность* (reliability). Стабильно работающий тостер не загорится, если в него вместо хлеба положат бублик, вилку или вообще ничего. В нем предусмотрены защитные меры против перегрева, и т. д.[1]

Стабильный код, среди прочего, проверяет входные данные на корректность. Если данные некорректны, код сообщает о проблеме и реагирует на нее соответствующим образом. Если ошибка фатальна, то стабильная программа аварийно завершается, сохранив максимально возможный объем пользовательских данных или результатов вычислений, выполненных к текущему моменту. В главе 5 описаны меры по достижению стабильности за счет жесткого следования

[1] Безопасность тостера — не шутка: по статистике, из-за несчастных случаев с тостерами в мире за год погибает более семисот человек.

спецификациям и отслеживания контрактов методов и инвариантов класса на стадии выполнения.

1.2.3. Удобство использования

Оценка усилий, необходимых для освоения кода и достижения целей с его помощью, простота использования.

Современные тостеры очень просты в использовании. В них нет специального рычага, который опускает хлеб и начинает приготовление тостов, или рукоятки для регулировки степени обжарки. Удобство использования программного кода тесно связано с проектированием его пользовательского интерфейса (UI, user interface). На него влияют такие дисциплины, как теория взаимодействия человека с компьютером и проектирование пользовательского опыта (UX, user experience). В этой книге тема удобства использования не рассматривается, потому что книга посвящена программным системам, не предназначенным для прямого взаимодействия с конечным пользователем.

1.2.4. Эффективность

Разумное потребление ресурсов.

Под «эффективностью тостера» можно понимать то, сколько времени и электроэнергии потребует выполнение им задачи. Время и память — два основных ресурса, потребляемых всеми программами. В главах 3 и 4 рассмотрены темы эффективности времени и памяти соответственно. Многим программам также требуется пропускная способность сети, подключения к базам данных и другие ресурсы. Между потреблением разных ресурсов обычно возникают компромиссы. Более мощный тостер может поджаривать хлеб быстрее, но потреблять больше электроэнергии (пиковой). Аналогичным образом выполнение некоторых программ можно ускорить за счет повышения затрат памяти (об этом позднее).

Хотя я перечисляю эффективность среди внешних свойств, ее истинная природа двойственна. Например, скорость выполнения определенно заметна для конечного пользователя, особенно когда она ограничена. С другой стороны, потребление других ресурсов (таких, как пропускная способность сети) скрыто от пользователя, а оценить его можно только при помощи специализированных инструментов или анализа исходного кода. Вот почему на рис. 1.2 эффективность в середине.

Эффективность в основном является нефункциональным свойством, потому что пользователя обычно не интересует, выполняется ли некоторая операция за одну или две миллисекунды или сколько передается килобайтов данных, один или два. Она становится функциональной проблемой в двух ситуациях:

- *Когда скорость выполнения критически важна,* гарантии быстрого ответа являются частью спецификации приложения. Представьте встроенное устройство, взаимодействующее с физическими датчиками и приводами. Время отклика его кода должно подчиняться точно заданным тайм-аутам. Несоблюдение тайм-аутов в лучшем случае приведет к функциональной непоследовательности, в худшем — к опасным для жизни происшествиям (в промышленных, медицинских или автомобильных приложениях).
- *Когда низкая эффективность ухудшает работу приложения.* Для любой программы, ориентированной на потребителя, существует предел заторможенности и нехватки памяти, с которыми готов мириться пользователь. Превышение этого порога ведет к функциональному дефекту программы.

1.3. ПРЕИМУЩЕСТВЕННО ВНУТРЕННИЕ СВОЙСТВА КОДА

Внутренние свойства проще оценить при просмотре исходного кода программы, а не при ее запуске. В следующих подразделах приводятся описания важнейших внутренних свойств.

1.3.1. Удобочитаемость

Ясность для коллег-программистов.

Удобочитаемость применительно к тостеру выглядит странно, пока мы не поймем, что все внутренние свойства основаны на структуре и дизайне. В международном стандарте свойств кода эта характеристика называется *анализируемостью*. Разобранный *удобочитаемый* тостер легко анализируется, потому что имеет ясную внутреннюю структуру, схему питания, таймер, нагревательные элементы, изолированные от электроники, и т. д.

Смысл удобочитаемости кода вытекает из самого термина: такой код понятен другому программисту (или автору, забывшему детали собственной программы). Удобочитаемость — чрезвычайно важное свойство кода, которое часто недооценивают. Этой теме посвящена глава 7.

1.3.2. Возможность повторного использования

Простота повторного использования кода в решениях похожих задач и объем необходимых для этого изменений также называются адаптируемостью.

Тостер можно считать повторно используемым, если его дизайн и составляющие подходят для создания других устройств. Например, стандартный шнур

питания совместим с различными бытовыми устройствами, таймер встраивается в СВЧ-печь и т. д.

Возможность повторного использования была одним из важнейших аргументов в пользу парадигмы объектно-ориентированного программирования. Но опыт показал, что представления о построении сложных систем из программных компонентов с высокой степенью повторного использования были преувеличены. В наши дни предпочтение отдается библиотекам и фреймворкам, спроектированным для повторного использования, над которыми строится (не такая уж тонкая) прослойка кода, специфичного для приложения и для повторного использования не предназначенного. Подробнее в главе 9.

1.3.3. Удобство тестирования

Возможность (и простота) написания тестов, которые позволят активизировать все важные аспекты поведения проекта и наблюдать их результаты.

Прежде чем обсуждать удобство тестирования тостеров, попробуем разобраться, как может выглядеть тест для тостера[1]. Например, положить термометры в щели для тостов и включить прибор. Успех оценивается в зависимости от того, насколько изменение температуры со временем приближается к заранее определенному номинальному значению. Более удобным является многократное автоматическое тестирование с минимальным участием человека. Например, машине проще сымитировать запуск при помощи кнопки, нежели рычага.

Код, удобный для тестирования, предоставляет программный интерфейс (API), который позволяет вызывающей стороне проверить все ожидаемые аспекты поведения. Например, `void`-метод (процедура) обладает меньшим удобством тестирования, чем метод, возвращающий значение. Приемы тестирования и удобство тестирования рассматриваются в главе 6.

1.3.4. Удобство сопровождения

Простота нахождения и исправления ошибок, а также эволюции программного кода.

Тостер, удобный в сопровождении, легко разбирается и не создает проблем с техническим обслуживанием. Его схемы доступны, а компоненты легко заменяются. Аналогичным образом код с удобным сопровождением легко читается и имеет модульную структуру, где существует разделение ответственности. Основной вклад в удобство сопровождения вносят удобство тестирования и удобочитаемость, рассматриваемые в главах 6 и 7.

---

[1] По некоторым данным, вопрос «как протестировать тостер?» часто встречается на собеседованиях для разработчиков.

МОДЕЛЬ FURPS

Крупные компании с сильными техническими традициями разрабатывают собственную модель свойств для процессов разработки. Например, в Hewlett-Packard была разработана известная модель FURPS, которая классифицирует характеристики кода на пять групп: функциональность, удобство использования, надежность, скорость выполнения и удобство сопровождения.

1.4. ВЗАИМОДЕЙСТВИЕ СВОЙСТВ КОДА

Одни свойства кода преследуют взаимно противоречащие цели, другие — идут рука об руку. Инженер, как водится, должен выбрать сбалансированное решение. У математиков для задач такого рода существует специальный термин — *многоцелевая оптимизация*, то есть поиск оптимальных решений с учетом нескольких противоречивых показателей. В отличие от абстрактной математической задачи, не все свойства кода могут быть выражены числом (взять хотя бы удобочитаемость). К счастью, искать нужно не оптимальное решение, а достаточно отвечающее целям.

В табл. 1.1 приведена сводка отношений между четырьмя свойствами, рассматриваемыми в книге. Эффективность по времени и по затратам памяти ухудшает удобочитаемость. Стремление к достижению максимальной скорости выполнения заставляет жертвовать абстракцией и писать низкоуровневый код. В Java это может привести к использованию примитивных типов вместо объектов, простых массивов вместо коллекций, а в самых крайних случаях — написанию критических частей на низкоуровневом языке (таком, как C) и связыванию их с основной программой через механизм Java Native Interface.

Таблица 1.1. Типичные взаимодействия между свойствами кода: ↓ означает «мешает», а «—» означает «не взаимодействует». (По материалам рис. 20.1 «Совершенный код» — см. раздел «Дополнительная литература» в конце главы)

	Удобочитаемость	**Стабильность**	**Эффективность по затратам памяти**	**Эффективность по времени**
Удобочитаемость				
Стабильность	–			
Эффективность по затратам памяти	↓	–		
Эффективность по времени	↓	↓	↓	

Минимизация требований к памяти также способствует применению примитивных типов и специальных кодировок, в которых одно значение используется для компактного представления разных сущностей (пример в разделе 4.4). Все эти приемы обычно вредят удобочитаемости и, как следствие, — удобству сопровождения. И наоборот, удобочитаемый код использует больше временных переменных и вспомогательных методов и держится подальше от низкоуровневых трюков повышения скорости выполнения.

Эффективность по времени и эффективность по затратам памяти также конфликтуют друг с другом. Например, стандартная стратегия повышения скорости выполнения подразумевает хранение в памяти дополнительной информации, вместо того чтобы каждый раз вычислять ее при необходимости. Типичный пример — различия между односвязными и двусвязными списками. Несмотря на то что «предыдущее» звено каждого узла теоретически можно определить перебором списка, хранение и поддержание ссылок позволяет выполнять удаление произвольных узлов с постоянным временем. Класс в разделе 4.4 достигает более эффективного времени выполнения за счет снижения эффективности по затратам памяти.

Чтобы максимизировать стабильность, необходимо добавить код проверки аномальных ситуаций и соответствующей реакции на них. Такие проверки отрицательно сказываются на общей скорости выполнения, хотя их влияние может быть весьма ограниченным. При этом они могут никак не отразиться на эффективности по затратам памяти. Также теоретически нет причин, по которым стабильный код должен быть менее удобочитаемым.

МЕТРИКИ КОДА

Свойства кода связаны с *метриками* — количественными характеристиками блока кода. В литературе предлагаются сотни разных метрик, две самые распространенные — число строк кода (LOC, lines of code) и цикломатическая сложность (оценка степени вложения и ветвления). Метрики позволяют объективно оценивать и контролировать текущее состояние проекта для принятия неких мер. Например, метод с высокой цикломатической сложностью требует более основательной работы по тестированию.

Современные интегрированные среды (IDE) вычисляют метрики с помощью либо встроенных ресурсов, либо специальных плагинов. Относительные преимущества этих метрик, их связь с общими свойствами кода, описанными в этой главе, а также их эффективное применение — в высшей степени неоднозначные темы в сообществе разработчиков. В этой книге будут использоваться метрики покрытия кода (глава 6).

Всем этим свойствам противостоит такой фактор, как время разработки. Коммерческие соображения подталкивают к быстрому написанию кода, но максимизация любого атрибута свойства требует сознательного выделения времени и ресурсов. Даже если руководство хорошо понимает перспективные преимущества тщательного проектирования кода, может быть достаточно трудно оценить, сколько времени потребуется для получения наилучшего результата. Технологические процессы разработки (весьма разнообразные) предлагают разные решения этой проблемы, в части которых задействованы метрики, упомянутые во врезке.

В этой книге мы не будем углубляться в споры по поводу технологических процессов (порой кажется, что их впору назвать «войнами»). Вместо этого основное внимание мы уделим тем свойствам кода, которые можно осмыслить в небольшом программном блоке, состоящем из одного класса с фиксированным API. Эффективность по времени и затратам памяти пройдет отбор вместе с надежностью, удобочитаемостью и универсальностью. Другие свойства, такие как удобство использования, из анализа будут исключены.

1.5. СПЕЦИФИЧНЫЕ СВОЙСТВА

Помимо свойств, описанных в предыдущих разделах, рассмотрим два свойства класса, которые формально не являются свойствами кода: потокобезопасность и лаконичность.

1.5.1. Потокобезопасность

Способность класса без проблем работать в многопоточной среде.

Потокобезопасность не является универсальным свойством кода, потому что она применима только в ограниченном контексте многопоточных программ. Тем не менее этот контекст встречается так часто, а проблемы синхронизации потоков настолько нетривиальны, что умение обращаться с базовыми примитивами синхронизации станет ценным навыком в инструментарии любого программиста.

Идея включить потокобезопасность в число внутренних свойств выглядит заманчиво, но будет ошибкой. В действительности от пользователя скрыто, последовательной или многопоточной является программа. В мире многопоточных программ потокобезопасность становится основным предусловием правильности, и свойство становится видимым. Кстати, ошибки, возникающие из-за дефектов потокобезопасности, очень трудно обнаружить из-за их внешней хаотичности и плохой воспроизводимости. Вот почему на рис. 1.2 потоковая безопасность находится в одной зоне с правильностью и стабильностью. Глава 8

посвящена обеспечению потокобезопасности и ловушкам параллельного выполнения.

1.5.2. Лаконичность

Написание самой короткой из возможных программ для решения имеющейся задачи.

Вообще, она не является целью: слишком лаконичная программа не внушает доверия. В приложении А есть занятное упражнение, в котором возможности языка доведены до предела, чтобы вы проверили свое владение Java или другим языком.

Тем не менее в некоторых практических сценариях компактность желательна. Низкопроизводительные встроенные системы (вроде смарт-карт в телефонах и кредитных картах) могут оснащаться самым малым объемом памяти, чтобы программа расходовала минимум памяти не только при выполнении, но и при хранении. В самом деле, большинство современных смарт-карт имеет всего 4 Кбайт оперативной памяти и 512 Кбайт долгосрочной памяти. В таких случаях играет роль даже простое количество инструкций, поэтому удобен компактный исходный код.

1.6. СКВОЗНОЙ ПРИМЕР: СИСТЕМА РЕЗЕРВУАРОВ ДЛЯ ВОДЫ

В этом разделе я представлю задачу программирования, которую мы будем многократно решать в книге, каждый раз ориентируясь на новое свойство кода. Сначала я опишу нужный API, за которым последует простой пример использования и предварительный вариант реализации.

Представьте, что вам нужно реализовать базовую инфраструктуру для новой социальной сети. Пользователи могут регистрироваться и, конечно, устанавливать дружеские связи. Эти связи симметричны (если я становлюсь вашим другом, то и вы автоматически становитесь моим, как на Facebook). Кроме того, у сети есть одна специфическая особенность: пользователь может отправить сообщение всем пользователям, с которыми он связан (прямо или косвенно). В книге я возьму важнейшие аспекты этого сценария и помещу их в упрощенный контекст, в котором не нужно беспокоиться о содержании сообщений или об атрибутах пользователей.

Вместо пользователей в задаче будет задействован набор резервуаров для воды. Все резервуары идентичны и имеют практически неограниченную емкость. В любой момент времени резервуар содержит некоторое количество жидкости,

и любые два резервуара могут быть соединены трубой. Вместо отправки сообщений мы будем доливать или отливать воду из резервуаров. Любые два и более соединенных резервуара сообщаются друг с другом, и с этого момента содержащаяся в них вода распределяется в равных пропорциях.

1.6.1. API

В этом разделе описан требуемый API для резервуаров с водой. Мы построим класс `Container` с открытым конструктором, который не получает аргументов и создает пустой резервуар, и тремя следующими методами:

- `public double getAmount()` — возвращает объем воды, содержащейся в резервуаре в настоящий момент;
- `public void connectTo(Container other)` — соединяет этот резервуар с другим;
- `public void addWater(double amount)` — добавляет `amount` единиц воды в резервуар. Этот метод автоматически распределяет в равных пропорциях воду между всеми резервуарами, соединенными (прямо или косвенно) с текущим.

  Метод можно использовать с отрицательным значением `amount`, чтобы отлить воду из резервуара. В этом случае группа соединенных резервуаров должна содержать достаточный объем воды для выполнения этого запроса — в резервуаре не должен остаться отрицательный объем воды.

Большинство реализаций, представленных в следующих главах, точно соответствуют этому API, кроме пары четко обозначенных исключений, когда настройка API помогает оптимизировать некоторое свойство кода.

Соединения между двумя резервуарами симметричны: вода может течь в обоих направлениях. Набор резервуаров, связанных симметричными соединениями, образует то, что называется в computer science *неориентированным, или ненаправленным, графом*. Во врезке приведена краткая информация о нем.

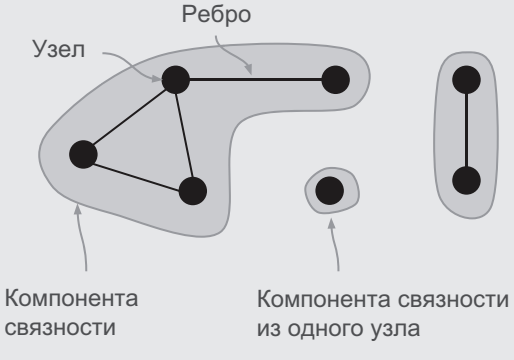
Чтобы реализовать `addWater` в сценарии с резервуарами, необходимо знать компоненты связности, потому что вода должна равномерно добавляться во все или сливаться из всех соединенных резервуаров. Алгоритмическая задача, лежащая в основе такого сценария, заключается в сохранении информации о компонентах связности при создании узла (нового объекта `Container`) и вставке ребер (методом `connectTo`) — разновидности *задачи о динамической связности графа*.

Такие проблемы занимают центральное место во многих приложениях, где используются сети. В социальных сетях компоненты связности представляют группы людей, связанных дружескими отношениями. При обработке графики

соединенные (смежные) области пикселов одного цвета помогают идентифицировать объекты в сцене. В компьютерных сетях обнаружение и хранение информации о связях между элементами — основа маршрутизации. В главе 9 рассмотрена область применения и ограничения для конкретной версии этой задачи.

НЕОРИЕНТИРОВАННЫЙ ГРАФ

В computer science термином *граф* обозначаются сети с попарно соединенными элементами. В этом контексте элементы также называются *узлами*, а соединения — *ребрами*. Если соединения симметричны, то граф называется *неориентированным*, потому что соединения не имеют определенного направления. Множество элементов, соединенных друг с другом (напрямую или косвенно), называется *компонентой связности*. В книге максимальная компонента связности называется *группой*.



1.6.2. Сценарий использования

В этом разделе представлен простой сценарий использования, в котором воплощен API из предыдущего раздела. Мы создадим четыре резервуара, нальем воду в два из них, а затем последовательно будем соединять их, пока они не образуют единую группу (рис. 1.3). В этом предварительном примере мы сначала наливаем воду, а затем соединяем резервуары. В общем случае эти операции могут свободно чередоваться. Более того, пользователь может в любой момент создать новые резервуары.

Я разделил сценарий использования (класс `UseCase` в сетевом репозитории https://bitbucket.org/mfaella/exercisesinstyle) на четыре части, чтобы в других главах вы могли легко найти конкретную точку и посмотреть, как разные реализации удовлетворяют одни и те же запросы. Эти четыре этапа изображены

на рис. 1.3. В первой части, соответствующей приведенному ниже фрагменту кода, мы посто создаем четыре резервуара. Изначально резервуары пусты и не соединены.

```
Container a = new Container();
Container b = new Container();
Container c = new Container();
Container d = new Container();
```

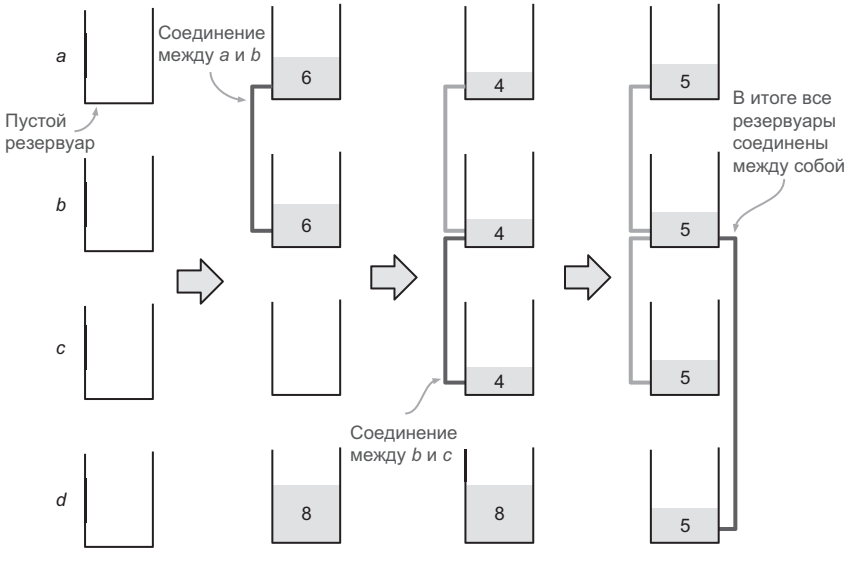


Рис. 1.3. Четыре этапа сценария использования: от четырех пустых изолированных резервуаров к одной группе соединенных резервуаров

Затем мы добавим воду в первый и последний резервуар и соединим их трубой. После этого выведем количество воды в каждом резервуаре, чтобы убедиться, что все работает в соответствии со спецификациями.

```
a.addWater(12);
d.addWater(8);
a.connectTo(b);
System.out.println(a.getAmount()+" "+b.getAmount()+" "+
                   c.getAmount()+" "+d.getAmount());
```

В конце предыдущего фрагмента резервуары `a` и `b` соединены, поэтому объем воды, залитой в `a`, распределяется по ним равномерно, тогда как резервуары `c` и `d` изолированы. Команда `println` должна выводить следующий результат:

```
6.0 6.0 0.0 8.0
```

А теперь соединим c с b и проверим, приведет ли создание нового соединения к автоматическому перераспределению воды между всеми соединенными резервуарами.

```
b.connectTo(c);
System.out.println(a.getAmount()+" "+b.getAmount()+" "+
                   c.getAmount()+" "+d.getAmount());
```

В этой точке резервуар c соединен с b и косвенно соединен с a. Теперь резервуары a, b и c являются сообщающимися сосудами, а общий объем содержащейся в них воды равномерно распределяется между ними. К резервуару d это не относится, поэтому результат выглядит так:

```
4.0 4.0 4.0 8.0
```

Обратите внимание на текущую точку сценария использования — вы встретите ее в следующих главах в стандартном сценарии и увидите, как одна ситуация представляется в памяти в разных реализациях.

Наконец, соединим d с b, чтобы все резервуары образовали одну связанную группу:

```
b.connectTo(d);
System.out.println(a.getAmount()+" "+b.getAmount()+" "+
                   c.getAmount()+" "+d.getAmount());
```

В финальном выводе уровень воды во всех резервуарах одинаков:

```
5.0 5.0 5.0 5.0
```

1.7. МОДЕЛЬ ДАННЫХ И ПРЕДСТАВЛЕНИЯ

Теперь нам известны требования к классу резервуара, и мы можем обратиться к проектированию фактической реализации. Спецификации фиксируют общедоступный API, поэтому на следующем шаге нужно разобраться, какие поля необходимы каждому объекту `Container`, а возможно, и самому классу (статические поля). Примеры в дальнейших главах показывают, что при проектировании возможно на удивление большое количество вариантов выбора полей в зависимости от того, на какое свойство мы ориентируемся. В этом разделе представлены некоторые общие наблюдения, применимые независимо от цели.

Прежде всего объекты должны включать информацию, достаточную для предоставления сервисов, требуемых спецификацией. Если этот базовый критерий выполнен, вам все равно придется принимать решения двух типов:

1. Хранить ли *дополнительную* информацию, даже если она не является абсолютно необходимой?

2. Как *закодировать* всю информацию, которую нужно сохранить? Какие типы данных или структуры подойдут? И какой объект(-ы) будет отвечать за нее?

Итак, хранение необязательной информации может объясняться двумя причинами. Во-первых, иногда вычисление информации требует бóльших затрат, чем ее хранение в готовом виде (что ускоряет выполнение кода). Представьте связанный список, который хранит свою длину в отдельном поле, даже если эта информация может быть вычислена «на ходу» сканированием списка и подсчетом количества узлов. Во-вторых, так резервируется место для будущих расширений (раздел 1.7.2).

Когда вы определили, какую информацию нужно хранить, переходите ко второму вопросу. Классы и объекты оборудуются полями соответствующих типов. Даже в простом сценарии с резервуарами этот шаг может быть весьма нетривиальным. Как показывает эта книга, есть несколько альтернативных решений, уместных для разных контекстов и целей.

В нашем сценарии информация, описывающая текущее состояние резервуара, состоит из двух аспектов: объем находящейся в нем воды и его соединения с другими резервуарами. Эти два аспекта рассматриваются в следующих двух разделах.

1.7.1. Хранение объема воды

Прежде всего, присутствие метода `getAmount` требует, чтобы резервуары «знали» о количестве хранящейся в них воды. Эта информация не обязательно должна храниться в резервуаре, достаточно дать резервуару возможность как-то получить значение и вернуть его. Кроме того, API требует, чтобы объем был представлен в формате `double`. Проще всего включить поля типа `double` в каждый резервуар. Но обратите внимание, что каждый резервуар в группе соединенных резервуаров содержит одинаковый объем воды. Возможно, будет лучше хранить информацию только в одном экземпляре — отдельном объекте, представляющем группу резервуаров. Тогда при вызове `addWater` будет достаточно обновить только один объект, даже если текущий резервуар соединен со множеством других.

Наконец, объем в группе можно хранить в специальном резервуаре, который выбирается как представитель группы. Таким образом, есть как минимум три подхода, которые выглядят разумно на данном этапе:

1. Каждый резервуар содержит актуальное поле для объема воды.
2. Отдельный объект-группа содержит поле для объема воды.
3. Всего один резервуар в каждой группе — *представитель* — содержит актуальное поле для объема воды, значение которого относится ко всем резервуарам в группе.

В следующих главах на разных реализациях мы рассмотрим эти и другие подходы. Также я подробно опишу плюсы и минусы каждого из них.

1.7.2. Хранение информации о соединениях

Вода, добавляемая в резервуар, должна равномерно распределяться между всеми резервуарами, с которыми он соединен (прямо или косвенно). Следовательно, каждый резервуар должен уметь находить все резервуары, соединенные с ним. При этом нужно принять важное решение: отличать ли прямые соединения от косвенных? Прямое соединение между `a` и `b` может быть создано только вызовом `a.connectTo(b)` или `b.connectTo(a)`, тогда как косвенные соединения возникают как следствия создания прямых соединений[1].

Выбор информации для хранения

Операции, которых требуют наши спецификации, не различают прямые и косвенные соединения, поэтому можно ограничиться хранением более общего типа: косвенных соединений. Но предположим, вы хотите добавить операцию `disconnectFrom` для отмены предыдущей операции `connectTo`. Если прямые соединения будут смешиваться с косвенными, правильно реализовать `disconnectFrom` не удастся.

Рассмотрите два сценария (рис. 1.4), где прямые соединения обозначены линиями между резервуарами. Если в памяти хранятся только косвенные соединения, то два сценария неразличимы: в обоих все резервуары соединены друг с другом. Таким образом, если одна и та же последовательность операций применяется в обоих сценариях, они должны реагировать абсолютно одинаково. С другой стороны, подумайте, что *должно* произойти, если клиент выполнит следующие операции:

```
a.disconnectFrom(b);
a.addWater(1);
```

Если эти две строки выполняются в первом сценарии (рис. 1.4, слева), то три резервуара будут взаимно соединенными, а значит, дополнительная вода должна равномерно распределиться между ними. И наоборот, во втором сценарии (рис. 1.4, справа) при отсоединении `a` от `b` резервуар `a` станет изолированным, поэтому дополнительная вода добавится только к `a`. Этот пример показывает, что хранение только косвенных соединений несовместимо с операцией `disconnectFrom`.

Итак, если вы ожидаете операцию `disconnectFrom`, храните прямые соединения явно и отдельно от косвенных. Но если у вас нет конкретной информации

[1] В математической терминологии косвенные соединения соответствуют *транзитивному замыканию* прямых соединений.

о перспективах развития программы, к таким соблазнам следует относиться с осторожностью. Известно, что программисты склонны к чрезмерным обобщениям и уделяют больше внимания гипотетическим преимуществам, а не связанным с ними затратам. Учтите, что затраты на дополнительные возможности не ограничиваются стадией разработки: каждый необязательный компонент класса нуждается в тестировании, документировании и сопровождении точно так же, как обязательный.

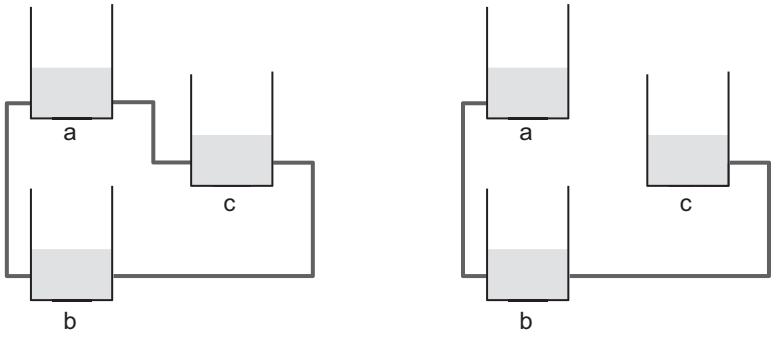


Рис. 1.4. Два сценария с тремя резервуарами. Линии между резервуарами обозначают прямые соединения

Кроме того, объем дополнительной информации, которую вы захотите включить, не ограничен. Что, если вы захотите удалить все соединения с возрастом более 1 часа? Нужно будет хранить время создания каждого соединения! А если вы захотите узнать количество потоков, создававших соединения? Нужно будет хранить набор всех потоков, которые создавали соединения, и т. д. В следующих главах я буду ограничиваться хранением только той информации, которая строго необходима для текущих целей[1], с несколькими четко выделенными исключениями.

Выбор представления

Наконец, если предположить, что вас устраивает хранение косвенных соединений, следующим шагом станет выбор фактического представления. Сначала он будет выбором между явным созданием нового класса (допустим, `Pipe`), представляющего соединение между двумя резервуарами, и хранением соответствующей информации непосредственно в объекте резервуара (*неявное* представление).

[1] В движении экстремального программирования этот принцип был формализован термином YAGNI («You aren't gonna need it», то есть «Вам это не понадобится»).

Первый вариант больше соответствует канонам ООП. В реальном мире резервуары соединяются трубами, а трубы представляют собой реальные объекты, отличающиеся от резервуаров. А значит, они должны моделироваться по отдельности. Но в спецификациях, представленных в этой главе, не упоминаются объекты труб `Pipe`, поэтому информация должна оставаться скрытой в резервуарах и неизвестной клиентам. Еще важнее, что такие объекты труб будут содержать минимум поведения: только две ссылки на соединяемые резервуары.

Создание лишнего класса не принесет особой пользы, обойдитесь без него. Резервуары должны иметь возможность добраться до компаньонов по группе без создания специализированного объекта — трубы. Но как организовать связывание соединенных резервуаров по ссылкам? Базовый язык и его API предоставляют много решений: массивы, списки, множества и т. д. Мы не будем их анализировать, потому что многие из них появятся в следующих главах (в частности, в главах 4 и 5).

1.8. HELLO, РЕЗЕРВУАРЫ! [NOVICE]

Чтобы сделать первый шаг, в этом разделе мы рассмотрим реализацию `Container`, которую мог бы написать неопытный программист, только что взявшийся за Java после работы на каком-нибудь структурированном языке (например, C). Этот класс станет началом длинной цепочки версий, приведенной в этой книге. Я присвоил каждой версии условное имя, чтобы вам было проще ориентироваться и сравнивать их. Этой версии присвоено имя Novice, а ее полное имя в репозитории — eis.chapter1.novice.Container.

1.8.1. Поля и конструктор

Даже опытные профессионалы когда-то были начинающими. Они разбирались в синтаксисе нового языка, не подозревая об огромных API, которые поджидали за углом. На первых порах массивы были основной структурой данных, а исправление ошибок в синтаксисе было настолько хлопотным, что о стиле программирования не беспокоились. После периода проб и ошибок начинающий программист сооружал класс, который успешно компилировался и вроде бы выполнял поставленные требования. Возможно, в исходном виде класс выглядел примерно так:

Листинг 1.1. Novice: поля и конструктор

```
public class Container {

   Container[] g;  ❶ Группа соединенных резервуаров
   int n;          ❷ Фактический размер группы
   double x;       ❸ Объем воды в резервуаре
```

```
public Container() {
   g = new Container[1000]; ❹ Внимание: волшебное число!
   g[0] = this;             ❺ Включить резервуар в группу
   n = 1;
   x = 0;
}
```

В этих строках содержится огромное количество мелких (и не очень) дефектов. Начнем с тех, которые лежат на поверхности и легко исправляются, а другие станут очевидными далее.

Три поля экземпляра предназначены для хранения следующей информации:

- `g` — массив всех резервуаров, соединенных с этим резервуаром (сам он тоже включается в группу, что следует из конструктора);
- `n` — количество резервуаров в `g`;
- `x` — объем воды в резервуаре.

Первая странность, которая немедленно выдает программиста-дилетанта, выбор имен переменных: они слишком короткие и совершенно неинформативные. Профессионал не назовет группу `g`, даже если мафия даст ему всего 60 секунд на взлом сверхзащищенной системы резервуаров для воды. А если серьезно, содержательные имена — первое правило удобочитаемого кода (глава 7).

Также есть проблема с видимостью: поля должны быть приватными. Вспомните, что *видимость по умолчанию* предоставляет более широкий доступ, нежели приватный: она допускает обращения из других классов, находящихся в том же пакете. Сокрытие информации (инкапсуляция) — фундаментальный ОО-принцип, позволяющий классам игнорировать внутреннее устройство других классов и взаимодействовать с ними через четко определенный открытый интерфейс (форма разделения ответственности). В свою очередь, это позволяет классам изменять свое внутреннее представление так, чтобы это не отражалось на существующих клиентах.

Принцип разделения ответственности также закладывает фундамент для всей книги. Многие реализации, представленные здесь, поддерживают тот же открытый API, то есть теоретически клиенты могут использовать любую версию по своему усмотрению. Способ, которым каждая реализация исполняет API, скрывается от внешнего мира благодаря спецификаторам видимости. На более глубоком уровне сама концепция оптимизации разных свойств кода может рассматриваться как экстремальное воплощение разделения ответственности — экстремальное до уровня простого дидактического инструмента, а не метода, применяемого на практике.

Далее, размер массива (шестая строка кода в листинге 1.1) определяется так называемым *волшебным числом* — константой без имени. Правила хорошего стиля требуют, чтобы каждая константа присваивалась `final`-переменной, чтобы (1) имя переменной документировало смысл константы и (2) значение

константы задавалось в одной точке, чтобы надежно использовать ее в нескольких местах.

Простой массив *по определению* ограничивает количество соединенных резервуаров: при слишком малой границе программа обречена на сбой, а при слишком большой неэффективно расходуется память. Более того, использование массива заставляет нас вручную отслеживать количество резервуаров, находящихся в группе (поле `n`). В Java API есть более эффективные варианты (глава 2), но простые массивы тоже пригодятся — в главе 5, где основной целью будет экономия памяти.

1.8.2. Методы getAmount и addWater

Проанализируем исходный код первых двух методов, приведенный в листинге 1.2.

Листинг 1.2. Novice: методы getAmount and addWater

```
public double getAmount() { return x; }

public void addWater(double x) {
   double y = x / n;
   for (int i=0; i<n; i++)
      g[i].x = g[i].x + y;
}
```

`getAmount` — тривиальный GET-метод, а `addWater` демонстрирует типичные проблемы с выбором имен переменных `x` и `y`, тогда как имя `i` приемлемо как традиционное имя для индекса массива. Если в последней строке листинга использовать оператор `+=`, то выражение `g[i].x` не будет повторяться дважды, а вам не придется переходить вперед-назад, чтобы убедиться в том, что команда действительно увеличивает ту же переменную.

Обратите внимание, что метод `addWater` не проверяет, является ли его аргумент отрицательным, и если является, хватит ли в группе воды для удовлетворения запроса. Подобные проблемы стабильности будут рассмотрены в главе 6.

1.8.3. Метод connectTo

Наконец, реализуем метод `connectTo`, задача которого — соединение двух групп резервуаров новой связью. После этой операции все резервуары в двух группах должны содержать одинаковый объем воды — стать сообщающимися сосудами. Сначала метод вычислит общий объем воды в группах и общий размер двух групп. Объем воды в резервуаре после слияния равен частному этих двух значений.

Также необходимо обновить массивы всех резервуаров в двух группах. Проще всего присоединить все резервуары второй группы ко всем массивам,

принадлежащим к первой группе, и наоборот. Это произойдет в листинге 1.3 с использованием двух вложенных циклов. Наконец, массив обновит поле размера `n` и поле объема `x` всех задействованных резервуаров.

Листинг 1.3. Novice: метод connectTo

```
public void connectTo(Container c) {
   double z = (x*n + c.x*c.n) / (n + c.n); ❶ Объем на резервуар после слияния
   for (int i=0; i<n; i++)                 ❷ Для каждого резервуара g[i] в первой группе
      for (int j=0; j<c.n; j++) {          ❸ Для каждого резервуара c.g[j] во второй группе
         g[i].g[n+j] = c.g[j];             ❹ Присоединение c.g[j] к группе g[i]
         c.g[j].g[c.n+i] = g[i];           ❺ Присоединение g[i] к группе c.g[j]
      }

   n += c.n;

   for (int i=0; i<n; i++) {               ❻ Обновление размеров и объемов
      g[i].n = n;
      g[i].x = z;
   }
}
```

Как видно из листинга, проблемы с именами проявляются наиболее серьезно именно в методе `connectTo`.

Из-за однобуквенных имен трудно понять, что происходит. Чтобы контраст был более разительным, перейдите немного вперед и взгляните на версию, оптимизированную для удобочитаемости (глава 7).

Удобочитаемость также может быть улучшена заменой трех циклов `for` расширенной конструкцией `for` (команда `foreach` в C#), но с представлением, основанным на массивах фиксированного размера, результат получится громоздким. Представьте, что последний цикл из листинга 1.3 был заменен следующим:

```
for (Container c: g) {
   c.n = n;
   c.x = z;
}
```

Безусловно, новый цикл лучше читается, но он будет аварийно завершен по исключению `NullPointer Exception`, как только переменная `c` выйдет за пределы ячеек, содержащих ссылку на резервуар. Проблема решается выходом из цикла сразу же при обнаружении `null`-ссылки:

```
for (Container c: g) {
   if (c==null) break;
   c.n = n;
   c.x = z;
}
```

Несмотря на полную неудобочитаемость, метод `connectTo` из листинга 1.3 является логически правильным с некоторыми ограничениями. Подумайте, что произойдет, если `this` и `c` уже соединены при вызове метода. Для конкретности предположим следующий сценарий использования с двумя новыми резервуарами:

```
a.connectTo(b);
a.connectTo(b);
```

Вы видите, что произойдет? Перенесет ли метод эту небольшую неосторожность от вызывающей стороны? Подумайте, прежде чем читать дальше. Я подожду...

Ответ: соединение двух уже соединенных резервуаров испортит их состояние. В итоге резервуар `a` будет хранить в своем массиве группы две ссылки на себя и две ссылки на `b`, а поле размера `n` будет равно 4 вместо 2. Нечто похожее происходит и с `b`. Что еще хуже, дефект проявится даже в том случае, если `this` и `c` соединены только *косвенно*, что не может считаться некорректным использованием от вызывающей стороны. Я говорю о сценарии следующего вида (и снова `a`, `b` и `c` — три новых резервуара):

```
a.connectTo(b);
b.connectTo(c);
c.connectTo(a);
```

Перед последней строкой резервуары `a` и `c` уже соединены, хотя и косвенно (рис. 1.4, справа). Последняя строка добавляет прямое соединение между ними, что вполне законно в соответствии со спецификациями. Возникает ситуация, изображенная на рис. 1.4 слева. Однако реализация `connectTo` из листинга 1.3 вместо этого добавляет вторую копию всех трех резервуаров во все массивы групп, ошибочно присваивая всем размерам групп значение 6 вместо 3.

Другое очевидное ограничение этой реализации заключается в том, что, если объединенная группа содержит более 1000 резервуаров (волшебное число), одна из следующих двух строк в листинге 1.3:

```
g[i].g[n+j] = c.g[j];
c.g[j].g[c.n+i] = g[i];
```

вызовет аварийное завершение нашей программы с выдачей исключения `ArrayIndexOutOfBoundsException`.

В следующей главе представлена эталонная реализация, которая решает бóльшую часть очевидных проблем, описанных выше, и при этом выдерживает баланс между разными свойствами кода.

ИТОГИ

- Свойства кода делятся на внутренние и внешние, а также на функциональные и нефункциональные.
- Некоторые свойства кода конфликтуют друг с другом, тогда как другие мирно сосуществуют.
- В книге программные свойства рассматриваются на сквозном примере с системой резервуаров для воды.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Эта книга пытается уместить на 300 страницах широкий диапазон тем, которые редко рассматриваются вместе. Это можно сделать только одним способом: весьма поверхностным изложением каждой темы. Из-за этого каждая глава завершается коротким списком источников информации, к которым можно обратиться за более глубоким изложением материала каждой главы.

- *Макконнелл С.* Совершенный код. — СПб.: БХВ, 2020. — 867 с.: ил.

  Полезная книга, посвященная стилю программирования и написанию хорошего во всех отношениях кода. Среди прочего, в ней обсуждаются свойства кода и их взаимодействие.

- *Spinellis D.* Code Quality: The Open Source Perspective. Addison Wesley, 2006.

  Автор знакомит читателя с атрибутами свойств, похожими на те, которые описаны в этой книге. При этом автор руководствуется почти противоположным принципом: вместо одного сквозного примера он приводит подборку фрагментов кода из разных популярных проектов с открытым кодом.

- *Kan S. H.* Metrics and Models in Software Quality Engineering. Addison Wesley, 2003.

  Кан систематично и глубоко рассматривает метрики кода, включая статистически обоснованные механизмы их числовой оценки, а также применение их для наблюдения и управления процессами разработки.

- *Davis C. W. H.* Agile Metrics in Action. Manning Publications, 2015.

  В главе 8 книги Дэвиса рассматриваются свойства кода и метрики для их оценки.

2 Эталонная реализация

В этой главе:

- ✓ Стандартные коллекции.
- ✓ Создание диаграмм для демонстрации структуры кода.
- ✓ Выражение скорости выполнения через *O*-большое.
- ✓ Оценка потребления памяти классом.

В этой главе рассмотрена версия класса `Container`, которая обеспечивает хороший баланс между разными свойствами, такими как чистота, эффективность и умеренное использование памяти.

Как говорилось в разделе 1.7, предполагается, что для системы резервуаров достаточно хранить и поддерживать набор косвенных соединений между резервуарами. На практике для этого с каждым резервуаром хранится ссылка на набор резервуаров, соединенных с ним прямо или косвенно (этот набор называется его *группой*). Вооружимся знаниями о JCF (Java collections framework) (см. врезку) и попробуем подобрать лучший класс для представления одной из таких групп.

JAVA COLLECTIONS FRAMEWORK

Многие стандартные коллекции появились в Java 1.2 и были серьезно переработаны в версии 1.5 (позднее переименованной в Java 5) для использования только что появившейся в языке поддержки обобщений. Полученный API — JCF — это одна из жемчужин экосистемы Java. JCF включает приблизительно 25 классов и интерфейсов и предоставляет такие стандартные структуры данных, как связанные списки, хеш-таблицы и сбалансированные деревья, а также средства синхронизации.

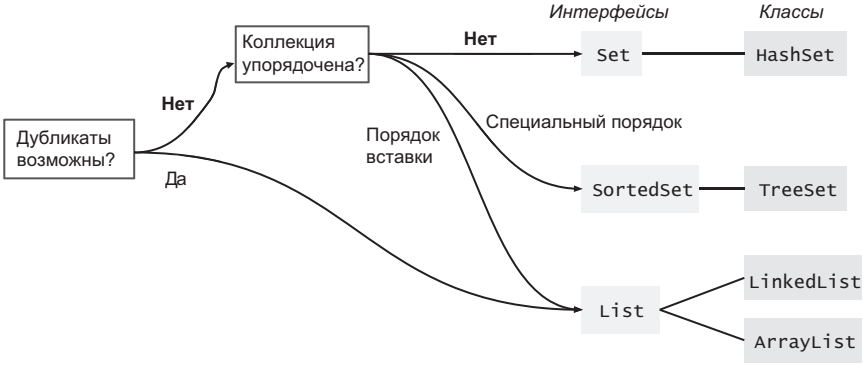


Рис. 2.1. Выбор интерфейса и класса Java для коллекции элементов

Выбирая тип для представления коллекции элементов, следует ответить на два вопроса: будет ли коллекция содержать дубликаты и важен ли порядок элементов? В нашем случае ответом на оба вопроса будет «нет». Другими словами, группы резервуаров функционируют как математические множества, соответствующие интерфейсу `Set` из JCF (рис. 2.1).

Затем выберем фактическую реализацию `Set` — класс, реализующий указанный интерфейс. Нет причин отказываться от самого распространенного и в общем случае самого эффективного варианта — `HashSet`.

НЕОЖИДАННЫЙ ВОПРОС 1

Какой интерфейс и класс коллекции вы бы выбрали для представления списка контактов в вашем телефоне?

КОЛЛЕКЦИИ C#

Иерархия коллекций C# несколько отличается от иерархии Java, но их наборы конкретных классов, экземпляры которых в конечном итоге будут создаваться в программе, очень похожи. Например, ниже перечислены ближайшие аналоги классов, упомянутых на рис. 2.1, в языке C#:

Java	C#
`HashSet`	`HashSet`
`TreeSet`	`SortedSet`
`ArrayList`	`List`
`LinkedList`	`LinkedList`

2.1. КОД [REFERENCE]

Спроектируем эталонную версию класса `Container`, начиная с полей и конструктора, и обозначим ее условным именем `Reference`. Опираясь на предыдущее обсуждение коллекций, создадим для каждого нового резервуара исходную группу в виде класса `HashSet`, состоящую только из этого резервуара.

> **ПРОГРАММИРОВАНИЕ В СООТВЕТСТВИИ С ИНТЕРФЕЙСОМ**
>
> Этим выражением обозначается общая концепция, согласно которой проектирование должно вестись вокруг API, а не вокруг конкретных реализаций. Сравните с методологией контрактного проектирования из главы 5. Объявление поля с самым общим типом интерфейса, который решает задачу, может рассматриваться как ограниченное применение этого принципа.

Следуя *программированию в соответствии с интерфейсом*, объявим поле группы с типом `Set`, а затем создадим его экземпляр в виде конкретного объекта `HashSet`. Считайте это сокрытием конкретного типа от остальных частей класса. Преимущество такого подхода заключается в том, что если позднее вы измените свое решение и замените конкретный тип `HashSet` другой реализацией `Set`, окружающий код изменять не придется, потому что он написан для интерфейса, а не для конкретной реализации.

Кроме того, каждый резервуар знает объем содержащейся в нем воды. Он кодируется значением `double`, которое неявно инициализируется нулем. Полученный код будет выглядеть примерно так:

Листинг 2.1. Reference: поля и конструктор

```
import java.util.*;                          ❶ В следующих листингах директивы import не приводятся
/* Резервуар для воды.                       ❷ Вместо этого произвольного комментария следует
 *                                             использовать Javadoc
 * by Marco Faella
 */
public class Container {

   private Set<Container> group;     ❸ Резервуары, соединенные с текущим
   private double amount;            ❹ Объем воды в резервуаре

   /* Создает пустой резервуар. */  ❺ Также должен быть комментарий Javadoc
   public Container() {
      group = new HashSet<Container>();
      group.add(this);               ❻ Группа начинается с этого резервуара
   }
```

Отметим, что, в отличие от `Novice`, в этой версии используется нормальная инкапсуляция и содержательные имена: поля объявлены приватными, а их имена информативны. Затем я намеренно прокомментировал код очень наивным способом, чтобы стиль комментариев лучше контрастировал с более принципиальным подходом, описанным в главе 7, где главной целью выступает удобочитаемость.

Прежде чем описывать разные методы, я представлю пару графических инструментов, которые пригодятся в следующих главах для наглядного сравнения разных версий резервуаров.

2.1.1. Диаграммы распределения памяти

Для каждой версии `Container`, использующей новый набор полей для представления данных, я буду приводить диаграмму *распределения памяти* — абстрактное представление того, как заданный набор резервуаров хранится в памяти. Это поможет вам осмыслить визуальную модель этого представления, а также сравнить разные версии. Для этого я всегда буду изображать одну ситуацию, а именно стандартный сценарий использования, описанный в главе 1, после выполнения первых трех частей. Напомню, что эти части создают четыре резервуара (от `a` до `d`) и выполняют следующие строки:

```
a.addWater(12);
d.addWater(8);
a.connectTo(b);
b.connectTo(c);
```

В этот момент три из четырех резервуаров объединены в группу, а четвертый изолирован (рис. 2.2). Диаграмма распределения памяти представляет собой упрощенную схему расположения объектов в памяти, напоминающую *диаграммы объектов* UML (см. следующий подраздел). Обе они отображают статические снимки набора объектов с указанием значений их полей и отношений между ними. В этой книге я предпочитаю собственный стиль диаграмм объектов, потому что он более интуитивен и хорошо адаптируется к излагаемой в каждом разделе точке зрения. На рис. 2.3 показано распределение памяти версии `Reference`

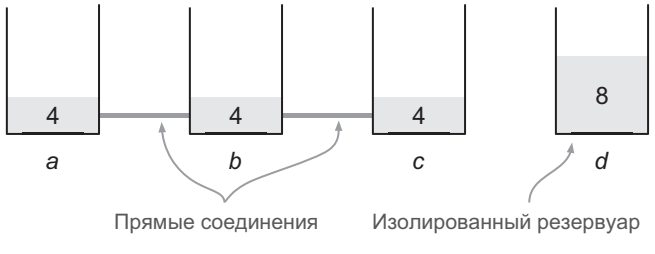


Рис. 2.2. Ситуация после выполнения первых трех частей сценария использования. Мы соединили резервуары от *a* до *c*, а затем налили воду в *a* и *d*

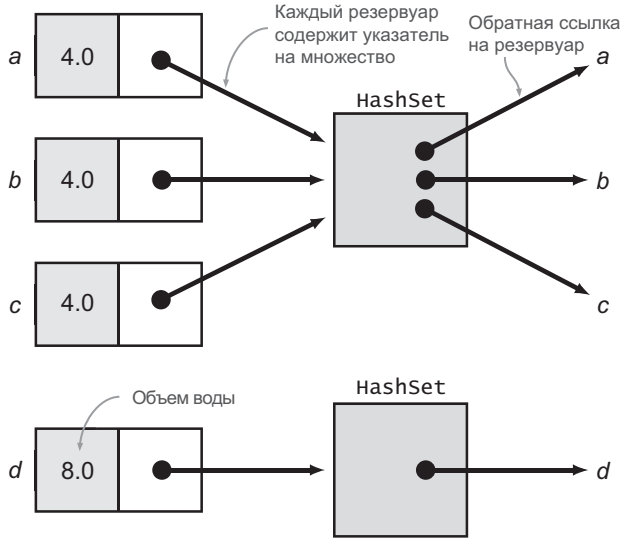


Рис. 2.3. Распределение памяти Reference после выполнения первых трех частей сценария использования. Чтобы не загромождать диаграмму, я обозначил ссылки от HashSet к резервуарам линиями, направленными к именам

после первых трех частей сценария использования. Как видите, я опустил многие низкоуровневые подробности (например, тип и ширину каждого поля в байтах) и скрыл внутреннюю структуру `HashSet`, потому что сейчас хочу сосредоточиться на том, в каком объекте хранится та или иная информация и какой объект содержит указатель на другой объект. Мы вернемся к распределению памяти `HashSet` в разделе 2.2.1.

Естественно, в работе вы будете чаще встречать стандартные диаграммы UML, поэтому я кратко напомню два основных типа диаграмм UML.

UML

Универсальный язык моделирования (UML, unified modeling language) — стандарт, предоставляющий широкий ассортимент диаграмм для описания различных аспектов программной системы. На практике чаще всего используются две части стандарта: диаграммы классов и диаграммы последовательности (глава 3).

Диаграмма классов UML

Диаграмма классов — описание статических свойств набора классов, в основном их взаимоотношений (таких, как наследование или включение). Только что

упоминавшиеся диаграммы объектов тесно связаны с диаграммами классов, но на них изображаются отдельные экземпляры этих классов.

Диаграмма классов для `Reference` показана на рис. 2.4. Блок `Container` комментариев не требует: в нем перечислены поля и методы, видимость которых обозначена как «плюс» (открытые) или «минус» (приватные). В блоке `HashSet` никакие поля или методы не указаны: для абстрактных или детализированных диаграмм это нормально.

Линия между двумя блоками называется *ассоциацией* и представляет отношение между двумя классами. В конце линии размещается описание роли каждого класса в ассоциации («участник» и «группа») и так называемое кардинальное число ассоциации. Последнее показывает, сколько экземпляров класса участвуют в отношении с одним экземпляром другого класса. В данном примере каждый резервуар принадлежит одной группе, а каждая группа включает одного или нескольких участников, что в UML обозначается символами `1..*`.

Хотя диаграмма классов на рис. 2.4 формально верна, для большинства практических целей она слишком детализирована. Диаграммы UML предназначены для описания *модели* системы, а не самой системы. Слишком детализированную диаграмму проще заменить исходным кодом. А значит, такие стандартные коллекции, как `HashSet`, обычно не упоминаются явно. Вместо этого они интерпретируются как одна из возможных реализаций ассоциаций между классами.

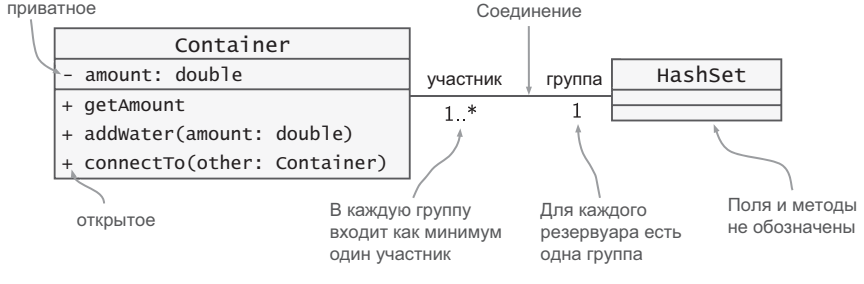


Рис. 2.4. Диаграмма классов UML для Reference (детализированная версия)

В данном случае можно заменить `HashSet` более абстрактной ассоциацией, связывающей класс `Container` с самим собой. Так вместо описания реализации вы передадите идею о том, что каждый резервуар может быть соединен с другими резервуарами в количестве от нуля и более (рис. 2.5).

НЕОЖИДАННЫЙ ВОПРОС 2

Используйте диаграмму классов для представления основных атрибутов классов Java, методов и их взаимоотношений.

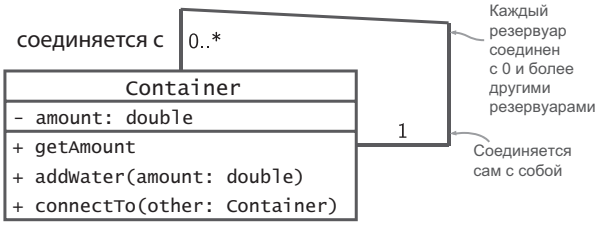


Рис. 2.5. Диаграмма классов UML для Reference (абстрактная версия)

Диаграмма объектов UML

Диаграммы объектов UML очень похожи на диаграммы классов. Чтобы объекты (то есть экземпляры классов) отличались от классов, их имена и типы подчеркиваются. Например, на рис. 2.6 изображена диаграмма объектов для `Reference` после выполнения первых частей сценария использования. Эта диаграмма соответствует абстрактной диаграмме классов на рис. 2.5, где класс `HashSet` не моделируется явно, а скрывается в ассоциации между резервуарами.

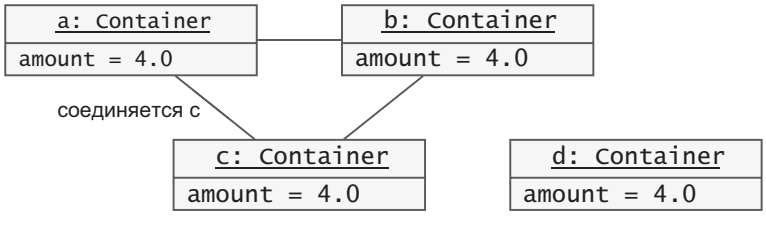


Рис. 2.6. Диаграмма объектов UML для Reference (абстрактная версия)

Из главы 3 вы узнаете еще об одном типе диаграмм UML: диаграмме последовательности, предназначенной для наглядного представления динамических взаимодействий между группами объектов.

2.1.2. Методы

Метод `getAmount` представляет собой тривиальный GET-метод, и не более того.

Листинг 2.2. Reference: метод getAmount

```
public double getAmount() { return amount; }
```

Затем можно переходить к методу `connectTo` (листинг 2.3[1]). Для начала заметим, что соединение двух резервуаров, по сути, означает слияние их двух групп. В результате метод изначально вычисляет общий объем воды в двух группах и объем воды в каждом резервуаре после слияния. Затем группа текущего резервуара изменяется с поглощением второй группы, а все резервуары второй группы включаются в новую большую группу. Наконец, объем воды в каждом резервуаре обновляется заранее вычисленной новой величиной.

Листинг 2.3 подробно прокомментирован для улучшения удобочитаемости. В современной практике этот метод был бы разбит на меньшие методы с информативными именами (глава 7).

Листинг 2.3. Reference: метод connectTo

```
public void connectTo(Container other) {

   // Если резервуары уже соединены, ничего не делать
   if (group==other.group) return;

   int size1 = group.size(),          ❶ Вычислить новый объем воды в каждом резервуаре
       size2 = other.group.size();
   double tot1 = amount * size1,
          tot2 = other.amount * size2,
          newAmount = (tot1 + tot2) / (size1 + size2);

   // Объединить две группы
   group.addAll(other.group);
   // Обновить группу резервуаров, соединенных с other   ❷ Эти комментарии можно
   for (Container c: other.group) { c.group = group; }     заменить вспомогательным
   // Обновить amount для всех                             методом с информативным
   // вновь присоединенных резервуаров                     именем
   for (Container c: group) { c.amount = newAmount; }
}
```

Метод `addWater` просто распределяет равные объемы воды по всем резервуарам в группе.

Листинг 2.4. Reference: метод addWater

```
public void addWater(double amount) {
   double amountPerContainer = amount / group.size();
   for (Container c: group) c.amount += amountPerContainer;
}
```

Как и в версии `Novice`, этот метод может получать отрицательные аргументы (обозначающие отлив воды) и не проверять, содержат ли резервуары достаточно воды для удовлетворения запроса. Возникнет риск отрицательного

[1] Здесь и далее я позволяю себе некоторые вольности в форматировании кода, например буду размещать цикл и его тело в одной строке, чтобы уместить листинг на одной странице.

объема воды в одном или нескольких резервуарах. (Проблемы стабильности обсуждаются в главе 5.) В следующих двух разделах мы проанализируем затраты памяти и времени для реализации, представленной в этом разделе, чтобы позднее сравнить их с показателями других версий.

2.2. ТРЕБОВАНИЯ К ПАМЯТИ

Несмотря на то что примитивные типы имеют фиксированный размер, оценка размера объекта Java в памяти — непростое дело. Существуют три фактора (связанные как с архитектурой, так и с разработчиком JDK), от которых зависит точный размер объекта:

- эталонный размер;
- заголовки объектов;
- заполнители в памяти.

Влияние этих факторов на размеры объектов определяется конкретной виртуальной машиной Java (JVM), используемой для запуска программы. Вспомните, что фреймворк Java базируется на двух официальных спецификациях: языка Java и виртуальной машины (VM). Разработчики могут реализовать собственный компилятор или виртуальную машину — на момент написания этой книги в Википедии перечислены 18 JVM, находящихся в активной разработке[1]. В следующих VM-зависимых обсуждениях я буду ссылаться на стандартную JVM компании Oracle, которая называется HotSpot.

Рассмотрим каждый из трех факторов более подробно. Во-первых, размер ссылки не фиксируется языком. На 32-разрядном оборудовании ее размер составляет 32 бита, но на современных 64-разрядных процессорах он может быть равен 32 или 64 битам благодаря *сжатию обычных указателей на объекты* (OOPs). Оно позволяет программе хранить ссылки в виде 32-разрядных значений, даже когда оборудование поддерживает 64-разрядные адреса, за счет адресации «всего» 32 Гб общего доступного пространства кучи. В следующих оценках затрат памяти предполагается фиксированный размер ссылки, составляющий 32 бита.

Во-вторых, все объекты в памяти начинаются с заголовка с некоторой стандартной информацией, необходимой для JVM. Как следствие, даже объект без полей (объект без *состояния*) занимает некоторую память. Конкретная структура заголовка объекта выходит за рамки книги[2], но знайте, что за него отвечают три возможности языка Java: отражение, многопоточность и сбор мусора.

---

[1] См. http://mng.bz/zlm6.

[2] Подробнее в исходном коде HotSpot (https://hg.openjdk.java.net/jdk10/jdk10/hotspot). Содержимое заголовков объектов описано в файле src/share/vm/oops/markOop.hpp.

СЖАТЫЕ OOPS

В сжатых OOPs неявно добавляются три нуля в конец каждого 32-разрядного адреса, поэтому хранимый адрес (к примеру) `0x1` будет интерпретирован как машинный адрес `0x1000`. В этом виде машинные адреса фактически занимают 35 бит, а программа может обращаться к объему до 32 Гб пространства. JVM также приходится принимать специальные меры для выравнивания всех переменных по 8-байтовым границам, потому что программа может обращаться только по адресам, кратным 8.

Подведем итог: эта технология экономит память для каждой ссылки, но увеличивает заполнители и провоцирует лишние временные затраты на преобразование хранимых адресов в машинные (быстрая операция сдвига влево). Сжатые OOPs включены по умолчанию, но автоматически отключаются, если вы сообщите виртуальной машине, что собираетесь использовать более 32 Гб памяти (при помощи ключей командной строки `-Xms` и `-Xmx`).

1. Отражение (reflection) требует, чтобы объекты знали свой тип. Следовательно, каждый объект должен хранить ссылку на свой класс или числовой идентификатор в таблице загруженных классов. Этот механизм позволяет оператору `instanceof` проверить динамический тип объекта, а методу `getClass` класса `Object` — вернуть ссылку на динамический класс объекта.

   Обратите внимание, что массивы должны хранить тип своих элементов, потому что каждая операция записи в массив проходит проверку типа во время выполнения (и может выдать исключение `ArrayStoreException`). Тем не менее эта информация *не* увеличивает затраты памяти массива, потому что она является частью информации типа и может совместно использоваться всеми массивами одного типа. Например, все массивы строк указывают на один объект `Class`, который представляет тип «массив строк».

2. При поддержке многопоточности с каждым объектом связывается *монитор* (для обращения к которому используется ключевое слово `synchronized`). Это значит, что заголовок должен содержать ссылку на объект монитора. Современные виртуальные машины создают монитор только при необходимости, когда несколько потоков конкурируют за монопольный доступ к такому объекту.

3. Механизм сбора мусора требует, чтобы для каждого объекта хранилась некоторая служебная информация, например *счетчик ссылок*. Современные алгоритмы сбора мусора назначают объектам разные *поколения* в зависимости от того, когда они были созданы. В этом случае заголовок будет содержать поле *возраст*.

В этой книге предполагается фиксированный объем заголовка — 12 байт на объект, типичный для современных 64-разрядных JVM. Кроме стандартного заголовка объекта, массивы хранят свою длину, что приводит к 16-байтовому общему объему служебных данных (даже пустой массив занимает 16 байт).

Наконец, аппаратные архитектуры стремятся, чтобы данные были выровнены по некоторым границам для более эффективной работы. Например, при обращениях к памяти используются адреса, кратные степени двойки (обычно 4 или 8). Это обстоятельство заставляет компиляторы и виртуальные машины применять *заполнение*: данные объекта в памяти дополняются пустым пространством для выравнивания полей, чтобы весь объект точно укладывался в целое количество слов. В книге мы будем игнорировать проблемы заполнения, зависящие от архитектуры.

РАЗМЕР ОБЪЕКТОВ В C#

Ситуация в C# очень похожа на то, что я описывал здесь для Java, и причины для лишних затрат памяти точно те же: для 32-разрядных архитектур используются 12-байтовые заголовки, а для 64-разрядных — 16-байтовые.

2.2.1. Требования к памяти в Reference

Теперь обратимся к фактическому распределению памяти в реализации `Reference`. Прежде всего, каждый объект `Container` содержит следующее:

- 12 байт служебной информации;
- 8 байт для поля `amount` (тип `double`);
- 4 байта для ссылки на множество, к которым прибавлен размер самого множества.

Чтобы оценить затраты памяти для `HashSet`, заглянем «за кулисы» и обратимся к реализации. Обычно `HashSet` реализуется с использованием массива связанных списков (называемых *гнездами*) и парой дополнительных полей для хранения служебной информации. В идеале все элементы распределяются по разным гнездам, а количество гнезд точно соответствует количеству элементов. Не углубляясь в подробности[1], скажу, что тогда минимальная реализация `HashSet` занимает приблизительно 52 байта. Каждый элемент множества требует одну ссылку (на его список) и список с одним элементом (приблизительно еще 32 байта). Я использую термин «минимальная реализация» вместо «пустое множество», потому что пустое множество `HashSet` в исходном состоянии имеет ненулевую исходную емкость (16 гнезд в текущем OpenJDK), но проще и логичнее списать эти затраты на первые элементы, которые будут вставлены. На рис. 2.7 подробно изображено внутреннее строение объектов с разбиением требований к памяти.

[1] Фактическая реализация `HashSet` использует `HashMap`, что усложняет анализ.

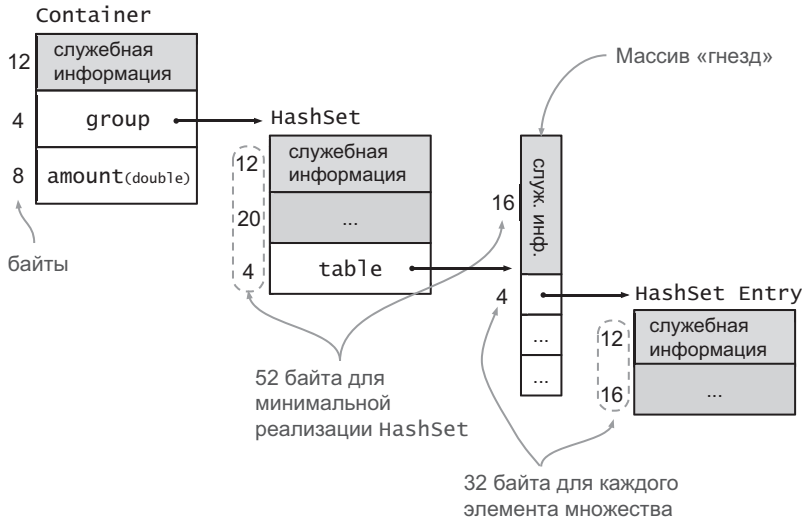


Рис. 2.7. Подробная схема распределения памяти для объекта резервуара в эталонной версии Reference. Оценки для HashSet предполагают идеальный размер гнезд и наличие идеальной функции хеширования, при которых на каждое гнездо приходится ровно один элемент

ОПРЕДЕЛЕНИЕ РАЗМЕРА ОБЪЕКТА

В JDK входит программа JOL (java object layout), которая анализирует внутреннее распределение памяти заданного класса, включая заголовок объекта. Она доступна по адресу http://openjdk.java.net/projects/code-tools/jol/.

НЕОЖИДАННЫЙ ВОПРОС 3

Класс `android.graphics.Rect` содержит четыре открытых поля типа `int`. Сколько байтов занимает объект `Rect`?

Чтобы получить фактические значения и упростить сравнения с другими реализациями, мы оценим затраты памяти для двух гипотетических сценариев: в одном используется 1000 изолированных резервуаров, а в другом — 1000 резервуаров, объединенных в 100 групп по 10 резервуаров в каждой. Для этих двух вариантов эталонной реализации будут получены результаты (табл. 2.1). Что вы скажете об этих числах: хороши они или плохи? 100 байт — не слишком ли много для изолированного резервуара?

Таблица 2.1. Требования к памяти Reference в двух условных сценариях

Сценарий	Размер (вычисления)	Размер (байты)
1000 изолированных резервуаров	1000 × (12 + 8 + 4 + 52 + 32)	108 000
100 групп по 10	1000 × (12 + 8 + 4) + 100 × (52 + 10 × 32)	61 200

Можно ли добиться чего-то большего? Трудно оценивать эти числа сами по себе. В следующих главах мы разработаем ряд альтернативных реализаций, и тогда вы сможете сравнить их затраты памяти и аргументированно ответить на предыдущие вопросы. (Внимание, спойлер: самая компактная версия (глава 4) требует всего 4 Кб для обоих сценариев, но не соответствует установленному API.)

2.3. ВРЕМЕННАЯ СЛОЖНОСТЬ

При оценке затрат памяти программы стандартной единицы измерения можно выбрать байты. Если отвлечься от низкоуровневых подробностей, рассмотренных в предыдущем разделе, как правило, Java-программа занимает одинаковый объем памяти при запуске на любом компьютере.

Измерить временны́е характеристики сложнее. На разных компьютерах одна программа может выполняться с совершенно разной продолжительностью. Вместо того чтобы измерять фактическое время выполнения, можно подсчитать количество *элементарных действий*, выполняемых программой. Упрощенно говоря, элементарным действием можно считать любую операцию (например, арифметическую или сравнение), выполняемую за постоянное время[1].

Другая проблема заключается в том, что одна функция может выполнять разное количество элементарных действий для разных входных данных. Для примера возьмем метод `connectTo` из листинга 2.3. На входе он получает два резервуара:

- Единственное *явное* входное значение — параметр `other` типа `Container`.
- Как метод экземпляра он также получает *неявный* параметр, так что текущий резервуар тоже фактически является входным значением.

Метод содержит два цикла `for`, длины (количества итераций) которых зависят от размера двух объединяемых групп резервуаров, то есть являются функцией входных данных.

В таких случаях одним или несколькими числовыми параметрами характеризуются аспекты входных данных, влияющие на время выполнения алгоритма.

[1] Формальное определение элементарного действия должно быть основано на формальной модели вычислений и звучать, например, как «любая операция, требующая постоянного количества тактов машины Тьюринга».

Происходит числовая оценка размера входных данных. Если количество элементарных действий алгоритма изменяется даже для входных данных одинакового размера, стоит рассматривать худший случай — максимальное число элементарных действий, выполняемых для любого ввода заданного размера.

Возвращаясь к методу `connectTo`, для начала рассмотрим два параметра: `size1` и `size2` — размеры двух объединяемых групп резервуаров. Используя эти параметры, можно проанализировать метод `connectTo` так:

Листинг 2.5. Reference: метод connectTo (комментарии опущены)

```
public void connectTo(Container other) {

   if (group==other.group) return;     ❶ одно действие

   int size1 = group.size(),
       size2 = other.group.size();
   double tot1 = amount * size1,                                  ❷ пять действий
          tot2 = other.amount * size2,
          newAmount = (tot1 + tot2) / (size1 + size2);

   group.addAll(other.group);                                     ❸ size2 действий
   for (Container c: other.group) { c.group = group; }            ❹ size2 действий
   for (Container c: group) { c.amount = newAmount; }             ❺ size1 + size2 действий
}
```

Я считаю за одно действие все, что не подразумевает цикла, имеет постоянное время выполнения и не зависит от параметров `size1` и `size2`. Я также опускаю многие подробности, когда помечаю строку `group.addAll` комментарием «size2 действий». В двух словах, эта оценка представляет ожидаемое количество действий, при условии что метод `hashCode` равномерно распределяет объекты по всему множеству представимых целых чисел.

ПРИМЕЧАНИЕ

Подробнее о хеш-таблицах и скорости их выполнения читайте в книгах по структурам данных (одна из них упомянута в конце этой главы).

Согласно этим рассуждениям, общее количество элементарных действий, выполняемых `connectTo`, равно

$$6 + 2 \times \texttt{size2} + (\texttt{size1} + \texttt{size2}) = 6 + \texttt{size1} + 3 \times \texttt{size2} \qquad (*)$$

Впрочем, следует признать, что число 6 в этом выражении выбрано несколько произвольно. Если посчитать строки ассемблера вместо строк Java, можно получить 6000 вместо 6, а если такты машины Тьюринга — 6 миллионов. По той же причине множитель 3 перед `size2` тоже выбран фактически произвольно. Иначе говоря, константы 3 и 6 зависят от *детализации*, выбранной для элементарных действий.

Можно иначе подсчитать действия и элегантно обойти проблему детализации — сосредоточиться на быстроте роста количества действий при увеличении параметров размера. Эта характеристика, называемая *порядком роста*, является одним из основных принципов теории сложности — раздела computer science. Порядок роста освобождает вас от выбора конкретной детализации элементарных действий и предоставляет оценки эффективности более абстрактные, но проще сопоставимые. В то же время порядок роста сохраняет *асимптотическое* поведение функции, то есть тенденцию ее поведения при больших значениях параметра(-ов).

На практике для обозначения порядка роста часто применяется запись *O-большое*. Например, выражение элементарных действий (*) принимает вид $O(size1 + size2)$, фактически скрывая все произвольные константы-слагаемые и множители. Запись подчеркивает, что количество действий линейно пропорционально `size1` и `size2`. Если говорить точнее, *O*-большое устанавливает *верхнюю границу* роста функции: выражение $O(size1 + size2)$ означает, что время выполнения растет *не более* чем в линейной зависимости от `size1` и `size2`.

Метод `connectTo` прост: он всегда выполняет равное количество действий для заданных значений `size1` и `size2`. Другие алгоритмы менее постоянны — скорость их выполнения зависит от характеристик ввода, не выражаемых параметрами размера. Например, при поиске заданного значения в неупорядоченном массиве значение может быть найдено немедленно (постоянное время) или не найдено посредством перебора всего массива (линейное время). Анализ сложности подразумевает работу со входными данными, требующими наибольшего количества действий (*наихудший случай*). Итак, стандартный способ оценки скорости выполнения алгоритмов называется *асимптотической сложностью в наихудшем случае* (или пессимистической асимптотической сложностью), и в неупорядоченном массиве такая сложность поиска обозначается $O(n)$. В табл. 2.2 представлены некоторые границы *O*-большого, их названия и примеры алгоритмов, характеризуемых этими границами. Для алгоритмов, работающих с массивами, параметром n обозначается размер массива.

Прежде чем углубляться в асимптотическую запись, немного упростим анализ `connectTo`, переключившись с двух параметров размера на один. Если обозначить через `n` общее количество созданных резервуаров, то значение `size1+size2` не превысит `n` (разные группы разъединены по определению). Так как верхняя граница $O(size1 + size2)$ продолжает действовать для нашей функции, будет действовать и граница $O(n)$, превышающая первую верхнюю границу. То есть время, требуемое для выполнения операции `connectTo`, растет максимум в линейной зависимости от общего количества резервуаров. Такая аппроксимация может показаться довольно грубой, и это действительно так. В конце концов, значения `size1` и `size2` с большой вероятностью будут намного меньше `n`. Тем не менее при всей приблизительности такой тип верхней границы будет достаточно точным, чтобы различать эффективность разных реализаций, представленных в следующих главах.

Таблица 2.2. Типичные границы сложности в записи через O-большое

Обозначение	Название	Пример
$O(1)$	Постоянное время	Проверка, равен ли первый элемент массива нулю
$O(\log n)$	Логарифмическое время	Бинарный поиск — умный способ поиска конкретного значения в отсортированном массиве
$O(n)$	Линейное время	Поиск максимального значения в несортированном массиве
$O(n \log n)$	Квазилинейное время	Сортировка массива методом сортировки слиянием
$O(n^2)$	Квадратичное время	Сортировка массива методом пузырьковой сортировки

НЕОЖИДАННЫЙ ВОПРОС 4

Имеется неупорядоченный массив целых чисел. С какой сложностью будет выполняться проверка того, является ли содержимое массива палиндромом?

ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ ЗАПИСИ ЧЕРЕЗ O-БОЛЬШОЕ

Когда кто-то говорит, что алгоритм характеризуется сложностью $O(f(n))$ для функции f, это означает, что $f(n)$ определяет верхнюю границу для количества элементарных действий, выполняемых алгоритмом для любых входных данных с размером n. И это логично, если оценивать размер входных данных одним параметром n.

Выражаясь более формально, запись через O-большое можно применить к любой функции $f(n)$, представляющей количество элементарных действий, необходимых алгоритму при выполнении для входных данных с размером n. Пусть функция $g(n)$ определяет фактическое количество «действий» — как бы они ни определялись, — выполняемых алгоритмом для входных данных с размером n. Следовательно, когда вы пишете, что алгоритм имеет временную сложность $O(f(n))$, это означает, что существуют два числа m и c, с которыми для всех $n \geq m$

$$g(n) \leq c \times f(n).$$

Другими словами, для достаточно больших входных данных фактическое количество действий заведомо не превышает значение функции f с некоторым постоянным множителем.

В теории сложности существуют и другие обозначения, определяющие только нижние границы или верхние и нижние границы одновременно и т. д.

2.3.1. Временная сложность в Reference

Теперь мы можем точно указать временну́ю сложность всех методов реализации `Reference`. Метод `getAmount` — простой GET-метод, требующий постоянного времени. Методам `connectTo` и `addWater` приходится перебирать все резервуары в группе. Так как размер группы может достигать всего множества всех резервуаров, их сложность в худшем случае находится в линейной зависимости от общего количества резервуаров n (табл. 2.3). Из главы 3 вы узнаете, как улучшить эти показатели временной сложности.

Таблица 2.3. Временные сложности для Reference, где n обозначает общее количество резервуаров

Метод	Временная сложность
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$O(n)$

2.4. ПРИМЕНИМ ИЗУЧЕННОЕ

В этом разделе (и в аналогичных разделах следующих глав) концепции, описанные в текущей главе, применяются в разных контекстах. Сквозной пример в книге связывает разные темы, поэтому для понимания важно прочитать код, тщательно разобраться в упражнениях и попытаться решить их самостоятельно. Если у вас нет времени или желания, по крайней мере *почитайте упражнения и их решения*. Надеюсь, вы обнаружите, что все решения тщательно объяснены и помогают глубже понять материал главы и исследовать различные классы из JDK и других библиотек.

Упражнение 1

1. Оцените сложность следующего метода:

```
public static int[][] identityMatrix(int n) {
   int[][] result = new int[n][n];
   for (int i=0; i<n; i++) {
      for (int j=0; j<n; j++) {
         if (i==j) {
            result[i][j] = 1;
         }
      }
   }
   return result;
}
```

2. Можно ли сделать его более эффективным без изменения результатов?

3. Если вы смогли предложить более эффективную версию, будет ли она обладать меньшей сложностью?

Упражнение 2

Класс `java.util.LinkedList<T>` реализует двусвязный список ссылок на объекты типа `T`. Просмотрите его исходный код[1] и оцените размер в байтах `LinkedList` с *n* элементами (без учета памяти, занятой `n` объектами).

Упражнение 3 (мини-проект)

Реализуйте класс `User`, представляющий пользователя в социальной сети, со следующей функциональностью:

- У каждого пользователя есть имя. Реализуйте открытый конструктор, получающий это имя.
- Пользователи могут устанавливать дружеские отношения при помощи следующего метода:

  ```
  public void befriend(User other)
  ```

  Дружеские отношения симметричны: вызов `a.befriend(b)` эквивалентен `b.befriend(a)`.
- Клиенты могут проверить, являются ли два пользователя прямыми или косвенными друзьями (друзья друзей), при помощи следующих двух методов:

  ```
  public boolean isDirectFriendOf(User other)
  public boolean isIndirectFriendOf(User other)
  ```

ИТОГИ

- Для наглядного представления структуры и поведения кода используются статические и динамические диаграммы (такие, как диаграммы объектов UML и диаграммы последовательности).
- Пустой объект Java занимает 12 байт памяти из-за заголовка. Асимптотическая сложность измеряет эффективность по времени способом, не зависящим от оборудования.
- Запись через *O*-большое — наиболее распространенный способ выражения асимптотической верхней границы временной сложности.

[1] На момент написания книги он доступен по адресу http://mng.bz/KElg.

ОТВЕТЫ НА ВОПРОСЫ И УПРАЖНЕНИЯ

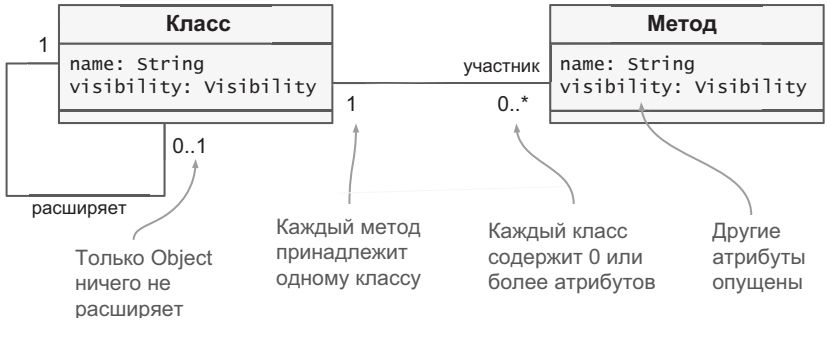
Неожиданный вопрос 1

Предположим, данные контакта состоят из имени и номера телефона. Обычно вы обращаетесь по имени к списку контактов, упорядоченному в алфавитном порядке. Это нестандартный порядок, основанный на содержимом объекта, поэтому, несмотря на имя («list»), список контактов лучше представляется интерфейсом `SortedSet`, стандартной реализацией которого является класс `TreeSet`.

На реальном телефоне контакт представляет собой более сложную сущность с множеством атрибутов, которые связываются между собой разными приложениями. А значит, контакты с большой вероятностью будут храниться в той или иной базе данных (например, Android использует SQLite).

Неожиданный вопрос 2

Диаграмма классов, представляющая классы и методы Java:



Неожиданный вопрос 3

Объект `android.graphics.Rect` занимает 28 байт: 12 байт для служебной информации и 4 × 4 байта для четырех целочисленных полей. Как принято в этой книге, эта оценка игнорирует выравнивание в памяти, из-за которого фактический размер увеличивается до следующего значения, кратного 8, то есть 32.

Неожиданный вопрос 4

Чтобы проверить, является ли массив четной длины `n` палиндромом, необходимо проверить, равны ли элементы `a[i]` и `a[n-1-i]` для всех `i` от `0` до `n/2`. Это `n/2` итераций, что соответствует порядку роста $O(n)$. (Постоянный множитель `1/2` не важен для асимптотической записи.)

Упражнение 1

1. Метод обладает сложностью $O(n^2)$, то есть квадратичной сложностью.
2. Более эффективная версия без вложенного цикла с командой `if`:

```
public static int[][] identityMatrix(int n) {
   int[][] result = new int[n][n]; ❶ Матрица инициализируется нулями
   for (int i=0; i<n; i++) {
      result[i][i] = 1;
   }
   return result;
}
```

3. Сложность новой версии остается квадратичной из-за выделения памяти во второй строке, которая неявно инициализирует все n^2 ячеек нулями.

Упражнение 2

Соответствующие строки из исходного кода `LinkedList`:

```
public class LinkedList<E> extends AbstractSequentialList<E> ... {
   transient int size = 0;
   transient Node<E> first;
   transient Node<E> last;

   ...
   private static class Node<E> {
      E item;
      Node<E> next;
      Node<E> prev;
      ...
   }
}
```

При беглом просмотре кода обнаруживаются суперклассы `AbstractSequentialList`, `AbstractList` и `AbstractCollection` в указанном порядке. Из них только `AbstractList` содержит поле экземпляра, используемое для поиска параллельных изменений в списке во время итерации:

```
protected transient int modCount = 0;
```

С учетом сказанного, список `LinkedList` с `n` элементами занимает:

- 12 байт служебной информации;
- 3 × 4 байта для трех полей `size`, `first` и `last`;
- 4 байта для унаследованного поля `modCount`.

Кроме того, для каждого элемента затраты составляют:

- 12 байт служебной информации;
- 3 × 4 байта для трех полей `item`, `next` и `prev`.

Итак, `LinkedList` с `n` элементами занимает $28 + n \times 24$ байта.

Упражнение 3

Спецификации отчасти напоминают сценарий с резервуарами, если не считать того, что прямые соединения (то есть дружеские отношения) отличаются от косвенных. Одно из возможных решений — явное хранение прямых соединений и вычисление косвенных соединений по мере необходимости. Тогда начальная версия класса может выглядеть так:

```
public class User {
   private String name;
   private Set<User> directFriends = new HashSet<>();

   public User(String name) {
      this.name = name;
   }

   public void befriend(User other) {
      directFriends.add(other);
      other.directFriends.add(this);
   }

   public boolean isDirectFriendOf(User other) {
      return directFriends.contains(other);
   }
```

Проверка неявных соединений требует обхода (неориентированного) графа. Простейший алгоритм такого рода — алгоритм *поиска в ширину* (BFS, breadth-first search), который поддерживает два множества узлов:

- *рубеж* (frontier) узлов, ожидающих посещения;
- множество уже *посещенных* (закрытых) узлов.

Возможная реализация BFS:

```
public boolean isIndirectFriendOf(User other) {
   Set<User> visited = new HashSet<>();
   Deque<User> frontier = new LinkedList<>();

   frontier.add(this);
   while (!frontier.isEmpty()) {
      User user = frontier.removeFirst();
      if (user.equals(other)) {
```

```
            return true;
        }
        if (visited.add(user)) {                    ❶ Если узел не посещался
            frontier.addAll(user.directFriends); ❷ addAll вставляет с конца
        }
    }
    return false;
}
```

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

На тему Java-программирования написаны тысячи книг вводного уровня. Мои любимые книги:

- *Хорстманн К. С.* Java. — 11-е изд. — М.; СПб.: Диалектика, 2020.

  Книга-монстр с подробными описаниями многих частей API, в основном предназначенная для преподавателей.

- *Sestoft P.* Java Precisely. MIT Press, 3rd edition, 2016.

  Не столько учебник вводного уровня, сколько компактный и подробный справочник по языку и ограниченному подмножеству API (включая коллекции и потоки Java 8).

Что касается временно́й сложности и записи *O*-большое, в любой вводной книге по алгоритмам приводится исчерпывающее описание этой темы. Классический вариант:

- *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ. — 3-е изд. — М.; СПб.: Диалектика, 2020.

И, наконец, книги о UML и сопутствующих приемах разработки:

- *Фаулер М.* UML. Основы. 3-е изд. / Пер. с англ. — СПб.: Символ Плюс, 2004. — 192 с., ил.

  Всего на 200 страницах приведено серьезное введение в синтаксис UML, при этом особое внимание уделено диаграммам классов и последовательностей.

- *Ларман К.* Применение UML 2.0 и шаблонов проектирования: введение в объектно-ориентированный анализ, проектирование и интеративную разработку. — 3-е изд. — М. [и др.]: Вильямс, 2013.

  Этот том, значительно превосходящий книгу Фаулера по количеству страниц и набору тем, выходит за рамки UML и предоставляет систематическое введение в ОО-анализ и проектирование. Второе издание на английском языке можно загрузить бесплатно.

Часть II
Свойства программного кода

В этой части мы займемся различными свойствами кода и их оптимизацией. В главе 3 мы рассмотрим эффективность — как по времени, так и по затратам памяти. Алгоритмы и структуры данных станут средствами достижения этой цели.

В главах 5 и 6 центральное место займет надежность и будут использованы такие приемы, как контрактное проектирование и тестирование. В главе 7 я порекомендую приемы написания удобочитаемого кода. Наконец, в главах 8 и 9 мы займемся расширенными аспектами программирования, относящимися к потокобезопасности и возможности повторного использования.

3 Жажда скорости: эффективность по времени

В этой главе:

- ✓ Сравнение скорости выполнения основных структур данных, включая списки, множества и деревья.
- ✓ Оценка скорости выполнения в худшем случае и средней скорости выполнения структуры данных в долгосрочной перспективе.
- ✓ Концентрация вычислительной нагрузки в конкретном методе класса или ее распределение по всем методам.

Достижение максимально возможной скорости выполнения задачи привлекало программистов еще с доисторических времен программирования на перфокартах. В самом деле, можно сказать, что почти вся computer science подчинена удовлетворению этой потребности. В этой главе я представлю три реализации резервуара, оптимизирующие скорость разными способами. Почему три? Разве нельзя привести одну *лучшую*? Дело в том, что одной лучшей версии не существует — это один из главных выводов этой главы.

Базовые учебные курсы программирования и даже вводные курсы computer science упускают этот факт из виду. В последних широко рассматриваются вопросы эффективности по времени, особенно в алгоритмах и классах структур данных. В этих учебных курсах и учебниках рассматривается одна задача за раз, будь то обход графа или балансировка дерева. Если взять одну алгоритмическую задачу с заданным входом и желаемым выходом, можно сравнить два любых

алгоритма по скорости. Выяснится, что самым быстрым является решение с наименьшей асимптотической временной сложностью в наихудшем случае, и именно так ведется анализ для одиночных вычислительных задач.

Но реальные задачи программирования, включая наш пример с резервуарами, работают не так. Они не получают вход, вычисляют результат и завершаются. Им нужна группа взаимодействующих методов или аспектов функциональности, которые могут использоваться любое количество раз. Причем разные структуры данных могут лучше проявлять себя в одном методе и хуже — в другом, сокращая сложность первого и замедляя второй. По этой причине часто не существует однозначно лучшего решения — требуются компромиссы.

ЧАСТИЧНЫЙ ПОРЯДОК

В контексте с множеством методов (как у нас) временная сложность в наихудшем случае подразумевает *частичный порядок* между реализациями — отношение между парами элементов, при котором не каждая пара является сравнимой. Например, применим отношение «быть потомком» к парам людей. Майк и Анна принадлежат отношению, если Майк — потомок Анны. Если же два человека *a* и *b* не связаны между собой, ни пара (*a*;*b*), ни пара (*b*;*a*) не принадлежит отношению, то есть отношение является частичным порядком. В таких отношениях могут быть элементы, *не меньшие* любого другого (верхние элементы).

Экономисты называют такие элементы *оптимальными по Парето*. *Парето-фронтом* называется множество всех элементов, оптимальных по Парето. Если интерпретировать отношение «быть потомком» как «быть меньше», мифические Адам и Ева будут единственными верхними элементами, потому что они не меньше (то есть не являются потомками) любого другого человека.

Рассмотрим другой пример, имеющий отношение к компьютерам: правила преобразования между примитивными типами Java подразумевают отношение частичного порядка между ними. Таким образом int меньше (то есть преобразуется в) long, тогда как boolean и int несравнимы.

Если при проектировании класса нет информации, сколько раз и в какой последовательности вызывается каждый метод (нет *профиля использования*), лучшее, что можно сделать, — это выбрать реализацию, профиль скорости выполнения которой оптимален по Парето. В такой реализации никакой метод не может быть улучшен без ухудшения скорости выполнения другого метода. В этой главе представлены три реализации, оптимальные по Парето, для задачи с резервуарами.

В этой главе представлены три реализации класса резервуара, и все они соответствуют API из главы 1. Эти реализации различаются по скорости выполнения, но среди них нет такой, которая бы всегда работала быстрее других (по крайней мере в сложности наихудшего случая). Однако можно измерить *среднюю* скорость выполнения заданной реализации при выполнении длинной последовательности операций. Как показывают простые тесты из раздела 3.4, по среднему значению третья реализация быстрее других во всех сценариях, кроме самых экзотических.

НЕОЖИДАННЫЙ ВОПРОС 1

Назовите частичный порядок, существующий между классами в программе Java.

3.1. ДОБАВЛЕНИЕ ВОДЫ С ПОСТОЯННЫМ ВРЕМЕНЕМ [SPEED1]

В этом разделе я покажу, как оптимизировать метод `addWater`, имеющий линейную сложность в эталонной реализации `Reference` (глава 2). Оказывается, его сложность можно сократить до постоянного времени без повышения сложности двух других методов класса. Вряд ли можно рассчитывать на что-то лучшее.

В реализации `Reference` проблема метода `addWater` заключалась в необходимости посещения всех резервуаров, соединенных с текущим, и обновления объемов содержащейся в них воды. Это неэффективно, особенно из-за того, что *все соединенные резервуары содержат одинаковые объемы воды*. Для повышения эффективности можно вынести поле `amount` из класса `Container` в новый класс `Group`. Все резервуары, входящие в одну группу, будут ссылаться на один объект `Group`, который содержит объем воды в каждом из резервуаров группы.

Новая версия `Container`, которая называется `Speed1`, содержит одно поле:

```
private Group group = new Group(this);
```

Каждый резервуар содержит ссылку на объект нового класса `Group`, который представляет собой вложенный класс (листинг 3.1). Конструктору передается `this`, так что новая группа в начале своего существования содержит этот резервуар. Каждый объект `Group` содержит два поля экземпляра: объем воды в каждом резервуаре группы и множество всех резервуаров в группе. Таким образом, каждый резервуар знает свою группу, а каждая группа знает все входящие в нее резервуары.

Класс `Group` объявлен с ключевым словом `static`, потому что группа не должна намертво связываться с создавшим ее резервуаром. Он объявлен приватным (`private`), потому что не должен быть доступен клиентам напрямую. Так как весь класс объявлен `private`, нет смысла изменять его видимость по отношению к конструктору и полям.

Листинг 3.1. Speed1: вложенный класс Group

```
private static class Group {
   double amountPerContainer;
   Set<Container> members;   ❶ Множество всех соединенных резервуаров

   Group(Container c) {
      members = new HashSet<>();
      members.add(c);
   }
}
```

На рис. 3.1 изображена ситуация после выполнения первых трех частей основного сценария использования. Напомню, что эти три части создают четыре резервуара (a, b, c и d) и вызывают следующие методы:

```
a.addWater(12);
d.addWater(8);
a.connectTo(b);
b.connectTo(c);
```

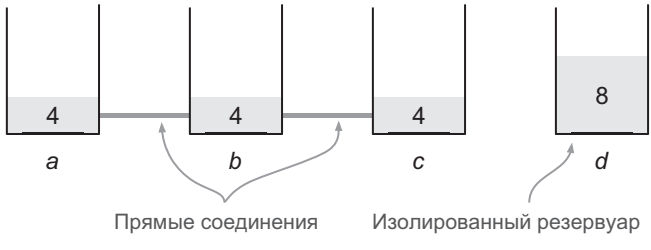


Рис. 3.1. Ситуация после выполнения первых трех частей основного сценария. Резервуары от *a* до *c* соединены, а в *a* и *d* наливается вода

Метод `connectTo` очень похож на одноименный метод из эталонной реализации `Reference` (https://bitbucket.org/mfaella/exercisesinstyle).

На рис. 3.2 изображено распределение памяти `Speed1` на данный момент сценария использования. Так как резервуары соединены в две группы, существуют два объекта типа `Group`, каждый из которых содержит ссылку на множество всех резервуаров, принадлежащих группе, а также объем воды в каждом из этих резервуаров.

Методы чтения и записи `Container` работают непосредственно с объектом `Group`:

Листинг 3.2. Speed1: методы getAmount и addWater

```
public double getAmount() { return group.amountPerContainer; }

public void addWater(double amount) {
   double amountPerContainer = amount / group.members.size();
   group.amountPerContainer += amountPerContainer;
```

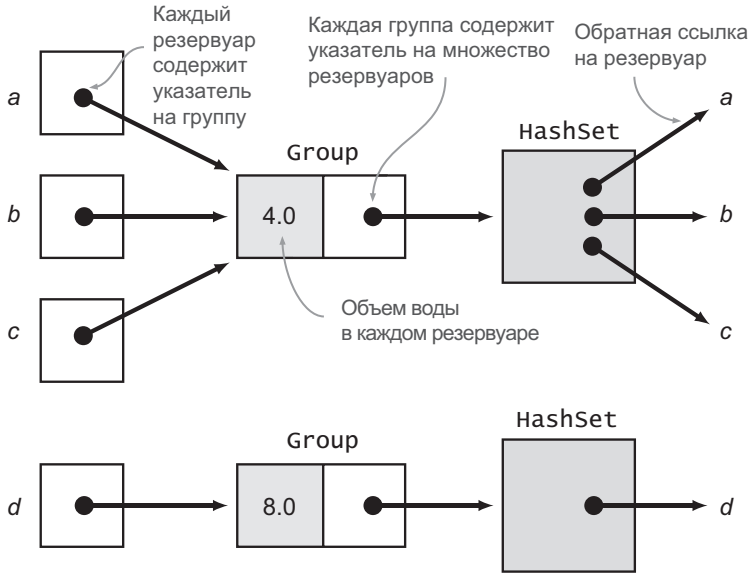


Рис. 3.2. Распределение памяти реализацией Speed1 после выполнения первых трех частей основного сценария

3.1.1. Временная сложность

По аналогии с `Reference`, метод `connectTo` все равно должен перебрать все резервуары в группе, что приведет к временным сложностям (табл. 3.1). Как видите, «узким местом» реализации является метод `connectTo`.

Таблица 3.1. Временные сложности для Speed1, где n обозначает общее количество резервуаров

Метод	Временная сложность
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$O(1)$

Два действия в методе `connectTo` требуют линейного времени для завершения:

1. Слияние элементов двух групп методом `addAll`.
2. Уведомление элементов одной из объединяемых групп о том, что их группа изменилась.

Первый шаг легко заменяется более быстрым альтернативным решением. Перейдите от множеств на связанные списки, и *вуаля*: слияние двух коллекций

превратится в операцию с постоянным временем. Избежать второго шага намного сложнее. Собственно, невозможно заставить `connectTo` выполняться за постоянное время без повышения временной сложности `getAmount`. Но если вам непременно понадобится `connectTo` с постоянным временем, примените реализацию из следующего раздела.

3.2. ДОБАВЛЕНИЕ СОЕДИНЕНИЙ ЗА ПОСТОЯННОЕ ВРЕМЯ [SPEED2]

Цель этого раздела — уменьшение сложности `connectTo` до постоянного времени, которое приведет к новой версии класса резервуара — `Speed2`. Для этого мы используем два приема:

1. Представим группы соединенных резервуаров в виде структуры данных, которая поддерживает операцию слияния с постоянным временем.
2. Максимально задержим обновление объема воды.

Для начала нам понадобится принципиально новый способ представления группы соединенных резервуаров: реализованный вручную циклический связанный список.

3.2.1. Представление групп в виде циклических списков

Циклический связанный список представляет собой последовательность узлов, в которой каждый узел содержит ссылку на следующий узел с циклическим замыканием списка. В списке нет ни первого узла, ни последнего. Пустой циклический связанный список не содержит ни одного узла, тогда как в списке из одного узла этот узел указывает сам на себя как на следующий узел.

В приложении с резервуарами каждый резервуар представляет собой узел односвязного циклического списка, который содержит поле `amount` и одну ссылку `next`:

Листинг 3.3. Speed2: поля

```
public class Container {
   private double amount;
   private Container next = this;
```

НЕОЖИДАННЫЙ ВОПРОС 2

С какой сложностью выполняется удаление узла из односвязного циклического списка?

У циклических связанных списков есть одно важное свойство, из-за которого они так популярны: если у вас есть два произвольных узла из двух таких списков, то слияние этих списков выполняется за постоянное время (даже если списки являются односвязными) посредством перестановки указателей `next` двух узлов (рис. 3.3).

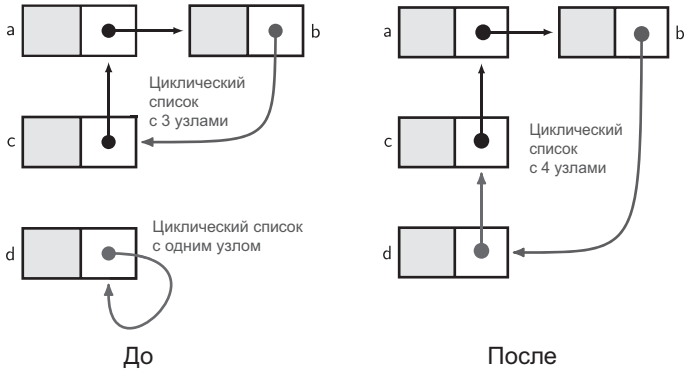


Рис. 3.3. Перестановка указателей next двух узлов (b и d) приводит к слиянию двух циклических связных списков

Тем не менее этот метод работает только в том случае, если два узла принадлежат разным спискам. Иначе перестановка ссылок ведет к обратному эффекту: список разбивается на два разных списка. А значит, эта реализация страдает от того же ограничения, которое упоминалось для `Novice`: `connectTo` работает правильно, только если соединяемые резервуары еще не соединены — даже косвенно.

Напрашивается вывод, что метод `connectTo` должен *проверять*, соединены ли два резервуара, прежде чем пытаться соединять их. Но для этого необходимо просканировать по крайней мере один из двух списков, а эта операция уже не выполняется с постоянным временем. Для достижения желаемой скорости выполнения с постоянным временем приходится мириться с подобным снижением защищенности программы. Положение будет исправлено в главе 5, когда мы займемся построением классов резервуаров с повышенной надежностью.

СОВЕТ

Стандартная реализация связанных списков на Java (LinkedList) не поддерживает конкатенацию с постоянным временем. Вызов `list1.addAll(list2)` перебирает все элементы `list2`.

КАК НАСЧЕТ ОБЫЧНОГО СВЯЗАННОГО СПИСКА?

Циклические списки — не единственная структура данных, обеспечивающая слияние за постоянное время. Обычный связанный список также обладает таким свойством, при условии что операция слияния может напрямую обращаться к первым и последним элементам двух списков («голове» и «хвосту»). Представьте, что вы можете напрямую обратиться к полям `head` и `tail` двух непустых связанных списков `list1` и `list2`. Слияние их посредством конкатенации выполнится следующими строками:

```
list1.tail.next = list2.head;
list1.tail = list2.tail;
```

В результате `list1` представит результат слияния, а `list2` не изменится.

Тем не менее связанные списки не удастся использовать для соединения резервуаров с водой за постоянное время. Чтобы понять почему, вспомните, что каждый резервуар должен иметь прямой доступ к первому и последнему элементу списка. При слиянии двух групп необходимо обновить все задействованные резервуары, чтобы они отражали новые значения `head` и `tail` после слияния. Такое обновление требует линейного времени.

На рис. 3.4 представлено распределение памяти в два момента во время выполнения основного сценария использования, с реализацией резервуаров из этого раздела (то есть `Speed2`). Как видите, структура точно совпадает с рис. 3.3, не считая того, что узлы в списках теперь представляют резервуары с водой.

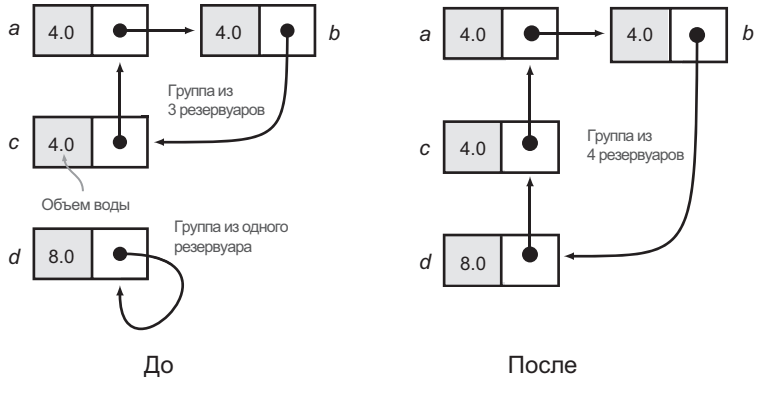


Рис. 3.4. Распределение памяти Speed2 при выполнении основного сценария использования, до и после b.connectTo(d). Перестановка указателей next объектов *b* и *d* приводит к слиянию двух групп в одну

В левой части схемы резервуары `a`, `b` и `c` соединены в одну группу, поэтому они циклически связаны друг с другом. Резервуар `d` остается изолированным, поэтому его `next` указывает на себя.

В правой части показан результат выполнения команды `b.connectTo(d)`. Перестановки указателей `next` объектов `b` и `d` достаточно, для того чтобы объединить два списка в один:

Листинг 3.4. Speed2: метод connectTo

```
public void connectTo(Container other) {      ❶ Переставляет поля next объектов this и other
   Container oldNext = next;
   next = other.next;
   other.next = oldNext;
}
```

3.2.2. Отложенные обновления

Метод `connectTo` выполняется с постоянным временем, потому что не обновляет объемы воды. В конце концов, программа работает с ними только при вызове `getAmount`. Как следствие, обновление можно *отложить* до следующего вызова `getAmount`. Этот стандартный прием в инструментарии программиста, называемый *отложенным вычислением*, является неотъемлемой частью функционального программирования. Отложенные вычисления задерживаются до того момента, когда их результаты действительно потребуются в программе.

НЕОЖИДАННЫЙ ВОПРОС 3

Предложите две свои задачи, которые вы выполняли бы немедленно (при первой возможности), и две активности, которые вы предпочли бы отложить.

В `addWater` используется тот же принцип отложенных вычислений, чтобы обновлялся только текущий резервуар без фактического распределения воды в группе. К сожалению, рано или поздно будет вызван метод `getAmount`, и вам придется расплачиваться за отложенное выполнение затратной операцией обновления, которая распределит объемы воды в равных пропорциях среди участников группы. Чтобы код стал более ясным, делегируйте обновление отдельному приватному методу `updateGroup`:

Листинг 3.5. Speed2: методы addWater и getAmount

```
public void addWater(double amount) {
   this.amount += amount; ❶ Обновляет только локальный резервуар
}
```

```
public double getAmount() {
   updateGroup();          ❷ Вспомогательный метод, отвечающий за распределение воды
   return amount;
}
```

ОТЛОЖЕННЫЕ ВЫЧИСЛЕНИЯ В JDK

Стандартные коллекции Java используют немедленное вычисление (противоположное отложенному). В Java 8 появилась библиотека streams — мощный фреймворк для манипуляций с последовательностями данных. Среди прочего, streams применяет отложенное вычисление. Рассмотрим список `list` с целочисленными элементами. Команда

```
list.sort(null);
```

немедленно сортирует список, потому что он использует немедленное вычисление. (Аргумент *null* означает, что целые числа имеют естественный порядок и функция сравнения не нужна.) С другой стороны, попробуйте преобразовать этот список в поток с последующей сортировкой потока:

```
Stream<Integer> stream = list.stream();
Stream<Integer> sortedStream = stream.sorted();
```

В отличие от предыдущего примера, в данном случае сортировка пока не выполняется. Метод `sorted` всего лишь устанавливает флаг, который *обещает* в какой-то момент отсортировать данные. Библиотека проводит фактическую сортировку данных при применении *терминальной операции* к потоку, то есть при преобразовании потока в коллекцию или при переборе его элементов (например, методом `forEachOrdered`).

Метод обновления (листинг 3.6) делает два прохода по циклическому списку, представляющему группу. При первом проходе он вычисляет общий объем воды в группе и подсчитывает количество содержащихся в ней резервуаров. При втором проходе информация, собранная при первом проходе, используется для фактического обновления объема воды в каждом резервуаре.

Листинг 3.6. Speed2: вспомогательный метод updateGroup

```
private void updateGroup() {
   Container current = this;
   double totalAmount = 0;
   int groupSize = 0;

   do {                                    ❶ Первый проход:
      totalAmount += current.amount;          подсчет объема и количества резервуаров
      groupSize++;
```

```
      current = current.next;
   } while (current != this);
   double newAmount = totalAmount / groupSize;

   current = this;
   do {                                    ❷ Второй проход:
      current.amount = newAmount;             обновление объемов
      current = current.next;
   } while (current != this);
}
```

При каждом проходе необходимо посетить каждый узел циклического списка. Чтобы избежать зацикливания, можно начать с любого узла, переходить по указателю `next` и остановиться при возвращении к исходному узлу.

Тут же приходит в голову пара вопросов:

1. Действительно ли необходимо вызывать `updateGroup` при каждом вызове `getAmount`? Может, удастся обойтись флагом для пометки уже обновленных резервуаров и избежать лишних вызовов `updateGroup`?
2. Нельзя ли переместить вызов `updateGroup` из `getAmount` в `addWater`? Ведь разумнее платить цену вычислений при записи, а не при чтении.

К сожалению, ни одно из этих потенциальных улучшений нельзя реализовать — если, конечно, вы хотите сохранить постоянное время выполнения операции.

Во-первых, представьте, что во все резервуары добавлен флаг обновления. Каждый раз, когда группа обновляется, ее резервуары помечаются как обновленные. Последующим вызовам `getAmount` для этих резервуаров не обязательно вызывать `updateGroup` — пока все нормально. Но предположим, что две группы были объединены вызовом `connectTo`. Флаги обновления их резервуаров необходимо сбросить, но сделать это за постоянное время не удастся[1].

Во-вторых, перемещение вызова `updateGroup` из `getAmount` в `addWater` — нормальная идея, но только если аналогичный вызов также будет включен в `connectTo`. В противном случае чтение объема сразу же после слияния групп даст устаревший результат. Это изменение также переведет `connectTo` в линейную сложность, что противоречит целям этого раздела.

Сложность реализации `Speed2` в наихудшем случае показана в табл. 3.2. Как и ожидалось, `connectTo` и `addWater` выполняются за постоянное время, потому что основная работа перемещена в метод `getAmount`, требующий линейного времени.

---

[1] Откровенно говоря, это можно сделать, если переместить флаг обновления из резервуаров в отдельный объект `Group`, по аналогии со `Speed1`. Но даже тогда сложность `getAmount` в наихудшем случае останется неизменной (линейной).

Таблица 3.2. Временные сложности для Speed2, где n обозначает общее количество резервуаров

Метод	Временная сложность
`getAmount`	$O(n)$
`connectTo`	$O(1)$
`addWater`	$O(1)$

3.3. ЛУЧШИЙ БАЛАНС: АЛГОРИТМЫ ПОИСКА ОБЪЕДИНЕНИЙ [SPEED3]

Как выясняется, наша маленькая задача с резервуарами напоминает классическую задачу нахождения объединения. В этом сценарии требуется поддерживать непересекающиеся множества элементов с выделенным элементом в каждом множестве, который называется *представителем*. Необходимо обеспечить поддержку следующих двух операций:

- Слияние двух множеств (операция *объединения*).
- Для заданного элемента поиск представителя из соответствующего множества (операция *поиска*).

Мы применим сценарий нахождения объединения к задаче с резервуарами и получим реализацию `Speed3`, наиболее производительную на практике.

При применении обобщенного сценария нахождения объединения к задаче с резервуарами множества представляют собой группы взаимно соединенных резервуаров. Представителем группы может быть любой резервуар, который будет использоваться для хранения «официального» объема воды для этой группы. Когда резервуар получит вызов `getAmount`, активизируется операция поиска для получения значения из представителя группы.

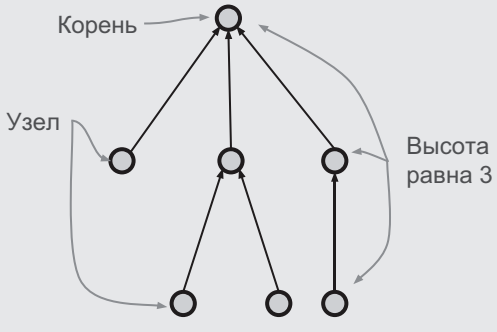
Многие достойные специалисты computer science занимались такими задачами и разработали следующий — доказуемо оптимальный — алгоритм, который требует представления группы в виде *дерева* резервуаров, где каждый резервуар знает только своего родителя в дереве. Корень каждого дерева является представителем группы. В корнях также должен храниться размер дерева (скоро вы поймете зачем).

НЕОЖИДАННЫЙ ВОПРОС 4

Вы пишете компилятор Java. Требуется представить отношения *подклассирования* между классами, в которых классы образуют структуру дерева. Какое дерево лучше использовать — дерево указателей на родителей или дерево указателей на дочерние узлы?

ДЕРЕВО УКАЗАТЕЛЕЙ НА РОДИТЕЛЕЙ

Дерево указателей на родителей — связанная структура данных, в которой каждый узел указывает ровно на один другой узел, называемый его *родителем*. Исключением является один специальный узел, называемый корнем: он не указывает на другой узел. Кроме того, все узлы могут перейти к корню по цепочке указателей. С этими ограничениями указатели заведомо не образуют цикл, поэтому деревья являются особой разновидностью направленного ациклического графа (DAG, directed acyclic graph).



Дерево указателей на родителей

Узлы, не имеющие дочерних узлов, называются листовыми узлами, или *листьями*. В дереве указателей на родителей каждому узлу известен его родитель, но родитель не содержит ссылок на его дочерние узлы. В результате по дереву можно перемещаться только по направлению от листьев к корню. *Высота* дерева равна длине самого длинного пути от любого узла к корню.

Деревья в computer science традиционно изображаются в перевернутом виде: корень расположен сверху, и дерево растет вниз.

Как следует из обсуждения алгоритма дерева, каждый объект резервуара будет содержать такие поля:

Листинг 3.7. Speed3: поля, конструктор не нужен

```
public class Container {
   private double amount;
   private Container parent = this; ❶ Изначально каждый резервуар
   private int size = 1;              является корнем своего дерева
```

Корень дерева распознается по условию `parent==this`. Из листинга 3.7 видно, что каждый новый резервуар изначально является корнем своего дерева

и единственным содержащимся в нем узлом. Поля `amount` и `size` используются только для корневых резервуаров. Для других резервуаров они только впустую расходуют память. Реализация, оптимизированная по памяти, может улучшить эту ситуацию[1].

3.3.1. Поиск представителя группы

Чтобы получить желаемую скорость выполнения, недостаточно просто представить группы резервуаров в виде деревьев — указателей на родителей. Необходимо использовать два приема при операциях с деревом:

1. Во время операции поиска применить *сжатие пути*.
2. Во время операции объединения применить *политику связывания по размеру*.

Начнем с операции поиска и сжатия пути. Все операции с `Container` должны находить представителя группы, потому что информация об объеме воды хранится именно в нем. С учетом предшествующего обсуждения деревьев — указателей на родителей найти представителя группы несложно: следуйте по ссылкам на родителей, пока не будет достигнут корень дерева, который можно узнать по условию `parent==this`. При сжатии пути *каждый встреченный узел преобразуется в прямого потомка корня*. Дерево модифицируется в процессе перемещения для повышения эффективности будущих операций.

Выделим задачу поиска корня в приватный вспомогательный метод с именем `findRootAndCompress` (листинг 3.8). Метод перемещается по дереву от резервуара вверх до корня дерева по указателям на родителя. На этом пути он обновляет указатели на родителей всех обнаруженных резервуаров прямыми указателями на корень. Затем при каждом повторном вызове для любого из этих объектов он завершится за постоянное время, потому что немедленно найдет корень.

Для примера возьмем три резервуара `x`, `y` и `root`, соединенные таким образом, что `root` — представитель группы, `y` — его дочерний узел, а `x` — дочерний узел `y` (рис. 3.5, слева).

Вызов `x.findRootAndCompress()` должен вернуть ссылку на `root`, а также сократить путь, ведущий от `x` к `root`, преобразуя все промежуточные резервуары на пути от `x` к `root` в дочерний узел `root`. В этом примере единственным резервуаром, который может быть преобразован, является сам узел `x`, потому что `y` уже является непосредственным дочерним узлом `root`. Желательная ситуация после вызова изображена на рис. 3.5 справа. Сложная на первый взгляд задача сжатия элегантно решается рекурсивной реализацией из трех строк:

---

[1] В главе 4 приводится возможное решение проблемы, в частности, в упражнении 3.

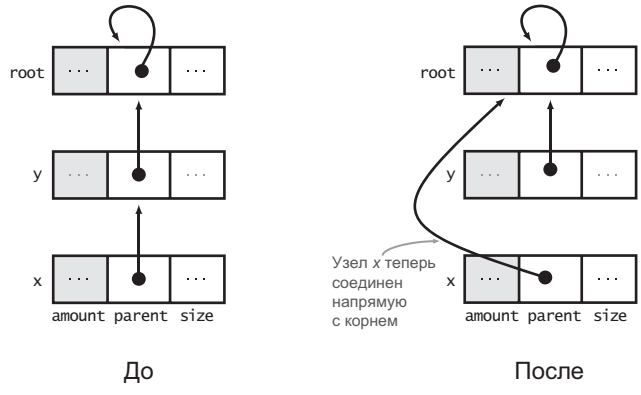


Рис. 3.5. Распределение памяти трех соединенных резервуаров в Speed3, до и после вызова x.findRootAndCompress(). После вызова резервуар x стал прямым дочерним узлом root. Поля amount и size опущены как несущественные

Листинг 3.8. Speed3: вспомогательный метод findRootAndCompress

```
private Container findRootAndCompress() {
   if (parent != this) ❶ Проверяет, является ли this корнем своего дерева
      parent = parent.findRootAndCompress(); ❷ Рекурсивно ищет корень
   return parent;                              и назначает его родителем this
}
```

Разобраться в логике рекурсивных методов бывает непросто. Проанализируем этот листинг шаг за шагом. Каждый раз, когда вы вызываете `findRootAndCompress` для корневого резервуара, он просто возвращает сам резервуар (`this`). Если вызвать метод для резервуара, находящегося ниже, метод вызовет себя для его родителя. Если родитель все еще не является корнем группы, то метод снова вызовет себя для его родителя, и так далее, пока метод в конечном итоге не будет вызван для корня. В этот момент начнется каскад `return`, который распространится от корня к месту исходного вызова. При этом метод обновляет все ссылки `parent`, чтобы они указывали на `root`.

Возвращаясь к примеру с тремя резервуарами, за ходом выполнения `x.findRootAndCompress();` можно проследить на *диаграмме последовательности* UML (рис. 3.6). Если такие диаграммы вам незнакомы, обратитесь к врезке.

Начиная с вызова `x.findRootAndCompress()`, на рис. 3.6 представлена последовательность дальнейших действий: `findRootAndCompress` вызывает себя для `y`, а затем для `root`. В этой точке ссылка на `root` возвращается по всей цепочке исходной вызывающей стороне, при этом все промежуточные указатели `parent` обновляются указателем на `root`. Как упоминалось ранее, это приводит к итоговой структуре памяти на рис. 3.5 справа. Теперь узел `x` связан напрямую с `root` в результате сжатия.

ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ UML

Диаграммы последовательности (вроде изображенной на рис. 3.6) представляют взаимодействия между объектами во времени. Каждый объект представляется блоком, соединенным с его пунктирной вертикальной *линией жизни*. Время течет сверху вниз, а вызовы методов (или *сообщения*) представляются стрелками от линии жизни вызывающей стороны к линии жизни вызываемой стороны. Сообщение запускает выполнение метода. Графически сообщение изображается пустым вертикальным прямоугольником активации, который рисуется поверх линии жизни вызывающей стороны. Если вы хотите сильнее выделить возвращаемое значение метода (как на рис. 3.6), добавьте пунктирную стрелку от прямоугольника активации к вызывающей стороне.

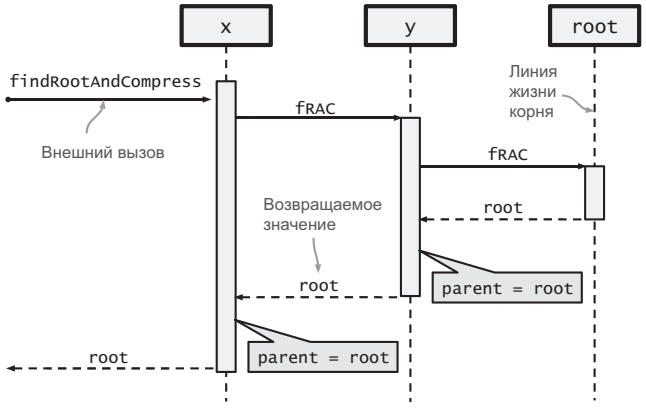


Рис. 3.6. Диаграмма последовательности для вызова x.findRootAndCompress() — сокращенно fRAC — в сценарии с тремя резервуарами, изображенном на рис. 3.5. Выноски не входят в стандарт UML

После реализации `findRootAndCompress` методы `getAmount` и `addWater` становятся элементарными: они получают корень своей группы, а затем читают или обновляют свое поле `amount`:

Листинг 3.9. Speed3: методы getAmount и addWater

```
public double getAmount() {
   Container root = findRootAndCompress(); ❶ Получение корня и сжатие пути

   return root.amount; ❷ amount читается из root
}
```

```
public void addWater(double amount) {
   Container root = findRootAndCompress(); ❸ Получение корня и сжатие пути
   root.amount += amount / root.size;      ❹ Добавление воды в корневой резервуар
}
```

3.3.2. Соединение деревьев

Структура дерева позволяет использовать простой алгоритм соединения, который находит корни двух объединяемых групп, и один из корней преобразуется в дочерний узел другого корня (рис. 3.7).

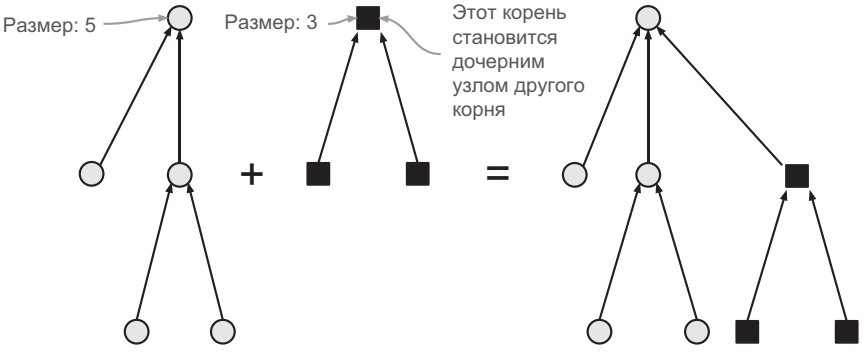


Рис. 3.7. Слияние двух деревьев в соответствии с политикой связывания по размеру. Меньшее дерево присоединяется к корню большего дерева

Для ограничения высоты полученного дерева необходимо применить следующее правило: меньшее дерево (с меньшим количеством узлов) присоединяется к корню большего дерева. Если два дерева имеют одинаковые размеры, присоединяемое дерево выбирается произвольно. Это правило, называемое *политикой связывания по размеру*, является важным шагом навстречу желаемой скорости выполнения, как объясняется в следующем разделе. Из-за этой политики корни должны знать размеры своих деревьев, отсюда и необходимость поля `size` в каждом резервуаре.

В листинге 3.10 показана возможная реализация метода `connectTo`. Она начинается с нахождения корней двух объединяемых групп. Затем она проверяет, совпадают ли корни, то есть что два резервуара уже соединены. Без этого шага возникает ошибка (а вернее, недостаточная защищенность) версий `Novice` и `Speed2`. При соединении двух резервуаров, уже принадлежащих одной группе, структура окажется в некорректном состоянии. После этого метод вычисляет новый объем воды, который должен быть помещен в каждый резервуар, и объединяет два дерева в соответствии с описанной политикой связывания по размеру.

Листинг 3.10. Speed3: метод connectTo

```
public void connectTo(Container other) {
   Container root1 = findRootAndCompress(), ❶ Находит два корня
             root2 = other.findRootAndCompress();
   if (root1==root2) return;                ❷ Проверка необходима!
   int size1 = root1.size, size2 = root2.size;
   double newAmount = ((root1.amount * size1) +
                       (root2.amount * size2)) / (size1 + size2);

   if (size1 <= size2) {                    ❸ Политика связывания по размеру
      root1.parent = root2;                 ❹ Присоединяет первое дерево
      root2.amount = newAmount;                к корню второго
      root2.size += size1;
   } else {
      root2.parent = root1;                 ❺ Присоединяет второе дерево
      root1.amount = newAmount;                к корню первого
      root1.size += size2;
   }
}
```

Теперь у вас есть вся информация, необходимая для обычного моделирования основного сценария использования и получения распределения памяти (рис. 3.8), соответствующего ситуации после первых трех частей сценария. В этот момент b является представителем для группы, содержащей резервуары a, b и c, тогда как резервуар d изолирован и, как следствие, сам является своим представителем.

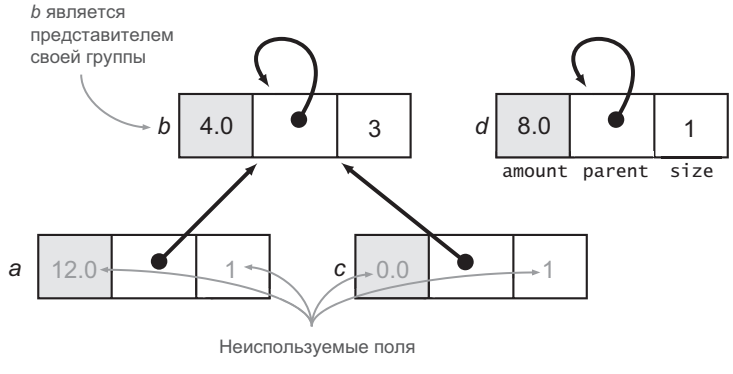


Рис. 3.8. Распределение памяти Speed3 после выполнения первых трех частей основного сценария. Поля amount и size объектов a и c окрашены в серый цвет, потому что они содержат устаревшие значения, не актуальные для поведения объектов. Только поля представителей группы актуальны и обновлены

3.3.3. Временная сложность в наихудшем случае

Так как все методы резервуаров начинаются с вызова findRootAndCompress (дважды в случае connectTo), для сравнения Speed3 с предыдущими реализациями

резервуара необходимо оценить сложность этого метода в наихудшем случае. Так как `findRootAndCompress` является рекурсивным методом, не содержащим циклов, его сложность равна количеству рекурсивных вызовов (*глубине* рекурсии), которая, в свою очередь, равна длине пути от этого резервуара до корня дерева. В худшем случае метод вызывается для резервуара, находящегося на максимальном расстоянии от корня, то есть на расстоянии высоты дерева. При этом все еще необходимо определить максимальную высоту, которую может достичь дерево с заданным количеством узлов. Здесь на помощь приходит политика связывания по размеру, которая гарантирует, что высота дерева находится в *логарифмической зависимости от размера*. Например, высота дерева, представляющего группу из 8 резервуаров, не может превышать 3 (напомним, что $3 = \log_2 8$, потому что $2^3 = 8$).

На рис. 3.9 представлена серия операторов объединения, которая строит дерево с логарифмической высотой. Фокус заключается в объединении деревьев равного размера. Для каждого такого слияния высота полученного дерева увеличивается на 1, а количество узлов удваивается. Следовательно, высота постоянно равна логарифму размера по основанию 2.

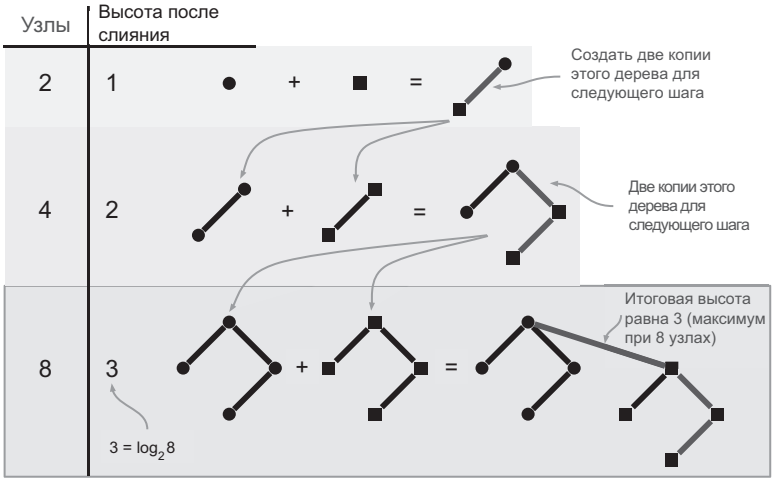


Рис. 3.9. Последовательность операций объединения при построении дерева, высота которого связана логарифмической зависимостью с его размером. Это максимальная высота, достижимая при заданном количестве узлов

Вы уже видели, что вызовы `findRootAndCompress` требуют логарифмического времени. Так как этот метод вызывается всеми тремя открытыми методами, мы получим временные сложности в наихудшем случае (табл. 3.3).

Обратите внимание: несмотря на то что один конкретный вызов `x.findRootAndCompress()` требует логарифмического времени, метод сжатия

Таблица 3.3. Временные сложности для Speed3, где n обозначает общее количество резервуаров

Метод	Временная сложность
`getAmount`	$O(\log n)$
`connectTo`	$O(\log n)$
`addWater`	$O(\log n)$

пути гарантирует, что будущие вызовы того же метода для того же резервуара, а также для любого резервуара, находящегося на пути от `x` до корня дерева, будут выполняться за постоянное время. Это наблюдение наводит на мысль, что назначение логарифмической сложности всем трем методам резервуара несколько некорректно, хотя и формально правильно, потому что эта сложность применима только к первому вызову заданного резервуара. В следующем разделе мы решим эту проблему и перейдем на другую разновидность анализа сложности. Пока я буду использовать сложности в наихудшем случае из табл. 3.3 для сравнения скоростей выполнения трех реализаций в этой главе.

На рис. 3.10 изображено графическое представление сложности методов `getAmount` и `connectTo` для трех версий этой главы. Как и ожидалось, ни одна из них не показывает лучших результатов постоянно. Только `Speed1` гарантирует постоянное время для `getAmount`. Аналогичным образом `Speed2` обеспечивает лучшую скорость выполнения `connectTo`. `Speed3` выдерживает баланс между двумя методами, в результате чего обоим назначается одинаковая логарифмическая сложность. Один и тот же метод в двух разных реализациях то улучшает скорость выполнения, то ухудшает ее. Как объяснялось в начале главы, в научной терминологии его можно назвать *оптимальным по Парето*.

Необходимо проанализировать контекст приложения и определить, с какой частотой клиенты будут вызывать каждый метод (рис. 3.10). Если чаще вызывается метод `getAmount`, для проекта следует выбрать реализацию `Speed1`. И наоборот, если клиенты с наибольшей вероятностью вызовут `connectTo`, актуальна `Speed2`.

Из следующего раздела вы узнаете, что анализ наихудшего случая такого рода некорректен для `Speed3`, где скорость выполнения в полной мере проявляется при анализе *амортизированной сложности*. Наихудший случай в `Speed3` встречается настолько редко, что его анализ не даст нам достаточно информации.

3.3.4. Амортизированная временная сложность

Если стандартный анализ ориентируется на один прогон алгоритма, амортизированный анализ исследует *серию* запусков, что подходит для алгоритмов, выполняющих дополнительные операции, улучшающие будущие вызовы. Эти

дополнительные операции можно сравнить с *инвестициями*. Анализ для одного выполнения алгоритма учтет вложения, но не будущую прибыль, и только серия запусков алгоритма позволит оценить будущий выигрыш от затрат.

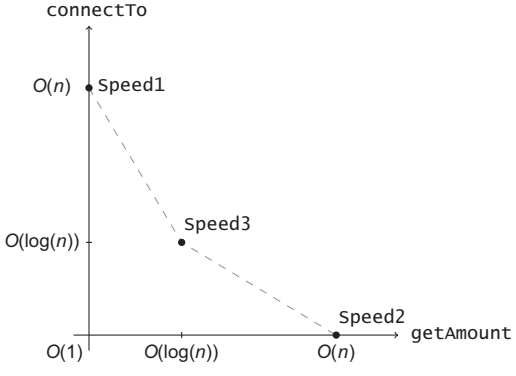


Рис. 3.10. Графическое представление временной сложности в наихудшем случае для методов getAmount и connectTo в реализациях Speed1, Speed2 и Speed3. Пунктирная линия, соединяющая три реализации, представляет Парето-фронт

В нашем случае сжатие в `findRootAndCompress` требует дополнительных затрат. Для нахождения корня сжатие ускоряет будущие вызовы, но не является обязательной мерой.

Для выполнения амортизированного анализа необходимо выбрать серию произвольной длины m, выполняемую с набором из n резервуаров. Так как нас интересуют затраты в долгосрочной перспективе, можно предположить, что $m > n$. Затем необходимо выбрать, какую долю из m операций составляют `connectTo`, `getAmount` и `addWater`. Обратите внимание: значимыми являются только $n - 1$ вызовов `connectTo`, поскольку потом все резервуары соединяются в одну группу. Разумно анализировать серии, имеющие следующую структуру:

1. Серия состоит как минимум из n операций.
2. Серия содержит $n - 1$ вызовов `connectTo`.
3. Остальными операциями являются либо `getAmount`, либо `addWater`.

Теперь по правилам стандартного анализа сложности можно спросить, в какой зависимости от количества элементарных действий находится порядок роста, выполняемых каждой такой серией (то есть наихудший случай среди всех серий, удовлетворяющих предположениям). Реальный анализ выходит за рамки книги, за подробностями обращайтесь к разделу «Дополнительная литература» в конце главы. Более того, даже *определение* верхней границы сложности — достаточно нетривиальная задача! Самая точная верхняя граница для серии из m операций определяется не самой простой функцией: она чуть больше постоянной,

но значительно меньше квазилинейной ($m \log m$). Ее можно записать в виде $O(m \times \alpha(n))$, где α — *обратная функция Аккермана*.

Таблица 3.4. Амортизированная временная сложность для Speed3, где α(n) — обратная функция Аккермана

Сценарий	Амортизированная временная сложность
Серия из m операций с n резервуарами	$O(m \times \alpha(n))$

ФУНКЦИЯ АККЕРМАНА

Вильгельм Аккерман предложил одноименную функцию $A(k; j)$ в 1928 году. Он был учеником знаменитого математика Давида Гильберта и прославленным ученым. Это первый известный пример функции, которая является алгоритмически вычислимой, но не вычисляется через ограниченный набор операций, то есть не является *примитивной рекурсивной*. Ключевое свойство этой функции состоит в том, что она чрезвычайно быстро растет даже при малых значениях аргументов, например, $A(2; 1) = 7$; $A(3; 1) = 2047$; $A(4; 1) > 10^{80}$.

Обратная функция Аккермана $\alpha(n)$ определяется как наименьшее целое k, для которого $A(k; 1) \geq \alpha(n)$. Из-за быстрого роста $A(k; 1)$ малых значений k достаточно, чтобы сделать ее больше n. В частности, $\alpha(n)$ не превышает 4 для всех значений $n < 10^{80}$ — приблизительного числа атомов в известной Вселенной.

Обратная функция Аккермана является фактически постоянной, так что верхняя граница $O(m \times \alpha(n))$ эквивалентна $O(m)$ для всех практических целей. Так как мы обсуждаем сложность серии из m операций, верхняя граница $O(m)$ означает, что в долгосрочной перспективе каждая операция выполняется за постоянное время. Ни на что лучшее рассчитывать не следует. Эксперименты из раздела 3.4.1 показывают, что в этом случае амортизированный анализ намного более актуален, чем стандартный анализ наихудшего случая, в результате чего `Speed3` заметно обходит своих конкурентов в типичном сценарии.

3.3.5. Амортизированный анализ массивов с изменяемым размером

Амортизированный анализ деревьев нахождения объединений слишком сложен, чтобы описывать его во всех подробностях, поэтому мы возьмем более простой, но чрезвычайно важный случай: массивы с автоматически изменяемым размером, такие как классы Java `Vector` и `ArrayList`, или класс

`List` в C#. Эти классы хранят коллекции переменного размера в непрерывной области памяти с произвольным доступом к любому элементу коллекции за постоянное время. Для этого коллекция хранится в массиве, размеры которого изменяются при необходимости. К сожалению, расширять массивы «на месте» невозможно[1]. В случае расширения класс должен выделить новый массив большего размера, а затем скопировать содержимое старого массива в новый. С какими затратами сопряжена такая операция? С какой сложностью в эту коллекцию будут добавляться новые элементы, если учесть, что любое добавление может инициировать затратную операцию изменения размера? Амортизированная сложность — правильный инструмент для получения ответов на эти вопросы. Вот что говорится в документации Oracle при описании класса `ArrayList`:

> *Операция добавления выполняется за* амортизированное постоянное время, иначе говоря, *добавление n элементов требует времени O(n).*

Чтобы понять почему, проанализируем текущую реализацию `ArrayList` из OpenJDK[2]. Легко определить, что исходная емкость пустого `ArrayList` составляет 10 ячеек, а приватный метод `grow` отвечает за расширение используемого массива. Внутри метода присутствуют следующие критические строки:

```
int newCapacity = oldCapacity + (oldCapacity >{}> 1);
...
elementData = Arrays.copyOf(elementData, newCapacity);
```

Оператор `>>` — поразрядный сдвиг вправо — хорошо делит целые числа на 2, поэтому первая строка увеличивает емкость на 50 %. Каждый раз, когда возникает необходимость в увеличении массива, он расширяется не на один элемент, а на 50%. Эта стратегия чрезвычайно важна для контроля за амортизированной сложностью. Метод выделяет массив с новой емкостью вызовом `Arrays.copyOf` — статического вспомогательного метода, который выделяет память для нового массива и копирует в нее существующий массив.

Теперь рассмотрим серию из n вставок (метод `add`) в новую коллекцию `ArrayList` и вычислим ее общую сложность. Нужно узнать, сколько раз будет выделяться память для используемого массива. Назовем это число k. Каждое новое выделение увеличивает емкость в 1,5 раза. Если исходная емкость была равна 10, то после k повторных выделений она составит $10 \times 1{,}5^k$. Чтобы контейнер мог обеспечить n вставок, эта емкость должна быть равна как минимум n:

$$10 \times 1{,}5^k \geq n.$$

---

[1] На самом деле низкоуровневые языки предоставляют такую возможность. Обращайтесь к описанию функции `realloc` из стандартной библиотеки C.

[2] В настоящее время доступна по адресу http://mng.bz/j5m8.

Другими словами, $k \geq \log_{1,5}\frac{n}{10}$. Не беспокойтесь, логарифм скоро исчезнет; достаточно знать, что $1{,}5^{(\log_{1,5} x)} = x$. Целое по определению число k является наименьшим целым числом, превышающим $\log_{1,5}\frac{n}{10}$[1]. Чтобы упростить вычисления, ослабьте ограничение о том, что k — целое число, и установите $k = \log_{1,5}\frac{n}{10}$. Эта аппроксимация не повлияет на окончательный результат.

В серии из n вставок первые 10 будут выполняться быстро (стоимость 1), потому что исходная емкость пустой коллекции `ArrayList` равна 10. Одиннадцатая вставка инициирует первый вызов `grow`, увеличивая емкость до 15. Стоимость этого вызова тоже будет равна 15, потому что вызов `Arrays.copyOf` (см. предыдущие строки кода) должен скопировать 10 значений из старого массива в новый, и *инициализировать пять новых ячеек* значением `null`. Подведем итог: в этом случае стоимость n вставок может быть выражена:

$$\text{стоимость}(n) = \underbrace{1+1+\ldots+1}_{\text{10 вставок}} + \underbrace{15}_{\text{рост}} + \underbrace{1+1+\ldots+1}_{\text{5 вставок}} + \underbrace{22}_{\text{рост}} + 1 + 1 + \ldots$$

что можно привести к следующему виду:

$$\begin{aligned}
\text{стоимость}(n) &= 10 + (15+5) + (22+7) + (33+11) + \ldots = \\
&= 10 + (15\times 1 + 5\times 1) + (15\times 1{,}5 + 5\times 1{,}5) + \left(15\times(1{,}5)^2 + 5\times(1{,}5)^2\right) + \ldots = \\
&= 10 + \sum_{i=0}^{k}\left(15\times(1{,}5)^i + 5\times(1{,}5)^i\right) = \\
&= 10 + 20\sum_{i=0}^{k}(1{,}5)^i
\end{aligned}$$

Теперь применим стандартную формулу для суммы первых k степеней константы a:

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1}-1}{a-1},$$

Применив эту формулу для $a = 1{,}5$ и $k = \log_{1,5}\frac{n}{10}$, получаем

[1] Никакое целое число не может быть *равно* $\log_{1,5}\frac{n}{10}$. А вы знаете почему?

$$\text{стоимость}(n) = 10 + 20 \times \frac{1{,}5^{\left(\log_{1,5}\frac{n}{10}+1\right)} - 1}{1{,}5 - 1} =$$

$$= 10 + 20 \times \frac{1{,}5 \times 1{,}5^{\left(\log_{1,5}\frac{n}{10}\right)} - 1}{0{,}5} =$$

$$= 10 + 20 \times 2 \times \left(1{,}5 \times \frac{n}{10} - 1\right) =$$

$$= 10 + 60 \times \frac{n}{10} - 40 =$$

$$= 6 \times n - 30 =$$

$$= O(n).$$

Подведем итог. Стоимость n вставок линейно пропорциональна n, что означает, что средняя долгосрочная стоимость одиночной вставки является постоянной величиной. Это вычисление подтверждает, что вызовы `grow` становятся более затратными при повышении расстояния между ними в серии вставок. Если эти затраты разнесены по длинной серии вставок, средняя нагрузка на каждую операцию остается постоянной (6 в данной формуле).

Со вставкой в `ArrayList` тоже все гладко. Анализ, показывающий, что долгосрочная стоимость каждой вставки постоянна, подчеркивает, что скорость выполнения серии вставок неравномерна. Большинство вставок обходится очень дешево, но время от времени вставка запускает полное копирование всех ранее вставленных элементов (вернее, копирование ссылок). На профессиональном жаргоне: вставка в `ArrayList` является операцией с высокой скоростью передачи (имеет постоянное время в долгосрочном периоде), страдающей от высокого разброса (изменчивости по времени). Напротив, вставка в `LinkedList` обладает аналогичной скоростью передачи, но практически без разброса, потому что каждая вставка занимает одинаковое время (необходимое для выделения памяти для нового узла в списке).

3.4. СРАВНЕНИЕ РЕАЛИЗАЦИЙ

В предыдущих разделах были разработаны три реализации резервуаров, которые оптимизируют скорость выполнения разными способами. Для оценки скорости выполнения был использован анализ сложности в наихудшем случае и амортизированной сложности. При анализе наихудшего случая мы рассмотрели один метод, для которого предположили худшие возможные входные данные, тогда как при амортизированном анализе мы задействовали разные методы в серии операций произвольной длины. В обоих случаях мы определили только порядок роста (подробнее в разделе 2.3). В этом есть как свои плюсы (упрощение сравнений), так и минусы (потеря связи с фактическим временем выполнения). Если

вы не склонны ничего принимать на веру, как и я, давайте попробуем экспериментально убедиться в том, что теоретические метрики скорости выполнения соответствуют фактическому времени выполнения.

3.4.1. Эксперименты

Начнем с простого эксперимента, в котором три реализации из этой главы будут сравниваться с эталонной реализацией `Reference` в следующем тестовом сценарии:

1. Создайте 20 000 резервуаров и добавьте воду в каждый резервуар ($20k$ вызовов конструктора и $20k$ вызовов `addWater`).
2. Соедините резервуары в 10 000 пар, добавьте воду в каждую пару и запросите объем в каждой паре ($10k$ `connectTo`, $10k$ `addWater` и $10k$ `getAmount`).
3. Соединяйте пары резервуаров, пока все резервуары не будут соединены в одну группу. После каждого соединения добавляйте воды и запрашивайте полученный объем ($10k$ `connectTo`, $10k$ `addWater` и $10k$ `getAmount`).

Я выбрал общее количество резервуаров методом проб и ошибок, чтобы время выполнения было достаточно большим для демонстрации различий между реализациями и достаточно малым для многократного выполнения эксперимента за короткий промежуток времени. В табл. 3.5 приведено время выполнения, полученное на моем компьютере. Как и ожидалось, все классы из этой главы существенно превосходят `Reference` по скорости выполнения — выигрыш составляет до двух порядков. В частности, наша лучшая попытка `Speed3` работает в 500 раз быстрее. С другой стороны, `Speed2` работает на порядок медленнее `Speed1` и `Speed3` (но при этом значительно быстрее `Reference`). Как упоминалось ранее, `Speed2` — довольно нетипичная реализация, которая имеет смысл только в том случае, если запросы `getAmount` редки по сравнению с другими операциями. К тестируемому сценарию это не относится.

Таблица 3.5. Первый эксперимент: время выполнения для сбалансированного сценария использования с 20 000 резервуаров

Версия	Время (мс)
Reference	2300
Speed1	26
Speed2	505
Speed3	6

Чтобы подтвердить эти наблюдения, можно выполнить измененный сценарий использования, в котором удалены все вызовы `getAmount`, кроме одного (в конце). Этот странный сценарий спроектирован так, чтобы создать наиболее

благоприятные условия для `Speed2`. При запуске эксперимента на моей машине были получены результаты, приведенные в табл. 3.6. Как видите, реализация `Speed2` теперь сопоставима по скорости выполнения со `Speed3`, тогда как на трех других реализациях изменения, по сути, не отразились. Это демонстрирует, что запрос `amount` является очень малозатратной операцией во всех остальных версиях. Второй эксперимент подтверждает, что реализация `Speed3` на практике является оптимально сбалансированным вариантом улучшения скорости выполнения.

Таблица 3.6. Второй эксперимент: время выполнения для сценария использования с 20 000 резервуаров и одним вызовом getAmount

Версия	Время (мс)
Reference	2300
Speed1	25
Speed2	4
Speed3	5

БЕНЧМАРКИНГ

Сравнение скорости выполнения программ, написанных на Java (или любом другом языке, выполняемым виртуальной машиной), требует особого внимания. И компилятор, и VM могут вносить в программу серьезные изменения, что может привести к значительным изменениям кода, показатели которого будут измеряться. Для примера возьмем две стандартные оптимизации:

- Компилятор может удалить некоторые строки кода, если придет к выводу, что они ни на что не влияют.
- VM может переключаться между интерпретацией байт-кода и компиляцией его в платформенный код (так называемая *JIT-компиляция*).

Существуют некоторые обходные решения для предотвращения таких оптимизаций:

- Проследите за тем, чтобы каждая операция имела видимый эффект (например, вывод на печать или в файл).
- Несколько раз выполните код, прежде чем переходить к хронометражу. *Холодные прогоны* побудят VM откомпилировать разделы, критичные по скорости выполнения, и результаты станут более содержательными.

Кроме того, в поставку Java входит стандартный бенчмарк-фреймворк JMH (Java Microbenching Harness), который предоставляет средства для более точного контроля над компилятором и оптимизациями VM.

3.4.2. Теория и практика

Стандартный анализ сложности наихудшего случая объявил три наши реализации фактически несравнимыми (табл. 3.7): каждая реализация в каких-то сценариях использования превосходит другие, но ни одна из них не будет оптимальной всегда. Точнее, `Speed1` работает быстрее всего, если соединения изменяются относительно редко, а операции добавления, удаления и получения объема воды выполняются часто. Реализация `Speed2` оптимальна, если вы постоянно добавляете новые соединения, а также часто добавляете и удаляете воду, но редко запрашиваете текущий уровень воды в каком-либо резервуаре. Наконец, `Speed3` выглядит как удачный компромисс, в котором все операции выполняются не особенно быстро и занимают практически одинаковое время. Соответственно, она хорошо подходит для сценариев, в которых у вас нет четкого представления о том, как клиенты будут использовать класс (профиль использования). Амортизированный анализ и эксперименты показывают, что `Speed3` действительно является самой быстрой версией во всех ситуациях, кроме аномальных (как во втором эксперименте).

Таблица 3.7. Временная сложность в наихудшем случае для трех реализаций из этой главы и эталонной реализации Reference из главы 2

Версия	getAmount	addWater	connectTo
Reference	$O(1)$	$O(n)$	$O(n)$
Speed1	$O(1)$	$O(1)$	$O(n)$
Speed2	$O(n)$	$O(1)$	$O(1)$
Speed3	$O(\log n)$	$O(\log n)$	$O(\log n)$

Это не означает, что анализ сложности в наихудшем случае следует отбросить. Он остается самой удобной формальной основой для сравнения алгоритмов решения изолированных одноразовых задач. Кроме того, к нему прилагается асимптотическая запись (O-большое и т. д.) — мощная абстракция, применимая ко всем разновидностям анализа, включая амортизированный анализ или анализ среднего случая.

АНАЛИЗ СРЕДНЕГО СЛУЧАЯ

Анализ среднего случая — еще одна разновидность анализа сложности. Вместо того чтобы сосредоточиться на наихудшем возможном входе для заданного алгоритма, он оценивает среднюю сложность по всем возможным вариантам входных данных (предполагается, что все они имеют равную вероятность).

Не стоит забывать уточнение «наихудший случай» и помнить, что сложные алгоритмы (такие, как нахождение объединения) могут тратить время на действия,

которые окупятся со временем. Их преимущества проявляются только с длинными сериями операций.

Из этой главы вы узнали, что при проектировании класса, поддерживающего взаимодействие с клиентами метода, может быть недостаточно анализа сложности каждого метода по отдельности. Во-первых, скорости выполнения разных методов могут зависеть друг от друга. Как и в нашем примере, вычислительную нагрузку можно переместить из одного метода в другой, то есть ускорить один метод за счет другого. Во-вторых, иногда ценой дополнительных временных затрат можно ускорить будущее выполнение одного или нескольких методов.

В первом случае (взаимодействие между методами) анализ сложности в сочетании с профилем использования может направить вас к лучшему решению. *Профиль использования* характеризует то, как клиенты взаимодействуют с классом. Типичная информация профиля включает относительную частоту и порядок вызова методов классов. По этой информации можно определить, какой метод наиболее критичен для выполнения и требует максимальной скорости.

Во втором сценарии амортизированный анализ (вроде того, который был проведен для `Speed3`) позволяет формально подтвердить полезность временных вложений в долгосрочной перспективе. Оба метода в основном требуют мозгового штурма, а не написания программ и их запуска. На практике самый простой (хотя и не самый быстрый) способ заключается в реализации и профилировании разных решений. Он же будет и самым точным, если рабочие условия будут сходны с теми, которые использовались для профилирования.

3.5. А ТЕПЕРЬ СОВСЕМ ДРУГОЕ

В этом разделе я применю методы повышения скорости выполнения, рассмотренные ранее, к другому примеру. Собственно, начиная с этой главы, каждая глава будет иметь следующую структуру:

1. Мы медленно и в подробностях разберем знакомый пример с резервуарами.
2. Я представлю другой пример, но задам только общее направление, а детали оставлю вам.
3. В конце каждой главы я приведу пару упражнений, которые помогут закрепить материал.

В этом разделе рассматривается следующая задача: нужно спроектировать класс `IntStats` для вычисления обобщающей статистики по списку целых чисел. Класс должен предоставлять три открытых метода[1]:

[1] В Java 8 существует похожий класс `IntSummaryStatistics`, но он не вычисляет медиану.

- `public void insert(int n)` — добавляет целые числа в список в произвольном порядке;
- `public double getAverage()` — возвращает среднее арифметическое целых чисел, вставленных к текущему моменту;
- `public double getMedian()` — возвращает медиану целых чисел, вставленных к текущему моменту. *Медианой* называется значение, лежащее в середине упорядоченной последовательности целых чисел. Например, медианой последовательности 2; 10; 11; 20; 100 является 11. Медиана четного количества чисел определяется как среднее арифметическое двух центральных элементов последовательности. Например, медиана последовательности 2; 10; 11; 20 равна 10,5. Медиана последовательности целых чисел может быть вещественным числом.

При этом необходимо учесть три требования к скорости выполнения, описанные в следующих подразделах.

3.5.1. Быстрая вставка

Спроектируйте класс `IntStats`, чтобы операции `insert` и `getAverage` выполнялись за постоянное время.

Следующая реализация обеспечивает вставку и вычисление среднего за постоянное время. Для простоты определение медианы начинается с сортировки списка, поэтому оно требует квазилинейного времени $O(n \log n)$. `getMedian` можно реализовать за линейное время, но такие алгоритмы выходят за рамки книги[1].

```
public class IntStats {
   private long sum;  ❶ Текущая сумма всех целых чисел
   private List<Integer> numbers = new ArrayList<>();

   public void insert(int n) {
      numbers.add(n);
      sum += n;
   }

   public double getAverage() {
      return sum / (double) numbers.size();
   }

   public double getMedian() {
      Collections.sort(numbers);     ❷ Библиотечный метод сортировки списка
      final int size = numbers.size();
      if (size \% 2 == 1) {          ❸ Нечетный размер
         return numbers.get(size/2);
```

[1] Поищите информацию в интернете или в книгах по алгоритмам из раздела «Дополнительная литература» в конце главы.

```
      } else {                          ❹ Четный размер
         return (numbers.get(size/2 -1) + numbers.get(size/2)) / 2.0;
      }
   }
}
```

3.5.2. Быстрые запросы

Спроектируйте класс `IntStats`, чтобы операции `getAverage` и `getMedian` выполнялись за постоянное время.

Вычислительную нагрузку вы сможете вынести из `getMedian` в `insert`, сортируя последовательность после каждой вставки. Как следствие, время вставки возрастет от постоянного до квазилинейного $O(n \log n)$.

Чуть более интересное решение заключается в ведении отсортированного списка, в котором новые числа сразу вставляются в правильной позиции:

```
public void insert(int n) {
   int i = 0;
   for (Integer k: numbers) {
      if (k >= n) break;  ❶ Остановиться на первом числе, большем либо равным n
      i++;
   }
   numbers.add(i, n);     ❷ Вставить n в позиции i
   sum += n;
}
```

Два других метода могут остаться такими же, как в предыдущей версии, не считая того, что `getMedian` не нужно выполнять сортировку, потому что последовательность уже отсортирована. Так `insert` выполнится за линейное время, тогда как `getAverage` и `getMedian` достаточно постоянного времени.

Наконец, переключившись со списка на сбалансированное дерево (похожее на `TreeSet`), вы сможете понизить сложность `insert` от линейной до логарифмической.

3.5.3. Все операции

Спроектируйте класс `IntStats`, чтобы *все* три открытых метода выполнялись за постоянное время.

К сожалению, это невозможно. Более того, невозможно даже реализовать `insert` и `getMedian` за постоянное время. Определение медианы за постоянное время требует, чтобы она всегда была актуальной. Следовательно, каждый вызов `insert` должен обновлять медиану, а это, в свою очередь, требует нахождения следующего большего или меньшего элемента. Для этого можно воспользоваться простым поиском с линейным временем или структурой данных, поддерживающей целые

числа в порядке, как описано в разделе «Быстрые запросы». В обоих случаях вставка уже не выполнится с постоянным временем.

Формально можно доказать, что если такая структура данных существует, произвольные данные можно сортировать за линейное время, а это, как известно, невозможно.

3.6. РЕАЛЬНЫЕ СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

Рассуждения, которые я привожу в этой главе, пригодятся во многих приложениях, критичных по скорости выполнения. Несколько рекомендаций:

- Эффективность по времени стоит учитывать при работе с современными алгоритмами машинного обучения. Процесс обучения модели включает: (1) большие объемы данных, (2) эксперименты, пробы и ошибки для определения оптимальной модели. Таращиться в монитор, пока модель работает над схождением решения, неинтересно и непродуктивно. Популярные фреймворки глубокого обучения используют современные компьютерные архитектуры, для чего операции выражаются в виде модели, функционирующей как вычислительный граф в процессе обучения. Эти графы распределяются между процессорами для параллельного выполнения.
- Даже если вы думаете, что сможете обойтись субоптимальной системой (например, медлительной моделью глубокого обучения), там, где важна скорость реакции, могут возникнуть проблемы. Если результат запроса на поиск книг об алгоритмах в интернет-магазине будет возвращен только через 10 минут, скорее всего, вы начнете искать в другом месте. Собственно, медленная система может оказаться менее популярным вариантом, даже если генерируемые ею рекомендации намного более релевантны. (На практике почти всегда существует компромисс между степенью точности и эффективностью по времени.)
- Существуют и другие случаи, в которых эффективность по времени может немедленно повысить прибыль компании, для которой вы работаете. Высокочастотные торговые операции с финансовыми продуктами выполняются буквально за микросекунды, а эффективные алгоритмы с чрезвычайно низкими характеристиками задержки не просто желательны, а необходимы. Как нетрудно представить, высокочастотные торговые операции выполняются автоматически, а торговля на скорости вдвое меньшей, чем у конкурента, вряд ли приведет компанию к счастливому будущему.
- Плохо спроектированная система высокочастотной торговли, которая может вытеснить из бизнеса нескольких сотрудников, — печальное событие, но последствия сбоя системы реального времени могут быть катастрофическими. Система реального времени проектируется для реакции на физические про-

цессы, а эффективность по времени становится ограничением: либо система работает в рамках заданных временных границ, либо она вообще не считается работоспособной.

В энергосетях система автоматического регулирования процесса производства энергии отправляет управляющие сигналы для регулирования выходной мощности электростанций, направленного на соблюдение баланса генерирования нагрузки и потребления. Несвоевременная выдача сигнала может привести к катастрофическим последствиям, например отключениям электричества.

3.7. ПРИМЕНИМ ИЗУЧЕННОЕ

Рассмотрим аспекты функциональности, которые можно добавить к резервуарам:

- `groupSize` — метод экземпляра, который не получает параметров и возвращает количество резервуаров, соединенных с текущим прямо или косвенно;
- `flush` — метод экземпляра, который не получает параметров и не имеет возвращаемого значения. Метод опустошает все резервуары, соединенные с текущим прямо или косвенно.

Упражнение 1

Добавьте метод `groupSize` в три реализации из этой главы без добавления новых полей или изменения других методов.

1. Какой будет асимптотическая сложность наихудшего случая во всех трех вариантах?
2. Сможете ли вы изменить `Speed2` так, чтобы выполнение `groupSize` занимало постоянное время без повышения асимптотической сложности других методов?

Упражнение 2

Добавьте метод `flush` в три реализации резервуаров из этой главы без изменения других методов.

1. Какой будет асимптотическая сложность наихудшего случая во всех трех вариантах?
2. Сможете ли вы изменить `Speed2` так, чтобы выполнение `flush` занимало постоянное время без повышения асимптотической сложности других методов?

Упражнение 3 (мини-проект)

1. Спроектируйте два класса `Grid` и `Appliance`, представляющие электросеть и устройство, которое может питаться от сети. Каждая сеть (или устройство) характеризуется максимальной поставляемой (или потребляемой) мощностью. Устройство подключается к сети методом `plugInto`, а для его включения и отключения используются методы экземпляра `on` и `off`. (В исходном состоянии любое новое устройство отключено.) Подключение устройства к другой сети автоматически отключает его от первого. Если включение устройства приводит к перегрузке сети, метод `on` должен выдать исключение. Наконец, метод `residualPower` класса `Grid` возвращает оставшуюся свободную мощность в сети.

   Убедитесь, что решение будет работать со следующим сценарием использования:

```
Appliance tv = new Appliance(150), radio = new Appliance(30);
Grid grid = new Grid(3000);

tv.plugInto(grid);
radio.plugInto(grid);
System.out.println(grid.residualPower());
tv.on();
System.out.println(grid.residualPower());
radio.on();
System.out.println(grid.residualPower());
```

   В этом сценарии должен выводиться следующий результат:

```
3000
2850
2820
```

2. Сможете ли вы спроектировать эти два класса, чтобы все их методы выполнялись за постоянное время?

Упражнение 4

1. Если `ArrayList` будет увеличивать массив на 10 % при заполнении, останется ли амортизированная сложность `add` постоянной?
2. В этом случае изменение размера будет выполняться чаще. Насколько именно?

ИТОГИ

- Существуют разные способы оптимизации класса.
- Самые затратные вычисления, которые могут понадобиться классу, можно перемещать в соответствии с профилем использования.

- Циклический связанный список — удобная структура данных для слияния двух последовательностей, начиная с произвольных элементов.
- Деревья указателей на родителей — структура данных, хорошо подходящая для нахождения объединений.
- Амортизированный анализ — формальный способ описания средней скорости выполнения класса на основании длинной серии операций.

ОТВЕТЫ НА ВОПРОСЫ И УПРАЖНЕНИЯ

Неожиданный вопрос 1

Между парами классов в Java-программах существуют два частичных порядка: (1) является подклассом, (2) является внутренним классом.

Неожиданный вопрос 2

Удаление заданного узла из односвязного циклического списка занимает линейное время $O(n)$. Начиная от этого узла, необходимо обойти весь список до предшественника удаляемого узла. Тогда указатель `next` предшественника обновится в обход удаляемого узла.

Неожиданный вопрос 3

Я очень легко найду две задачи, которые я бы предпочел выполнить как можно позднее: мытье машины и посещение стоматолога. Сложнее найти задачи, за которые мне бы хотелось взяться как можно скорее.

Неожиданный вопрос 4

Дерево указателей на родителей будет более уместным. Для компиляции класса необходимо знать его непосредственный суперкласс. Например, в каждом конструкторе компилятор вставляет вызов конструктора суперкласса. Но чтобы откомпилировать заданный класс, знать его подклассы не обязательно.

Упражнение 1

1. Для `Speed1` (постоянное время):

```
public int groupSize() {
   return group.members.size();
}
```

Для Speed2 (линейное время):

```
public int groupSize() {
   int size = 0;
   Container current = this;
   do {
      size++;
      current = current.next;
   } while (current != this);
   return size;
}
```

Для Speed3 (логарифмическое время в наихудшем случае, постоянное амортизированное время):

```
public int groupSize() {
   Container root = findRootAndCompress();
   return root.size;
}
```

2. Что касается второй части упражнения, временная сложность groupSize легко улучшается добавлением поля экземпляра groupSize, которое обновляется во время выполнения connectTo.

Упражнение 2

1. Для Speed1 (постоянное время):

```
public void flush() {
   group.amountPerContainer = 0;
}
```

Для Speed2 (линейное время):

```
public void flush() {
   Container current = this;
   do {
      current.amount = 0;
      current = current.next;
   } while (current != this);
}
```

Для Speed3 (логарифмическое время в наихудшем случае, постоянное амортизированное время):

```
public void flush() {
   Container root = findRootAndCompress();
   root.amount = 0;
}
```

2. Чтобы добиться выполнения flush с постоянным временем в Speed2 без повышения сложности другого метода, flush должен осуществить отложенную пометку текущего резервуара, которая закодирует факт очищения группы. Но за этим событием могут последовать другие вызовы addWater и connectTo и даже другие действия flush, и специальная пометка может превратиться в сложную историю событий, произошедших с резервуаром с момента последнего вызова getAmount. То есть реализация flush с постоянным временем (а следовательно, локальная) требует хранения в каждом резервуаре неограниченной истории событий, которую нужно воспроизводить при вызове getAmount, что повышает сложность операции выше линейной.

Упражнение 3

1 и 2. Все операции можно выполнить за постоянное время. Для этого в Grid сохраняется остаточная мощность, которая постоянно обновляется при всех операциях. Следует заметить, что сетям не нужно располагать информацией о своих устройствах. Достаточно каждому устройству хранить ссылку на сеть, к которой оно подключено в данный момент, или null, если оно еще не подключено. В итоге получаем следующую структуру для Grid:

```
public class Grid {
   private final int maxPower;
   private int residualPower;
   ...
```

и для Appliance:

```
public class Appliance {
   private final int powerAbsorbed;
   private Grid grid;
   private boolean isOn;
   ...
```

Устройства должны каким-то образом обновлять остаточную мощность своей сети при включении и выключении. Вместо того чтобы обращаться к полю residualPower напрямую, лучше воспользоваться методом Grid, который выдаст нужное исключение при перегрузке сети:

```
void addPower(int power) {
   if (residualPower + power < 0)
      throw new IllegalArgumentException("Not enough power.");
   if (residualPower + power > maxPower)
      throw new IllegalArgumentException("Maximum power exceeded.");
   residualPower += power;
}
```

Этот метод должен быть доступным только для устройств, но такое разграничение доступа невозможно в Java, если Grid и Appliance являются разными классами верхнего уровня. Чтобы частично скрыть этот метод, можно поместить два

класса в отдельный пакет и предоставить пакетную (по умолчанию) видимость `addPower`, как это сделал я.

Я решил выдавать исключение `IllegalArgumentException` при перегрузке сети, хотя `IllegalStateException` описывает ситуацию ничуть не хуже. В самом деле, ошибочное состояние обусловлено несовместимостью значения аргумента (`power`) с текущим состоянием поля объекта (`residualPower`). В таких случаях Джошуа Блох рекомендует выдавать исключение `IllegalArgumentException` («Java. Эффективное программирование», пункт 72) и использовать `IllegalStateException` только в том случае, если никакое другое значение аргумента не подойдет.

Полный код классов `Appliance` и `Grid` доступен в репозитории (https://bitbucket.org/mfaella/exercisesinstyle).

Упражнение 4

1. Да. Увеличение массива на *любой* процент, включая незначительные 10 %, приводит к постоянной амортизированной сложности вставок. Чтобы доказать это, достаточно заменить множитель 1,5, использовавшийся в вычислениях из раздела 3.3.5, другим множителем, например 1,1 для 10 %.

   Правильный выбор процента помогает выдержать баланс между затратами времени и памяти. Чем меньше процент, тем больше будет константа, а следовательно, тем меньше эффективность по времени. С другой стороны, более консервативный процент экономит память, потому что емкость `ArrayList` в общем случае будет ближе к размеру.

2. Как я уже говорил, увеличение с множителем f приводит к $\log_f \frac{n}{10}$ повторных выделений при выполнении n вставок. Соответственно, увеличение на 10 % вместо 50 % приводит к следующему возрастанию числа повторных выделений памяти:

$$\frac{\log_{1,1} \frac{n}{10}}{\log_{1,5} \frac{n}{10}} = \log_{1,1} 1,5 \approx 4,25.$$

   При увеличении на 10 % количество увеличений возрастает в 4,25 раза по сравнению с увеличением на 50 %.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Есть несколько книг о стандартных алгоритмах, в которых рассматриваются задачи нахождения объединений и вычисления амортизированной сложности.

- *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ. 3-е изд. — М.; СПб: Диалектика, 2020.
- *Kleinberg J. and Tardos E.* Algorithm Design. Pearson, 2005.
- После краткого обзора алгоритмов нахождения объединений Кевин Уэйн (Kevin Wayne) из Принстонского университета создал слайды по материалам книги «Algorithm Design». Вы легко найдете их в интернете.

Я не рассматривал приемы улучшения скорости выполнения, специфичные для Java. Вместо этого я сосредоточился на высокоуровневых алгоритмических принципах, не зависящих от языка. Книга Скотта Оукса поможет заполнить этот пробел: в ней описаны возможности настройки VM для потребностей конкретного приложения.

Значительная часть книги посвящена сбору мусора, так как в этой области есть несколько конкурирующих алгоритмов с разными профилями скорости выполнения, ни один из которых не является универсальным. Кроме того, в книге обсуждаются разнообразные средства мониторинга и профилирования, доступные в Java:

- *Oaks S.* Java Performance: The Definitive Guide. O'Reilly Media, 2014.

4 Эффективность по затратам памяти

В этой главе:

- ✓ Написание классов, эффективных по затратам памяти.
- ✓ Сравнение требований к памяти для основных структур данных, включая массивы, списки и множества.
- ✓ Оценка компромиссов между скоростью выполнения и потреблением памяти.
- ✓ Использование локальности в памяти для ускорения выполнения.

Иногда программистам требуется хранить данные в минимально возможном объеме памяти. Однако это редко происходит из-за недостаточного объема памяти устройства, для которого предназначена программа, или из-за большого объема данных. Например, видеоигры часто приближаются к пределам возможностей оборудования. Сколько бы гигабайтов памяти ни было установлено на каждой следующей консоли, вскоре играм ее будет не хватать и вы начнете упаковывать данные всевозможными странными способами.

В этой главе предполагается, что программа для управления резервуарами будет иметь дело с миллионами, а может быть, даже миллиардами резервуаров, и как можно большее их количество требуется уместить в основной памяти. Разумеется, потребление памяти для каждого резервуара должно быть сведено к минимуму. Но вам не придется беспокоиться о памяти, используемой

временными локальными переменными, потому что они существуют только в течение короткого периода вызова метода.

Для каждой реализации в этой главе потребление памяти сравнивается с показателями `Reference` (глава 2). А пока вспомните, какие поля используются в этом классе:

```
public class Container {
   private Set<Container> group;   ❶ Резервуары, соединенные с текущим
   private double amount;          ❷ Объем воды в резервуаре
```

4.1. ПЕРВЫЕ ШАГИ [MEMORY1]

Даже несколько простых приемов позволят превзойти показатели `Reference`. Во-первых, маловероятно, чтобы для представления объема воды в резервуаре понадобится точность или диапазон чисел с двойной точностью, так что можно сэкономить 4 байта на резервуар, уменьшив размер поля `amount` с `double` на `float`. Аргумент `addWater` и возвращаемый тип `getAmount` тоже необходимо изменить соответствующим образом, то есть слегка изменить открытый API. Обратите внимание: полученный класс все еще остается на 100 % совместимым со сценарием использования `UseCase` (глава 1), потому что сценарий передает объем воды в виде целых чисел, а целочисленные аргументы совместимы с параметрами как `float`, так и `double`.

> **ТИПЫ ДАННЫХ ДЛЯ ЭКОНОМИИ ПАМЯТИ**
>
> Типы данных уменьшенного размера играют ограниченную роль в основном API Java, но они хорошо поддерживаются в более специализированных контекстах, в которых возможна нехватка памяти. Например, Android предоставляет класс `FloatMath` для выполнения основных математических операций с `float` вместо `double`. Кроме того, в спецификации Java для смарт-карт (Java Card) многие целые числа, используемые в API, кодируются типом `short` или `byte`.

НЕОЖИДАННЫЙ ВОПРОС 1

Если ваша программа содержит 10 вхождений строкового литерала `Hello World`, сколько памяти будет расходоваться на хранение этих строк?

Что касается поля `group`, в эталонной реализации для него был выбран тип `Set`, потому что он ясно выражает, что группы не упорядочены и не содержат дубликатов. Отказываясь от четко выраженного намерения и переключаясь на `ArrayList`, можно сэкономить значительный объем памяти. В конце

концов, `ArrayList` представляет собой тонкую обертку для обычного массива, так что общие затраты памяти для дополнительного элемента составят всего 4 байта. Новый класс `Container`, который называется `Memory1`, будет начинаться так:

Листинг 4.1. Memory1: поля и методы getAmount, конструктор не нужен

```
public class Container {
   private List<Container> group; ❶ Будет инициализироваться ArrayList
   private float amount;

   public float getAmount() { return amount; }
```

Кроме того, если многие резервуары никогда не соединяются с группой, можно сэкономить место за счет создания списка только в тот момент, когда в нем возникнет реальная необходимость (*отложенная инициализация*). Другими словами, если поле `group` содержит `null`, значит, объект представляет изолированный резервуар. Этот вариант позволяет не включать явный конструктор, но методам `connectTo` и `addWater` придется рассматривать изолированные резервуары как особые случаи.

В общем случае при переходе с `Set` на `List` необходимо действовать осторожно, потому что вы теряете возможность автоматически исключать дубликаты. В `Reference` эта возможность не использовалась, потому что группы, объединяемые методом `connectTo`, гарантированно были разъединены в исходном состоянии. Реализация `connectTo`, которую вы в итоге получите:

Листинг 4.2. Memory1: метод connectTo

```
public void connectTo(Container other) {
   if (group==null) { ❶ Если резервуар изолирован, инициализировать его группу
       group = new ArrayList<>();
       group.add(this);
   }
   if (other.group==null) { ❷ Если резервуар other изолирован, инициализировать его группу
       other.group = new ArrayList<>();
       other.group.add(other);
   }
   if (group==other.group) return; ❸ Проверить, соединены ли резервуары

   int size1 = group.size(),  ❹ Вычислить новый объем воды
       size2 = other.group.size();
   float tot1 = amount * size1,
         tot2 = other.amount * size2,
         newAmount = (tot1 + tot2) / (size1 + size2);

   group.addAll(other.group);  ❺ Объединить две группы
   for (Container x: other.group) { x.group = group; }
   for (Container x: group) { x.amount = newAmount; }
}
```

Наконец, метод `addWater` также должен учитывать особый случай изолированного резервуара для предотвращения разыменования `null`-указателя:

Листинг 4.3. Memory1: метод addWater

```
public void addWater(double amount) {
   if (group==null) {     ❶ Если резервуар изолирован, обновить локально
      this.amount += amount;
   } else {
      double amountPerContainer = amount / group.size();
      for (Container c: group) {
         c.amount += amountPerContainer;
      }
   }
}
```

В завершение этого раздела рассмотрим распределение памяти этой реализации. Как обычно, предполагается, что вы выполнили первые три части основного сценария использования `UseCase`, в которых создаются четыре резервуара (от `a` до `d`) и выполняются следующие строки:

```
a.addWater(12);
d.addWater(8);
a.connectTo(b);
b.connectTo(c);
```

Сценарий изображен на рис. 4.1, а соответствующее распределение памяти в `Memory1` — на рис. 4.2. Оно очень похоже на `Reference`, не считая того что вместо `HashSet` используется `ArrayList`, а также значения `null` в резервуаре `d` (вместо ссылки на коллекцию `HashSet` с одним элементом). Третье отличие от `Reference` — использование типа `float` вместо `double` для хранения объема воды (на диаграмме не обозначено).

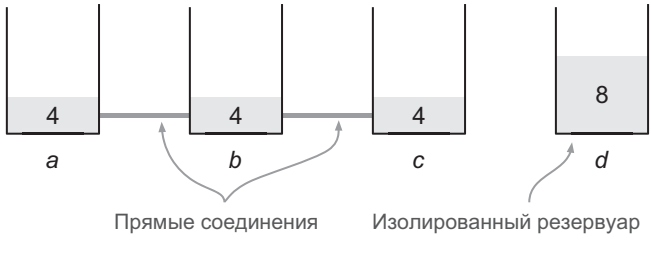


Рис. 4.1. Ситуация после выполнения первых трех частей основного сценария. Резервуары от *a* до *c* соединены, а в *a* и *d* наливается вода

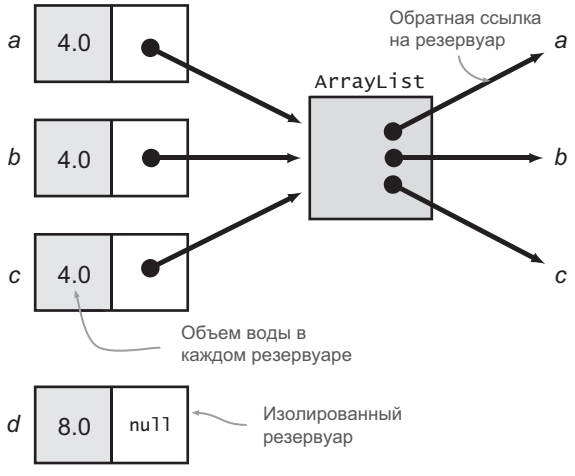


Рис. 4.2. Распределение памяти реализации Memory1 после выполнения первых трех частей основного сценария

4.1.1. Временная сложность и затраты памяти

Чтобы оценить потребление памяти в `Memory1`, начнем с оценки размера коллекции `ArrayList`, во внутренней реализации которой используется массив и пара служебных полей. Длина внутреннего массива называется *емкостью* `ArrayList` (не путайте с размером — количеством элементов, уже хранящихся в коллекции). Требования к памяти `ArrayList` таковы:

- 12 байт — стандартный заголовок объекта;
- 4 байта — целочисленное поле с количеством структурных изменений (вставок и удалений), выполнявшихся со списком (это поле используется для выдачи исключения при изменении списка во время итерации);
- 4 байта — целочисленное поле размера;
- 4 байта — ссылка на массив;
- 16 байт — стандартная память массива;
- 4 байта — каждая ячейка массива.

Так как коллекция `ArrayList` с n элементами содержит как минимум n ячеек массива, она занимает не менее $40 + 4n$ байт (рис. 4.3).

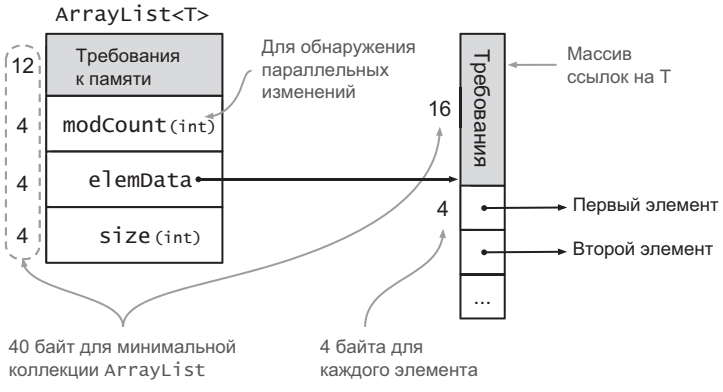


Рис. 4.3. Распределение памяти ArrayList

На практике емкость коллекции `ArrayList` часто превышает ее размер. При добавлении дополнительного элемента в заполненную коллекцию `ArrayList` класс создает массив большего размера и копирует старое содержимое в новый массив. Для общего ускорения выполнения в увеличенном массиве должно присутствовать свободное место — например, он не должен увеличиться только на одну ячейку. Как я уже говорил (раздел 3.3.5), фактическая емкость возрастает на 50 %. В результате в любой момент времени емкость коллекции `ArrayList` составляет от 100 до 150 % от ее размера. В следующих оценках предполагается среднее значение 125 %.

Наши результаты должны примерно соответствовать оценкам в табл. 4.1. Изолированный резервуар (первый сценарий) не содержит `ArrayList`. Память занимают только сами объекты резервуаров: сумма 4 байт для ссылки на группу (содержит `null`), 4 байт для поля `amount` и 12-байтового заголовка объекта. Если упорядочить резервуары в 100 групп по 10 (второй сценарий), то к затратам памяти 1000 объектов резервуаров прибавятся затраты 100 коллекций `ArrayList`.

Таблица 4.1. Требования к памяти для Memory1

Сценарий	Размер (вычисление)	Размер (в байтах)	Процент от Reference
1000 изолированных	1000 × (12 + 4 + 4)	20 000	19 %
100 групп по 10	1000 × (12 + 4 + 4) + 100 × × (40 + 10 × 1,25 × 4)	29 000	47 %

Как видно из табл. 4.1, несколько простых изменений `Reference` позволяют сэкономить значительный объем памяти. В частности, идея выделения памяти для списков, когда они впервые понадобятся, обеспечивает наибольшую

экономию в первом сценарии, когда все резервуары изолированы (и ни один список не создается). 50 %-ная экономия во втором сценарии обусловлена заменой `HashSet` на `ArrayList`.

Обратите внимание: экономия памяти, достигнутая в этом разделе, практически не отражается на скорости выполнения, потому что три операции имеют ту же сложность, что и в реализации `Reference` (табл. 4.2). Но по сравнению с `Reference` класс читается хуже. Во-первых, объявление поля `group` с типом `List` скрывает, что группы в действительности представляют собой неупорядоченные коллекции без дубликатов. Во-вторых, обработка изолированных резервуаров как особых случаев становится избыточным усложнением ради экономии памяти.

Таблица 4.2. Временные сложности для Memory1, где n обозначает общее количество резервуаров; совпадают со сложностями Reference

Метод	Временная сложность
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$O(n)$

Высокие дополнительные затраты памяти `HashSet` и других стандартных коллекций — хорошо известный факт, поэтому некоторые фреймворки предоставляют их альтернативы. Например, класс `SparseArray` в Android — это карта с целочисленными ключами и ссылочными значениями — реализуется на базе двух массивов одинаковой длины: в первом хранятся ключи, упорядоченные по возрастанию, а во втором — соответствующие значения. В такой структуре данных за улучшенную эффективность расходования памяти приходится платить ухудшением временной сложности: поиск значения, соответствующего заданному ключу, требует бинарного поиска по массиву ключей, что, в свою очередь, требует логарифмического времени. В упражнении 2 в конце главы вам будет предложено продолжить анализ класса `SparseArray`.

Если требуется хранить только примитивные значения, некоторые библиотеки (например, GNU Trove — https://bitbucket.org/trove4j/trove) предоставляют специализированные реализации множеств и карт, которые избегают упаковки каждого значения в соответствующий класс.

4.2. ПРОСТЫЕ МАССИВЫ

Во второй реализации для экономии памяти `Memory2` мы заменим коллекцию `ArrayList`, представляющую группу, простым массивом, длина которого в точности равна размеру группы. Начнем так:

Листинг 4.4. Memory2: поля и методы getAmount, конструктор не нужен

```
public class Container {
   private Container[] group;
   private float amount;

   public float getAmount() { return amount; }
```

Как и в `Memory1`, память для массива `group` можно выделять только по необходимости, то есть когда этот резервуар соединен как минимум с одним другим резервуаром (рис. 4.4).

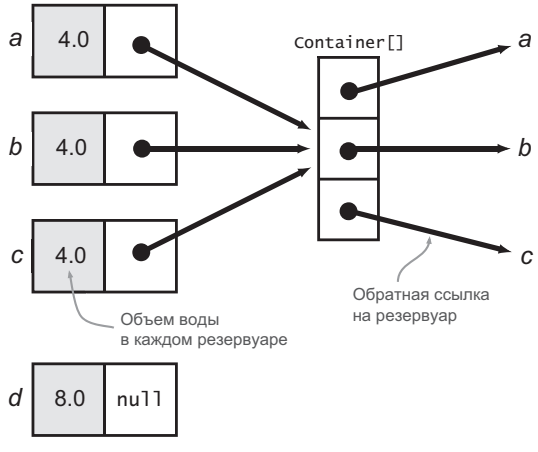


Рис. 4.4. Распределение памяти в реализации Memory2 после выполнения первых трех частей основного сценария

Метод `connectTo` очень похож на одноименный метод из `Memory1`, хотя его код выглядит чуть более громоздко из-за низкого уровня абстракции. Например, слияние двух коллекций `ArrayList` сводится к простому вызову `addAll`, тогда как слияние двух массивов требует повторного выделения памяти для одного из них и последующего перебора другого.

Листинг 4.5. Memory2: метод connectTo

```
public void connectTo(Container other) {
   if (group==null) {   ❶ Если резервуар изолирован, инициализировать его группу
      group = new Container[] { this };
   }

   if (other.group==null) {                    ❷ Если резервуар other изолирован,
      other.group = new Container[] { other };    инициализировать его группу
   }
   if (group == other.group) return;           ❸ Проверить, соединены ли резервуары
```

```
   int size1 = group.length,                          ❹ Вычислить новый объем воды
       size2 = other.group.length;
   float amount1 = amount * size1,
         amount2 = other.amount * size2,
         newAmount = (amount1 + amount2) / (size1 + size2);

   Container[] newGroup = new Container[size1 + size2]; ❺ Выделить память
                                                          для новой группы
   int i=0;
   for (Container a: group) {   ❻ Для каждого резервуара в 1-й группе...
      a.group = newGroup;       ❼ ...обновить его группу
      a.amount = newAmount;     ❽ ...обновить его объем
      newGroup[i++] = a;        ❾ ...и присоединить к newGroup
   }
   for (Container b: other.group) { ❿ Сделать то же для 2-й группы
      b.group = newGroup;
      b.amount = newAmount;
      newGroup[i++] = b;
   }
}
```

Наконец, метод `addWater` почти идентичен одноименному методу из `Memory1`, только тип переменных для объема воды в нем `float` вместо `double`:

Листинг 4.6. Memory2: метод addWater

```
public void addWater(float amount) {
   if (group==null) {
      this.amount += amount;
   } else {
      float amountPerContainer = amount / group.length;
      for (Container c: group) {
         c.amount += amountPerContainer;
      }
   }
}
```

4.2.1. Временная сложность и затраты памяти

Простой массив, содержащий ссылки на n резервуаров, занимает $16 + 4n$ байт, что ведет к оценкам в табл. 4.3 для стандартных сценариев. Изолированный резервуар (первый сценарий) не выделяет массив `group`, а его потребление памяти, как и в `Memory1`, — 20 байт. Если разделить резервуары на 100 групп по 10 (второй сценарий), каждая группа будет массивом из 10 ячеек. Служебная информация массива займет 16 байт, а его реальное содержимое — 4×10 байт.

К сожалению, экономия памяти, достигаемая с `Memory2`, незначительна по сравнению с `Memory1`. Изолированные резервуары занимают одинаковый объем памяти, а группы резервуаров, представленные массивами, лишь немногим более компактны по сравнению с `ArrayList`. Основная экономия возникает при

использовании *плотных* массивов, длина которых в точности соответствует потребности (вместо относительно свободного хранения в `ArrayList`).

Таблица 4.3. Требования к памяти для Memory2

Сценарий	Размер (вычисление)	Размер (в байтах)	Процент от Reference
1000 изолированных	1000 × (12 + 4 + 4)	20 000	19 %
100 групп по 10	1000 × (12 + 4 + 4) + 100 × × (16 + 10 × 4)	25 600	42 %

Самое время вспомнить и сравнить требования к памяти для трех стандартных структур данных, которые использовались для представления групп резервуаров: `HashSet` (`Reference` и `Speed1`), `ArrayList` (`Memory1`) и простые массивы (`Memory2`). Я не включаю в сравнение классы `Speed2` и `Speed3`, потому что они базируются на специализированных структурах данных, недоступных для общего использования.

В табл. 4.4 приведена сводка этих требований к памяти. Оценка размера для простого массива ссылок на объекты вычисляется легко: 16 байт служебной информации и 4 байта для каждой ссылки. Подробный анализ размера `ArrayList` приведен в разделе 4.1.1. Предполагается, что емкость равна размеру. (На практике емкость может превышать размер не более чем на 50 %.) Аналогичные упрощающие предположения применялись при анализе размера `HashSet` в разделе 2.2.1.

Таблица 4.4. Требования к памяти для распространенных коллекций (предполагается, что емкость ArrayList и HashSet равна их размеру). Второй столбец снабжен пометкой «минимум» вместо «пустой», потому что он не учитывает исходную емкость этой коллекции по умолчанию

Тип	Размер (минимум)	Размер (каждый дополнительный элемент)
Массив	16	4
`ArrayList`	40	4
`LinkedList`	24	24
`HashSet`	52	32

Как видно из таблицы, массив и `ArrayList` очень близки в отношении требований к памяти, но коллекция `ArrayList` более полезна благодаря автоматическому изменению размера и другим вспомогательным функциям. Кроме того, код с `ArrayList` более удобочитаем благодаря высокому уровню абстракции. А еще `ArrayList` лучше взаимодействует с обобщениями, чем массив.

НЕОЖИДАННЫЙ ВОПРОС 2

Для параметра типа `T` почему `new T[10]` не является допустимым выражением Java?

`HashSet` принадлежит к другой, более громоздкой категории из-за упаковывания каждого вставляемого элемента в новый объект. Тем не менее эта коллекция предоставляет уникальные возможности за постоянное время:

- проверку принадлежности (метод `contains`);
- исключение дубликатов (метод `add`);
- удаление произвольного элемента (метод `remove`).

Если вашему приложению необходимы эти функции, использование `HashSet` обычно более чем окупает повышенное потребление памяти (если она есть).

Что касается производительности, временная сложность `Memory2` оказывается такой же, как у `Memory1` и `Reference` (табл. 4.5). В конце концов, `Memory2` — разновидность `Memory1`, которая использует простые массивы вместо `ArrayList`.

Таблица 4.5. Временные сложности для Memory2, где n обозначает общее количество резервуаров; совпадают со сложностями Reference

Метод	Временная сложность
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$O(n)$

Почему умная политика изменения размера `ArrayList`, которая гарантирует амортизированное постоянное время вставки, не дает преимуществ в `Memory2`? Это объясняется тем, что выигрыш в скорости выполнения *обеспечивает* `connectTo`, но другие операции, выполняемые методом, этот выигрыш скрывают. Например, `Memory1` объединяет две группы строкой

```
group.addAll(other.group);
```

где `group` и `other.group` — две коллекции `ArrayList`. `Memory2` вместо этого выполняет строку

```
Container[] newGroup = new Container[size1 + size2];
```

Первый вариант в общем случае эффективнее второго, потому что дополнительной емкости первой коллекции `ArrayList` может быть достаточно для вставки *всех* элементов из *второй* коллекции `ArrayList` без выделения новой памяти. Тем не менее, обе версии `connectTo` затем перебирают все элементы из обеих старых групп. В результате в асимптотическом выражении последующие циклы

перекрывают более раннюю экономию, что в записи *O*-большое дает одну и ту же сложность $O(n)$ для обеих версий `connectTo`.

Существует еще одна причина предпочесть `Memory2`: эта реализация абсолютно автономна, в ней не задействованы никакие другие классы из Java API. В некоторых случаях это может быть полезно, потому что класс способен функционировать в крайне ограниченной исполнительной среде благодаря системе модулей в Java 9.

4.3. ОТКАЗ ОТ ОБЪЕКТОВ [MEMORY3]

Даже пустой объект Java занимает 12 байт, поэтому в сценарии использования из главы 1 резервуар ни при каких условиях не может занимать меньше памяти. Теперь допустим, что API можно изменить так, чтобы данные занимали минимально возможную память, но при этом сохранилась функциональность исходного класса. Попробуем получить текущий объем воды в резервуаре, изменить текущий объем и соединить два резервуара. Какой наименьший объем памяти нужен для хранения информации, реально необходимой для этой функциональности?

Чтобы заметно сократить потребление памяти, можно отказаться от создания объекта для каждого резервуара, но клиент все равно должен идентифицировать резервуары. Для этого мы предоставим клиенту *дескриптор* (handle) — значение, которое однозначно идентифицирует резервуар. Ссылка на объект резервуара — прекрасный дескриптор, но к ней прилагаются 12-байтовые затраты, которых нам хотелось бы *избежать*. Нужна альтернатива.

4.3.1. API без объектов

Вместо того чтобы создавать объект для каждого резервуара, позволим клиенту идентифицировать резервуар по целочисленному значению — дескриптору резервуара на стороне клиента. Необходимую информацию (объем воды и взаимные связи) нужно сохранить в подходящей структуре данных. Какая структура данных с ходу приходит в голову? Правильно, массивы.

Соответственно, вместо конструктора в класс будет включен статический метод, который возвращает идентификатор нового резервуара:

```
int id = Container.newContainer();
```

Тогда вместо того, чтобы вызвать `c.getAmount()` для объекта резервуара, клиент вызовет статический метод, которому передастся идентификатор резервуара:

```
float amount = Container.getAmount(id);
```

Аналогичным образом для соединения двух резервуаров клиенты передадут их идентификаторы статическому методу `connect`:

```
Container.connect(id1, id2);
```

Очевидно, что эта реализация противоречит всем ОО-канонам, но мы доведем наши предположения до предела.

Чтобы поближе познакомиться с полученным API, проверьте, как сценарий использования из главы 1 (`UseCase`) преобразуется при замене объектов резервуаров целочисленными идентификаторами. Вспомните первые строки `UseCase`:

```
Container a = new Container();
Container b = new Container();
Container c = new Container();
Container d = new Container();

a.addWater(12);
d.addWater(8);
a.connectTo(b);
System.out.println(a.getAmount() + " " + b.getAmount() + " " +
                   c.getAmount() + " " + d.getAmount());
```

При отказе от объектов резервуаров каждый резервуар идентифицируется целым числом, а приведенный фрагмент принимает следующий вид:

```
int a = Container.newContainer(), b = Container.newContainer(),
    c = Container.newContainer(), d = Container.newContainer();

Container.addWater(a, 12);
Container.addWater(d, 8);
Container.connect(a, b);
System.out.println(Container.getAmount(a) + " " +
                   Container.getAmount(b) + " " +
                   Container.getAmount(c) + " " +
                   Container.getAmount(d));
```

Хотя я рекомендую использовать простые массивы по соображениям эффективного расходования памяти, на практике вы скорее выберете их по соображениям *временной* эффективности, потому что массивы всегда обладают преимуществами локальности для *кэширования*. В двух словах, обращение к данным, расположенным близко в памяти (как в массиве), происходит быстрее, чем обращение к случайно распределенным данным (как в связанном списке). Это связано с организацией кэша процессора — буфера памяти, закрывающего разрыв скорости выполнения между процессором и основной памятью. Кэш хранит маленькие блоки смежных данных. Близкое хранение двух взаимосвязанных элементов данных в памяти повышает вероятность того, что при загрузке одного блока данных в кэш также будет записан второй элемент данных, что приведет к ускорению последующих операций. Например, так располагаются поля одного

объекта. А значит, обращение к одному полю с большой вероятностью ускорит обращения к остальным полям того же объекта.

ИЕРАРХИЯ ПАМЯТИ

Память современных компьютеров образует иерархию уровней, каждый из которых больше и медленнее уровня, расположенного над ним. На верхнем уровне располагаются регистры процессора, которые обычно занимают несколько сотен байтов. Регистры — единственный тип памяти, которая по скорости работы сопоставима со скоростью обработки данных процессором: запись и чтение в регистр могут выполняться на каждом такте процессора.

Под регистрами располагается кэш, разделенный на несколько уровней и занимающий несколько мегабайтов. Чтение из кэша верхнего уровня (то есть перемещение данных из кэша верхнего уровня в регистр) занимает несколько тактов, тогда как чтение непосредственно из основной памяти требует *сотен* тактов.

Кэш делится на *строки*, каждая из которых занимает несколько машинных слов. Каждый раз, когда программа адресует новую ячейку памяти, в кэш загружается полная строка, начинающаяся с адреса этой ячейки. Если ячейка является первым элементом массива, еще несколько элементов массива будут автоматически загружены в ту же строку кэша и будут готовы к быстрой загрузке в регистры по запросу.

Рассмотрите новую архитектуру AMD Zen для процессоров настольных систем. Кэш состоит из трех уровней, длина каждой строки составляет 64 байта. Ниже перечислены основные характеристики иерархии памяти (http://mng.bz/E1lX):

Уровень	Размер (на ядро)	Задержка (такты)
Регистры	128 байт	1
Кэш L1	32 Кбайт	4
Кэш L2	512 Кбайт	17
Кэш L3	2 Мбайт	40
Основная память	16 Гбайт (типично)	~300

НЕОЖИДАННЫЙ ВОПРОС 3

Если `set` имеет тип `HashSet`, стоит ли ожидать, что вызов `set.contains(x)` ускорит последующий вызов `set.contains(y)`? А если `set` имеет тип `TreeSet`?

В языке Java для хранения коллекции элементов способом, обеспечивающим локальности для кэширования, используются только массивы или классы, основанные на массивах (такие, как `ArrayList`). Однако в обобщенных коллекциях (таких, как `ArrayList`) могут храниться только ссылки, так что локальность для кэширования ограничивается только самими ссылками, но не данными, на которые они указывают. Например, `ArrayList<Integer>` содержит массив ссылок на `Integer`. Ссылки будут храниться в смежной области памяти, а сами целочисленные значения — нет. То же относится к простому массиву объектов `Integer` (рис. 4.5).

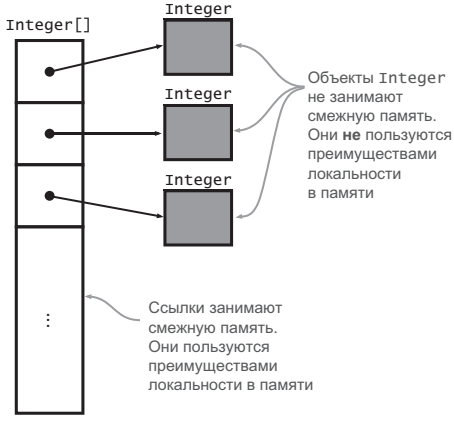


Рис. 4.5. Массив объектов Integer. Сам массив занимает непрерывный блок памяти, но объекты Integer, указатели на которые хранятся в массиве, разбросаны в памяти

Для объединения автоматического изменения размеров и других удобных возможностей `ArrayList` с локальностью для кэширования массива примитивных значений понадобятся внешние библиотеки, такие как GNU Trove. Например, класс GNU Trove `TIntArrayList` представляет массив примитивных целых чисел с изменяемым размером.

4.3.2. Поля и метод getAmount

Новый класс `Memory3` идентифицирует резервуары и группы резервуаров по целочисленным идентификаторам. Он использует два массива класса (статические поля) для кодирования необходимой информации:

- Массив `group` связывает идентификатор резервуара с идентификатором его группы.
- Массив `amount` связывает идентификатор группы с объемом воды в каждом резервуаре этой группы.

Для заданного идентификатора резервуара метод `getAmount` обращается к массиву `group`, чтобы получить идентификатор группы для этого резервуара, а затем к массиву `amount` за объемом воды в этой группе:

Листинг 4.7. Memory3: поля и метод getAmount

```
public class Container {
   private static int group[] = new int[0];   ❶ От идентификатора резервуара к его группе
   private static float amount[] = new float[0]; ❷ От идентификатора группы
                                                    к объему на резервуар

   public static float getAmount(int containerID) {
      int groupID = group[containerID];
      return amount[groupID];
}
```

Инициализация этих полей массивами нулевого размера выглядит странно, однако благодаря этому не нужно предусматривать особую обработку создания первого резервуара. Код становится более однородным. Всегда можно обращаться к массивам `group.length` и `amount.length`, потому что они никогда не равны `null`.

4.3.3. Создание резервуаров фабричным методом

Теперь займемся статическим методом `newContainer`, который заменяет конструктор. Метод, возвращающий новый экземпляр класса, часто называется *фабричным методом*. `newContainer` не создает экземпляр объекта, потому что так задумана реализация `Memory3`. Однако возвращая идентификатор нового резервуара, он выполняет функции фабричного метода.

Чтобы реализовать метод `newContainer`, следует учесть, что этот метод должен обновлять массивы при появлении нового резервуара, а затем возвращать его идентификатор. Так как новый резервуар появляется с собственной группой, в оба массива необходимо добавить новую ячейку. Для этого используется статический метод `Arrays.copyOf`, который копирует массив в массив (возможно) другого — меньшего или большего — размера. Если новый размер меньше, то метод отбрасывает лишние элементы. Если больше, как в нашем случае, — добавляются нули, как обычно при создании нового массива. Нулевое значение по умолчанию хорошо подходит для новой ячейки `amount`, потому что резервуары в исходном состоянии пусты. С другой стороны, необходимо явно присвоить новой ячейке `group` идентификатор новой группы, для которого можно выбрать наименьшее целое число, которое еще не является идентификатором группы, — другими словами, размер старого массива `amount`. Результат будет выглядеть примерно так:

Листинг 4.8. Memory3: метод newContainer

```
public static int newContainer() {
   int nContainers = group.length,
       nGroups = amount.length;
```

```
    amount = Arrays.copyOf(amount, nGroups + 1);    ❶ Присоединить 0 к amount
    group = Arrays.copyOf(group, nContainers + 1); ❷ Присоединить 0 к group
    group[nContainers] = nGroups; ❸ Задать идентификатор группы нового резервуара

    return nContainers; ❹ Вернуть идентификатор нового резервуара
}
```

ФАБРИЧНЫЕ МЕТОДЫ И ПАТТЕРН ФАБРИЧНЫЙ МЕТОД

Любой метод, возвращающий новый экземпляр класса, называется фабричным. По сравнению с конструктором он обладает следующими преимуществами:

- Он не обязан возвращать объект конкретного класса; подойдет любой подтип объявленного возвращаемого типа.

  Например, класс `EnumSet` (реализация `Set`, элементы которой должны принадлежать заданному перечислению) предоставляет новые множества только при вызове различных статических фабричных методов. Для ускорения выполнения они возвращают разные реализации (подклассы `EnumSet`) в зависимости от размера используемого перечисления.

- Несмотря на то что я писал ранее, фабричный метод не обязан возвращать действительно новый объект.

  Он может кэшировать или заново использовать объекты, при условии что это не создаст проблем — например, если эти объекты неизменяемы. Так, фабричный метод `Integer.valueOf` упаковывает примитивное целое число в неизменяемый объект `Integer`, который может быть или не быть новым.

С другой стороны, ФАБРИЧНЫЙ МЕТОД (записываю в верхнем регистре, чтобы было понятно, что речь идет о паттерне) — один из основных паттернов проектирования, определенных в известной книге Банды четырех[1]. Естественно, паттерн строится на основе фабричного метода, но в специфичном контексте он подразумевает, что класс должен предоставить объект всем клиентам, оставляя своим подклассам возможность изменять фактический тип объекта. Например, интерфейс `Iterable` с методом `iterator` может рассматриваться как применение этого паттерна.

Кроме того, стоит запретить клиентам создавать объекты `Container`. Можно добавить приватный конструктор с пустым телом, чтобы компилятор не добавил конструктор по умолчанию. Сравним разные варианты достижения этой цели.

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2021. — 448 с.

КЛАССЫ, НЕ ДОПУСКАЮЩИЕ СОЗДАНИЕ ЭКЗЕМПЛЯРОВ

Версия `Memory3` класса резервуара содержит только статические компоненты и не предназначена для создания экземпляров. Если вы хотите запретить создание объектов этого класса во время компиляции, язык Java предлагает:

- преобразовать класс в интерфейс;
- объявить класс абстрактным (`abstract`);
- предоставить приватный конструктор (как единственный конструктор).

Первые два способа не подходят, потому что они предлагают клиенту расширить класс, что бессмысленно для класса, не допускающего создание экземпляров. Третий способ предотвращает как создание экземпляра, так и расширение. В самом деле, при попытке расширить такой класс вы поймете, что конструктор из подкласса не может включить обязательный вызов конструктора из суперкласса.

Также третий способ используется в классах JDK, не допускающих создание экземпляров. Так, вспомогательные классы `Math`, `Arrays` и `Collections` не имеют состояния (не содержат изменяемых полей) и предназначены только для предоставления служебных функций. `Memory3` не является вспомогательным классом, потому что он хранит информацию в своих полях. Это модуль, распространяющий свою функциональность за рамками ОО-канона.

НЕОЖИДАННЫЙ ВОПРОС 4

Как бы вы спроектировали класс, у которого может быть создан только один экземпляр (паттерн Одиночка)?

На рис. 4.6 представлено распределение памяти этой реализации после выполнения первых трех частей основного сценария. Резервуары объединены в две группы с идентификаторами 0 и 1. В массиве `group` хранятся идентификаторы групп всех резервуаров, а в массиве `amount` — объем воды на резервуар в каждой группе.

Чтобы массив `amount` имел оптимальный размер, необходимо проследить за тем, чтобы между идентификаторами групп не было разрывов. Допустим, такой разрыв существует, а трем группам были присвоены идентификаторы 0, 1 и 5. Они соответствуют индексам массива `amount`, причем массив использует шесть ячеек для трех групп. Следует поддерживать следующий *инвариант класса*:

> Если существует n групп, то их идентификаторами являются целые числа от 0 до $n - 1$.

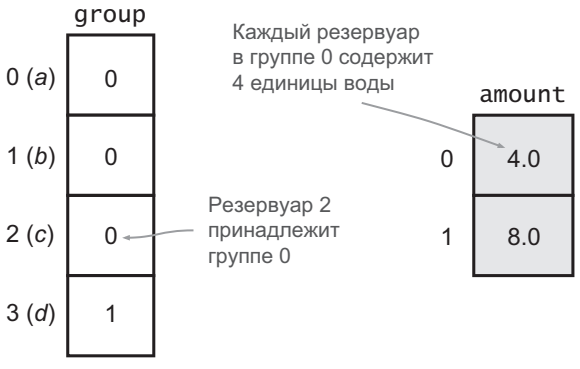


Рис. 4.6. Распределение памяти реализации Memory3 после выполнения первых трех частей основного сценария

Инвариант класса — свойство, которое остается истинным постоянно, кроме времени выполнения методов этого класса. Таким образом, методы этого класса могут рассчитывать на то, что инварианты истинны в начале их выполнения, а также должны оставаться истинными при завершении. (Мы вернемся к этой теме в главе 6.)

Предыдущий инвариант гарантирует, что массив `amount` имеет минимально возможную длину: его размер точно соответствует общему количеству групп. Это легко обеспечивается при условии, что группы только добавляются, но не удаляются. К сожалению, каждая операция соединения удаляет группу, объединяя две группы в одну. Следовательно, после того как операция удалит группу, необходимо проделать некоторую дополнительную работу для переупорядочения идентификаторов групп. Простое решение для удаления разрывов в последовательности идентификаторов — перемещение разрыва в конец последовательности, для чего отсутствующий идентификатор (разрыв) присваивается группе, которая в настоящее время содержит наибольший идентификатор. За эту операцию отвечает метод `removeGroupAndDefrag`, описанный в следующем разделе.

4.3.4. Соединение резервуаров по идентификаторам

Рассмотрим метод `connect` (листинг 4.9). Его структура вам уже знакома, кроме некоторых специальных возможностей `Memory3`:

- Размер группы недоступен напрямую: вспомогательный метод `groupSize` вычисляет его, подсчитывая количество вхождений заданного идентификатора группы в массиве `group`.
- Он объединяет две группы, присваивая идентификатор первой группы всем резервуарам второй группы.

- В конце вспомогательный метод `removeGroupAndDefrag` должен переупорядочить идентификаторы групп.

Листинг 4.9. Memory3: метод connect

```
public static void connect(int containerID1, int containerID2) {
   int groupID1 = group[containerID1],
       groupID2 = group[containerID2],
       size1 = groupSize(groupID1),   ❶ Описание метода приводится ниже
       size2 = groupSize(groupID2);
   if (groupID1 == groupID2) return;  ❷ Проверить, соединены ли резервуары

   float amount1 = amount[groupID1] * size1,  ❸ Вычислить новый объем воды
         amount2 = amount[groupID2] * size2;
   amount[groupID1] = (amount1 + amount2) / (size1 + size2);

   for (int i=0; i<group.length; i++) {  ❹ Присвоить идентификатор первой группы
      if (group[i] == groupID2) {           элементам второй группы
        group[i] = groupID1;
      }
   }
   removeGroupAndDefrag(groupID2);
}
```

Как обычно, метод `connect` должен вычислить новый объем воды в каждом резервуаре после соединения. Для этого ему нужно знать размеры двух объединяемых групп. Но размер группы нигде не хранится, и его придется вычислить, подсчитывая резервуары с заданным идентификатором группы. Эта задача решается приватным вспомогательным методом `groupSize`:

Листинг 4.10. Memory3: вспомогательный метод groupSize

```
private static int groupSize(int groupID) {
   int size = 0;
   for (int otherGroupID: group) {
      if (otherGroupID == groupID) {
         size++;
      }
   }
   return size;
}
```

Наконец, как объяснялось ранее, метод `removeGroupAndDefrag` отвечает за удаление группы с поддержанием инварианта класса[1]. Чтобы понять, как он работает, отметим, что когда `connect` вызывает `removeGroupAndDefrag` с аргументом k, ни одна ячейка массива `group` не содержит значение k — ни один резервуар уже не принадлежит к группе k. Тем не менее идентификатор группы k нельзя просто

[1] `defrag` в имени метода происходит от служебной операции файловой системы — дефрагментации, в ходе которой происходит перемещение блоков, чтобы файлы занимали смежное пространство.

удалить, потому что в последовательности идентификаторов появится разрыв, что противоречит инварианту класса. Вместо этого нужно присвоить идентификатор k другой группе и обновить два массива соответствующим образом. Допустим, перед удалением идентификаторы групп лежали в диапазоне от 0 до $n - 1$. Самое простое, что можно сделать, — это присвоить идентификатор k группе $n - 1$, а затем полностью удалить идентификатор $n - 1$.

ПОТОКИ ДАННЫХ

Метод `groupSize` идеально подходит для демонстрации потенциальных преимуществ библиотеки потоков в Java 8. Массив `group` можно преобразовать в поток целых чисел (интерфейс `IntStream`) и отфильтровать их по предикату (интерфейс `IntPredicate`), оставив только те значения, которые соответствуют заданному идентификатору группы. Затем останется подсчитать оставшиеся значения терминальной операцией `count`.

Также можно использовать лямбда-выражения, которые тоже поддерживаются Java 8, для определения фильтрующего предиката с намного более коротким синтаксисом (вместо анонимного класса).

Полученная в результате однострочная команда заменяет все тело `groupSize`:

```
return Arrays.stream(group)
             .filter(otherGroupID -> otherGroupID == groupID)
             .count();
```

Расширенное применение потоков продемонстрировано в главе 9.

В листинге 4.11 цикл `for` присваивает идентификатор группы k всем резервуарам, ранее связанным с группой $n - 1$. Строка `amount[groupID]` копирует объем воды старой группы $n - 1$ в новую группу k. Последняя строка `amount` отсекает последнюю ячейку от массива `array`, фактически стирая группу $n - 1$ и восстанавливая инвариант класса. В конце идентификаторы групп лежат в диапазоне от 0 до $n - 2$, как и требовалось.

Листинг 4.11. Memory3: вспомогательный метод removeGroupAndDefrag

```
private static void removeGroupAndDefrag(int groupID) {
   for (int containerID=0; containerID<group.length; containerID++) {
      if (group[containerID] == amount.length-1) {
         group[containerID] = groupID;
      }
   }
   amount[groupID] = amount[amount.length-1];
   amount = Arrays.copyOf(amount, amount.length-1);
}
```

На рис. 4.7 показан результат выполнения следующих трех строк из переработанного сценария, представленного ранее:

```
Container.addWater(a, 12);
Container.addWater(d, 8);
Container.connect(a, b);
```

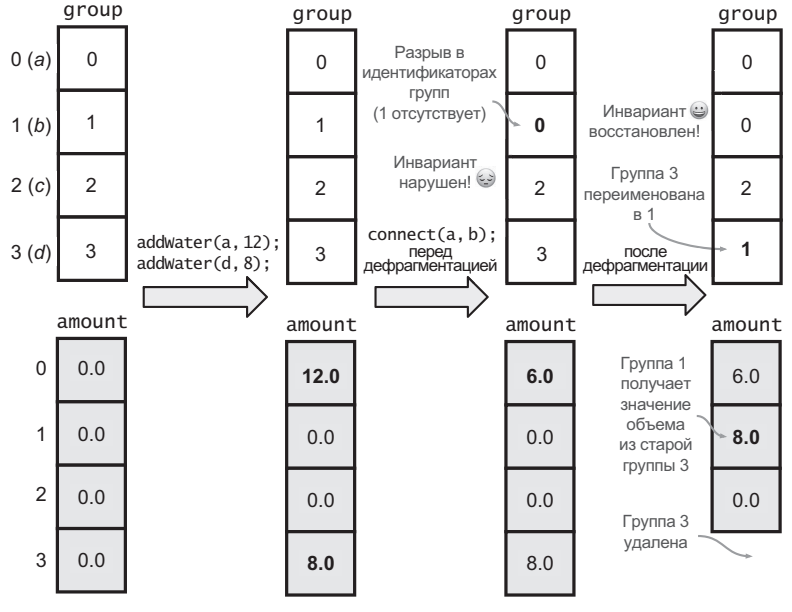


Рис. 4.7. Метод removeGroupAndDefrag удаляет группу при восстановлении инварианта класса

В частности, на рисунке изображена ситуация ближе к концу `connect`, до и после вызова `removeGroupAndDefrag` (последняя строка в листинге 4.9). Перед вызовом инвариант не выполняется, потому что идентификатор группы 1 не назначен никакому контейнеру. После вызова процедура дефрагментации переименовывает группу 3 в 1 и перемещает ее объем в `amount[1]`, восстанавливая инвариант.

Возможно, вас интересует, что произойдет, если удаляется самая последняя группа ($k = n - 1$). Быстрая проверка показывает, что этот случай не нуждается ни в какой специальной обработке. В самом деле, если $k = n - 1$, то цикл `for` не выполняется, потому что условие команды `if` всегда ложно. Следующее присваивание (строка `amount[groupID]`) также ничего не меняет, а последняя ячейка (`amount`) просто удаляет последнюю ячейку из массива `amount`.

4.3.5. Временная сложность и затраты памяти

Эта реализация базируется на двух простых массивах, размеры которых равны, соответственно, количеству резервуаров и количеству групп. Это позволяет легко вычислить затраты памяти (табл. 4.6).

Таблица 4.6. Требования к памяти для Memory3

Сценарий	Размер (вычисление)	Размер (в байтах)	Процент от Reference
1000 изолированных	4 + 16 + 1000 × 4 + 4 + 16 + + 1000 × 4	8040	7 %
100 групп по 10	4 + 16 + 1000×4 + 4 + 16 + + 100 × 4	4440	7 %

Отказ от объектов резервуаров обеспечивает серьезную экономию памяти без замедления выполнения — по крайней мере в отношении асимптотической сложности, которая остается такой же, как в `Reference`. На практике все предыдущие реализации `connect(To)` и `addWater` перебирают только обрабатываемые группы, тогда как в `Memory3` эти методы должны перебирать *все* резервуары. И `connect`, и `addWater` должны знать размер группы, что, в свою очередь, требует перебора массива всех резервуаров (массив `group`). При большом количестве резервуаров эти циклы могут даже замедлить общее выполнение по сравнению с `Reference`.

Это еще не все. Мы сосредоточились на трех открытых методах, но обратите внимание, что метод `newContainer`, играющий роль конструктора, выполняется за линейное время из-за вызовов `Arrays.copyOf`. Во всех предыдущих реализациях конструктор не содержит циклов и выполняется за постоянное время.

4.4. ЧЕРНАЯ ДЫРА [MEMORY4]

Последняя реализация в этой главе — `Memory4` — ухитряется использовать всего 4 байта для каждого дополнительного резервуара за счет более высокой временной сложности. Идея этой реализации состоит в использовании одного статического массива с одной ячейкой на каждый резервуар, выполняющей сразу две функции. Для некоторых индексов массив содержит индекс следующего резервуара той же группы, как если бы группы хранились в связанных списках. Для резервуаров, у которых нет следующего резервуара (они изолированы или завершают свой список), в массиве хранится объем воды этого резервуара (и каждого резервуара той же группы).

Я предлагаю хранить и индексы, и объемы воды в одном массиве. Первые — целые числа, вторые — вещественные. Какой тип должен иметь массив? В голову

приходят два варианта, приводящие к одинаковым затратам памяти (4 байта на резервуар):

1. Массив типа `int`, в котором содержимое ячейки должно интерпретироваться как объем воды и его можно делить на постоянную величину (фактически реализация чисел с фиксированной точкой). Например, если все объемы будут делиться на 10 000, они будут определяться с 5 цифрами в дробной части.
2. Массив типа `float`, в котором, чтобы содержимое ячейки интерпретировалось как индекс, необходимо убедиться, что значение является неотрицательным целым числом. В конце концов, неотрицательные целые числа (до некоторого значения) являются частным случаем чисел с плавающей точкой.

В листинге 4.12 я выбрал второй вариант, который выглядит проще, хотя, как вы вскоре увидите, у него есть свои недостатки.

Листинг 4.12. Memory4: поле — конструктор не нужен

```
public class Container {
   private static float[] nextOrAmount;
```

Как при чтении содержимого ячейки отличать *следующие* значения от *объемов* воды? Можно воспользоваться доисторическим трюком и закодировать один из двух случаев положительными числами, а другой — отрицательными. Положительное число можно будет интерпретировать как индекс следующего резервуара, а отрицательное число будет обозначать объем воды в этом резервуаре с *обратным* знаком. Например, если `nextOrAmount[4] == -2.5`, это означает, что резервуар 4 является последним в своей группе (или изолированным) и содержит 2,5 единицы воды.

Есть небольшая проблема: в формате с плавающей точкой «положительный нуль» не отличается от «отрицательного». Эту неоднозначность можно устранить, считая, что нуль всегда обозначает объем, и никогда не использовать его в качестве индекса следующего резервуара. Чтобы не терять нулевую ячейку, увеличьте все индексы, хранящиеся в массиве, на 1 (*смещение*). Например, если за резервуаром 4 следует резервуар 7, `nextOrAmount[4] == 8`.

На рис. 4.8 представлено распределение памяти этой реализации после выполнения первых трех частей основного сценария. Значение 2,0 в первой ячейке — смещенный указатель на следующий резервуар — означает, что первый резервуар (`a`) связан с резервуаром под номером 1 (`b`). Значение –4,0 в третьей ячейке указывает, что `c` является последним резервуаром в своей группе, а каждый резервуар в этой группе содержит 4,0 единицы воды.

В листинге 4.13 представлен код метода `getAmount`. Он переходит к следующим значениям, как в связанном списке (вторая строка), пока не найдет последний резервуар в списке, который распознается по отрицательному или нулевому значению. Это значение представляет собой объем воды в резервуаре с обратным

знаком. Обратите внимание на –1 в конце третьей строки кода (удаление смещения) и знак минус после `return` для возврата объема воды с правильным знаком.

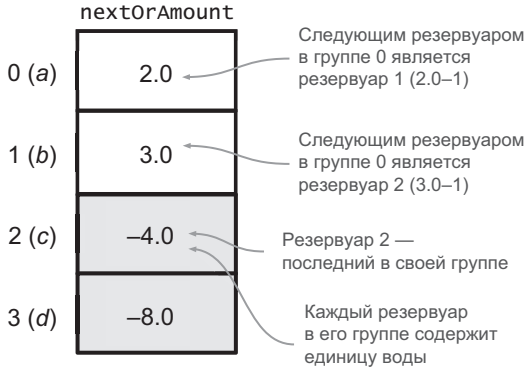


Рис. 4.8. Распределение памяти Memory4 после выполнения первых трех частей основного сценария. Массив из четырех float выполняет сразу две функции: связывание резервуаров, принадлежащих к одной группе, и хранение объемов воды

Листинг 4.13. Memory4: метод getAmount

```
public static float getAmount(int containerID) {
   while (nextOrAmount[containerID]>0) { ❶ Найти последний элемент группы
      containerID = (int) nextOrAmount[containerID] -1; ❷ Удалить смещение
   }
   return -nextOrAmount[containerID]; ❸ Восстановить правильный знак
}
```

У `float`, использующегося для представления индексов массивов, есть еще один скрытый недостаток. Теоретически индексы массивов могут охватывать весь диапазон неотрицательных 32-разрядных целых чисел: от 0 до $2^{31} - 1$ (приблизительно 2 млрд, также обозначается `Integer.MAX_VALUE`). Формат с плавающей точкой имеет существенно больший диапазон, но с изменяющимся *разрешением*. Расстояние между двумя соседними числами изменяется в зависимости от размера (рис. 4.9). Для малых значений (близких к нулю) следующее число с плавающей точкой расположено чрезвычайно близко. Для больших значений следующее число с плавающей точкой находится дальше. В какой-то момент расстояние превышает 1, и ряд чисел с плавающей точкой начинает пропускать целочисленные значения.

Например, из-за расширенного диапазона тип `float` способен точно представить число 1E10 (10^{10} или 10 млрд), чего не позволяет сделать целочисленный тип. Оба типа могут представить значение 1E8 (100 млн), но если переменная `float` содержит 1E8, то при увеличении на 1 она останется равной 1E8. У чисел с плавающей точкой не хватает значащих цифр для представления числа 100 000 001.

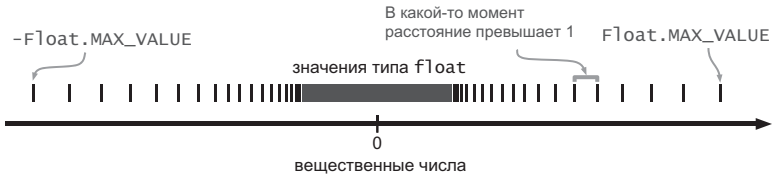


Рис. 4.9. Отношения между вещественными числами и значениями типа float

Расстояние между 1E8 и следующим числом типа `float` превышает 1. Хотя число 1E8 входит в диапазон чисел `float`, оно не входит в *непрерывный целочисленный диапазон* `float`, то есть в диапазон целых чисел, которые могут быть представлены точно и без разрывов. В табл. 4.7 приведены непрерывные целочисленные диапазоны для большинства числовых примитивных типов.

Таблица 4.7. Сравнение непрерывных целочисленных диапазонов примитивных типов. Непрерывным целочисленным диапазоном называется множество (неотрицательных) целых чисел, которые могут быть представлены точно и без разрывов

Тип	Значащие биты	Значащие десятичные цифры	Непрерывный целочисленный диапазон
`int`	31	9	от 0 до $2^{31} - 1 \approx 2 \times 10^9$
`long`	63	18	от 0 до $2^{63} - 1 \approx 9 \times 10^{18}$
`float`	24	7	от 0 до $2^{24} - 1 \approx 16 \times 10^6$
`double`	53	15	от 0 до $2^{53} - 1 \approx 9 \times 10^{15}$

НЕОЖИДАННЫЙ ВОПРОС 5

Выберите тип данных и исходное значение переменной x таким образом, чтобы цикл `while (x+1==x) {}` выполнялся бесконечно.

Использование `float` в качестве индекса массива — не лучшая идея. Оно сработает, только если индексы остаются в непрерывном целочисленном диапазоне, границы которого заметно меньше `Integer.MAX_VALUE`. Чтобы уточнить, насколько меньше, нужно учесть, что неотрицательные целые числа содержат 31 значащий бит, тогда как неотрицательные числа с плавающей точкой имеют только 24 значащих бита. Так как $31 - 24 = 7$, порог для `float` в $2^7 = 128$ раз меньше `Integer.MAX_VALUE`.

Если создать более 2^{24} резервуаров, начнут происходить странные вещи и потребуется включить проверки времени выполнения в метод `newContainer`. Но эта

глава посвящена потреблению памяти, поэтому будем придерживаться плана и оптимизировать только одно свойство кода за раз, а с факторами надежности подождем до главы 6. Остальной исходный код `Memory4` можно найти в репозитории (https://bitbucket.org/mfaella/exercisesinstyle).

4.4.1. Временная сложность и затраты памяти

Один статический массив из `Memory4` требует 4 байт для хранения ссылки на массив, 16 байт стандартных затрат массивов и 4 байт для каждой ячейки. В этой реализации заданное количество резервуаров всегда занимает одинаковый объем памяти, независимо от того, как они соединены. В табл. 4.8 приведены оценки затрат памяти для двух наших обычных сценариев.

Таблица 4.8. Требования к памяти для Memory4

Сценарий	Размер (вычисление)	Размер (в байтах)	Процент от Reference
1000 изолированных	$4 + 16 + 1000 \times 4$	4020	4 %
100 групп по 10	$4 + 16 + 1000 \times 4$	4020	7 %

За крайнюю экономию памяти приходится платить замедлением выполнения, как видно из табл. 4.9. Методы `connect` и `addWater` должны вычислять размер группы по заданному индексу произвольного резервуара группы. Для этого приходится возвращаться к первому резервуару группы, а затем обходить весь виртуальный список резервуаров для определения длины. Найти первый резервуар в группе не так просто, ведь это единственный элемент группы, на который не ссылается другой указатель. Чтобы найти его, необходимо обойти список в обратном порядке, что требует квадратичного времени.

Таблица 4.9. Временные сложности для Memory4

Метод	Временная сложность
`getAmount`	$O(n)$
`connectTo`	$O(n^2)$
`addWater`	$O(n^2)$

4.5. БАЛАНС ЗАТРАТ ПАМЯТИ И ВРЕМЕНИ

Начнем с краткой сводки требований по памяти для четырех версий резервуаров из этой главы и сравним их с реализацией `Reference` из главы 2.

Таблица 4.10. Требования к памяти для всех реализаций этой главы и реализации Reference. Не забудьте, что Memory3 и Memory4 предоставляют другой API без объектов

Сценарий	Версия	Байты	Процент от Reference
1000 изолированных	`Reference`	108 000	100 %
	`Memory1`	20 000	19 %
	`Memory2`	20 000	19 %
	`Memory3`	8040	7 %
	`Memory4`	4020	4 %
100 групп по 10	`Reference`	61 200	100 %
	`Memory1`	29 000	47 %
	`Memory2`	25 600	42 %
	`Memory3`	4440	7 %
	`Memory4`	4020	7 %

Как видно из табл. 4.10, разумный выбор коллекций и способов кодирования позволяет добиться значительной экономии памяти. Чтобы выйти за рамки, представленные служебными затратами объектов, нам пришлось нарушить API из главы 1 и идентифицировать резервуары целыми числами вместо объектов резервуаров. Все реализации этой главы также жертвуют удобочитаемостью — и, как следствие, удобством сопровождения. Стремление к эффективности использования памяти ведет к использованию низкоуровневых типов (в основном массивов) вместо высокоуровневых коллекций и специальных кодировок, вплоть до применения значений `float` в качестве индексов массивов в `Memory4`. Во многих рабочих средах такие приемы считаются нежелательными, но им находится место в узкоспециализированных ситуациях с жесткими ограничениями по памяти, как в некоторых встроенных системах, или с необходимостью хранить огромные объемы данных в основной памяти.

Как упоминалось в главе 1, эффективности по затратам памяти и времени часто вступают в конфликт. В этой и предыдущей главах были приведены как положительные, так и отрицательные примеры такого рода. На рис. 4.10 изображены требования к затратам памяти и времени для семи реализаций из этих глав, а также реализации `Reference` из главы 2. Вспомните, что в `Memory3` и `Memory4` заметная экономия памяти достигается за счет изменения API резервуаров.

Самыми сложными реализациями из двух глав являются те, которые максимизируют соответствующее свойство программного кода: `Speed3` обеспечивает максимальную скорость выполнения, а `Memory4` — максимальную эффективность использования памяти. Кроме того, при сокращении требований к памяти до

4 байт на резервуар в Memory4 временная сложность повышается до квадратичной. Этого следует ожидать — подобные компромиссы типичны при выборе баланса между временем и затратами памяти.

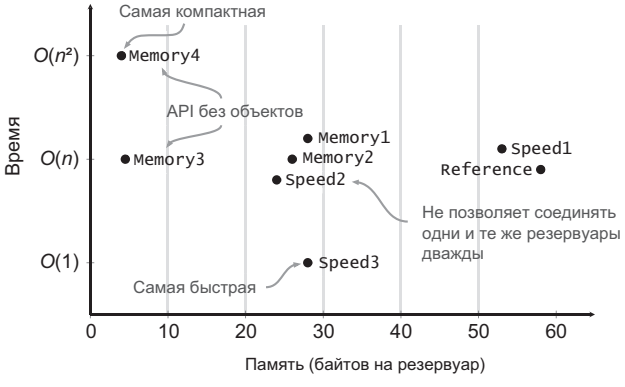


Рис. 4.10. Профили скорости выполнения реализаций из глав 3 и 4, а также Reference. В качестве показателя затрат памяти используется среднее количество байтов на резервуар в сценарии 2 (100 групп по 10 резервуаров). В качестве показателя затрат времени используется максимальная сложность по трем методам класса (при этом для Speed3 оценивается амортизированная сложность)

С другой стороны, Speed3 демонстрирует отличную эффективность: затраты памяти близки к Memory2 — минимуму, которого нам удалось добиться без отказа от стандартного API. Следовательно, в большинстве ситуаций при отсутствии жестких ограничений по памяти следует выбирать Speed3.

4.6. А ТЕПЕРЬ СОВСЕМ ДРУГОЕ

Пришло время применить методы экономии памяти в другом сценарии: работе с мультимножествами. *Мультимножеством* называется множество, которое может содержать дубликаты. Так, мультимножество {*a*, *a*, *b*} отлично от {*a*, *b*}, но неотличимо от {*a*, *b*, *a*}, потому что порядок элементов не важен.

Спроектируем реализацию мультимножества MultiSet<T>, которая эффективно расходует память и поддерживает следующие методы:

- `public void add(T elem)` — вставляет `elem` в мультимножество;
- `public long count(T elem)` — возвращает количество вхождений `elem` в мультимножество.

Для сравнения и выбора между разными реализациями можно руководствоваться следующими вопросами:

1. Предположим, вы вставляете n разных объектов с возможностью многократной вставки одного объекта и всего есть m вставок (то есть m по крайней мере не меньше n). Сколько байт потребуется для их хранения?
2. Какова временная сложность операций `add` и `count` в вашей реализации?

Как выясняется, существуют две реализации, оптимальные по затратам памяти, а выбор между ними зависит от предполагаемого количества дубликатов.

4.6.1. Малое количество дубликатов

Если предполагаемое количество дубликатов невелико, можно воспользоваться одним массивом для объектов и присоединять каждый вставляемый объект в конец массива — как для первого вхождения, так и для дубликатов.

Как упоминалось в этой главе, использование `ArrayList` вместо простого массива абсолютно оправданно, потому что коллекция занимает чуть больше памяти, но очень сильно упрощает реализацию. Более того, в отличие от массивов, `ArrayList` хорошо работает с обобщениями.

Реализация должна выглядеть примерно так:

```
public class MultiSet<T> {
   private List<T> data = new ArrayList<>();

   public void add(T elem) {
      data.add(elem);
   }
   public long count(T elem) {
      long count = 0;
      for (T other: data) {
         if (other.equals(elem)) {
            count++;
         }
      }
      return count;
   }
}
```

С новой библиотекой потоков можно переписать метод `count` в однострочной реализации:

```
public long count(T elem) {
   return data.stream().filter(x -> x.equals(elem)).count();
}
```

Метод `add` выполняется за постоянное (амортизированное) время (раздел 3.3.5), а `count` — за линейное время. Затраты памяти после m вставок n разных объектов составят $56 + 4 \times m$ байт (не зависит от n):

- 12 байт — служебная информация объекта `MultiSet`;
- 4 байта — ссылка на `ArrayList`;
- 40 байт — минимальная коллекция `ArrayList` (табл. 4.4);
- $4 \times m$ байт для ссылок на элементы мультимножества.

4.6.2. Большое количество дубликатов

Если дубликаты встречаются часто, лучше использовать два массива: для хранения самих объектов и для хранения количества повторений каждого объекта. Если вы знакомы с библиотекой коллекций, то догадаетесь, что эта задача идеально подходит для `Map`. Однако обе стандартные реализации `Map` (`HashMap` и `TreeMap`) представляют собой связанные структуры и занимают намного больше памяти, чем две коллекции `ArrayList`.

В итоге у вас получится нечто такое:

```
public class MultiSet<T> {
   private List<T> elements = new ArrayList<>();
   private List<Long> repetitions = new ArrayList<>();
   ...
```

Остаток реализации я оставлю вам для самостоятельной работы. Проследите, чтобы i-й элемент `repetitions` (который вы получаете от `repetitions.get(i)`) содержал количество повторений объекта `elements.get(i)`.

Для ускорения выполнения вставка должна проверять первый массив и определять, что вставляется: новый объект или дубликат. В худшем случае оба метода `add` и `count` будут выполняться за линейное время.

Затраты памяти после m вставок n разных объектов составят $100 + 28 \times n$ байт (не зависит от m):

- 12 байт — служебная информация объекта `MultiSet`;
- 2×4 байта — ссылки на две коллекции `ArrayList`;
- 2×40 байт — две минимальные коллекции `ArrayList`;
- $4 \times n$ байт для хранения ссылок на уникальные элементы (первый массив);
- $(4 + 20) \times n$ байт для хранения счетчиков `Long` на уникальные элементы (второй массив). (Каждый объект `Long` занимает $12 + 8 = 20$ байт.)

Решение с двумя массивами наиболее эффективно по памяти, если $100 + 28 \times n < 56 + 4 \times m$, то есть в среднем каждый объект представлен в коллекции не менее 7 раз ($m > 11 + 7 \times n$).

4.7. РЕАЛЬНЫЕ СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

В главах 3 и 4 рассматривались два основных фактора, влияющих на эффективность алгоритма: время и затраты памяти. Было показано, что задача может быть решена разными способами (например, с использованием `ArrayList` вместо `HashSet` для хранения групп резервуаров). Выбор того или иного метода обычно приводит к компромиссу между эффективностью по времени и затратам памяти. Лучший выбор зависит от контекста решаемой задачи. Рассмотрим пару сценариев с высокой эффективностью по затратам памяти.

- В области машинного обучения центральное место занимают наборы данных. Обычно наборы данных представляются плотной матрицей строк исторических данных, включающих *переменные*. Возьмем сложный набор данных, состоящий из направленного графа, в котором узлами являются веб-страницы, а направленные ребра представляют ссылки между ними. Теоретически можно представить этот набор данных в виде матрицы смежности — квадратной матрицы, строки и столбцы которой представляют узлы графа (веб-страницы), а значения элементов показывают, существует ли ребро (ссылка) от одной веб-страницы к другой (значение 1) или не существует (значение 0). Если граф является разреженным, большинство ячеек матрицы остается неиспользованным, что приведет к потере памяти. В таком случае можно подумать о представлении, эффективном по памяти, но теряющем часть эффективности по времени.
- Смартфоны в наши дни оснащаются почти таким же объемом памяти, как стандартные портативные компьютеры. Однако когда компания Google разрабатывала ОС Android в начале 2000-х годов, ситуация была иной. Система Android также должна была работать на устройствах с существенно меньшими объемами памяти, чем у современных телефонов. По этой причине в API Android можно найти следы усилий по экономии памяти. Например:
  - Пакет `android.util` содержит несколько классов, предоставляющих альтернативы для стандартных коллекций Java с меньшими затратами памяти. Например, `SparseArray` — эффективная по памяти реализация карты (или ассоциативного массива), связывающей целочисленные ключи с объектами. (В упражнении 2 этой главы вам будет предложено проанализировать этот класс.)
  - Все классы Android, относящиеся к работе с графикой, используют для представления координат, углов поворота и т. д. значения `float` с одинарной

точностью вместо значений `double`. Пример можно найти в классе `android.graphics.Camera`.

- XML широко используется для обмена данными между разнородными системами. В стандартной схеме взаимодействия приложение разбирает XML, сохраняет контент в реляционной базе данных и, наконец, сохраняет XML в виде BLOB (binary large object). Далее бизнес-логика и запросы выполняются с использованием реляционной схемы, и события загрузки исходной разметки XML происходят редко. То есть лучше проектировать процесс, эффективный по затратам памяти, который сжимает XML-документы перед их сохранением в базе данных.

4.8. ПРИМЕНИМ ИЗУЧЕННОЕ

Упражнение 1

Прочитайте описание мультимножества из раздела 4.6. Библиотека Google Guava (https://github.com/google/guava) содержит интерфейс `Multiset` и его различные реализации в пакете `com.google.common.collect`. Основные методы `Multiset<E>`:

- `public boolean add(E elem)` — вставляет `elem` в мультимножество и возвращает `true` (для совместимости с интерфейсом `Collection`);
- `public int count(Object elem)` — возвращает количество вхождений `elem` в мультимножестве.

Просмотрите исходный код класса `HashMultiset` и ответьте на следующие вопросы.

1. Какой временной сложностью обладают его методы `add` и `count`?
2. Как вы считаете, оптимизирован ли этот класс по затратам памяти, времени или их балансу?

Подсказка: вам также понадобится заглянуть в исходный код `HashMultiset` и его абстрактного суперкласса `AbstractMapBasedMultiSet`.

Упражнение 2

Класс Android `android.util.SparseArray` представляет собой эффективную по затратам памяти реализацию массива объектов, индексами которого могут быть произвольные целые числа вместо непрерывного ряда индексов, начинающегося с 0. Соответственно, он служит заменой для `Map<Integer,Object>`. В его внутренней реализации используются два массива: для индексов (ключей) и для объектов (значений).

Просмотрите исходный код класса `android.util.SparseArray` (http://mng.bz/DNZA) и ответьте на следующие вопросы.

1. Сколько памяти потребуется для хранения пустого объекта `SparseArray`, созданного вызовом `new SparseArray()`?
2. Сколько памяти понадобится для объекта `SparseArray`, содержащего 100 объектов с непрерывными индексами 0–99 (не считая памяти, занимаемой самими объектами)?
3. Сколько памяти понадобится для объекта `SparseArray`, содержащего 100 объектов со случайными целочисленными индексами?

Упражнение 3

Из раздела 3.3 вы узнали, что в `Speed3` только резервуары, являющиеся представителями своей группы, используют свои поля `amount` и `size`. Для других резервуаров эти поля неактуальны. Проведите рефакторинг `Speed3`, чтобы устранить эту неэффективность по памяти без изменения открытого API реализации.

Подсказка: не забудьте, что объекты резервуаров создаются перед соединением, чтобы клиенты могли хранить ссылки на них, а объекты в Java не могли динамически изменять свой тип.

Упражнение 4 (мини-проект)

Класс `UniqueList<E>` является индексированным списком фиксированного размера, не содержащий дубликатов, который предоставляет следующий открытый интерфейс:

- `public UniqueList(int capacity)` — создает пустой контейнер `UniqueList` с заданной емкостью `capacity`;
- `public boolean set(int index, E element)` — вставляет заданный элемент в позицию с заданным индексом и возвращает `true`, если индекс лежит в диапазоне от 0 до `capacity-1` и элемент не входит в коллекцию под другим индексом. В противном случае метод не изменяет список и возвращает `false`;
- `public E get(int index)` — возвращает элемент с заданным индексом или `null`, если индекс недействителен или содержит пустое (неприсвоенное) значение.

Ориентируясь на этот интерфейс, сделайте следующее:

1. Реализуйте класс `UniqueList` способом, эффективным по затратам памяти.
2. Реализуйте класс `UniqueList` способом, эффективным по времени.

ИТОГИ

- Высокоуровневые коллекции, такие как `HashSet`, обычно улучшают скорость выполнения и удобочитаемость кода, но требуют больших затрат памяти по сравнению с низкоуровневыми альтернативами.
- При катастрофической нехватке памяти можно избежать затрат, связанных с хранением объектов, перейдя на целочисленные идентификаторы.
- Хранение данных в смежной памяти повышает быстродействие благодаря локальности данных для кэширования.
- Числа с плавающей точкой имеют более широкий диапазон представления данных, чем целые числа, но с изменяющимся разрешением.

ОТВЕТЫ НА ВОПРОСЫ И УПРАЖНЕНИЯ

Неожиданный вопрос 1

Благодаря механизму, называемому *интернированием строк*, в памяти хранится только одна копия каждого строкового литерала.

Что касается памяти, занимаемой одной строкой `Hello World`, до Java она была бы представлена в UTF-16: по 2 байта на символ. Начиная с Java 9, функциональность *компактных строк* распознает, что эта конкретная строка содержит только ASCII-символы, и переключается на однобайтовую кодировку. В обоих случаях символы хранятся в байтовом массиве. Кроме самих символов, добавляются следующие составляющие:

- 12 байт — служебная информация объекта `String`;
- 4 байта — для кэширования хеш-кода строки;
- 4 байта — для ссылки на байтовый массив;
- 1 байт — флаг, определяющий кодировку (традиционную или компактную);
- 16 байт — служебная информация для байтового массива.

Итак, единственная копия `Hello World` (11 символов) занимает

$$11 + 12 + 4 + 4 + 1 + 16 = 48 \text{ байт.}$$

Неожиданный вопрос 2

Взаимодействие между массивами и обобщениями ограничивается двумя противоречащими решениями из области проектирования языка:

- Компилятор «стирает» неограниченные параметры-типы и заменяет их на `Object`.

- Массивы хранят свой статический тип (и используют его для проверки каждой записи в массив).

В результате если бы конструкция `new T[10]` была допустима, вновь созданный объект вел бы себя в точности как `new Object[10]`. Но это не то, чего ожидает программист, — отсюда и решение считать первое выражение недопустимым.

Неожиданный вопрос 3

Да, вызов `set.contains(x)` может оказать небольшое положительное влияние на последующий вызов `set.contains(y)`, потому что первый вызов загрузит в кэш часть массива гнезд этой коллекции `HashSet`. (Внутренняя структура `HashSet` представлена на рис. 2.7.) Если объекты `x` и `y` имеют близкие хеш-коды, второй вызов может найти ссылку на гнездо `y` в кэше.

Те же соображения применимы к `TreeSet`, но по другой причине. `TreeSet` является полностью связанной структурой данных, в которой поиск элемента требует отслеживания пути по дереву. Второй вызов `set.contains(y)` может извлечь пользу из факта, что первые узлы пути, ведущего к `y`, могут уже находиться в кэше. (Все пути начинаются от одного корневого узла, поэтому по крайней мере этот узел с большой вероятностью все еще находится в кэше.)

Неожиданный вопрос 4

Классы-одиночки (или *синглетные* классы) — стандартный способ предоставления единой точки доступа к некоторой низкоуровневой функциональности. Класс-одиночка создается объявлением приватного конструктора как единственного конструктора и предоставлением открытого метода, который всегда возвращает один и тот же экземпляр. Этот экземпляр обычно хранится в приватном статическом поле класса.

Если единственный экземпляр создается по требованию при первом вызове метода (отложенная инициализация), необходимо крайне осторожно отнестись к возможным проблемам с потоковой безопасностью. Речь идет о так называемой проблеме безопасной инициализации; подробнее о ней можно узнать в книге *Java Concurrency in Practice* Брайана Гетца (Brian Goetz) и др. (раздел «Дополнительная литература» главы 8).

Неожиданный вопрос 5

Для переменной `x` можно выбрать тип `float` или `double` и исходное значение за пределами непрерывного целочисленного диапазона этого типа, например `float x = 1E8`.

Это лишь один из многих интересных вопросов, встречающихся в книге *Java Puzzlers* Джошуа Блоха (Joshua Bloch) и Нила Гефтера (Neal Gafter).

Упражнение 1

Начните исследование с конкретного класса `HashMultiset`, который расширяет `AbstractMapBasedMultiset` и использует вспомогательный класс `Count` (в строке `super`), представляющий целое число, которое можно модифицировать на месте, — по сути, изменяемую версию `Integer`.

```
public final class HashMultiset<E> extends AbstractMapBasedMultiset<E> {

   public static <E> HashMultiset<E> create() {   ❶ Фабричный метод
      return new HashMultiset<E>();
   }

   private HashMultiset() {                       ❷ Приватный конструктор
      super(new HashMap<E, Count>());             ❸ Вызов конструктора суперкласса
   }
```

Как видно из листинга, открытый фабричный метод создает пустую коллекцию `HashMultiset` (строка `public static`). При этом вызывается приватный конструктор (строка `private`), который, в свою очередь, передает новый объект `HashMap` конструктору суперкласса (строка `super`). Затем взгляните на соответствующую часть суперкласса `AbstractMapBasedMultiSet`, в которой вы найдете непосредственное поле экземпляра (`backingMap` в следующем фрагменте), поддерживающее всю реализацию:

```
abstract class AbstractMapBasedMultiset<E> extends AbstractMultiset<E>
                                           implements Serializable {
   private transient Map<E, Count> backingMap;
```

По этим фрагментам можно сделать вывод, что внутренняя структура `HashMultiset` должна быть картой, связывающей объекты с целыми числами, которую реализует `HashMap`, сохранив количества вхождений каждого элемента. `HashSet` является реализацией `Set`, эффективной по времени (по аналогии `HashMap` и `Map`)[1]. Оба класса ориентированы на эффективность по времени в ущерб для памяти. Теперь ответим на два вопроса, поставленных в упражнении:

1. Методы `add` и `count` обладают постоянной временной сложностью, потому что используют постоянное количество вызовов базовых методов `HashMap`, обладающих постоянной временной сложностью. Действуют обычные проблемы хешированных структур данных: хеш-функция, предоставляемая методом `hashCode`, должна равномерно распределять объекты по диапазону целых чисел.
2. Класс `HashMultiset` оптимизирован для эффективности по времени.

[1] Во внутренней реализации `HashSet` фактически представляет собой коллекцию `HashMap`, у которой все ключи используют одно и то же фиктивное значение.

Упражнение 2

Сначала рассмотрим поля экземпляра `SparseArray`, перечисленные в следующем фрагменте кода. Поле `mGarbage` — флаг, используемый для задержки фактического удаления элемента до того момента, когда его отсутствие станет видимым (разновидность *отложенного выполнения* — см. главу 3).

```
public class SparseArray<E> implements Cloneable {
   private boolean  mGarbage = false;
   private int[]    mKeys;
   private Object[] mValues;
   private int      mSize;
```

Ниже приведены два сокращенных конструктора, задействованных в вызовах `new SparseArray()`. Строка `mValues` — эффективный способ выделения памяти для массива, специфичный для Android.

```
public SparseArray() {
   this(10);   ❷ Исходная начальная емкость — 10 элементов
}
public SparseArray(int initialCapacity) {
   ...
   mValues = ArrayUtils.newUnpaddedObjectArray(initialCapacity);
   mKeys = new int[mValues.length];
   mSize = 0;
}
```

Предыдущих фрагментов достаточно, чтобы ответить на вопросы:

1. Из этой главы и главы 2 вы узнали о размерах объектов и массивов и можете вычислить размеры всех полей `SparseArray`, кроме поля `mGarbage`, потому что примитивный тип `boolean` мы не обсуждали. Хотя его значение можно закодировать одним битом, его потребление памяти зависит от VM. В текущей версии HotSpot `boolean` занимает 1 байт — наименьшую единицу памяти, адресуемую процессором. Как обычно в этой книге, я игнорирую увеличение фактического размера объектов для выравнивания их по адресам, кратным 8.

   С учетом сказанного пустой массив `SparseArray` занимает:

   - 12 байт — служебная информация объекта `SparseArray`;
   - 12 байт — поля `mKeys`, `mValues` и `mSize`;
   - 1 байт — поле `mGarbage`;
   - 16 байт — служебная информация массива для `mKeys`;
   - 10 × 4 байт — исходный массив `mKeys` длины 10;
   - 16 байт — служебная информация массива для `mValues`;
   - 10 × 4 байт — исходный массив `mValues` длины 10.

   Итого: 137 байт.

2. У коллекции `SparseArray` с сотней объектов, индексируемых от 0 до 99, два массива `mKeys` и `mValues` должны иметь длину не менее 100. После небольшой корректировки вычислений для вопроса 1 получается 857 байт.
3. Значения индексов не влияют на структуру `SparseArray` (именно это означает «разреженность» в имени). В результате потребление памяти в этом сценарии остается таким же, как в вопросе 2, — 857 байт.

Упражнение 3

Чтобы обеспечить максимальную эффективность по затратам памяти, нормальный резервуар должен содержать только поле `parent` типа `Container`. Для представителей групп (то есть корней дерева) это поле указывает на специальный объект, содержащий поля `amount` и `size`. Типом этого вспомогательного объекта должен быть подкласс `Container`, к которому необходимо применить понижающее преобразование для перехода от предполагаемого класса `Container` к его фактическому подклассу.

Решение сокращает потребление памяти обычных резервуаров, но увеличивает размер представителей группы, потому что при этом добавляется лишний объект, не использовавшийся в `Speed3`. Эффективность по затратам памяти улучшается только в том случае, если большинство резервуаров соединено друг с другом, а количество групп невелико.

Исходный код доступен в репозитории (https://bitbucket.org/mfaella/exercisesinstyle) в виде класса `eis.chapter4.exercises.Container`.

Упражнение 4

1. Версия `Memory2` показывает, что экономия памяти, полученная с переходом на простой массив по сравнению с `ArrayList`, незначительна, так что в версии `UniqueList`, эффективной по затратам памяти, следует использовать `ArrayList`. Однако это означает, что проверка, входит ли элемент в список, займет линейное время.

   Реализацию усложняют две проблемы:

   - С уже занятым индексом можно использовать только методы `set` и `get` из интерфейса `List`. Конструктор должен изначально заполнить список требуемым количеством значений `null`.
   - Методы `set` и `get` выдают исключение, если индекс выходит за пределы диапазона, тогда как спецификации упражнения требуют специальных возвращаемых значений (`false` и `null` соответственно). Вот почему вам придется вручную проверять, находится ли индекс в диапазоне.

   Полученный код выглядит так:

```
public class CompactUniqueList<E> {
   private final ArrayList<E> data;
```

```
    public CompactUniqueList(int capacity) {
       data = new ArrayList<>(capacity);
       for (int i=0; i<capacity; i++) {   ❶ Заполнить null
          data.add(null);
       }
       assert data.size() == capacity;    ❷ Простая проверка
    }

    public boolean set(int index, E element) {
       if (index<0 || index>=data.size() || data.contains(element))
          return false;
       data.set(index, element);   ❸ Выдать исключение при недопустимом индексе
       return true;
    }

    public E get(int index) {
       if (index<0 || index>=data.size())
          return null;
       return data.get(index); ❹ Выдает исключение при недействительном индексе
    }
}
```

2. В реализации, эффективной по времени, все операции должны выполняться как можно быстрее — лучше за постоянное время. В данном случае можно добиться нужного результата, сохраняя элементы двух структур данных одновременно: списком для быстрой индексируемой выборки и множеством для быстрого исключения дубликатов. Класс содержит следующие поля:

```
public class FastUniqueList<E> {
   private final ArrayList<E> dataByIndex;
   private final Set<E> dataSet;
```

Конструктор и метод `get` очень похожи на предыдущий пример, они доступны в репозитории (https://bitbucket.org/mfaella/exercisesinstyle). Только метод `set` демонстрирует взаимодействие двух полей.

```
public boolean set(int index, E element) {
   if (index<0 || index>=dataByIndex.size() || dataSet.contains(element))
      return false;
   E old = dataByIndex.set(index, element); ❶ Возвращает объект, ранее
   dataSet.remove(old);                       находившийся по этому индексу
   dataSet.add(element);
   return true;
}
```

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Вряд ли можно найти книги, посвященные исключительно методам экономии памяти. Попытки втиснуть больше данных в меньший объем памяти обычно

приводят к неуклюжим схемам кодировки и непонятным программам, как `Memory4`, а ясность кода обладает намного большей ценностью, чем память.

Чтобы ограничить потребление памяти при сохранении удобочитаемости кода, можно выбрать структуры данных, более эффективно расходующие память, как было сделано в `Memory1` при переключении с `HashSet` на `ArrayList`. Чтобы больше узнать о временной сложности и затратах памяти стандартных алгоритмов и структур данных, обращайтесь к книгам, упоминавшимся в конце главы 3.

Другие полезные советы можно найти в следующих книгах:

- *Oaks S.* Java Performance: The Definitive Guide. O'Reilly Media, 2014.

  Помимо приемов ускорения выполнения, в книге освещены средства работы с памятью, включая методы определения объектов, занимающих наибольший объем памяти, а также различные приемы экономии памяти.

- *White E.* Making Embedded Systems. O'Reilly Media, 2011.

  Глава «Как делать больше меньшими средствами» содержит рекомендации по экономии памяти в программировании для встроенных систем. Особое внимание уделено сокращению сегмента кода и сегмента данных программы.

5 Надежность за счет мониторинга

В этой главе:

- ✓ Написание спецификаций методов в форме контрактов.
- ✓ Соблюдение контрактов во время выполнения.
- ✓ Использование директив `assert`.
- ✓ Проверка инвариантов классов как упрощенная альтернатива для постусловий.

Надежностью программных продуктов называется степень, в которой поведение системы соответствует ожиданиям в различных эксплуатационных условиях. В этой главе будут исследованы основные методы для поиска и предотвращения неожиданного поведения программ. Но сначала *определим* ожидаемое поведение блока кода (то есть его *спецификацию*). В соответствии со структурой книги я сосредоточусь на поведении отдельного класса, такого как `Container`. Популярный способ организации спецификаций в ОО-программах и находящихся в них классах основан на методологии *контрактного проектирования*.

5.1. КОНТРАКТНОЕ ПРОЕКТИРОВАНИЕ

В повседневном понимании *контрактом* называется соглашение, в котором каждая сторона принимает на себя обязательства в обмен на особые права. Собственно, то, что является обязательством для одной стороны, становится

правом для другой. Например, тарифный план является контрактом между компанией-оператором и владельцем телефона. Оператор обязан предоставить услугу связи, а владелец телефона обязан оплачивать эту услугу, так что каждая сторона получает особые права при выполнении обязательств другой стороной.

Методология контрактного проектирования предполагает связывание контрактов с программными артефактами, прежде всего с отдельными методами. Контракт метода включает *предусловие*, *постусловие* и, возможно, *штраф*.

5.1.1. Предусловия и постусловия

Предусловие формулирует требования для правильного функционирования метода. Оно определяет допустимые значения параметров и текущее состояние объекта (для методов экземпляров). Например, предусловие метода для вычисления квадратного корня может указать, что аргумент должен быть неотрицательным.

Вызывающая сторона обязана соблюдать предусловие вызываемого метода. По аналогии с обычными контрактами, предусловие является обязательством для вызывающей стороны и дает особые права вызываемой. Сам метод может либо пассивно предположить, что предусловие выполняется, либо активно проверить, выполняется ли оно, и реагировать соответствующим образом.

Предусловие должно включать только те свойства, которые находятся под полным контролем вызывающей стороны. Например, метод, который получает имя файла в аргументе и открывает этот файл, не может указать в своих предусловиях, что этот файл должен существовать, потому что вызывающая сторона не может быть в этом уверена на 100 %. (Другой процесс может в любой момент удалить этот файл.) В этом случае метод все равно может выдать исключение, но это исключение будет проверяемым, что заставит вызывающую сторону обработать его.

ЧИСТЫЕ МЕТОДЫ И ПОБОЧНЫЕ ЭФФЕКТЫ

Метод, единственным результатом применения которого является возвращение некоторого значения, называется *чистым* (pure). Любое другое последствие, будь то вывод на экран или обновление поля экземпляра, называется побочным эффектом. При повторном вызове с теми же аргументами чистый метод возвращает тот же результат — это свойство называется *ссылочной прозрачностью*. (Вспомните, что текущий объект является неявным входным параметром метода экземпляра.) Функциональные языки, такие как Haskell или Scheme, основаны на концепциях чистых функций и ссылочной прозрачности. Тем не менее любая полезная программа должна в конечном итоге взаимодействовать со своим окружением, поэтому функциональные языки упаковывают эти необходимые побочные эффекты в специально обозначенные модули.

С другой стороны, постусловие определяет результат применения метода и описывает его возвращаемое значение, а также все изменения в состоянии любых объектов. В большинстве хорошо спроектированных классов изменения должны ограничиваться текущим объектом, но так получается не всегда. Например, метод `connectTo` в нашем сквозном примере должен изменить несколько резервуаров для достижения желаемого эффекта.

Постусловие также должно выдавать *штраф* при нарушении предусловия вызывающей стороной. В Java типичный штраф состоит из выдачи непроверяемого исключения. На рис. 5.1 приведено графическое представление контракта метода экземпляра.

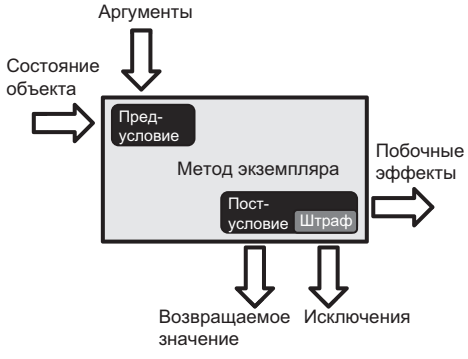


Рис. 5.1. Высокоуровневая структура контракта метода экземпляра. Все последствия выполнения метода, кроме возвращаемого значения, называются побочными эффектами

НЕОЖИДАННЫЙ ВОПРОС 1

Почему штраф не должен сводиться к выдаче *проверяемого* исключения?

Например, контракт метода `next` интерфейса `java.util.Iterator` выглядит так:

- *Предусловие*: итератор еще не достиг конца (в эквивалентной формулировке — вызов `hasNext` возвращает `true`).
- *Постусловие*: возврат следующего элемента при переборе и перемещение итератора на одну позицию.
- *Штраф:* если предусловие нарушается, метод выдает `NoSuchElement Exception` (непроверяемое исключение).

Вызов `next` при достижении итератором конца перебора нарушает предусловие и является ошибкой со стороны клиента. (Ошибка лежит за пределами метода `next`.) И наоборот, реализация `next`, которая не перемещает итератор

к следующему элементу, нарушает постусловие. В данном случае ошибка кроется в самом методе.

На рис. 5.2 изображены зависимости данных, в которых задействованы разные части контракта. Предусловие определяет допустимые значения аргументов и состояние объекта перед вызовом. Именно поэтому в блок «предусловие» входят две стрелки. Например, в предусловии `Iterator::next` упоминается только состояние итератора, потому что метод не получает аргументов.

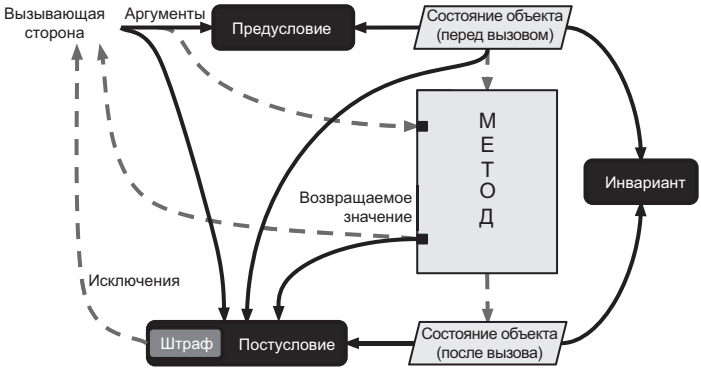


Рис. 5.2. Подробная структура контракта для метода экземпляра. Сплошные стрелки изображают зависимости данных, относящиеся к содержанию контракта. Пунктирные стрелки изображают стандартные взаимодействия во время выполнения, происходящие фактически во время выполнения независимо от контракта

Поскольку постусловие описывает все изменения, производимые методом, в нем могут упоминаться следующие данные:

- Возвращаемое значение (как главный результат применения метода).
- Старое и новое состояние объекта: старое состояние как входные данные, которые могут влиять на поведение метода, и новое состояние как другой результат применения метода.
- Значения аргументов как входные данные.
- Другие побочные эффекты, производимые глобально доступными объектами или статическими методами, такими как вызов `System.out.println`.

На рис. 5.2 последний случай опущен, а другие обозначены стрелками, входящими в постусловие. Например, в постусловии `Iterator::next` явно упоминается возвращаемое значение и неявно — старое и новое состояния итератора («возвращает следующий элемент при переборе и перемещает итератор на одну позицию»).

5.1.2. Инварианты

Кроме контрактов методов, с классами также могут связываться инварианты. Инвариант представляет собой условие, относящееся к полям класса, которое всегда должно оставаться истинным, кроме как во время изменения объекта методом класса.

Инварианты являются *статическими правилами целостности*: в них упоминается состояние объекта в один конкретный момент времени. С другой стороны, постусловия являются *динамическими правилами целостности*, потому что они сравнивают состояние объектов до и после вызова метода.

Как следует из названия, инварианты должны выполняться как до, так и после вызова метода. Соответственно, на рис. 5.2 в инвариант входят стрелки как из старого, так и из нового состояния объекта.

Исходное состояние каждого объекта, установленное конструктором, должно удовлетворять инвариантам, а все открытые методы отвечают за их поддержание. У приватных методов такой обязанности нет, потому что их роль сводится к поддержке работы открытых методов. Как следствие, выполнение приватного метода экземпляра обычно происходит в контексте некоторого выполняемого открытого метода экземпляра, из которого он был вызван (прямо или косвенно). Так как текущий объект может находиться в процессе изменения, приватный метод может обнаружить его в промежуточном состоянии, нарушающем инвариант, а также оставить его в противоречивом состоянии. Только в конце открытого метода состояние объекта снова должно стать целостным и инварианты должны быть восстановлены.

5.1.3. Правильность и стабильность

Надежность программного кода можно уточнить из двух свойств: правильности и стабильности. Различия между ними заключаются в типе окружения системы. При оценке правильности вы представляете свою систему в *номинальном* окружении, которое соответствует ее ожиданиям. В таком дружественном контексте предусловия методов свято соблюдаются, внешние входные данные поступают своевременно и в правильном формате, а все необходимые ресурсы доступны. Если система работает правильно, она будет вести себя по плану во всех дружественных средах.

В принципе, правильность является логическим свойством: она либо есть, либо ее нет. Частичная правильность особого смысла не имеет. Тем не менее обычно бывает непрактично проектировать абсолютно формальные и полные спецификации, а как только спецификация размывается, пропадает и правильность. В маленьком управляемом мире примера с резервуарами мы заранее разрабатываем четкие спецификации и убеждаемся в том, что класс правилен

в отношении этих спецификаций. Затем мы исследуем некоторые приемы, повышающие нашу *уверенность* в правильности кода. Эти приемы будут полезны в реальных ситуациях, когда вы не сможете позволить себе тратить месяцы на один класс (как я во время написания книги).

Под термином «стабильность» понимается поведение системы в исключительных или неожиданных средах. Типичные ситуации — нехватка памяти или дискового пространства на хосте, неверный формат или недопустимые значения внешних входных данных, вызов методов с нарушением предусловий и т. д. Стабильная система должна *корректно* работать в таких ситуациях, когда определение *корректности* зависит от контекста.

Например, если критический ресурс недоступен, программа может попробовать немного подождать и запросить его снова — и так пару раз, прежде чем отказаться от дальнейших попыток и завершиться. Если проблема не исчезает, а в более общем варианте если завершение остается единственно возможным выходом, программа должна четко проинформировать пользователя о характере проблемы. Кроме того, она должна по возможности минимизировать потерю данных, чтобы пользователь мог позднее возобновить свою работу, словно ничего не произошло.

НЕОЖИДАННЫЙ ВОПРОС 2

Если программа выдает какие-то результаты на печать, какой должна быть корректная реакция на отсутствие бумаги в принтере?

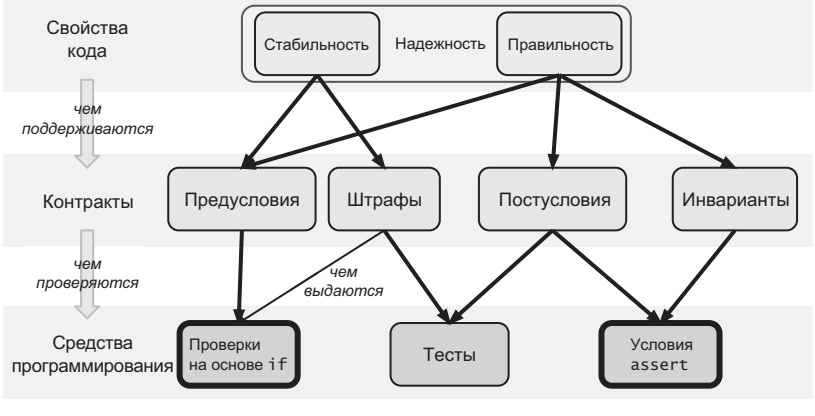


Рис. 5.3. Отношения между атрибутами надежности, спецификациями на базе контрактов и средствами программирования

На рис. 5.3 представлены отношения между двумя свойствами кода, которые входят в понятие надежности, различными типами спецификаций, рассмотренных

ранее, и тремя средствами программирования, которые будут использоваться в этой и следующей главах.

Правильность определяется соблюдением контракта, включающего пред- и постусловия, а также необязательным набором инвариантов класса. Штраф не связан напрямую с правильностью, потому что он вступает в действие только при нарушении предусловия вызывающей стороной. Как следствие, он является проблемой стабильности.

Три средства программирования помогают в реализации и обеспечении соблюдения контрактов:

- Простые проверки на основе `if` гарантируют, что вызывающая сторона вызывает метод так, как положено, с соблюдением его предусловий, в противном случае выдавая соответствующий штраф.
- Директивы `assert` в Java полезны для контроля над постусловиями и инвариантами, особенно в приложениях, критичных по безопасности.
- Наконец, тесты повышают нашу уверенность в надежности кода, в основном за счет проверки постусловий и активизации штрафов.

В этой и следующей главах будут подробно рассмотрены рекомендации, относящиеся к каждому из этих средств. А пока заметим, что первые два пункта являются средствами *мониторинга*, активными в процессе обычного функционирования программного кода. Напротив, тестирование выполняется перед основной работой и отдельно от нее.

5.1.4. Проверка контрактов

Многие ошибки связаны с нарушением предусловий методов. Чтобы эти проблемы выявлялись как можно ранее, методы должны проверять свои предусловия во время выполнения и выдавать подходящее исключение, если они не выполняются. Иногда это называется *защитным программированием*. Для этой цели обычно используются два стандартных класса исключений:

- `IllegalArgumentException` — если значение аргумента нарушает предусловие;
- `IllegalStateException` — если текущее состояние объекта несовместимо с вызываемым методом экземпляра или значениями аргументов, например, при попытке чтения из файла, который уже закрыт.

Директивы `assert` представляют связанный, но более специализированный механизм проверки. Эти команды выглядят так:

```
assert условие : "Error message!";
```

При выполнении строка вычисляет логическое условие и выдает исключение `AssertionError`, если оно ложно. Строка сообщения об ошибке передается

выдаваемому исключению и выводится в том случае, если исключение не перехвачено. `assert` очень похожа на следующую команду:

```
if (!условие) {
   throw new AssertionError("Error message!");
}
```

Сейчас `assert` выглядит как укороченная версия обычной проверки на основе `if`. Но между ними есть принципиальное различие: по умолчанию JVM не выполняет директивы `assert`. Их необходимо явно активизировать ключом командной строки `-ea` или соответствующей настройкой IDE. Если `assert` отключены, то программа не несет затраты, связанные с вычислением соответствующих логических условий.

ДИРЕКТИВЫ ASSERT В C#

Директивы `assert` в C# отличаются от `assert` в Java в двух отношениях: они реализуются вызовом статических методов `Debug.Assert` и `Trace.Assert`, а управление их выполнением осуществляется на стадии компиляции, а не на стадии выполнения. Компилятор игнорирует вызовы `Debug.Assert` при компиляции программы в окончательной версии, тогда как вызовы `Trace.Assert` компилируются и выполняются всегда.

Стандартная проверка на основе `if` выполняется всегда, но если для проверки условия используется `assert`, вы сможете включать или отключать эту проверку при каждом выполнении. Обычно директивы `assert` включаются в процессе разработки, а потом возвращаются в стандартное отключенное состояние в итоговой версии. Похоже, директивы `assert` выигрывают во всех отношениях: они более компактны и дают больше возможностей для управления ими. Стоит ли использовать их для *всех* проверок времени выполнения? Оказывается, в некоторых случаях гибкость, присущая `assert`, становится недостатком. В таких случаях проверки должны оставаться на своем месте всегда, даже в окончательной версии.

Контрактное проектирование предоставляет простые рекомендации для определения, какие проверки должны быть включены постоянно:

- Проверки предусловий для открытых методов должны быть активны всегда, поэтому для них следует использовать обычные условия `if`.
- Все остальные проверки должны быть активны только во время разработки. К их числу относятся постусловия и проверки инвариантов, а также проверки предусловий для методов, не являющихся открытыми. Для них используются директивы `assert`.

Эти рекомендации можно обосновать. Нарушения предусловий происходят оттого, что вызывающая сторона не соблюдает контракт метода. С другой стороны,

постусловие или нарушение инварианта связано с проблемами внутри самого класса. Возьмем следующее ключевое предположение:

> Процессы разработки и тестирование гарантируют, что каждый класс избавлен от внутренних проблем.

Под *внутренней проблемой* я подразумеваю ошибку, которая проявляется даже в том случае, если клиенты класса соблюдают все правила, обусловленные контрактами. На данный момент просто примем это предположение как есть — вскоре мы обсудим его достоверность. Если это предположение выполняется, то некорректное поведение программы может происходить только из-за того, что один класс некорректно используется другим классом. В системе с качественной инкапсуляцией это может происходить только через открытые методы. Следовательно, для выявления таких ошибок будет достаточно проверок предусловий для открытых методов, и их следует оставить включенными. Обратите внимание, что проверка предусловий во время выполнения не исправляет проблему, а только выявляет ее как можно ранее в процессе выполнения, чтобы вы могли найти корневую причину происходящего.

Насколько разумно предположение об отсутствии внутренних проблем? В конечном итоге это зависит от качества и интенсивности процесса разработки. Чем они выше, тем более вероятно, что предусловие будет выполняться. Под качеством процесса разработки я имею в виду применение разработчиками передовых практик отрасли. Под интенсивностью (или трудозатратами) понимается количество людей и время, используемое для разработки — и особенно тестирования — каждого класса. Например, можно ожидать, что только маленькие классы будут полностью избавлены от внутренних проблем. Неспроста написание малых классов считается одним из канонических принципов ООП.

5.1.5. Более широкая картина

На рис. 5.4 методы, описанные в этой и следующей главах, представлены в более широкой перспективе. В книге основное внимание уделяется стилям и приемам программирования, которые даже программист-одиночка может применять в повседневной деятельности. Но по крайней мере два типа взаимодействий между программистами могут внести свой вклад в качество кода, в частности в надежность.

Прежде всего это расширение контроля: коллега-программист просматривает ваш код и оценивает его в соответствии со стандартами компании. Такой контроль может осуществляться на периодических встречах или в виде непрерывного взаимодействия между двумя коллегами (*парное программирование*).

Кроме того, некоторые программные инструменты могут автоматически проверять различные свойства кода, расширяя набор проверок, уже выполняемых

Рис. 5.4. Широкий взгляд на улучшение качества кода

компилятором. Такие инструменты можно разделить приблизительно на три категории от простейших до сложных:

- *Средства проверки стиля* — эти программы могут выполнять только относительно поверхностные проверки, контролирующие удобочитаемость и единообразие (глава 7), косвенно улучшая надежность и удобство сопровождения.

  Пример функциональности: убедиться, что отступы расставлены правильно и имеют постоянный размер (равное количество пробелов на каждом уровне).

  Пример программы: CheckStyle[1].

- *Статические анализаторы* — инструменты, способные выполнять семантический анализ по аналогии с фазой проверки типов компилятором. Средства проверки стиля и статические анализаторы также известны как *линтеры*.

  Пример функциональности: проверить, не содержит ли анонимный класс метод, который не может вызываться (метод, который не переопределяет другой метод и не используется другими методами класса).

  Пример программы: SpotBugs[2], SonarQube[3].

- *Формальные верификаторы* — эти инструменты, рожденные в основном в результате академических исследований, понимают программу на более глубоком уровне, чем типичный компилятор, а именно: они могут моделировать выполнение программы для полных наборов значений — процесс, называемый *символическим выполнением*.

---

[1] На момент написания книги доступна по адресу https://checkstyle.sourceforge.io.
[2] На момент написания книги доступна по адресу https://spotbugs.github.io.
[3] На момент написания книги доступна по адресу www.sonarqube.org.

Пример функциональности: проверить, может ли целочисленная переменная стать отрицательной[1].

Пример программы: KeY[2].

Обычно компания сама выбирает набор практик контроля качества и инструментов, наиболее подходящих для решаемой задачи. Требования к разработке видеоигры сильно отличаются от ожиданий клиента в области здравоохранения или армии. А теперь вернемся к нашей обычной точке зрения, направленной на улучшение удобочитаемости одного блока кода еще до того, как им займутся ваши коллеги-программисты или программные инструменты.

5.2. КОНТРАКТНОЕ ПРОЕКТИРОВАНИЕ РЕЗЕРВУАРОВ

Теперь все готово к тому, чтобы применить рекомендации контрактного проектирования к резервуарам и их эталонной реализации `Reference`. Но сначала определимся с контрактами для методов резервуаров, краткая сводка которых приведена в табл. 5.1. Я не включил в таблицу конструктор, потому что его контракт просто определяет, что он создает пустой резервуар.

Таблица 5.1. Контракты методов Container

Метод	Предусловие	Постусловие	Штраф
`getAmount`	Нет	Возвращает текущий объем в этом резервуаре	Нет
`connectTo`	Аргумент не равен `null`	Объединяет две группы резервуаров и перераспределяет воду	NPE†
`addWater`	Если аргумент отрицателен, в группе достаточно воды	Распределяет воду в равных пропорциях между всеми резервуарами в группе	IAE‡

† NPE = `NullPointerException`

‡ IAE = `IllegalArgumentException`

Как видно из табл. 5.1, контракты всего лишь предоставляют структурированный способ представления ожидаемого поведения методов, который явно отличает предположения от гарантий. По сравнению с описаниями методов из

[1] Технически это свойство является неопределенным. Формальный верификатор попытается доказать его или опровергнуть, но успешность таких попыток не гарантирована.

[2] На момент написания книги доступна по адресу www.key-project.org.

главы 1 эти контракты добавляют описание предусловий и соответствующих штрафов:

- `connectTo` требует, чтобы его аргумент был отличен от `null`, и при нарушении должен выдавать исключение `NullPointerException` (NPE);
- `addWater` при использовании с отрицательным аргументом (допустим, `-x`) требует, чтобы общий объем воды в резервуарах, соединенных с текущим, был не меньше `x`, и при нарушении должен выдавать исключение `IllegalArgumentException` (IAE).

Оба стандартных класса исключений *непроверяемы* и являются подклассами `RuntimeException`.

Предусловие, которое требует, чтобы аргумент был отличен от `null`, встречается очень часто — как и путаница с типом исключения, наиболее подходящим для этого случая. Следующая врезка проливает свет на этот вопрос.

NULLPOINTEREXCEPTION ИЛИ ILLEGALARGUMENTEXCEPTION

Какое исключение должно выдаваться при получении недопустимого аргумента null — NPE или IAE? Этот вопрос породил целую серию вопросов и ответов на сайте StackOverflow и в известной книге «Java. Эффективное программирование». Вероятно, этот факт свидетельствует о внимании программистов к деталям.

Ниже перечислены основные аргументы в пользу обоих вариантов.

NPE:

- Сразу видно, какое фактическое значение вызвало проблему.

IAE:

- Сразу видно, что проблема обусловлена нарушением предусловия.
- Четко отличается от исключения NPE, генерируемого JVM.

Хотя аргументы в пользу IAE выглядят сильнее, по общепринятым соглашениям предпочтение обычно отдается NPE, на что указывает авторитетная книга «Java. Эффективное программирование» (пункт 72 в третьем издании) и вспомогательные методы из класса `Object`:

```
public static Object requireNonNull(Object x)
public static Object requireNonNull(Object x, String message)
```

Эти методы выдают NPE, если значение `x` равно `null`, в противном случае возвращая сам `x`. Начиная с Java 7, рекомендуется именно так обеспечивать проверку того, что параметры отличны от `null`.

Теперь рассмотрим инварианты классов. В идеале инварианты должны точно описывать, какие состояния объектов соответствуют контрактам. А если подробнее, они должны сообщать, какие значения полей могут быть получены после серий допустимых операций. Для реализации `Reference` это приводит к следующим инвариантам:

I1. Для каждого резервуара поле `amount` неотрицательно.

I2. Каждый резервуар принадлежит ровно одной группе.

I3. Для каждого резервуара поле `group` отлично от `null` и указывает на группу, содержащую `this`.

I4. Все резервуары, принадлежащие одной группе, содержат одно и то же значение `amount`.

Посмотрим, как эти инварианты относятся к контрактам в табл. 5.1. Инвариант I1 интуитивен: резервуар не может включать отрицательный объем воды. Предусловие `addWater` отвечает за защиту этого инварианта против внешних атак, а именно попыток понизить уровни воды ниже нуля. Инварианты I2 и I3 являются следствием нашей политики, относящейся к группам резервуаров: все они начинают с одного резервуара, а затем объединяются попарно. Конструктор устанавливает эти инварианты, а метод `connectTo` должен сохранить их правильным объединением групп. Наконец, инвариант I4 утверждает связь между группами и объемами. `addWater` и `connectTo` обязаны поддерживать эту связь, выраженную их постусловиями.

Интересное упражнение — проверить, что четыре инварианта являются *полными*. Построить любой пул резервуаров, удовлетворяющих им, «с нуля» произвольной последовательностью вызовов конструкторов и методов, если при удалении любого инварианта исчезнет возможность существования такого пула[1].

НЕОЖИДАННЫЙ ВОПРОС 3

Является ли следующее утверждение действительным инвариантом для класса `Container`: «При передаче нуля методу `addWater` все резервуары остаются неизменными»?

Итак, после того как мы четко сформулировали контракты и инварианты, ими можно воспользоваться для укрепления правильности и стабильности реализации `Reference`. Путь к предусловиям и постусловиям вполне понятен: мы проверяем их в начале и в конце их метода, разумно используя `if` или `assert`.

---

[1] Например, если удалить инвариант I1, появляется возможность существования изолированного резервуара с отрицательным объемом воды. Воспроизвести такой сценарий допустимой последовательностью вызовов конструкторов и методов не удастся.

Что касается инвариантов, необходимо определиться с тем, когда их следует проверять, то есть с какой частотой и в каких точках программы. Вспомните, что инварианты должны выполняться в начале и в конце каждого (открытого) метода. В одном крайнем случае можно проверять все инварианты во всех этих точках. В другом крайнем случае от проверки инвариантов можно вообще отказаться, потому что *нормальная проверка пред- и постусловий автоматически гарантирует выполнение инвариантов*. Слабость такого подхода проявляется в слове *нормальная*. В следующем разделе мы реализуем версию `Container`, в которой каждый метод тщательно проверяет свои пред- и постусловия, и вы увидите, насколько сложно и затратно выполнение этих проверок. Затем в разделе 5.4 постусловия будут заменены инвариантами, которые обычно проще проверяются.

Все версии `Container` в этой главе используют те же поля, что и `Reference`; повторим их для удобства:

```
public class Container {
   private Set<Container> group;   ❶ Резервуары, соединенные с текущим
   private double amount;          ❷ Объем воды в текущем резервуаре
```

5.3. РЕЗЕРВУАРЫ, ПРОВЕРЯЮЩИЕ СВОИ КОНТРАКТЫ [CONTRACTS]

В этом разделе будет разработана версия `Container`, методы которой проверяют свои предусловие и постусловие при каждом вызове.

5.3.1. Проверка контракта addWater

Начнем с метода `addWater`. Его контракт уже приводился в табл. 5.1, но я повторю его для удобства:

- *Предусловие*: если аргумент отрицателен, то в группе достаточно воды.
- *Постусловие*: вода распределяется в равных пропорциях между всеми резервуарами в группе.
- *Штраф:* `IllegalArgumentException`.

Метод имеет простое предусловие, которое можно проверить стандартной командой `if` в соответствии с правилами, рассмотренными ранее.

Для проверки постусловий следует применять директивы `assert`, чтобы эти проверки можно было легко отключить в рабочей версии кода. До сих пор я выражал постусловие `addWater` в несколько неопределенной форме. Что значит — равномерно распределить воду? Очевидно, в конце метода все резервуары в группе должны содержать одинаковый объем воды. Но это еще не все. Общий объем

воды во всей группе должен быть равен сумме старого общего объема и объема добавленной воды. Чтобы проверить это свойство, необходимо сохранить некоторую информацию в начале метода. Затем эта информация используется в конце для сравнения предполагаемого изменения состояния объекта с фактическим.

Ситуация наводит на мысль, что метод должен состоять из четырех шагов:

1. Проверка предусловия простой командой `if`.
2. Сохранение текущего объема воды в группе во временной переменной для последующей проверки постусловия.
3. Выполнение операции добавления воды.
4. Проверка постусловия с использованием данных, сохраненных на шаге 2.

Кроме того, нужно помнить, что при отключении `assert` должны исчезнуть все затраты времени и памяти. Поэтому шаги 2 и 4 следует выполнять только при включенных директивах `assert`. На шаге 4 это делается просто: достаточно вызвать `postAddWater` как условие `assert`. С шагом 2 дело обстоит сложнее, потому что он не имеет естественного выражения в форме `assert`. Чтобы преобразовать его в `assert`, необходимо упаковать присваивание в фиктивное сравнение (листинг 5.1), которое всегда дает результат `true`. В данном случае можно проверить, что старый объем воды в группе положительный. С этим приемом единственной оставшейся затратой ресурсов даже при отключении `assert` будет выделение памяти для переменной `oldTotal` в стеке[1].

НЕОЖИДАННЫЙ ВОПРОС 4

Как присвоить флагу значение `true`, только когда директивы `assert` включены?

В листинге 5.1 приведена возможная реализация, которая делегирует шаги 2 и 4 двум новым вспомогательным методам.

```
public void addWater(double amount) {
   double amountPerContainer = amount / group.size();
   if (this.amount + amountPerContainer < 0) { ❶ Проверка предусловия
      throw new IllegalArgumentException(
                "Not enough water to match the addWater request.");
   }
   double oldTotal = 0; ❷.❼ Сохранение данных для постусловия
   assert (oldTotal = groupAmount()) >= 0; ❷.❶ Фиктивная директива assert

   for (Container c: group) { ❸ Фактическое обновление
```

[1] Если вам не нравится прием с фиктивной директивой `assert`, используйте альтернативное решение — флагу присваивается `true` при включенных `assert` (Как? См. неожиданный вопрос 4), а обычная команда `if` используется для пропуска некоторых операций при отключенных `assert`.

```
      c.amount += amountPerContainer;
   }
   assert postAddWater(oldTotal, amount) : ❹ Проверка постусловия
               "addWater failed its postcondition!";
}
```

Реализация `addWater` в листинге 5.1 делегирует две задачи новым вспомогательным методам: `groupAmount` вычисляет новый объем воды в группе резервуаров, а `postAddWater` отвечает за проверку постусловия `addWater`. Код `groupAmount` просто суммирует значения всех полей `amount` в текущей группе:

Листинг 5.2. Contracts: вспомогательный метод groupAmount

```
private double groupAmount() { ❶ Возвращает общий объем воды в группе
   double total = 0;
   for (Container c: group) { total += c.amount; }
   return total;
}
```

В свою очередь, метод `postAddWater` разбивает свою задачу на две: сначала он проверяет, что все резервуары в текущей группе содержат одинаковый объем воды, а затем — что общий объем воды в группе равен сумме старого объема и добавленного объема. (Версия `postAddWater` в листинге 5.3 является предварительной — ниже будет приведена улучшенная версия.)

Листинг 5.3. Contracts: предварительная версия вспомогательного метода postAddWater

```
private boolean postAddWater(double oldTotal, double addedAmount) {
   return isGroupBalanced() &&
         groupAmount() == oldTotal + addedAmount; ❶ Точное сравнение значений double
}

private boolean isGroupBalanced() { ❷ Проверяет, что все группы содержат одинаковый
   for (Container x: group) {          объем воды
      if (x.amount != amount) return false;
   }
   return true;
}
```

Как видите, проверка постусловия требует больше строк кода, чем исходный незащищенный метод! Простое количество строк может привести к выводу, что при программировании постусловия вероятность ошибиться выше, чем при написании исходного метода. Оправданны ли эти усилия? Если проверка сводится к простому повторению тех же вычислений, которые выполнялись методом, то усилия определенно бессмысленны. Тем не менее если вы найдете другой и, хотелось бы надеяться, более простой способ проверки правильности результата, то два разных алгоритма будут проверять друг друга. Даже ошибка в процедуре постусловия поможет вам лучше понять имеющийся класс.

Теперь вы запускаете свою версию `addWater` на простом примере с включенными директивами `assert`, и... она ломается! VM сообщает об ошибке в постусловии `addWater`. Сбой `assert` генерируется следующим кодом — удастся ли вам обнаружить проблему?

```
Container a = new Container(), b = new Container(), c = new Container();
a.connectTo(b);
b.connectTo(c);
a.addWater(0.9);
```

Проблема кроется в сравнении двух значений `double` в `postAddWater` (листинг 5.3). Если вы не используете числа с плавающей точкой регулярно, легко забыть, что они не полностью повторяют условные вещественные числа. Как следствие, результат выражения `(a/b) * b` отличен от `a.`

Например, число 0,9 не имеет точного представления в двоичном виде. Его двоичная запись периодична, поэтому число будет храниться в приближенном виде. При делении его на 3 и добавлении к трем резервуарам будут выполнены другие приближенные вычисления. В итоге, когда вы просуммируете объемы в каждом резервуаре в группе, сумма будет несколько отличаться от ожидаемой. При суммировании мы вычисляли объем воды в группе двумя разными способами, а затем сравнивали результаты оператором `==`. Из-за приближенного представления две стороны не будут в точности равны. Подробный анализ вычислений выходит за рамки книги, но информацию о нем можно найти в ресурсах, перечисленных в разделе «Дополнительная литература» в конце главы. В нашей ситуации после вызова `addWater` будут получены следующие значения:

```
expected amount: 0.9
actual amount: 0.8999999999999999
```

Все это наводит на мысль, что сравнения с плавающей точкой всегда должны выполняться в пределах некоторой погрешности, размер которой зависит от диапазона обрабатываемых чисел. Допустим, объем жидкости измеряется в литрах (или галлонах), а наши резервуары способны вмещать десятки и сотни литров. В таком сценарии можно безопасно не учитывать отдельные капли воды, поэтому разумно установить погрешность равной, скажем, $0{,}0001 = 10^{-4}$ литра — приблизительному объему одной капли.

В результате мы получим следующую улучшенную версию `postAddWater`.

Листинг 5.4. Contracts: вспомогательные методы postAddWater и almostEqual

```
private boolean postAddWater(double oldTotal, double addedAmount) {
   return isGroupBalanced() &&
          almostEqual(groupAmount(), oldTotal + addedAmount);
}
```

```
private static boolean almostEqual(double x, double y) {
   final double EPSILON = 1E-4;    ❶ Погрешность округления
   return Math.abs(x-y) < EPSILON;
}
```

НЕОЖИДАННЫЙ ВОПРОС 5

Что произойдет, если передать `addWater` «не число» (`Double.NaN`)?

5.3.2. Проверка контракта connectTo

Рассмотрим метод `connectTo` и его контракт:

- *Предусловие*: аргумент отличен от `null`.
- *Постусловие:* две группы резервуаров объединяются, и перераспределяется вода.
- *Штраф:* `NullPointerException`.

Подобные предусловия встречаются так часто, что JDK предоставляет способ их стандартной реализации — статический метод `Objects.requireNonNull(arg,msg)`. Как объяснялось ранее, метод выдает NPE с пользовательским сообщением, если значение `arg` равно `null`, в остальных случаях выдавая само значение `arg`.

С другой стороны, правильная проверка постусловия создает немало проблем. Сначала постусловие преобразуется в список проверок, которые должны быть выполнены с полями экземпляра. Обозначим через `G` множество резервуаров, на которые `this.group` указывает в конце `connectTo(other)`. Постусловие требует, чтобы выполнялись следующие свойства:

1. Множество `G` не пустое, а его элементами являются резервуары, принадлежащие двум старым группам `this` и `other`.
2. Все резервуары в `G` должны указывать на `G` по ссылке `group`.
3. Все резервуары в `G` должны содержать одинаковые значения `amount`, равные общему объему воды в двух старых группах, разделенному на количество резервуаров в `G`.

Чтобы проверить свойство 1, необходимо сохранить старые группы `this` и `other` перед слиянием в начале `connectTo`. Метод может изменять эти группы, поэтому сохранять нужно копии. Свойство 2 можно проверить без предварительной информации — достаточно перебрать все резервуары в `G` и убедиться, что их поле `group` указывает на `G`. Наконец, для проверки свойства 3 необходимо знать значения полей `amount` перед слиянием (или по крайней мере сумму этих значений во всех резервуарах, соединенных с `this` или `other`). Подведем итог. Перед слиянием необходимо сохранить следующую информацию:

- копии групп `this` и `other`;
- общий объем воды в этих группах.

Введем вложенный класс `ConnectPostData` для хранения этой информации:

Листинг 5.5. Contracts: вложенный класс ConnectPostData

```
private static class ConnectPostData { ❶ Хранит данные, необходимые для проверки
   Set<Container> group1, group2;          постусловия
   double amount1, amount2;
}
```

Теперь можно написать заготовку кода `connectTo` по той же схеме из четырех шагов, что в `addWater`. Как и прежде, при отключенных директивах `assert` следует стараться держать затраты на минимальном уровне. В листинге 5.6 единственной затратой при отключенных `assert` остается выделение памяти для локальной переменной `postData` (в пятой строке). Этот эффект достигается внедрением вызова `saveConnectPostData` в фиктивную директиву `assert`, которая всегда дает успешный результат (в шестой строке).

Код, в котором реализуется соединение групп, будет тем же, что для `Reference`, поэтому я исключил его из следующего листинга для удобочитаемости.

Листинг 5.6. Contracts: метод connectTo (сокращенный)

```
public void connectTo(Container other) {
   Objects.requireNonNull(other, ❶ Проверка предусловия
           "Cannot connect to a null container.");
   if (group==other.group) return;

   ConnectPostData postData = null; ❷ Подготовка данных постусловия
   assert (postData = saveConnectPostData(other)) != null; ❸ Фиктивная
                                                            директива assert
                                  ... ❹ Реализация (то же, что для Reference)

   assert postConnect(postData) : ❺ Проверка постусловия
          "connectTo failed its postcondition!";
}
```

Методы `saveConnectPostData` и `postConnect` соответственно сохраняют необходимую информацию и используют ее для проверки, выполняется ли постусловие:

Листинг 5.7. Contracts: методы saveConnectPostData и postConnect

```
private ConnectPostData saveConnectPostData(Container other) {
   ConnectPostData data = new ConnectPostData();
   data.group1 = new HashSet<>(group); ❶ Поверхностная копия
   data.group2 = new HashSet<>(other.group);
   data.amount1 = amount;
   data.amount2 = other.amount;
```

```
   return data;
}

private boolean postConnect(ConnectPostData postData) {
   return areGroupMembersCorrect(postData)
      && isGroupAmountCorrect(postData)
      && isGroupBalanced()
      && isGroupConsistent();
}
```

Ради удобочитаемости метод `postConnect` делегирует свою задачу четырем разным методам, роли которых перечислены в табл. 5.2.

Код `isGroupBalanced` уже приводился ранее (листинг 5.3). А теперь кратко рассмотрим код, который проверяет, правильно ли прошло слияние старых групп (листинг 5.8). Сначала он убеждается, что новая группа содержит все элементы из двух старых групп (вторая и третья строки). Чтобы удостовериться, что новая группа не содержит *дополнительных* элементов[1], он проверяет, что размер новой группы равен сумме размеров двух старых групп (четвертая строка).

Табл. 5.2. Четыре метода, используемых для проверки постусловия connectTo

Метод	Проверяемое свойство
`areGroupMembersCorrect`	Новая группа содержит все элементы из двух старых групп
`isGroupConsistent`	Каждый контейнер в новой группе указывает на группу
`isGroupAmountCorrect`	Размер новой группы равен сумме размеров двух старых групп
`isGroupBalanced`	Все резеруары в новой группе имеют равное количество воды

Листинг 5.8. Contracts: вспомогательный метод areGroupMembersCorrect

```
private boolean areGroupMembersCorrect(ConnectPostData postData) {
   return group.containsAll(postData.group1)
      && group.containsAll(postData.group2)
      && group.size() == postData.group1.size() +
                         postData.group2.size();
}
```

[1] Можно сказать, что проверка размера избыточна. В самом деле, если у всех резервуаров выполняются инварианты перед вызовом `connectTo`, то `connectTo` не сможет получить доступ к резервуару, не входящему в одну из двух объединяемых групп. Даже ошибочная реализация может создать группу *меньшего* размера, но не *большего*.

АВТОМАТИЧЕСКИ ПРОВЕРЯЕМЫЕ КОНТРАКТЫ

В этой книге я представляю контракты как дисциплину для структурирования решений вокруг четко определенных API. Некоторые языки и средства программирования поднимают эту концепцию на новый уровень: они позволяют формально определять контракты на специальном языке и автоматически проверять их соответствующей программой, статически (во время компиляции) или динамически (во время выполнения).

Например, язык программирования Eiffel поддерживает пред- и постусловия в виде команд `require` и `ensure`. Вполне закономерно, что Eiffel создал Бертран Мейер, также разработавший методологию контрактного проектирования. Язык даже позволяет постусловиям обратиться к старому значению поля при входе в текущий метод. Можно приказать компилятору преобразовать аннотации контракта в проверки времени выполнения.

В Java встроенная поддержка контрактов отсутствует, но существует ряд инструментов, которые пытаются заполнить этот пробел, например KeY (www.key-project.org) и Krakatoa (http://krakatoa.lri.fr). Обе системы поддерживают спецификации, написанные на языке Java Modeling Language, и предоставляют полуавтоматическую статическую проверку контрактов.

5.4. РЕЗЕРВУАРЫ, ПРОВЕРЯЮЩИЕ СВОИ ИНВАРИАНТЫ [INVARIANTS]

Из предыдущего раздела видно, насколько сложной бывает правильная проверка постусловий. Удобная альтернатива заключается в периодической проверке инвариантов. Напомню инварианты, которые ранее были установлены для `Reference` в этой главе:

I1. Для каждого резервуара поле `amount` неотрицательно.

I2. Каждый резервуар принадлежит ровно одной группе.

I3. Для каждого резервуара поле `group` отлично от `null` и указывает на группу, содержащую `this`.

I4. Все резервуары, принадлежащие одной группе, содержат одно и то же значение `amount`.

Если класс написан правильно, а его клиенты используют его так, как положено (с соблюдением предусловий всех методов), все постусловия и инварианты

будут выполняться. Программная ошибка в методе может привести к нарушению постусловия. В свою очередь, нарушение постусловия может привести к нарушению инварианта. Если предполагать, что предусловия соблюдаются, нарушение постусловия предшествует и становится причиной нарушения инвариантов. С другой стороны, нарушение постусловия не обязательно преобразуется в нарушение инварианта.

Допустим, что метод `addWater` содержит следующую ошибку: по запросу на добавление x литров воды он добавляет только $x/2$ литров. Так как метод оставляет все объекты в корректном состоянии, эта реализация пройдет все проверки инвариантов. Дело в том, что инварианты представляют собой *статические проверки*, которые относятся только к текущему состоянию объектов. С другой стороны, они терпят фиаско при проверке постусловия, выполняемого в `Contracts`.

Подведем итог: проверка постусловий, как это делалось в предыдущем разделе, обычно безопаснее, но обходится дороже. Напротив, проверка инвариантов проще, но иногда сопряжена с риском: ошибки программирования, которые обнаруживаются проверкой постусловия, могут пройти проверку инвариантов.

Когда следует проверять инварианты? Как я уже говорил, можно проверять их в начале и в конце всех методов, а также в конце всех конструкторов. Это стандартное, хотя и радикальное решение, которое можно применять в любом контексте. С другой стороны, можно действовать умнее и избежать лишних проверок, сосредоточившись на тех методах, которые могут привести к нарушению инварианта.

Предположим, вы доверяете конструктору, который должен изначально установить все инварианты. Конструктор `Reference` настолько прост, что на это вполне можно рассчитывать. Какие методы могут нарушить инварианты? Инварианты являются свойствами состояний объектов, так что только те методы, которые изменяют состояние полей, могут стать потенциальными нарушителями инвариантов.

Рассмотрим три открытых метода `Reference`:

- Очевидно, метод `getAmount` используется только для чтения, а значит, не может нарушить никаких инвариантов.
- Метод `addWater` изменяет поле `amount`, поэтому он теоретически может нарушить инварианты I1 и I4 всех резервуаров, с которыми работает.
- Наконец, метод `connectTo` является наиболее критичным, потому что он изменяет оба поля многих резервуаров. При неправильной реализации он может нарушить все инварианты этих резервуаров.

Эти наблюдения обобщены в табл. 5.3.

Таблица 5.3. Что изменяет каждый метод и какие инварианты он может нарушить

Метод	Изменяемые поля	Инварианты, которые может нарушить
`getAmount`	–	–
`connectTo`	`amount` и `group`	I1, I2, I3, I4
`addWater`	`amount`	I1, I4

Один из способов предотвращения лишней работы заключается в том, чтобы проверять инварианты только *в конце тех методов, которые могут их нарушить*. Мы реализуем эти проверки с использованием директив `assert`, фактически рассматривая инварианты как постусловия этих методов. Такое упрощение безопасно в том смысле, что вы все еще можете связать нарушение инварианта с тем методом, который его вызвал. Действительно, так как состояние объектов начинается с соблюдением инвариантов и оно правильно инкапсулировано (то есть приватно), только открытые методы могут нести ответственность за их нарушение. Как только в одном открытом методе произойдет ошибка, она проявится как нарушение директивы `assert` из этого метода.

В соответствии с табл. 5.3, мы можем сосредоточиться на методах `connectTo` и `addWater`. Оба этих метода способны изменять состояние объектов, поэтому необходимо проверять инварианты всех объектов, с которыми они работают. Это особенно неудобно для метода `connectTo`: согласно его контракту, вызов вида `a.connectTo(b)` должен изменять состояние всех резервуаров, которые в начале метода соединены с `a` или `b`. Однако в той точке, в которой мы планируем проверять инварианты (то есть в конце метода), мы не знаем, какие резервуары были ранее соединены с `a` или `b`, и можем только неявно доверять правильности реализации самого метода.

5.4.1. Проверка инвариантов в connectTo

Как было показано в предыдущем обсуждении, проверка инвариантов в конце `connectTo` может проходить двумя путями:

1. В начале метода сохранить текущие группы `this` и `other` (вернее, их копии), чтобы потом проверить инварианты всех задействованных объектов.
2. Проверять состояние только тех объектов, которые входят в единую сформированную группу, в конце метода.

Вариант 1 безопаснее, но он напоминает тяжелую работу, которую мы выполняли в предыдущем разделе для проверки постусловия `connectTo`. Я рекомендую вариант 2: он более практичен и обеспечивает частичное доверие к методу. Он предполагает, что метод правильно объединяет две ранее

существовавшие группы в одну и проверяет все остальные свойства, участвующие в инварианте.

У вас должно получиться что-то вроде листинга 5.9, где задача проверки инварианта доверяется приватному вспомогательному методу. Середина `connectTo` не приводится, так как она полностью совпадает с кодом `Reference`.

Листинг 5.9. Invariants: метод connectTo (сокращенный) и его вспомогательный метод

```
public void connectTo(Container other) {
   Objects.requireNonNull(other, ❶ Проверка предусловия
                          "Cannot connect to a null container.");

   ... ❷ Реализация (то же, что для Reference)

   assert invariantsArePreservedByConnectTo(other) :
          "connectTo broke an invariant!"; ❸ Проверка инвариантов
}

private boolean invariantsArePreservedByConnectTo(Container other) {
   return group == other.group &&
          isGroupNonNegative() &&
          isGroupBalanced() &&
          isGroupConsistent();
}
```

При выборе варианта 2 не нужно сохранять состояние каких-либо объектов в начале `connectTo`. Вы можете проверить предусловие (во второй строке), выполнить операцию соединения (как в `Reference`) и, наконец, проверить (упрощенные) инварианты (в строке `assert`).

В проверке инвариантов задействованы еще три вспомогательных метода. Реализация `isGroupBalanced` уже была приведена в предыдущем разделе. Сейчас видно, что она проверяет инвариант I4. Вот два других метода проверки инвариантов:

Листинг 5.10. Invariants: два вспомогательных метода для проверки инвариантов

```
private boolean isGroupNonNegative() { ❶ Проверка инварианта I1
   for (Container x: group) {
      if (x.amount < 0) return false;
   }
   return true;
}

private boolean isGroupConsistent() {  ❷ Проверка инвариантов I2, I3
   for (Container x: group) {
      if (x.group != group) return false;
   }
   return true;
}
```

Чтобы убедиться, что мы не перехватываем все нарушения инвариантов, возьмем сценарий на рис. 5.5. В левой части («До») три резервуара объединены в две группы: резервуар `a` изолирован, `b` и `c` соединены. Объем воды в этом примере нас не интересует; будем считать, что он одинаковый во всех резервуарах. Такое состояние дел удовлетворяет всем инвариантам.

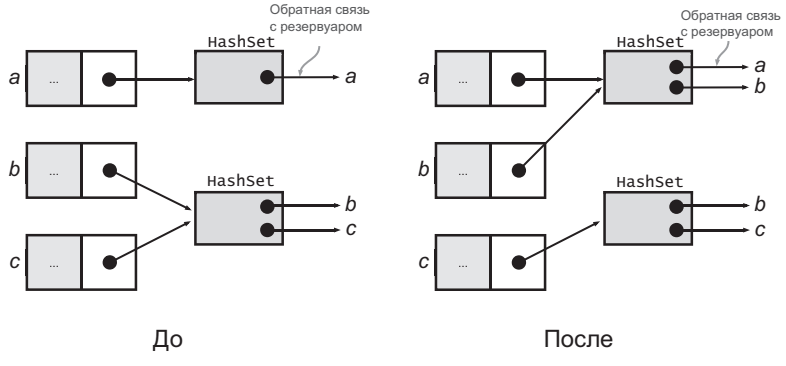


Рис. 5.5. Ситуация до и после некорректной операции a.connectTo(b). Проверка в Invariants не выявляет подобные нарушения. Объемы воды не указаны как несущественные

Теперь представьте, что некорректная реализация `a.connectTo(b)` приводит к ситуации в правой части. Вместо объединения всех резервуаров в одну группу эта реализация обновляет группу `a`, включая в нее `a` и `b`, а резервуар `b` теперь указывает на свою новую группу. При этом резервуар `c` и его группа остаются без изменений. Как следствие, резервуар `c` все еще «верит», что он входит в группу, включающую `b` и `c`.

В этой ситуации нарушается инвариант I2, потому что `b` принадлежит двум разным группам, но `Invariants` не обнаруживает проблему. В самом деле, выбирая вариант 2, который был описан ранее, вы проверяете только то, что два соединяемых резервуара (`a` и `b`) ссылаются на одну группу, а поле `group` всех резервуаров в этой группе действительно ссылается на эту группу (метод `isGroupConsistent`).

Ошибка на рис. 5.5 была бы выявлена, если бы мы выбрали вариант 1 вместо варианта 2. Кроме того, `Contracts` обнаружил бы ее как нарушение постусловия.

5.4.2. Проверка инвариантов в addWater

Реализация `addWater` следует той же схеме, что и `connectTo`. Как обсуждалось ранее (табл. 5.3), достаточно проверить соблюдение инвариантов I1 и I4, потому что только эти инварианты могут быть теоретически нарушены методом `addWater`.

Как видно из листинга 5.11, проверка инвариантов доверяется приватному вспомогательному методу, который вызывает два других метода, уже встречавшихся в предыдущих разделах.

Листинг 5.11. Invariants: метод addWater и его вспомогательный метод

```
public void addWater(double amount) {
   double amountPerContainer = amount / group.size();

   if (this.amount + amountPerContainer < 0) { ❶ Проверка предусловия
      throw new IllegalArgumentException(
         "Not enough water to match the addWater request.");
   }

   for (Container c: group) {
      c.amount += amountPerContainer;
   }
   assert invariantsArePreservedByAddWater() : ❷ Проверка инвариантов
          "addWater broke an invariant!";
}

private boolean invariantsArePreservedByAddWater() {
   return isGroupNonNegative() && isGroupBalanced();
}
```

5.5. А ТЕПЕРЬ СОВСЕМ ДРУГОЕ

Применим приемы, описанные в этой главе, в другом, более сухом примере (ни капли воды). Возьмем класс `BoundedSet<T>`, представляющий множество с ограниченным размером и возможностью отслеживания порядка вставки элементов. Фиксированный максимальный размер `BoundedSet`, который называется *емкостью*, устанавливается при вызове конструктора. Класс содержит следующие методы:

- `void add(T elem)` добавляет заданный элемент в ограниченное множество. Если в результате добавления размер множества превышает его емкость, то метод удаляет из множества *самый старый* элемент (вставленный первым). Добавление элемента, который уже принадлежит множеству, приводит к его *обновлению* (то есть делает его самым новым элементом во множестве);
- `boolean contains(T elem)` возвращает `true`, если ограниченное множество содержит заданный элемент.

Такая функциональность часто применяется для хранения небольшого количества часто используемых элементов данных (аналог кэша). Ее пример — команда меню для открытия в программе последних документов и возможность выбора недавно использовавшихся программ в меню Windows Пуск.

5.5.1. Контракты

Первым шагом на пути к созданию надежной реализации станет подробное формулирование контрактов методов с четким выделением предусловий и постусловий. В нашем случае к неформальным описаниям методов добавлять почти ничего не придется, потому что эти два метода не имеют предусловий: их можно вызывать в любой момент с любым аргументом (кроме `null`). Для `add` мы получим следующий контракт:

- *Предусловие:* аргумент отличен от `null`.
- *Постусловие:* заданный элемент добавляется в ограниченное множество. Если в результате добавления размер множества превышает его емкость, то метод удаляет из множества *самый старый* элемент (вставленный первым). Добавление элемента, который уже принадлежит множеству, приводит к его *обновлению* (то есть делает его самым новым элементом в множестве).
- *Штраф:* `NullPointerException`.

Для `contains` можно явно выразить в постусловии, что метод не изменяет свое множество:

- *Предусловие:* аргумент отличен от `null`.
- *Постусловие:* метод возвращает `true`, если ограниченное множество содержит заданный элемент. Это ограниченное множество при этом не изменяется.
- *Штраф:* выдает `NullPointerException`.

5.5.2. Эталонная реализация

Начнем с простой реализации `BoundedSet`. Она поможет более четко увидеть затраты, связанные с проверками контрактов. Начнем с выбора внутреннего представления ограниченного множества. Связанный список станет хорошим кандидатом, потому что он позволяет хранить элементы отсортированными по времени вставки и эффективно удалять самый старый элемент специальным методом `removeFirst`. Но это не значит, что вставка в ограниченном множестве будет выполняться за постоянное время. Чтобы обновить элемент, уже присутствующий во множестве, необходимо просканировать список, удалить элемент из его текущей позиции, а затем добавить его в начало списка, а на это потребуется линейное время.

Базовая структура класса выглядит так:

```
public class BoundedSet<T> {
   private final LinkedList<T> data;
   private final int capacity;
```

```
   public BoundedSet(int capacity) {  ❶ Конструктор
      this.data = new LinkedList<>();
      this.capacity = capacity;
}
```

Далее следуют два метода. Как видите, связанный список позволяет написать очень простую реализацию, за что приходится расплачиваться снижением производительности (один из типичных компромиссов, которые так часто встречаются в этой книге).

```
public void add(T elem) {
   if (elem==null) {
      throw new NullPointerException();
   }
   data.remove(elem); ❶ Удаляет elem, если он уже присутствует
   if (data.size() == capacity) {
      data.removeFirst(); ❷ Если список полон, удалить самый старый элемент
   }
   data.addLast(elem); ❸ Добавляет elem как самый новый элемент
}
public boolean contains(T elem) {
   return data.contains(elem);
}
```

5.5.3. Проверка контрактов

Как это ранее делалось с резервуарами для воды, спроектируем защищенную реализацию `BoundedSet`, методы которой активно проверяют свои контракты.

Сосредоточимся на постусловии `add` — самой интересной части обоих контрактов. Так как метод `add` должен изменять состояние ограниченного множества определенным образом, защищенный метод `add` должен начать с создания копии текущего состояния ограниченного множества. В конце `add` приватный вспомогательный метод будет сравнивать текущее состояние ограниченного множества с копией, созданной в начале `add`, и убеждаться в том, что она была изменена в соответствии с контрактом.

В современных условиях для предоставления функциональности копирования в классе рекомендуется использовать *копирующий конструктор*[1], то есть конструктор, получающий другой объект того же класса:

```
public BoundedSet(BoundedSet<T> other) { ❶ Копирующий конструктор
   data = new LinkedList<>(other.data);
   capacity = other.capacity;
}
```

[1] Если вас интересует, почему этот способ следует использовать вместо метода `clone` в новых классах, обратитесь к пункту13 книги «Java. Эффективное программирование».

Как упоминалось при обсуждении резервуаров с водой, вы должны проследить за тем, чтобы все, связанное с проверкой постусловия, включая исходное копирование, выполнялось при включенных директивах `assert`. Как и прежде, эта задача решается упаковкой исходного копирования в фиктивную директиву `assert`.

```
public void add(T elem) {
   BoundedSet<T> copy = null;
   assert (copy = new BoundedSet<>(this)) != null; ❶ Фиктивная директива assert

   ... ❷ Реализация

   assert postAdd(copy, elem) : ❸ Проверка постусловия
          "add failed its postcondition!";
}
```

Наконец, в следующем листинге приведен приватный вспомогательный метод, отвечающий за непосредственную проверку постусловия. Сначала он проверяет, что добавленный элемент находится в начале текущего списка. Затем он создает копию текущего списка, чтобы удалить новейший элемент из текущего и старого списка. В этот момент он сравнивает позиции всех остальных элементов до и после вызова `add` — они должны совпадать. Эту проверку можно поручить методу `equals` списков.

```
private boolean postAdd(BoundedSet<T> oldSet, T newElement) {
   if (!data.getLast().equals(newElement)) { ❶ newElement должен быть в начале
      return false;
   }
   ❷ Удаление newElement из старого и нового множества
   List<T> copyOfCurrent = new ArrayList<>(data);
   copyOfCurrent.remove(newElement);
   oldSet.data.remove(newElement);
   if (oldSet.data.size()==capacity) { ❸ Если список полон, удалить самый старый элемент
      oldSet.data.removeFirst();
   }
   ❹ Все остальные объекты должны остаться неизменными и следовать в том же порядке
   return oldSet.data.equals(copyOfCurrent);
}
```

Как и в случае с водяными резервуарами, проверка постусловия требует бóльших усилий, чем рассматриваемая операция `add`, — как на стадии программирования, так и во время выполнения. Это доказывает, что подобные проверки следует зарезервировать для особых ситуаций, например для программ, критичных по безопасности, или для особенно сложных функций.

5.5.4. Проверка инвариантов

Вспомните, что инвариант представляет собой статическое свойство полей класса, которое должно выполняться в любых ситуациях, кроме изменения

объекта под воздействием метода. При выбранном представлении ограниченных множеств (поля `data` и `capacity`) действительное ограниченное множество характеризуется только двумя свойствами целостности:

I1. Длина списка данных не должна превышать его емкость.

I2. В данных списка не должно быть дубликатов.

Любой список и любое целое число, удовлетворяющие этим двум инвариантам, являются допустимыми, и теоретически можно получить их законной серией операций с изначально пустым ограниченным множеством. Эти инварианты можно проверять в приватном вспомогательном методе:

```
private boolean checkInvariants() {
   if (data.size() > capacity) ❶ Инвариант I1
      return false;
   ❷ Далее идет инвариант I2
   Set<T> elements = new HashSet<>();
   for (T element: data) {
      if (!elements.add(element)) ❸ Если add возвращает false, элемент является дубликатом
         return false;
   }
   return true;
}
```

И снова сосредоточимся на методе `add`. Он содержит тривиальную строку кода, которая не может испортить состояние объекта.

С другой стороны, защищенная версия `add` может проверять инварианты в конце каждого вызова. Как обычно, такая проверка заключается в директиву `assert`, чтобы все улучшения надежности можно было легко включать и отключать (но помните, что по умолчанию они должны быть отключены).

```
public void add(T elem) {
   ... ❶ Реализация

   assert checkInvariants() : "add broke an invariant!";
}
```

Некоторые потенциальные ошибки в `add` могут остаться незамеченными при проверке инвариантов, но будут выявлены более тщательными проверками постусловий из предыдущего раздела. Например, представьте, что `add` удаляет самый старый элемент, даже если ограниченное множество не заполнено.

Проверка инварианта не заметит проблемы, потому что дефект не переведет ограниченное множество в некорректное состояние. А точнее, состояние дел после `add` не станет некорректным *само по себе*. Оно будет некорректно относительно *истории* объекта, но инварианты не обращают внимания на историю. С другой стороны, постусловие выявит дефект, сравнив состояние ограниченного множества до и после операций `add`.

5.6. РЕАЛЬНЫЕ СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

Рефакторинг `addWater` для соблюдения принципов контрактного проектирования не был простым делом. Для проверок предусловий и постусловий пришлось написать больше кода, чем для непосредственной бизнес-логики. Ключевой вопрос: стоит ли оно того? Приведу несколько реальных примеров.

- Вы работаете в небольшой фирме, которой банк доверил разработку программного обеспечения для операций в банкоматах. Сроки поджимают, так как банк существенно вырос и ему нужно заменить старую систему проведения операций, не справляющуюся с расширением сети. Чтобы выдержать сроки, руководитель проекта в вашей команде принимает катастрофическое решение: сосредоточиться на бизнес-логике, чтобы иметь возможность быстро выдать решение. К счастью, банки не доверяют никому. У банка имеется собственная команда тестировщиков, которые тщательно проверяют все перед запуском в эксплуатацию. Оказалось, что в элегантно построенной программе, которую разработала ваша команда, есть небольшой дефект: пользователь может снять больше средств, чем имеет на счету, из-за того что разработчики пропустили проверку предусловия. Время, потраченное на обеспечение надежности при разработке, предотвратит будущие потрясения.
- Вы проводите рефакторинг библиотеки для использования новых средств языка программирования, появившихся в последней версии. Или хотите переработать существующий код для расширения его функциональности. Издержки некачественного проектирования могут быть неочевидными при первом выпуске библиотеки, но со временем проблемы накапливаются. Для совокупных издержек, вызванных некачественным проектированием, даже появился специальный термин: *технический долг*. По мере накопления технический долг может даже повлиять на будущую эволюцию библиотеки. Контрактное проектирование и сопутствующие средства программирования помогают взять под контроль технический долг за счет явного определения спецификаций и надежности.
- При создании новых продуктов разработчики часто сталкиваются с непростым выбором: какой язык программирования использовать? Ответ зависит от многих факторов, не последним из которых являются сложность системы и ее надежность. Чем сложнее архитектура системы, тем сложнее заставить систему вести себя корректно и надежно при возникновении непредвиденных ситуаций. Если надежность особенно важна, проверьте, позволяет ли ваш язык программирования выражать контракты *способом, который может проверяться во время компиляции*. Возможно, вам даже придется изменить парадигму программирования, чтобы обнаруживать больше дефектов во время выполнения. Например, известно, что функциональное программирование способствует повышению надежности, но за счет больших усилий в отдельных случаях оно может снижать производительность.

Не обманывайте себя: ошибки неизбежны. Вот почему я определяю стабильность как *способность системы корректно реагировать в ситуациях, способных привести к сбоям, а не как характеристику системы, спроектированной для предотвращения всех возможных причин сбоев*. Современные распределенные системы подвержены сбоям по своей природе и создаются с учетом принципа: частичными сбоями, непоследовательностями и переупорядочением сообщений между узлами невозможно управлять. Сбои становятся частью контракта проектирования, чтобы вы могли корректно их обрабатывать.

5.7. ПРИМЕНИМ ИЗУЧЕННОЕ

Упражнение 1

1. Запишите контракт метода `add` интерфейса `java.util.Collection`. (Да, можно заглянуть в документацию.)
2. Сделайте то же для метода `add` класса `java.util.HashSet`.
3. Сравните два контракта. Чем они отличаются?

Упражнение 2

Реализуйте статический метод `interleaveLists`, определяемый следующим контрактом:

- *Предусловие:* метод получает в аргументах два списка `List` равной длины.
- *Постусловие:* метод возвращает новый список `List`, содержащий все элементы двух списков с чередованием элементов.
- *Штраф:* если хотя бы один из списков равен `null`, метод выдает `NullPointerException`. Если списки имеют разную длину, метод выдает `IllegalArgumentException`.

Проследите, чтобы предусловие проверялось всегда, а постусловие — только при включенных директивах `assert`. Попробуйте свести к минимуму затраты при отключенных `assert`.

Упражнение 3

Объект типа `java.math.BigInteger` представляет целое число произвольного размера, закодированное в виде массива целых чисел. Проверьте исходный код в OpenJDK (http://mng.bz/Ye6j) и найдите следующие приватные методы:

```
private BigInteger(int[] val)
private int parseInt(char[] source, int start, int end)
```

1. Запишите контракт приватного конструктора. Проследите, чтобы в предусловие были включены все предположения, необходимые конструктору для нормального завершения. Выполняет ли конструктор активную проверку своего предусловия?
2. Сделайте то же самое для метода `parseInt`.

Упражнение 4

Следующий метод должен возвращать наибольший общий делитель двух заданных целых чисел (не беспокойтесь: полностью понимать его не обязательно). Измените его так, чтобы он проверял свое постусловие при включенных директивах `assert`, и протестируйте его на 1000 пар целых чисел. (Приведенный ниже код содержится в классе `eis.chapter5.exercises.Gcd` в репозитории https://bitbucket.org/mfaella/exercisesinstyle.)

Подсказка: попробуйте проверить постусловие простейшим возможным способом. Не сомневайтесь в правильности самой проверки.

```
private static int greatestCommonDivisor(int u, int v) {
   if (u == 0 || v == 0) {
      if (u == Integer.MIN_VALUE || v == Integer.MIN_VALUE) {
         throw new ArithmeticException("overflow: gcd is 2\^{}31");
      }
      return Math.abs(u) + Math.abs(v);
   }
   if (Math.abs(u) == 1 || Math.abs(v) == 1) {
      return 1;
   }
   if (u > 0) { u = -u; }
   if (v > 0) { v = -v; }
   int k = 0;
   while ((u & 1) == 0 && (v & 1) == 0 && k < 31) {
      u /= 2;
      v /= 2;
      k++;
   }
   if (k == 31) {
      throw new ArithmeticException("overflow: gcd is 2\^{}31");
   }
   int t = (u & 1) == 1 ? v : -(u / 2);
   do {
      while ((t & 1) == 0) { t /= 2; }
      if (t > 0) { u = -t; }
      else { v = t; }
      t = (v - u) / 2;
   } while (t != 0);
   return -u * (1 <{}< k);
}
```

ИТОГИ

- Надежность программы начинается с четких спецификаций.
- Стандартная форма спецификаций определяется в контексте контрактов методов и инвариантов классов.
- Предусловия открытых методов должны проверяться во всех фазах процесса разработки.
- Проверяйте другие предусловия, постусловия и инварианты только при необходимости, в ходе разработки или в программах, критичных по безопасности.
- Директивы `assert` позволяют включать или отключать некоторые проверки при любом запуске программы.

ОТВЕТЫ НА ВОПРОСЫ И УПРАЖНЕНИЯ

Неожиданный вопрос 1

Выдача проверяемого исключения в качестве штрафа заставляет вызывающую сторону обработать это исключение, либо перехватывая его, либо объявляя в секции `throws`. Это неудобно, потому что штрафа можно избежать простым соблюдением предусловий. Проверяемые исключения предназначены для аномальных ситуаций, которых невозможно избежать, потому что они находятся за пределами прямого контроля вызывающей стороны.

Неожиданный вопрос 2

Чтобы корректно отреагировать на отсутствие бумаги в принтере, следует уведомить пользователя и дать ему возможность сделать повторную попытку или отменить печать. Некорректной реакцией будет аварийное завершение программы или простое игнорирование запроса печати.

Неожиданный вопрос 3

Предлагаемое свойство сравнивает состояние объекта до и после вызова метода. Это задача для постусловия, а не для инварианта. Инварианты могут обращаться только к текущему состоянию объекта.

Неожиданный вопрос 4

Инициализируйте флаг значением `false`, а потом присвойте ему `true` в фиктивной директиве `assert`:

```
boolean areAssertionsEnabled = false;
assert (areAssertionsEnabled = true) == true;
```

Неожиданный вопрос 5

Вспомните, что «не число» (`NaN`) — одно из специальных значений чисел с плавающей точкой (как положительная и отрицательная бесконечности). Для `NaN` установлены специальные арифметические правила. В контексте вопроса для нас важны следующие:

- `NaN / n` дает `NaN`
- `NaN + n` дает `NaN`
- `NaN < n` дает `false`
- `NaN == NaN` дает `false` (это не ошибка!)

В коде `addWater` в `Contracts` (листинг 5.1) мы видим, что передача `NaN` как значения параметра `amount` проходит проверку предусловия, потому что `this.amount + amountPerContainer < 0` дает `false`. В дальнейших строках полю `amount` всех резервуаров в группе присваивается значение `NaN`. Наконец, если предположить, что директивы `assert` включены, метод проверяет свое постусловие при помощи метода `postAddWater` (листинг 5.4). `NaN` провалит оба теста `isGroupBalanced()` и `almostEqual()`, и вызов завершится с ошибкой `AssertionError`.

Если директивы `assert` отключены (как должно быть по умолчанию), вызов `getAmount` незаметно присваивает всем резервуарам группы `NaN`. Поэтому контракт `addWater` следует доработать, чтобы он поддерживал `NaN` и другие специальные значения более разумно, например объявлял их недействительными из предусловия.

Упражнение 1

1. Контракт абстрактного метода обычно сложнее контракта конкретного метода. Абстрактный метод не имеет реализации — по сути, это *чистый контракт*, который должен быть четким и подробным. Ситуация становится еще более деликатной в таких интерфейсах, как `Collection`, который, являясь корнем иерархии коллекций, должен поддерживать множество разных специализаций (а именно 34 по всем классам и интерфейсам).

   В документации Javadoc для `Collection.add` содержится много полезной информации. Начнем с квалификатора «необязательной операции». Его можно интерпретировать как определение *двух альтернативных контрактов* для этого метода. Во-первых, реализация может отказаться от поддержки вставок, как это делают неизменяемые коллекции. В этом случае должен соблюдаться следующий контракт:

   - *Предусловие:* вызовы недопустимы.
   - *Постусловие:* нет.
   - *Штраф:* `UnsupportedOperationException`.

Если класс, реализующий `Collection`, поддерживает вставку, он должен выполнять другой контракт. Такой класс может свободно выбрать предусловие `add` для ограничения допустимых вставок, но он должен выдавать штрафы при отказе во вставке, как описывает следующий контракт:

- *Предусловие* определяется реализацией.
- *Постусловие:* проверка, содержит ли коллекция заданный элемент. Метод возвращает `true`, если коллекция изменилась в результате вызова.
- *Штраф*:

  `ClassCastException` — если аргумент недействителен из-за своего типа.

  `NullPointerException` — если аргумент равен `null`, а коллекция не допускает элементы `null`.

  `IllegalArgumentException` — если аргумент недействителен по другой причине.

  `IllegalStateException` — если аргумент не может быть вставлен в данный момент.

Обратите внимание: в контракте не указано, в каких условиях вставка изменяет используемую коллекцию. Это бремя возлагается на подклассы.

2. Класс `HashSet` определяет контракт `add` следующим образом:
   - *Предусловие:* нет. (Все аргументы допустимы.)
   - *Постусловие*: заданный элемент вставляется в коллекцию, если в ней не присутствует равный ему элемент (проверяется `equals`). Метод возвращает `true`, если коллекция не содержит заданный элемент перед вызовом.
   - *Штраф:* нет.
3. Контракт `HashSet` указывает, что коллекция не содержит дубликатов. Попытка вставки дубликата ошибкой не является: она не нарушает предусловия и не выдает исключения. Коллекция просто остается без изменений.

Упражнение 2

Ниже приведен код метода `interleaveLists`. Обратите внимание на то, как обычные команды `if` проверяют предусловие, тогда как постусловие доверяется отдельному методу, вызываемому только при включенных директивах `assert`.

```
public static <T> List<T> interleaveLists(List<? extends T> a,
                                          List<? extends T> b) {
   if (a==null || b==null)
      throw new NullPointerException("Both lists must be non-null.");
   if (a.size() != b.size())
      throw new IllegalArgumentException(
```

```
                "The lists must have the same length.");

   List<T> result = new ArrayList<>();
   Iterator<? extends T> ia = a.iterator(), ib = b.iterator();
   while (ia.hasNext()) {
      result.add(ia.next());
      result.add(ib.next());
   }

   assert interleaveCheckPost(a, b, result);
   return result;
}
```

Код вспомогательного метода, отвечающего за проверку постусловия:

```
private static boolean interleaveCheckPost(List<?> a, List<?> b,
                                           List<?> result) {
   if (result.size() != a.size() + b.size())
      return false;

   Iterator<?> ia = a.iterator(), ib = b.iterator();
   boolean odd = true;
   for (Object elem: result) {
      if ( odd && elem != ia.next()) return false;
      if (!odd && elem != ib.next()) return false;
      odd = !odd;
   }
   return true;
}
```

Упражнение 3

Сначала обратим внимание на некоторые подробности документирования этих приватных методов. На официальной Javadoc-странице `BigInteger` никакие приватные методы не упоминаются. Это стандартное поведение Javadoc, которое можно изменить при помощи параметра командной строки `--show-members private`. Тем не менее в исходном коде конструктор сопровождается полным комментарием в стиле Javadoc, тогда как методу предшествует краткий комментарий в произвольном формате. Очевидно, конструктор считается достаточно важным, чтобы заслуживать более подробной документации. Из главы 7 вы узнаете больше о Javadoc и рекомендациях по документированию. А теперь выделим контракты из этих комментариев и из кода.

1. В документации Javadoc указано, что значение `val` не должно изменяться в ходе выполнения конструктора. Это свойство относится к многопоточным контекстам, где программа может выполнять другой код одновременно с этим конструктором. Как следствие, это требование не соответствует классической форме контракта, представленной в этой главе, так как она адаптирована для последовательных программ.

С другой стороны, при беглом взгляде на исходный код конструктора становится ясно, что по неявному предположению массив `val` отличен от `null` и не пуст, в результате чего мы приходим к следующему контракту:

- *Предусловие:* значение `val` отлично от `null` и не пусто.
- *Постусловие:* создается объект `BigInteger`, соответствующий целому числу, закодированному в `val` в формате с обратным порядком байтов (big-endian) в дополнительном двоичном коде.
- *Штраф*:

  `NullPointerException` — если значение `val` равно `null`.

  `NumberFormatException` — если значение `val` пустое (длина 0).

  Конструктор активно проверяет, не пуст ли массив `val`. Нет необходимости проверять `val` на `null`, потому что в этом случае автоматически выдается `NullPointerException`.[1]

2. Комментарий перед `parseInt` гласит: «Предполагается, что `start < end`». По сути, это явно выраженное предусловие. Просматривая тело метода, вы также заметите, что исходный аргумент должен быть отличен от `null`, а `start` и `end` должны быть действительными индексами в `source`. Наконец, каждый символ в заданном интервале `source` должен быть цифрой. Эти наблюдения можно выразить в форме контракта следующим образом:

   - *Предусловие:* `source` содержит отличную от `null` последовательность цифровых символов, а `start` и `end` являются действительными индексами для `source`, причем `start < end`.
   - *Постусловие:* метод возвращает целое число, представляемое цифрами между индексами `start` и `end`.
   - *Штраф*:

     `NullPointerException` — если значение `source` равно `null`.

     `NumberFormatException` — если хотя бы один символ в заданном интервале не является цифрой.

     `ArrayIndexOutOfBoundsException` — если `start` и `end` не являются допустимыми индексами в `source`.

   Метод активно проверяет, что каждый символ в интервале является цифрой. Проверка `null` опускается как избыточная. Единственное предусловие, явно выраженное в документации, не проверяется: при вызове метода со

[1] Возможно, вы предпочтете явно проверить это условие, чтобы прояснить свои намерения и снабдить исключение более конкретным сообщением об ошибке.

значениями `start ≥ end` никакие исключения не выдаются, вместо чего возвращается целое число, соответствующее символу `source[start]`.

Попутно заметим, что этот метод не использует никакие поля экземпляров, поэтому он должен быть статическим.

Упражнение 4

Код этого примера представляет собой слегка отредактированный фрагмент класса `Fraction11`[1] из проекта Apache Commons. В нем используется в высшей степени нетривиальный алгоритм Дональда Кнута. Так как метод изменяет свои аргументы, а эти аргументы нужны для проверки постусловия, исходные значения необходимо сохранить в двух дополнительных переменных. Затем в конце метода можно проверить постусловие при помощи вспомогательного метода.

```
private static int greatestCommonDivisor(int u, int v) {
   final int originalU = u, originalV = v;
      ❶ Исходная процедура (изменяет u и v)
   int gcd = -u * (1 <{}< k);
   assert isGcd(gcd, originalU, originalV) : "Wrong GCD!";
   return gcd;
}
```

Что касается вспомогательного метода `isGcd`, я уже говорил, что предпочтение следует отдавать простейшему решению. В данном случае можно просто применить определение «наибольшего общего делителя» и проверить, что:

- `gcd` действительно является общим делителем `originalU` и `originalV`;
- ни одно большее число не является общим делителем.

```
private static boolean isGcd(int gcd, int u, int v) {
   if (u \% gcd != 0 || v \% gcd != 0) ❶ Проверяет, что gcd является общим делителем
      return false;
   for (int i=gcd+1; i<=u && i<=v; i++) ❷ Проверяет все большие числа до меньшего из u и v

      if (u \% i == 0 && v \% i == 0)
         return false;
   return true;
}
```

Эта реализация `isGcd` крайне неэффективна: она линейно зависит от размера меньшего из чисел `u` и `v`. Разумнее было бы воспользоваться классическим алгоритмом Евклида (https://ru.wikipedia.org/wiki/Алгоритм_Евклида) или вызвать уже реализованную процедуру нахождения наибольшего общего делителя, например метод `BigInteger.gcd()` из JDK.

---

[1] Полное имя класса — `org.apache.commons.lang3.math.Fraction`.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

- *Мейер Б.* Объектно-ориентированное конструирование программных систем — М.: Русская редакция: Интернет-университет информационных технологий, 2005.

  В этой книге была формально представлена методология разработки на основе контрактов и язык программирования Eiffel, разработанный для ее поддержки.

- *Muller J.-M. et al.* Handbook of Floating-Point Arithmetic. Birkhäuser, 2010.

  Хотите произвести впечатление на друзей и семью своими познаниями в области вычислений с плавающей точкой? Изучите этот 500-страничный том. К книге прилагается бесплатная кандидатская степень по computer science.

- *Goldberg D.* What Every Computer Scientist Should Know About Floating-Point Arithmetic // ACM Computing Surveys. 1991. Vol. 23.

  На 44 страницах эта статья, как и обещает ее название, рассказывает все, что должен знать каждый специалист о вычислениях с плавающей точкой, но без кандидатской степени.

- *Блох Д.* Java. Эффективное программирование. — 3-е изд. — М.; СПб.: Диалектика, 2018.

  Третье издание знаменитой книги, посвященной практическим приемам программирования на Java. Книга написана одним из проектировщиков платформы Java.

6 Надежность за счет тестирования

В этой главе:

- ✓ Проектирование набора модульных тестов.
- ✓ Применение критериев покрытия ввода.
- ✓ Оценка покрытия кода.
- ✓ Оценка и улучшение тестируемости кода.

Даже разработчики, которые никогда не слышали о контрактном проектировании, знают, что такое тестирование: это последняя фаза разработки программного продукта, когда злые люди, называемые тестировщиками, находят ваши блестящие задумки, направленные на экономию времени, и называют их «ошибками». Впрочем, шутки в сторону — тесты играют все более важную роль в современных процессах разработки ПО. Одна хорошо известная методология, называемая *разработкой через тестирование* (TDD, test-driven development), рекомендует писать тесты до, а не после рабочего кода. В этом случае тесты используются как спецификации, а остальная часть системы пишется для их прохождения[1].

Материал этой главы не зависит от вашего личного отношения к тестам. Мы дополним класс `Reference` (и любую реализацию, соответствующую API)

[1] О TDD можно узнать из книги *Growing Object-Oriented Software, Guided by Tests*, упоминаемой в разделе «Дополнительная литература» в конце главы.

набором тестов, которые выглядят разумно и пытаются покрыть всю возможную функциональность. Следуя теме книги, я сосредоточусь на *модульном тестировании* — то есть тестировании отдельного класса. Позднее мы критически проанализируем API резервуара с водой в свете тестируемости, а также предложим ряд улучшений в духе передовой практики.

6.1. ОСНОВНЫЕ ПОНЯТИЯ ТЕСТИРОВАНИЯ

Тестирование — основная проверочная деятельность в области разработки ПО. Как следствие, в литературе можно найти множество разнообразных теорий и приемов, связанных с этой областью. Как и другие темы, в книге я представлю основы тестирования поверхностно. Для углубленного изучения этой темы существует много специализированных ресурсов, часть из которых представлена в разделе «Дополнительная литература» в конце этой главы.

Целью тестирования является обнаружение и удаление как можно большего количества ошибок для повышения вашей уверенности в правильности и стабильности программы. Точнее, нельзя ожидать, что сложная программа когда-либо будет *полностью* свободна от ошибок, поэтому тестирование должно быть направлено на выявление всех серьезных дефектов — тех, которые с большой вероятностью часто и быстро возникают при повседневном использовании программы. Тестирование редко выявляет более коварные и хитроумные ошибки. Условия для появления таких ошибок возникают только при длительном и интенсивном использовании.

Умение проектировать хорошие тесты развивается на основе практики в неменьшей степени, чем на теоретической базе. Так как я не смогу немедленно сформировать у вас практическую основу, я сделаю то, что в моих силах: изложу применение теоретических принципов к конкретному примеру.

6.1.1. Покрытие в тестировании

Чтобы повысить вероятность выявления всех серьезных дефектов, следует применить систематический подход к тестированию, в котором центральное место занимает концепция покрытия. В самом деле, покрытие является одной из важнейших тем в проектировании тестов, причем с этим термином связывается целый ряд смыслов. В общем случае *покрытием* называется степень, в которой тесты пытаются моделировать разные части системы. Были разработаны два общих способа измерения оценки покрытия: покрытие кода и покрытие входа. Под *покрытием кода* понимается процент исходного кода, который по крайней мере однократно выполняется заданным набором тестов. Как будет показано в этой главе, этот процент может измеряться разными способами. Покрытие кода традиционно ассоциируется с тестированием *по принципу белого ящика*, которое

получило свое название из-за того, что оно предполагает наличие информации о тестируемом продукте (SUT, software under testing) и его исходном коде.

С другой стороны, *покрытие входа* игнорирует внутреннее строение тестируемой программы и концентрируется только на ее API. В общих чертах оно предлагает проанализировать набор возможных входных значений для выявления меньшего набора репрезентативных входных значений. Покрытие входа ассоциируется с тестированием *по принципу черного ящика* в том смысле, что оно не зависит от исходного кода SUT.

Два типа покрытия дополняют друг друга, и в этой главе будут рассмотрены оба варианта. Начнем с покрытия входа и спроектируем наборы тестов, которые стремятся предоставить разностороннюю подборку входных значений. Затем специальная программа будет использована для оценки покрытия кода, обеспечиваемого этими тестами. Иначе говоря, покрытие входа будет использоваться как цель проектирования, а покрытие кода — как форма проверки тестового плана.

6.1.2. Тестирование и контрактное проектирование

Прежде чем углубляться в подробности, сравним цель тестирования с целью методов, представленных в предыдущей главе, в которых центральное место занимала проверка контракта.

Проверка предусловий открытых методов и реакция в виде соответствующего штрафа являются простейшей формой защитного программирования и обычно относятся к числу оптимальных практик. Тестирование никоим образом не заменяет ее, а скорее подкрепляет. В самом деле, позднее в этой главе мы разработаем некоторые тесты, которые направлены на проверку того, что эти защитные меры находятся на своем месте.

Проверка постусловий и инвариантов внутри метода — совсем другое дело: их целью является выявление проблем *внутри* класса, то есть они решают те же задачи, что и модульное тестирование. Следовательно, эти приемы могут рассматриваться как альтернатива тестированию.

По сравнению с этими методами тестирование обладает двумя преимуществами, из-за которых оно намного чаще применяется на практике:

- Тестирование выводит инварианты и проверки постусловий за пределы самого класса. Это очень удобное решение, с которым классы остаются компактными и простыми и которое четко разделяет ответственность между классами и между разработчиками. У компании появляется возможность распределения разработки и тестирования по разным командам.
- Тестирование заставляет тщательно выбирать набор входных значений, которые будут передаваться тестируемому продукту. В других методологиях

этот аспект отсутствует. Другими словами, реализация постусловия или проверок инвариантов внутри методов — всего лишь половина дела. Без систематической стратегии вызова конкретных методов с конкретными входными значениями (что равносильно плану тестирования) такие проверки могут не выявить ошибок ни на одной стадии разработки или производства программы. Тестирование предоставляет вам возможность руководить процессом, при этом метрики покрытия поддерживают вашу уверенность в правильности и стабильности тестируемого продукта.

Рисунок 6.1, приводившийся в предыдущей главе, связывает тесты со свойствами кода и контрактным проектированием. Тесты проверяют, что методы соблюдают их постусловия и что они реагируют на недопустимый вход (определяемый по предусловиям) соответствующими штрафами. Тем самым тесты повышают надежность продукта, выявляя дефекты и упрощая их устранение.

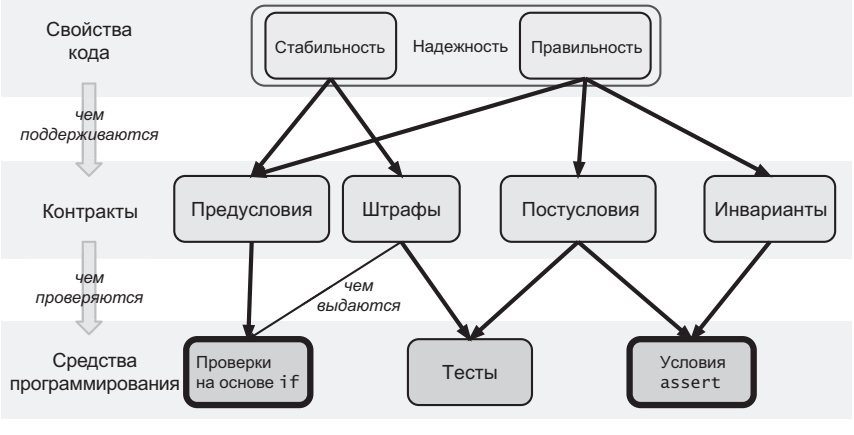


Рис. 6.1. Отношения между атрибутами надежности, спецификациями на базе контрактов и средствами программирования

В особенно критичном коде может быть полезно дополнить тестирование некоторыми приемами из предыдущей главы. Например, разместить проверки инвариантов, чтобы систему можно было в любой момент запустить в режиме контроля стабильности. Если аномальное поведение пройдет тестирование и будет обнаружено в ходе эксплуатации, вам будет проще диагностировать и устранить его.

НЕОЖИДАННЫЙ ВОПРОС 1

Какие части контракта метода важны для тестирования этого метода?

Современное тестирование основано на способности быстро и многократно выполнять изменяющуюся подборку тестов. Процесс автоматически поддерживается библиотеками и фреймворками, самым популярным из которых является семейство xUnit, включающее JUnit для Java и NUnit для языков .NET. Если вы еще не знакомы с JUnit, в следующем разделе приводится краткое введение.

6.1.3. JUnit

JUnit — стандартный фреймворк модульного тестирования для Java. Он предоставляет открытый код для набора тестов. Следующие тесты базируются на JUnit 4.0, поэтому я начну с краткого обзора этого фреймворка.

В JUnit широко применяются *аннотации* Java.

АННОТАЦИИ JAVA

Аннотация представляет собой метку, начинающуюся с символа `@`, которая присоединяется к методу непосредственно перед его сигнатурой. Большинству программистов знакома аннотация `@Override`:

```
public class Employee {
   private String name, salary;
   ...
   @Override
   public String toString() {
      return name + ", monthly salary " + salary;
   }
}
```

`@Override` в этом фрагменте сообщает компилятору, что метод `toString` предназначен для переопределения. Другими словами, программист приказывает компилятору выполнить дополнительную проверку: если целевой метод не переопределит метод суперкласса, то компиляция завершится неудачей. Аннотация `@Override` не имеет аргументов, но у других аннотаций они могут присутствовать (скоро я приведу пример).

На практике аннотации представляют собой обобщенный механизм присоединения метаданных к элементам программ. Помимо методов, их также можно применять к классам, полям, локальным переменным, параметрам методов и т. д. Аннотации — пассивные элементы, содержащие дополнительную информацию об элементе программы. Их можно передавать байт-коду и читать во время выполнения с использованием механизма отражения. Программисты могут легко определять собственные аннотации и писать программы, которые интерпретируют эти аннотации для изменения или расширения поведения программ.

В JUnit каждый тест оформляется в виде метода, а набор взаимосвязанных тестов образует класс. Не все методы в классе должны представлять тест. Чтобы указать, что метод представляет тест, пометьте его аннотацией `@Test`:

```
@Test
public void testSomething() { ... }
```

Если ожидается, что тест выдаст исключение (что часто случается при тестировании на стабильность), сообщите JUnit класс ожидаемого исключения в значении атрибута `expected` аннотации `@Test`:

```
@Test(expected = IllegalArgumentException.class)
public void testWrongInput() { ... }
```

Как показывают эти примеры, методы тестов не возвращают значения. Успех или неудача теста определяются соответствующим *тестовым условием JUnit* (не путайте с директивой `assert` в Java!). Тестовое условие в JUnit представляет собой один из многочисленных статических методов фреймворка для сравнения фактического результата операции с ожидаемым. Каждый раз, когда проверка тестового условия завершается неудачей, выдается исключение `AssertionError`, как и директива `assert` в Java. JUnit перехватывает эти исключения, продолжает выполнять все остальные тесты в наборе и выдает итоговый отчет со сводкой результатов всех тестов.

АТРИБУТЫ C#

В языке C# для присоединения метаданных к элементам программ используются атрибуты. Они работают аналогично аннотациям Java, и их можно узнать по квадратным скобкам. Например, аналогом аннотации Java `@Deprecated` является атрибут C# `[Obsolete]`.

Самыми распространенными тестовыми условиями являются следующие методы `public static void` из класса `org.junit.Assert`:

- `assertTrue(String message, boolean condition)` — тест проходит, если значение `condition` истинно. Это самое общее тестовое условие JUnit, которое позволяет использовать любую нестандартную проверку, возвращающую логическое значение. Строка `message` в этом и последующих методах присоединяется к исключению, выданному при нарушении условия, и позднее будет включена в итоговый отчет;
- `assertFalse(String message, boolean condition)` — проверка, обратная предыдущей: тест проходит, если значение `condition` ложно;
- `assertEquals(String message, Object expected, Object actual)` — тест проходит, если значения `expected` и `actual` одновременно равны `null` или

друг другу (при сравнении методом `equals`). Аналогичные условия могут получать примитивные типы `long`, `float` и `double` вместо `Object`. Тем не менее версии с плавающей точкой считаются устаревшими[1];

- `assertEquals(String message, double expected, double actual, double delta)` — тест проходит, если значения `expected` и `actual` лежат в диапазоне `delta` друг от друга. Здесь `delta` определяет погрешность сравнения. Как упоминалось в разделе 5.3, числа с плавающей точкой должны сравниваться с некоторой погрешностью для округления. Вскоре я вернусь к этому вопросу.

JUnit может запускаться из командной строки, но намного чаще запускается как часть IDE, чтобы вы могли легко выполнить тесты и наглядно проанализировать их результаты.

6.2. ТЕСТИРОВАНИЕ РЕЗЕРВУАРОВ [UNITTESTS]

Пора вернуться к нашим резервуарам. Этот раздел немного отличается от других, потому что в нем мы будем разрабатывать не очередную версию класса `Container`, а набор тестов для проверки его функциональности. Какая версия `Container` будет тестироваться? Так как мы будем действовать по схеме черного ящика, мы не станем привязываться к какой-то конкретной реализации `Container`. Вместо этого тесты будут ориентироваться на API (глава 1). Это позволит выполнить тесты с любыми реализациями, соответствующими данному API (раздел 6.2.4). Если же вам хочется ориентироваться на конкретную реализацию — возьмите `Reference`.

Код приведенных ниже тестов содержится в классе `eis.chapter6.UnitTests`, который можно найти в репозитории https://bitbucket.org/mfaella/exercisesinstyle.

6.2.1. Инициализация тестов

Следующие тесты ориентированы на API, который используется нормальными клиентами. Как следствие, внутреннее состояние объектов невозможно проверить напрямую. Метод `getAmount`, по сути, является единственной доступной нам обратной связью (и единственным методом, возвращающим значение). Мы еще вернемся к этому ограничению в этой главе.

Все наши тесты должны работать с одним или несколькими объектами `Container`. Вместо того чтобы создавать резервуары в начале каждого теста, можно избежать повторения кода за счет добавления в класс полей `Container` и инициализации их в методе, помеченном аннотацией JUnit `@Before`. Метод с аннотацией `@Before` будет выполняться перед каждым тестом. Такие объекты, совместно

[1] Еще они *всегда дают отрицательный результат*.

используемые несколькими тестами, называются *тестовой оснасткой* (text fixture). Соответственно, наш тестовый класс в исходном состоянии выглядит так:

```
public class UnitTests {
   private Container a, b; ❶ Тестовые оснастки

   @Before                 ❷ Приказывает JUnit выполнять этот метод перед каждым тестом
   public void setUp() {
      a = new Container();
      b = new Container();
}
```

УДОБОЧИТАЕМЫЕ ТЕСТОВЫЕ УСЛОВИЯ С СОПОСТАВИТЕЛЯМИ HAMCREST

В примерах, приводившихся до настоящего момента, я использовал базовый вариант записи тестовых условий JUnit. Существует другое, лучшее решение — использование библиотеки Hamcrest, входящей в JUnit. Библиотека позволяет выражать проверяемое условие в более удобочитаемом виде, то есть строить *объект-сопоставитель* (matcher) и передавать его тестовому условию `assertThat`.

Например, базовое тестовое условие

```
assertEquals("new container is not empty", 0, a.getAmount(), 0);
```

может быть переписано в виде следующего тестового условия Hamcrest:

```
assertThat("new container is not empty", a.getAmount(), closer(0, 0));
```

Также условия Hamcrest позволяют получать более четкие диагностические сообщения при неудачной проверке. Чтобы оценить различия, представьте, что пустой резервуар в исходном состоянии ошибочно содержит 0,1 единицы воды. Первая проверка на базе `assertEquals` выдаст при неудаче следующее сообщение:

```
new container is not empty
expected:<0.0> but was:<0.1>
```

тогда как проверка средствами Hamcrest даст более подробную информацию:

```
new container is not empty
Expected: a numeric value within <0.0> of <0.0>
     but: <0.1> differed by <0.0> more than delta <0.0>
```

Я буду использовать сопоставители Hamcrest во втором примере этой главы (раздел 6.4).

Для полноты можно пометить метод, который должен выполняться после каждого теста, парной аннотацией `@After`. Она полезна, если тестовые оснастки должны освобождать ресурсы при закрытии. Более того, можно присоединять аннотации `@BeforeClass` и `@AfterClass` к статическим методам, которые должны выполняться однократно до или после всей последовательности тестов в текущем классе. Например, эти аннотации можно использовать для создания и уничтожения вычислительно-затратных оснасток, общих для нескольких тестов (например, подключений к базам данных и каналов связи).

Начнем с проектирования нашего первого теста `Container`, который будет проверять, что конструктор работает так, как положено. Поскольку у конструктора нет входных данных, мы вызовем его однократно и проверим единственное свойство, которое API позволяет проверить, пуст ли только что созданный резервуар.

```
@Test
public void testNewContainerIsEmpty() {
   assertTrue("new container is not empty", a.getAmount() == 0);
}
```

В данном случае два числа с плавающей точкой можно сравнивать точно, потому что нет причин для приближенного представления значения классом. В предыдущем фрагменте я использовал метод `assertTrue`, потому что я считаю, что он читается лучше эквивалентного вызова `assertEquals`, который выглядит так:

```
assertEquals("new container is not empty", 0, a.getAmount(), 0);
```

6.2.2. Тестирование addWater

Перейдем к тестированию поведения метода `addWater`. Его входные данные включают параметр и текущее состояние резервуара. Так как эти входные данные могут принимать огромное количество значений, необходимо найти систематический способ выбора входных значений, которые будут передаваться тестируемому методу. Стандартный метод по принципу черного ящика называется *моделированием области входных значений*.

Моделирование области входных значений

Метод моделирования области входных значений помогает выбрать ограниченный набор интересных значений для передачи методу. Он состоит из трех шагов:

1. Определить небольшое количество важных *характеристик* входных значений. Характеристика представляет собой признак, по которому множество входных значений делится на конечное (желательно небольшое) число категорий, называемых *блоками*. Актуальные характеристики могут выбираться по типу входного значения или по контракту метода. Например, одна из стан-

дартных характеристик для целочисленного ввода подразумевает деление значений на три блока: отрицательные числа, нуль и положительные числа.

2. Объединить характеристики в конечное множество комбинаций. Например, на рис. 6.2 представлены две характеристики для входных данных типа `int`. В совокупности они определяют шесть возможных комбинаций, хотя одна из них пуста, потому что нуль традиционно считается четным числом.
3. Выбрать входное значение из каждой комбинации с помощью теста. Затем тест вызывает метод с выбранным входным значением и сравнивает результат с ожидаемым в соответствии с контрактом. (Правильным результатом может быть исключение).

Применим этот метод к `addWater`, а затем к `connectTo`.

Рис. 6.2. Две характеристики входных данных типа int: знак и четность. Совместно они разбивают множество целых чисел на пять подмножеств (комбинация «нуль, нечетное» противоречива)

Выбор характеристик

Первый способ выявления актуальных характеристик входных данных основан на простом анализе их типов. Примитивные типы данных обладают следующими характеристиками:

- Для числового типа естественно отличать нуль от других значений, потому что он проявляет особые арифметические свойства.
- Аналогичным образом API часто по-разному работает с положительными и отрицательными числами, причем отрицательные числа часто бывают нежелательными.
- Для всех ссылочных типов следует выделить значение `null`, требующее особой обработки.

- Наконец, для строк, массивов и коллекций особым случаем является пустой объект.

Наблюдения, относящиеся к характеристикам на основании типа, обобщены в табл. 6.1, и это лишь малая часть картины. Опытные тестировщики обычно используют много интересных стандартных характеристик. Например, строки могут охватывать все пространство символов (формально — *кодовых точек*) Юникода, а менее известные символы и алфавиты часто становятся источниками ошибок.

Таблица 6.1. Стандартные характеристики распространенных типов входных данных

Тип	Характеристика	Блоки
`int` и `long`	Знак	{отрицательное, нуль, положительное}
`float` и `double`	Знак и специальные значения	{отрицательное, нуль, положительное, бесконечность, NaN}
`String`	Длина	{`null`, пустая строка, непустая строка}
Массив или коллекция	Размер	{`null`, пустой массив / коллекция, непустой массив / коллекция}

НЕОЖИДАННЫЙ ВОПРОС 2

Какую характеристику вы бы выбрали для типа данных, представляющего дату?

Более интересным и результативным источником характеристик становится контракт тестируемого метода. Вы можете проанализировать как пред-, так и постусловие на актуальные свойства ввода. Для примера возьмем контракт `addWater`. Постусловие сообщает, что `addWater` распределяет добавленную воду по всем резервуарам, соединенным с текущим. Это условие применимо только к резервуарам, соединенным с текущим, поэтому первая характеристика будет отличать изолированные резервуары от соединенных. Данная характеристика — назовем ее C1 — является бинарной, то есть она делит входные значения на два блока: значения, с которыми текущим состоянием резервуара является изолированное состояние, и значения, с к которыми текущим состоянием резервуара является соединенное состояние.

Кроме того, предусловие предписывает, что при отрицательном аргументе метода в группе должно быть достаточно жидкости для удовлетворения запроса. Отсюда следует вторая характеристика — обозначим ее C2, — которая различает четыре случая (а следовательно, четыре блока):

- Аргумент положителен.
- Аргумент равен нулю.
- Аргумент отрицателен, и в группе достаточно воды (допустимое отрицательное значение).
- Аргумент отрицателен, и в группе недостаточно воды (недопустимое отрицательное значение).

В табл. 6.2 приведена сводка этих двух характеристик.

Таблица 6.2. Две характеристики, выбранные для тестирования addWater

Название	Характеристика	Блоки
C1	Резервуар соединен по крайней мере еще с одним	{`true`, `false`}
C2	Отношение между аргументом и объемом воды в текущей группе	{положительное, нуль, допустимое отрицательное, недопустимое отрицательное}

Выбор комбинаций блоков

Каждая характеристика разбивает входные значения на набор меньших блоков. Чтобы найти как можно больше дефектов, нужно протестировать некоторые (или все) *комбинации* блоков из разных характеристик. В нашем случае количество характеристик и количество блоков настолько мало, что мы можем полностью протестировать все восемь комбинаций блоков:

1. (C1 = `false`, C2 = положительное);
2. (`false`, нуль);
3. (`false`, допустимое отрицательное);
4. (`false`, недопустимое отрицательное);
5. (`true`, положительное);
6. (`true`, нуль);
7. (`true`, допустимое отрицательное);
8. (`true`, недопустимое отрицательное).

Назовем эту стратегию *полнокомбинаторным покрытием*. Обратите внимание: характеристики C1 и C2 независимы, так что все восемь комбинацией имеют смысл. В других случаях эта стратегия порождает слишком много комбинаций. В следующей врезке представлены некоторые альтернативные стратегии, дающие меньшее количество комбинаций.

КРИТЕРИИ ПОКРЫТИЯ ВХОДА

Исследователи и практики разработали много других критериев покрытия входных данных, которые тем или иным образом ограничивают общее количество выполняемых тестов. Часто они использовали два критерия:

- *Полновариантное покрытие* — этот критерий предполагает, что каждый блок из каждой характеристики включается по крайней мере в один тест.

  В случае `addWater` этому критерию удовлетворяет следующая подборка:

  1. (`true`, нуль);
  2. (`true`, положительное);
  3. (`true`, допустимое отрицательное);
  4. (`false`, недопустимое отрицательное).

  Также возможны альтернативные решения, каждое из которых содержит не менее четырех тестов, потому что вторая характеристика C2 состоит из четырех блоков.

- *Покрытие базового варианта* — в соответствии с этим критерием вы выбираете базовую комбинацию блоков, а затем изменяете характеристики по одной с перебором всех возможных значений этой характеристики. В нашем примере в качестве базового варианта можно выбрать следующий, в каком-то смысле наиболее типичный:

  1. (`true`, положительное).

  Изменение значения первой характеристики приводит к комбинации.

  2. (`false`, положительное).

  Изменение второй характеристики базовой комбинации генерирует следующие три комбинации, которые завершают подборку:

  3. (`true`, нуль).
  4. (`true`, допустимое отрицательное).
  5. (`true`, недопустимое отрицательное).

НЕОЖИДАННЫЙ ВОПРОС 3

Вы определили три независимые характеристики с n_1, n_2 и n_3 блоками. Сколько тестов понадобится для достижения полнокомбинаторного покрытия? А как насчет полновариантного покрытия и покрытия базового варианта?

Выбор значений

На последнем шаге моделирования области входных значений выберем один набор конкретных значений для каждой комбинации характеристик. Продолжая процесс для `addWater`, рассмотрим комбинацию 7 из следующего списка:

1. (C1 = `false`, C2 = положительное);
2. (`false`, нуль);
3. (`false`, допустимое отрицательное);
4. (`false`, недопустимое отрицательное);
5. (`true`, положительное);
6. (`true`, нуль);
7. (**`true`, допустимое отрицательное**);
8. (`true`, недопустимое отрицательное).

Наша цель на последнем шаге — спроектировать резервуар `c` и значение `amount` типа `double`, для которых следующий вызов:

```
c.addWater(amount);
```

попадает в 7-ю комбинацию блоков. Проще говоря, резервуар `c` должен быть соединен как минимум еще с одним резервуаром, значение `amount` должно быть отрицательным и в группе `c` должно быть достаточно воды для выполнения запроса. Этот сценарий достаточно легко подготовить с использованием API для резервуаров. В итоге будет получен тестовый метод следующего вида:

```
@Test
public void testAddValidNegativeToConnected() {
   a.connectTo(b); ❶ Подготовка нужного сценария
   a.addWater(10);
   a.addWater(-4); ❷ Тестируемая строка
   assertTrue("should be 3", a.getAmount() == 3);
}
```

Что же реально тестируется?

Метод `addWater` не возвращает значения. Как же узнать, сработал метод или нет?

Легко: нужно вызвать `getAmount` и сравнить ожидаемое значение с фактическим. Но как можно быть уверенным, что `getAmount` правильно возвращает текущее значение `amount`? Никак. И еще хуже, откуда мы знаем, что следующие две строки правильно настраивают тестовый сценарий?

```
a.connectTo(b);
a.addWater(10);
```

Ниоткуда.

Хотя предыдущий тест предназначался для `addWater`, в итоге он тестирует `connectTo`, `getAmount` и `addWater`! Если что-то пойдет не так, вы не узнаете, какой из трех методов в этом виноват. В большинстве случаев можно ожидать, что `getAmount` будет простым GET-методом, и с гораздо большей вероятностью можно подозревать ошибку в `addWater` или `connectTo`. Тем не менее это не всегда так: в версии `Speed3` из главы 3 реализация `getAmount` почти не уступает по сложности `addWater`. Обе реализации должны обойти дерево указателей на родителей до его корня:

Листинг 6.1. Speed3: методы getAmount и addWater

```
public double getAmount() {
   Container root = findRootAndCompress(); ❶ Получение корня и сжатие пути

   return root.amount; ❷ amount читается из root
}

public void addWater(double amount) {
   Container root = findRootAndCompress(); ❸ Получение корня и сжатие пути
   root.amount += amount / root.size;      ❹ Добавление воды в корневой резервуар
}
```

Ситуация становится безвыходной, если только не принять совершенно иной подход и позволить тестам обращаться к состоянию резервуаров напрямую (сделав все поля открытыми или разместив тесты в классе `Container`). Это станет шагом к тестированию по принципу белого ящика: тесты привяжутся к конкретной реализации и станут менее полезными (раздел 6.3).

В листинге 6.2 приведен код четырех тестов `addWater` для изолированного резервуара (C1 = `false`). Обратите внимание: для простоты я использую точное сравнение `double` (без погрешности округления), потому что добавляемая вода остается в этом резервуаре и у `addWater` нет причин что-либо округлять. Последний из этих тестов должен выдавать исключение, потому что мы намеренно нарушаем предусловие. Чтобы сообщить JUnit, исключение какого типа следует ожидать, используйте параметр `expected` аннотации `@Test`.

Листинг 6.2. UnitTests: четыре теста для addWater с изолированным резервуаром

```
@Test
public void testAddPositiveToIsolated() { ❶ C1 = false, C2 = положительное
   a.addWater(1);
   assertTrue("should be 1.0", a.getAmount() == 1);
}
@Test
public void testAddZeroToIsolated() { ❷ C1 = false, C2 = нуль
   a.addWater(0);
   assertTrue("should be 0", a.getAmount() == 0);
}
```

```
@Test
public void testAddValidNegativeToIsolated() { ❸ C1 = false, C2 = допустимое отрицательное
   a.addWater(10.5);
   a.addWater(-2.5);
   assertTrue("should be 8", a.getAmount() == 8);
}
@Test(expected = IllegalArgumentException.class)
public void testAddInvalidNegativeToIsolated() { ❹ C1 = false, C2=недопустимое
   a.addWater(-1);                                  отрицательное
}
```

Две характеристики из табл. 6.2, а именно изолированность резервуара и отношение между объемом воды в текущей группе и объемом, переданным в аргументе, станут хорошей отправной точкой для набора тестов. Тем не менее если вы решите, что тестирование должно быть более полным, вы наверняка добавите другие тесты. Например, если передается входное значение с плавающей точкой, вы должны учесть специальные значения, поддерживаемые такими типами: положительную / отрицательную бесконечность и «не число» (NaN). Сначала следует расширить контракт `addWater` и определить в нем реакцию на эти специальные значения (например, выдачу исключения). Затем можно перейти к добавлению характеристики, которая учитывает эти значения, что приведет к появлению новых комбинацией блоков и других тестов.

6.2.3. Тестирование connectTo

Перейдем к тестированию метода `connectTo`. Входные данные этого метода — его параметр и текущее состояние двух соединяемых резервуаров. Единственное предусловие — аргумент отличен от `null` (его нужно включить в анализ в характеристике C3). Можно перегрузить C3 так, чтобы характеристика также учитывала другое специальное значение параметра: `this`. Как выясняется, этот случай не был учтен при проектировании контракта `connectTo`. А теперь мы доработаем контракт и укажем, что попытка соединения резервуара с самим собой должна приводить к NOP[1].

Метод `connectTo` обеспечивает слияние двух групп резервуаров, поэтому нужно различать разные сценарии в зависимости от исходных размеров групп. Группы не могут быть пустыми: изолированный резервуар по определению образует отдельную группу, поэтому мы будем отличать группы с размером `1` от групп большего размера. Обозначим `2+` группу с размером, превышающим `1`. Эти размеры отражаются в характеристиках C4 и C5. Обратите внимание на то, что C5 («размер другой группы») включает дополнительное значение «none»,

[1] NOP означает отсутствие операции (no operation). Изначально это была мнемоника машинной команды, которая не делала ничего. Позднее сокращение приобрело более общий смысл и стало обозначать пустую (фиктивную) операцию.

которое применяется, когда аргумент метода равен `null`, то есть другая группа не существует.

Наконец, другая характеристика (C6) проверяет, совпадают ли эти две группы (соединены ли резервуары). Сводка выбранных характеристик приведена в табл. 6.3.

Таблица 6.3. Характеристики, выбранные для тестирования connectTo

Название	Характеристика	Блоки
C3	Значение аргумента	{`null`, `this`, `other`}
C4	Размер текущей группы	{`1`, `2+`}
C5	Размер другой группы	{`none`, `1`, `2+`}
C6	Две группы совпадают	{`true`, `false`}

На этот раз характеристики не полностью независимы друг от друга, то есть не все комбинации допустимы. Действуют следующие ограничения:

- Если аргумент `connectTo` равен `null` (C3 = `null`), то другой группы нет, так что C5 = `none`, а C6 = `false`.
- Если вы пытаетесь соединить резервуар сам с собой (C3 = `this`), другие характеристики будут равны либо (`1`, `1`, `true`), либо (`2+`, `2+`, `true`).
- Если соединяются два разных резервуара (C3 = `other`) и они соединены (C6 = `true`), размер их общей группы не может быть равен `1`.

К счастью, эти ограничения сокращают количество допустимых комбинаций с 36 до следующих 9:

1. (`other`, `1`, `1`, `false`);
2. (`other`, `2+`, `1`, `false`);
3. (`other`, `1`, `2+`, `false`);
4. (`other`, `2+`, `2+`, `false`);
5. (`other`, `2+`, `2+`, `true`);
6. (`this`, `1`, `1`, `true`);
7. (`this`, `2+`, `2+`, `true`);
8. (`null`, `1`, `none`, `false`);
9. (`null`, `2+`, `none`, `false`).

Мы будем выполнять по одному тесту для каждой из этих комбинаций, кроме последней, так как фактически нет смысла различать комбинации 9 и 8. В обоих случаях ожидаемое поведение сводится к простой выдаче `NullPointerException`.

Это наблюдение можно обобщить: если значение характеристики нарушает предусловие (и ведет к исключению), вы можете проверить его один раз, не обращаясь ко всем возможным комбинациям.

Неудивительно, что мы сталкиваемся с той же проблемой наблюдаемости, которая упоминалась ранее. Главным результатом применения `connectTo` является слияние двух групп, но API не предоставляет никаких средств для прямого анализа данных групп. Не существует открытого метода, который бы мог проверить, соединены ли два резервуара. Собственно, есть только один метод, который возвращает *какую-либо* информацию о состоянии резервуаров: `getAmount`. Вы можете проверить, что информация, возвращаемая `getAmount`, соответствует *объединению* двух групп, но тесты никак не смогут подтвердить, что группы были действительно объединены.

В листинге 6.3 содержится код тестов, соответствующих комбинациям `connectTo` с 1 по 3. Напомню, что все тесты могут использовать тестовые оснастки, определенные выше: два пустых изолированных резервуара `a` и `b`.

Листинг 6.3. UnitTests: три теста для connectTo

```
@Test
public void testConnectOtherOneOne() { ❶ C1 = other, C2 = 1, C3 = 1, C4 = false
   a.connectTo(b);      ❷ Тестируемая строка
   a.addWater(2);
   assertTrue("should be 1.0", a.getAmount() == 1);
}
@Test
public void testConnectOtherTwoOne() { ❸ C1 = other, C2 = 2+, C3 = 1, C4 = false
   Container c = new Container();
   a.connectTo(b);
   a.connectTo(c);      ❹ Тестируемая строка
   a.addWater(3);
   assertTrue("should be 1.0", a.getAmount() == 1);
}
@Test
public void testConnectOtherOneTwo() { ❺ C1 = other, C2 = 1, C3 = 2+, C4 = false
   Container c = new Container();
   b.connectTo(c);
   a.connectTo(b);      ❻ Тестируемая строка
   a.addWater(3);
   assertTrue("should be 1.0", a.getAmount() == 1);
}
```

6.2.4. Выполнение тестов

В табл. 6.4 приведена сводка 17 тестов, разработанных нами, которые будут выполнены для четырех разных реализаций: `Reference` из главы 2, «быстрой» реализации `Speed3` из главы 3 и двух «стабильных» реализаций `Contracts` и `Invariants` из главы 5.

Таблица 6.4. Количество прошедших тестов для разных реализаций. Результаты не зависят от включения или отключения директив assert

	Reference	Speed3	Contracts	Invariants
Конструктор	1/1	1/1	1/1	1/1
`addWater`	**6**/8	**6**/8	8/8	8/8
`connectTo`	8/8	8/8	8/8	8/8
Не прошедшие тесты	C2 = недопустимое отрицательное	C2 = недопустимое отрицательное	–	–

Первые две реализации не проходят два теста `addWater`, где мы пытаемся удалить больше воды, чем было доступно (C2 = недопустимое отрицательное). В самом деле, эти реализации не проверяют условие и спокойно поддерживают отрицательные объемы воды в резервуаре.

Мы намеренно построили две другие реализации, честно соблюдающие контракт, чтобы достойно пройти все тесты. Стоит заметить, что прохождение тестов не зависит от включения директив `assert`, потому что стандартные команды `if` всегда зависят от предусловий.

6.2.5. Оценка покрытия кода

Для покрытия кода, обеспечиваемого этими тестами, можно воспользоваться JaCoCo — фреймворком покрытия Java-программ с открытым кодом. Он собирает информацию на стадии выполнения при помощи *агента Java* — кода, выполняемого в фоновом режиме на JVM для анализа или изменения выполнения программы. После сбора информации JaCoCo выдает отчеты в разных форматах, включая страницы HTML со средствами навигации и расширенным контентом. Как и JUnit, JaCoCo хорошо интегрируется со многими популярными IDE, но также может запускаться из командной строки.

JaCoCo вычисляет различные критерии покрытия кода:

- *Покрытие команд* — процент выполняемых команд байт-кода.
- *Покрытие строк* — процент выполняемых строк исходного кода Java. Компилятор может преобразовать одну строку в несколько команд байт-кода. Строка кода считается выполненной, если выполнена хотя бы одна из этих команд. Таким образом, покрытие строк всегда выше покрытия команд.
- *Покрытие ветвей* — процент выполняемых условных ветвей в коде (то есть команд `if` и `switch`).

Инструкции по запуску JaCoCo из командной строки содержатся в файле UnitTests.java[1]; в нем же содержатся тесты, разработанные в этой главе. После выполнения тестов из JaCoCo вы получите отчет о покрытии, краткая сводка которого приведена в табл. 6.5. Из него следует, что нам удалось выполнить все команды байт-кода из реализаций `Reference` и `Speed3` — неплохой результат!

Таблица 6.5. Метрики покрытия, вычисленные для разных реализаций. Пометки об assert относятся к команде Java assert, а не к тестовым условиям JUnit

Версия	Команды	Строки	Ветви
`Reference`	100 %	100 %	100 %
`Speed3`	100 %	100 %	100 %
`Contracts` (`assert` выкл)	38 %	50 %	25 %
`Contracts` (`assert` вкл)	92 %	100 %	63 %
`Invariants` (`assert` выкл)	51 %	56 %	29 %
`Invariants` (`assert` вкл)	92 %	100 %	68 %

Кстати, если вы выполнили все команды байт-кода и не обнаружили ошибок, это еще не значит, что ошибок нет. Возможно, входные данные не выявляют ошибку. Например, программист-злоумышленник написал метод `addWater()` так, чтобы в нем происходил сбой при добавлении ϖ (`Math.PI`) литров воды. Крайне маловероятно, что эта ошибка будет выявлена при любом объеме тестирования по принципу черного ящика. Но при подробном анализе покрытия кода этот случай будет помечен как неисследованный и, возможно, ловушка будет найдена.

Для `Contracts` и `Invariants` покрытие зависит от того, включили ли вы директивы `assert` языка Java (при помощи параметра командной строки `-ea`). Если нет, тесты проанализируют лишь около 50 % строк исходного кода и еще меньше команд байт-кода, потому что код, относящийся к постусловиям и проверкам инвариантов, не будет выполняться. При включенных директивах `assert` будет достигнуто 100 %-ное покрытие строк. Полное покрытие команд и ветвей не достигается, потому что все проверки проходят и ветви «условие нарушено» не выполняются. Никакой объем тестирования не улучшит ситуацию, потому что тестируемый продукт работает правильно и эти ветви будут недоступны для тестирования.

[1] Этот файл можно найти в пакете `eis.chapter6` репозитория (https://bitbucket.org/mfaella/exercisesinstyle).

НЕОЖИДАННЫЙ ВОПРОС 4

Программа содержит директивы `assert`. Как ее тестировать — с включенными или выключенными директивами `assert`?

6.3. ТЕСТИРУЕМОСТЬ [TESTABLE]

Чтобы протестировать программный модуль, необходимо передать ему входные данные (*управляемость*) и наблюдать, что произойдет (*наблюдаемость*). Более того, если тестируемый модуль (UUT unit, under testing) зависит от других (например, метод вызывает метод другого класса) и тесты выявляют дефект, вы не знаете, какому из модулей принадлежит дефект. Вот почему правильное модульное тестирование требует, чтобы тестируемый модуль был изолирован.

В следующих подразделах аспекты тестирования и улучшения тестируемости раскрываются в контексте нашего примера. Улучшенная версия имеет такую же структуру, что и `Reference`, поля которой я повторю для удобства:

```
public class Container {
   private Set<Container> group;    ❶ Резервуары, соединенные с текущим
   private double amount;           ❷ Объем воды в резервуаре
```

Тем не менее тестируемость является свойством API, и улучшенная версия будет иметь несколько иной (на самом деле более содержательный) открытый интерфейс.

6.3.1. Управляемость

Управляемостью называется простота передачи произвольных входных данных тестируемому модулю. Класс `Container` обладает высокой управляемостью, потому что он получает входные данные прямо от клиентов через API.

Модули с плохой управляемостью получают входные данные из файлов, баз данных, сетевых подключений или, что еще хуже, — графических интерфейсов. В таких случаях для тестирования необходима инфраструктура, моделирующая другой конец канала связи. Я не буду углубляться в подробности, потому что они только отвлекут нас от текущего примера, а на эту тему написаны целые книги. Как обычно, некоторые рекомендации приводятся в разделе «Дополнительная литература» в конце главы.

НЕОЖИДАННЫЙ ВОПРОС 5

Допустим, вы добавили в класс `Container` статический метод, который строит набор объектов резервуаров из файла (метод *десериализации*). Как добавление этого метода повлияет на тестируемость?

6.3.2. Наблюдаемость

При разработке API резервуара с водой (см. главу 1) я стремился к простоте, а по шкале наблюдаемости он оставляет желать лучшего.

Во-первых, методы `connectTo` и `addWater` не возвращают никакого значения. Из тестируемости следует, что все методы должны возвращать некоторое значение для получения некоторой разновидности прямой обратной связи от любого вызова. Например, `connectTo` может вернуть хотя бы логическое значение того, соединены два резервуара или нет (листинг 6.4), по аналогии с тем, как метод `add` класса `Collection` сообщает, успешно ли была выполнена вставка.

Листинг 6.4. Testable: метод connectTo (сокращенный)

```
public boolean connectTo(Container other) {
   if (group==other.group) return false;

      ... ❶ Реализация (как у Reference)

   return true;
}
```

Или в более интересном варианте `addWater` может вернуть объем воды в этом резервуаре после текущего добавления:

Листинг 6.5. Testable: метод addWater

```
public double addWater(double amount) {
   double amountPerContainer = amount / group.size();
   for (Container c: group) { c.amount += amountPerContainer; }
   return this.amount;
}
```

Что касается наблюдения за текущим состоянием резервуара, `getAmount` — единственный метод, предоставляющий какую-либо обратную связь о состоянии резервуара. Такое представление получается крайне ограниченным: как будто мы рассматриваем интерьер комнаты через замочную скважину. Соединения резервуаров полностью скрыты, и их можно косвенно определить только по распределению воды между разными резервуарами. Мы можем достаточно легко добавить в API другие методы, чтобы расширить предоставляемую информацию и улучшить тестируемость. Например, будет естественно добавить метод для проверки того, соединены ли в настоящий момент два резервуара (листинг 6.6). В `Reference` эта проверка реализуется очень легко, потому что соединенные резервуары указывают на один объект группы.

Листинг 6.6. Testable: дополнительный метод isConnectedTo

```
public boolean isConnectedTo(Container other) {
   return group == other.group;
}
```

В главе 7 мы добавим несколько таких методов, но по другой причине: ради удобочитаемости.

НЕОЖИДАННЫЙ ВОПРОС 6

Допустим, в класс `Container` был добавлен открытый метод, который возвращает количество резервуаров, соединенных (прямо или косвенно) с текущим. Как новый метод повлияет на тестируемость?

6.3.3. Isolation: разделение зависимостей

Идея модульного тестирования заключается в проверке поведения одного модуля (например, класса) *в изоляции* от других. Неудачная попытка прохождения теста будет свидетельствовать о наличии дефекта в этом модуле, и искать ошибки в разных классах не придется. Пример с резервуарами является идеальным тестовым «модулем», потому что он не зависит от других классов, кроме стандартных классов JDK.

И наоборот, в большинстве реальных сценариев между классами существуют сложные связи, что затрудняет тестирование и диагностику ошибок. Чтобы свести к минимуму эти проблемы, можно воспользоваться такими методами, как *макетирование* (mocking) и создание *заглушек* (stubbing). Эти методы основаны на замене фактических зависимостей фиктивными, достаточно простыми, чтобы быть вне подозрений. Такие библиотеки, как Mockito и Powermock, помогают автоматизировать подобные задачи.

Распространенным способом улучшения тестируемости при наличии зависимостей является *внедрение зависимостей*. Если тестируемый класс создает объекты другого типа (например, `Container` создает `HashSet`), схема внедрения зависимостей предполагает, что такие объекты будут передаваться клиентом извне.

В нашем сценарии текущий конструктор из `Reference`:

```
public Container() {
   group = new HashSet<Container>();
   group.add(this);
}
```

может быть заменен следующим:

```
public Container(Set<Container> emptySet) {
   group = emptySet;
   group.add(this);
}
```

Новая версия обладает лучшей тестируемостью, потому что в наборе тестов `HashSet` можно заменить простой, возможно, фиктивной реализацией интерфейса `Set`. Она гарантирует, что любой дефект, выявленный тестом, происходит от кода `Container`, но не от `HashSet`. Такой путь может показаться абсурдным в этом контексте, потому что `HashSet` является одним из ключевых элементов JDK, пользующихся полным доверием. И все же запаситесь терпением: мы воспользуемся текущим примером для анализа достоинств и недостатков этого метода.

Версия конструктора с *внедрением* обладает двумя серьезными недостатками.

- Она раскрывает подробности реализации класса `Container` и нарушает инкапсуляцию. Тем самым вы не только сообщаете клиенту, что новым резервуарам необходимо множество, но и даже позволяете клиенту выбрать, какую именно разновидность множества следует использовать! Если позднее вы решите переключиться с представления на базе множества, как в `Reference`, на представление на базе дерева из `Speed3`, вам придется изменить открытый API резервуаров. Это общая проблема внедрения зависимостей: вы должны выдержать баланс между тестируемостью и ослаблением инкапсуляции.
- Требует передавать новый пустой объект `Set` для каждого создаваемого резервуара. У клиента появляется множество рисков допустить ошибку — например, передать множество, которое не является пустым, или передать одно множество более чем одному резервуару.

Для начала можно проверить, что переданное клиентом множество действительно пусто, и прервать выполнение в противном случае. Более того, чтобы предотвратить использование одного пустого множества для инициализации нескольких резервуаров, можно *скопировать* аргумент-множество (если выбранная клиентом реализация поддерживает клонирование).

```
public Container(Set<Container> emptySet) {
   if (!emptySet.isEmpty())
      throw new IllegalArgumentException("The set is supposed to be empty!");
   group = (Set<Container>) emptySet.clone();
   group.add(this);
}
```

Наконец, механизм отражения позволяет написать вариант, гарантирующий пустоту и избегающий клонирования. Вы получите объект `Class` и используете его для создания экземпляра нового пустого множества (листинг 6.7). Обратите внимание на использование обобщений для гарантии того, что предоставленный клиентом объект `Class` связан с реализацией `Set<Container>`. Но есть загвоздка: реализация множества, выбранная клиентом, должна предоставлять конструктор без аргументов. В противном случае метод `getDeclaredConstructor` выдаст исключение.

Листинг 6.7. Testable: конструктор, поддерживающий внедрение зависимостей

```
public Container(Class<? extends Set<Container>{}> setType)
       throws ReflectiveOperationException {
   group = setType.getDeclaredConstructor()
                  .newInstance();
   group.add(this);
}
```

На практике, вместо того чтобы реализовывать внедрение зависимостей «с нуля», лучше воспользоваться фреймворком:

ФРЕЙМВОРКИ ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ

Внедрение зависимостей (DI, Dependency Injection) поддерживается Java Enterprise Edition (сейчас известным как Jakarta EE) и рядом фреймворков Java, таких как Google Guice (маленькая библиотека) и Spring (большая библиотека для корпоративных приложений). Все они предоставляют следующую функциональность:

1. Пометка метода или конструктора как требующего внедрения зависимостей. Обычно для этого используется аннотация. Например, в Spring используется `@Autowired`, а в Guice и JEE — `@Inject`. Подобные взаимодействия, при которых вы приказываете фреймворку вызвать ваш код, также называются инверсией управления.
2. Конкретные классы связываются с внедряемыми параметрами.
3. Во время выполнения фреймворк берет на себя создание экземпляров соответствующих конкретных классов и передачу их соответствующему методу или конструктору.

6.4. А ТЕПЕРЬ СОВСЕМ ДРУГОЕ

Как обычно, применим методы из этой главы в другом примере. На этот раз позаимствуем пример — структуру данных ограниченного множества — из главы 5, потому что любой работе по тестированию (глава 6) должно предшествовать установление четко проработанных контрактов (глава 5).

Для резервуаров с водой я помог вам спроектировать тестовые сценарии, а затем рассказал о проблемах тестируемости и обсудил различные улучшения API. Для ограниченных множеств я пойду по противоположному пути, который ближе к тому, что вы делаете (или должны делать) на практике:

- Сначала мы спроектируем (или доработаем) API с учетом требований тестируемости.

- Затем спроектируем набор тестов.

Вспомните, что `BoundedSet` представляет множество с фиксированной емкостью, устанавливаемой во время конструирования, и обладает следующей функциональностью (глава 5):

- `void add(T elem)` — добавляет заданный элемент в ограниченное множество. Если в результате добавления размер множества превышает его емкость, то метод удаляет из множества *самый старый* элемент (вставленный первым).

  Добавление элемента, который уже принадлежит множеству, приводит к его *обновлению* (делает его самым новым элементом в множестве).

- `boolean contains(T elem)` — возвращает `true`, если ограниченное множество содержит заданный элемент.

В главе 5 мы решили использовать для представления ограниченного множества связанный список и емкость, то есть предельный размер:

```
public class BoundedSet<T> {
   private final LinkedList<T> data;
   private final int capacity;
```

А теперь проанализируем и улучшим тестируемость `BoundedSet`.

6.4.1. Повышение удобства тестирования

Можно заметить, что API ограниченного множества обладает плохой наблюдаемостью, потому что есть только один метод `contains`, предоставляющий информацию о его состоянии. Невозможно определить порядок вставки элементов или хотя бы узнать, какой элемент является самым старым (и, следовательно, должен быть удален следующим). Собственно, неизвестен даже текущий размер множества.

Как объяснялось в разделе, посвященном наблюдаемости, первым улучшением может стать добавление возвращаемого значения ко всем методам, у которых его нет. Например, можно дополнить `add` возвращаемым значением типа `T`, представляющим объект, вытесняемый из множества (если он есть). Это похоже на то, как `Map.put(key, val)` возвращает значение, ранее связанное с ключом.

Рассмотрите следующий обновленный контракт метода `add`. Кроме описания возвращаемого значения, он указывает, что `null`-аргумент недопустим. В противном случае возвращаемое значение `null` было бы неоднозначным:

- `T add(T elem)` — добавляет заданный элемент в ограниченное множество. Если в результате добавления размер множества превышает его емкость, то метод удаляет из множества и возвращает *самый старый* элемент (вставленный первым). В противном случае возвращается `null`.

Добавление элемента, который уже принадлежит множеству, приводит к его *обновлению* (делает его самым новым элементом во множестве).

Методу не может передаваться аргумент `null`.

Возвращение значения `add` — хорошее начало, но оно позволяет запросить состояние ограниченного множества *только тогда, когда вы его изменяете*. Чтобы дополнительно улучшить тестируемость, класс должен предоставить доступ ко всей информации, актуальной для его внешнего поведения (то есть ко всей информации, влияющей на поведение в восприятии клиента). В этом случае, кроме стандартного размера метода, необходимо иметь возможность проверки текущего порядка элементов, потому что этот порядок влияет на будущие вызовы `add` и `contains`. Сравним некоторые способы предоставления доступа к порядку элементов:

1. Предоставить клиенту прямой доступ к внутреннему списку объектов при помощи дополнительного метода:

```
public List<T> content() {
    return data;
}
```

   Не стоит и говорить, что это очень плохо, — клиент сможет вытворять с внутренним представлением все, что ему заблагорассудится!

2. Предоставить клиенту *копию* внутреннего списка объектов:

```
public List<T> content() {
   return new ArrayList<>(data);
}
```

   Это лучше, чем вариант 1, но неэффективно, ведь копирование требует линейного времени, а сторона вызова может изменить список, что бессмысленно и, скорее всего, приведет к ошибкам. (Сторона вызова может ошибочно полагать, что она изменяет само ограниченное множество.)

3. Предоставить клиенту *неизменяемое представление* внутреннего списка объектов, то есть объект, инкапсулирующий исходный список, у которого заблокированы все методы, способные его изменить (такие, как `add` и `remove`). Неизменяемые представления стандартных коллекций предоставляются парой статических методов из класса `Collections`. В данном случае проблема решается следующей однострочной реализацией:

```
public List<T> content() {
   return Collections.unmodifiableList(data);
}
```

   По сравнению с предыдущими решениями это лучше во всех отношениях: эффективно, потому что не требует копирования списка, и не создает рисков, потому что возвращаемый объект доступен только для чтения.

У всех трех решений есть один общий недостаток: вы обязуетесь использовать чрезвычайно выразительный возвращаемый тип — `List`. На данный момент это обязательство легко выполняется, потому что внутреннее представление само по себе является списком. Если в будущем вы измените свое решение относительно внутреннего представления (например, переключившись на массив), реализация может значительно усложниться. Следующее решение избегает этой проблемы через ограниченное представление содержимого: итератор вместо списка.

4. Предоставить клиенту итератор только для чтения, предназначенный для перебора содержимого. Напомню, что итератор может изменять свою коллекцию методом `remove`. Нужно проследить, чтобы `remove` был недоступен в возвращаемом итераторе. И снова задача решается при помощи неизменяемого представления:

```
public class BoundedSet<T> implements Iterable<T> {
   ...
   public Iterator<T> iterator() {
      return Collections.unmodifiableList(data).iterator();
}
```

В следующем разделе предполагается, что был выбран вариант 3 (метод, возвращающий неизменяемое представление списка), потому что он обеспечивает наилучшую тестируемость.

6.4.2. Набор тестов

Сосредоточимся на тестировании `add` — единственного метода, изменяющего ограниченное множество. При анализе контракта `add` можно выявить три характеристики, связанные с его поведением:

C1. Равен ли `null` передаваемый `add` аргумент? Если нет, можно ожидать выдачи исключения `NullPointerException` в качестве штрафа.

C2. Каков размер ограниченного множества перед вставкой? В частности, поведение `add` изменяется при заполненном ограниченном множестве (то есть если размер равен емкости множества). Также удобно исключить случай пустого ограниченного множества, потому что он может быть сопряжен с повышенным риском ошибок, как и все пограничные случаи.

C3. Присутствует ли аргумент `add` в ограниченном множестве перед вставкой? Это важно, потому что вставка уже присутствующего элемента не должна приводить к вытеснению любого элемента, даже если множество заполнено.

В табл. 6.6 приведена сводка этих характеристик и их возможных значений (блоков).

Таблица 6.6. Характеристики, выбранные для тестирования метода add ограниченных множеств

Название	Характеристика	Блоки
C1	Значение аргумента	{`null`, `other`}
C2	Размер множества перед вставкой	{пустое, заполненное, другое}
C3	Присутствие аргумента перед вставкой	{отсутствует, присутствует}

Количество осмысленных комбинаций блоков сокращается двумя ограничениями между этими характеристиками:

- Если элемент равен `null` (C1 = `null`), он не может присутствовать (C3 ≠ присутствует).
- Если ограниченное множество пусто перед вставкой (C2 = пустое), элемент не может присутствовать (C3 ≠ присутствует).

Из-за этих ограничений остаются следующие восемь комбинаций:

1. (C1 = `null`, C2 = пустое, C3 = отсутствует);
2. (`null`, полное, отсутствует);
3. (`null`, другое, отсутствует);
4. (другое, пустое, отсутствует);
5. (другое, полное, отсутствует);
6. (другое, другое, отсутствует);
7. (другое, полное, присутствует);
8. (другое, другое, присутствует).

Как упоминалось ранее в этой главе, первые три комбинации можно свернуть в одну, потому что эти случаи нарушают предусловие по одной и той же причине — аргумент `null`. В результате остаются шесть тестовых сценариев.

Чтобы реализовать тестовые сценарии в JUnit, начнем с инициализации ограниченного множества с емкостью 3, которое будет использоваться в качестве тестовой оснастки. Такая емкость ограниченна, но достаточно велика для поддержки всех интересных аспектов поведения ограниченных множеств.

```
public class BoundedSetTests {
   private BoundedSet<Integer> set; ❶ Тестовая оснастка

   @Before (2) Выполняется перед каждым тестом
   public void setUp() {
      set = new BoundedSet<>(3);
   }
```

Затем я приведу код первых трех тестов. На этот раз я воспользуюсь сопоставителями Hamcrest для написания более удобочитаемых условий `assert`:

- `is` — сквозной сопоставитель. Он ничего не проверяет, мы просто включаем его, чтобы условие читалось как на естественном языке;
- `nullValue` — `null` в терминологии Hamcrest;
- `Contains`, который сравнивает `Iterable` с явной последовательностью значений. Они считаются совпадающими, если содержат одинаковые элементы (при сравнении методом `equals`) в одинаковом порядке.

Каждый сопоставитель представляет собой статический метод из класса `org.hamcrest.Matchers`. Чтобы использовать неуточненные имена, необходимо выполнить их статическое импортирование.

В следующих тестах обратите внимание на то, как метод `content` — добавленный для удобства тестирования — работает в сочетании с сопоставителем `contains`. Возвращение списка (с таким же успехом можно использовать объект `Iterable`) позволит за один раз сравнить всю последовательность элементов с ее ожидаемым состоянием.

```
@Test(expected = NullPointerException.class) ❶ C1 = null
public void testAddNull() {
   set.add(null);
}

@Test
public void testAddOnEmpty() {   ❷ C1 = другое, C2 = пустое
   Integer result = set.add(1);
   assertThat("Wrong return value",
              result, is(nullValue())); ❸ Проверка null в Hamcrest
   assertThat("Wrong set content",
              set.content(), contains(1)); ❹ Сопоставитель для Iterable в Hamcrest
}
@Test
public void testAddAbsentOnNonFull() { ❺ C1 = другое, C2 = другое, C3 = отсутствует
   set.add(1);
   Integer result = set.add(2); ❻ Тестируемая строка
   assertThat("Wrong return value", result, is(nullValue()));
   assertThat("Wrong set content", set.content(), contains(1, 2));
}
```

В двух из приведенных тестов нарушается важная рекомендация использовать одно тестовое условие для каждого тестового правила. Модульные тесты должны быть целенаправленными — другими словами, каждый тест может не проходить только по одной причине. Как обычно при разработке, к таким рекомендациям стоит относиться с долей скепсиса. Ничто не мешает разбить каждый тест на два: один проверяет возвращаемое значение `add`, а другой проверяет состояние множества после вставки. Но исходные тесты настолько просты, что лишние

строки кода вряд ли нужны. Сообщение об ошибке в тестовом условии поясняет причину неудачи.

6.5. РЕАЛЬНЫЕ СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

Если вы были разработчиком, вам наверняка доводилось слышать фразы «я знаю, модульные тесты полезны, но у меня нет времени их писать» или «сначала завершим написание библиотеки, а если останется время — напишем модульные тесты». В первом случае расплата — всего лишь вопрос времени. Второй случай отражает устаревший подход к разработке: большинство тестов создавалось после написания продукта (так называемая *каскадная модель*). Рассмотрим некоторые сценарии использования, в которых тестирование может принести пользу.

- Вы работаете в команде разработки над успешной платформой связующего звена (middleware). Руководство потребовало, чтобы ваша команда раскрыла часть функциональности приложения, выполняемого в отделе финансов для начисления зарплаты, через REST-сервисы. Хоть вы и доверяете своим коллегам, вам бы хотелось избежать несанкционированных повышений зарплаты. Чтобы убедиться в правильности работы сервиса, вы решаете написать тесты. Тестирование REST-сервисов может быть достаточно хлопотным делом, но, к счастью, существуют библиотеки, которые помогут в создании чистых изолированных тестов для вашего API.
- Включение модели машинного обучения (ML, machine-learning) в рабочую версию продукта обычно означает, что она становится частью рабочего процесса. Допустим, автоматизированное задание запускается ежедневно рано утром, обращается с запросом к базе данных и экспортирует данные для передачи обученным моделям машинного обучения, чтобы они могли построить прогнозы продаж на завтра. Недавно принятый на работу специалист по работе с базой данных решает оптимизировать некоторые запросы. Но оказывается, что изменения влияют на формат результатов запросов, и после экспортирования данных процесс ломается. После инцидента команда базы данных решает написать модульные тесты, чтобы данные, экспортированные по результатам запросов, гарантированно соответствовали ожиданиям моделей машинного обучения для построения прогнозов.
- После окончания колледжа вы поняли, что пришло время преобразовать результаты исследований в коммерческий продукт. Вы приглашаете самых надежных друзей и после долгих переговоров решаете основать стартап. Проходит пара лет. До творения Билла Гейтса еще далеко, но выглядит неплохо: компания растет, как и кодовая база. Когда-то вы предвидели, что это случится, и написанные вами автоматизированные тесты станут страховкой для вашей команды разработки. Тесты развивались параллельно с остальной

кодовой базой. Их и надо писать *до* добавления новой функциональности. Этот принцип заложен в основу методологии разработки через тестирование (TDD, test-driven development): запрограммировать сценарий на основе ваших целей, выполнить тесты (которые должны завершиться неудачей), а затем вносить исправления, чтобы добиться прохождения тестов.

6.6. ПРИМЕНИМ ИЗУЧЕННОЕ

Упражнение 1

Спроектируйте и реализуйте план тестирования для метода `getDivisors`, определяемый следующим контрактом:

- *Предусловие:* метод получает целое число `n` в единственном параметре.
- *Постусловие:* метод возвращает список `List` с элементами `Integer`, содержащий все делители `n`. Для `n==0` возвращается пустой список. Для отрицательного `n` возвращается такой же список, как для значения с обратным знаком.

  Например, для 12 и –12 возвращается список `[1, 2, 3, 4, 6, 12]`.
- *Штраф:* нет (любые целые числа являются допустимыми аргументами).

Упражнение 2

Спроектируйте и реализуйте план тестирования для метода

```
public int indexOf(int ch, int fromIndex)
```

класса `String`, используя метод моделирования области входных значений.

Упражнение 3

1. Используя метод моделирования области входных значений, спроектируйте и реализуйте план тестирования для метода `interleaveLists`, определяемого следующим контрактом (то же, что упражнение 2 из главы 5):
   - *Предусловие:* метод получает в аргументах два списка `List` одинаковой длины.
   - *Постусловие:* метод возвращает новый список `List`, содержащий все элементы двух списков с чередованием.
   - *Штраф:* если по крайней мере один из списков равен `null`, метод выдает исключение `NullPointerException`. Если списки имеют разную длину, метод выдает исключение `IllegalArgumentException`.
2. Оцените покрытие кода, обеспечиваемое вашим планом, вручную или с помощью соответствующей программы.

Упражнение 4

Улучшите тестируемость обобщенного интерфейса `PopularityContest<T>`, представляющего конкурс популярности среди элементов (динамически расширяемого) множества объектов типа `T`. Интерфейс содержит следующие методы:

- `void addContestant(T contestant)` добавляет нового участника. Попытки добавления дубликатов игнорируются;
- `void voteFor(T contestant)` регистрирует голос за заданного участника. Если участник не входит в конкурс, выдается исключение `IllegalArgumentException`;
- `T getMostVoted()` возвращает участника, который получил наибольшее количество голосов на текущий момент. Если конкурс пуст (нет ни одного участника), выдается исключение `IllegalStateException`.

ИТОГИ

- Метод моделирования области входных значений помогает определить релевантные тестовые входные значения.
- Входные значения для разных параметров можно объединять разными способами, что приводит к большему или меньшему количеству тестов и разным уровням покрытия.
- Наборы тестов оцениваются по метрикам покрытия ввода и покрытия кода.
- Тестируемость можно улучшить на основе большей обратной связи от методов.
- Механизм внедрения зависимостей позволяет изолировать тестируемый класс, используя вместо зависимостей их более простые заменители.

ОТВЕТЫ НА ВОПРОСЫ И УПРАЖНЕНИЯ

Неожиданный вопрос 1

Все части контракта важны для тестирования. Так как постусловие описывает предполагаемый результат применения метода, оно диктует условия, которые будет проверять тест. Многие написанные вами модульные тесты будут отправлять допустимые входные значения тестируемому методу и проверять, что результаты соответствуют постусловию. Предусловие описывает диапазон допустимых входных значений. Наконец, как объясняется в этой главе, другие тесты будут отправлять недопустимые входные значения и проверять, что метод реагирует так, как описано в разделе «Штраф» контракта.

Неожиданный вопрос 2

Даты нередко становятся источником колоссальных неприятностей для программистов и тестировщиков. Даже если забыть о трудностях региональных стандартов и придерживаться грегорианского календаря, программам (а следовательно, и тестам) придется иметь дело с самыми разнообразными аномалиями. Во-первых, месяц может состоять из 28, 29, 30 или 31 дня; особенно капризен февраль. В табл. 6.7 приведена сводка трех возможных характеристик.

Таблица 6.7. Три возможные характеристики типа данных для представления даты

Название	Характеристика	Блоки
C1	Високосный год	{`true`, `false`}
C2	Длина месяца	{28, 29, 30, 31}
C3	День месяца	{первый, промежуточный, последний}

Неожиданный вопрос 3

- Полнокомбинаторное покрытие: $n_1 \times n_2 \times n_3$.
- Полновариантное покрытие: $\max\{n_1, n_2, n_3\}$.
- Покрытие базового варианта: $1 + (n_1 - 1) + (n_2 - 1) + (n_3 - 1) = n_1 + n_2 + n_3 - 2$.

Неожиданный вопрос 4

Зачем выбирать что-то одно? Сначала протестируйте с выключенными директивами `assert`, потому что рабочая версия программы должна выполняться так. Если какой-либо тест не будет пройден, снова выполните его с включенными директивами `assert`. Это упростит выявление дефекта.

Неожиданный вопрос 5

Добавление метода десериализации ухудшит тестируемость, потому что новый метод получит сложный вход из файла.

Неожиданный вопрос 6

В общем случае методы, доступные только для чтения, улучшают тестируемость, потому что они безопасны (в них труднее допустить ошибку) и предоставляют дополнительную возможность контроля состояния объектов. Таким образом, добавление метода `groupSize` улучшает тестируемость.

Упражнение 1

В качестве первой характеристики C1 можно выбрать знак входного значения `n`, как указано в списке стандартных признаков на базе типа (см. табл. 6.1). Вторая характеристика C2, следующая из постусловия, может быть основана на количестве делителей, возвращаемых методом. Она дает следующие четыре блока:

- Ни одного делителя — только для `n` = 0. Контракт указывает, что результатом должен быть пустой список.
- Один делитель — только для `n` = 1 и `n` = −1.
- Два делителя — для всех простых чисел и обратных им.
- Более двух делителей — для всех остальных вариантов входа.

Краткая сводка характеристик приведена в табл. 6.8.

Таблица 6.8. Характеристики для входного значения n метода getDivisors

Название	Характеристика	Блоки
C1	Знак	{отрицательное, нуль, положительное}
C2	Количество делителей	{нуль, один, два, более двух}

Как видите, две характеристики не являются независимыми, потому что C1 = нуль может существовать только в паре с C2 = нуль. В результате вместо 3 × 4 = 12 мы получим только семь осмысленных комбинаций и сможем применить полнокомбинаторное покрытие без особых усилий. Остальные пять тестов можно найти в репозитории (https://bitbucket.org/mfaella/exercisesinstyle) в классе `eis.chapter6.exercises.DivisorTests`:

```
@Test
public void testZero() { ❶ C1 = C2 = нуль
   List<Integer> divisors = getDivisors(0);
   assertTrue("Divisors of zero should be the empty list",
              divisors.isEmpty());
}

@Test
public void testMinusOne() { ❷ C1 = отрицательное, C2 = один
   List<Integer> divisors = getDivisors(-1);
   List<Integer> expected = List.of(1);
   assertEquals("Wrong divisors of -1", expected, divisors);
}
```

Упражнение 2

Документацию Javadoc для метода `indexOf` можно обобщить и воплотить в форме контракта следующим образом:

- *Предусловие:* нет (все вызовы допустимы).
- *Постусловие:* метод возвращает индекс первого вхождения заданного символа в строке, начиная поиск от заданного индекса. Возвращает –1, если символ не будет найден.

  Отрицательное значение `fromIndex` интерпретируется как нуль. С отрицательным значением `ch` возвращается –1.
- *Штраф* — нет.

При выборе хороших характеристик можно руководствоваться только постусловием кроме стандартных характеристик на базе типа. (Если вы их не помните, вернитесь к главе 6.1.) Первую характеристику C1 можно взять непосредственно из стандартных: пустая строка. Первый параметр `ch` представляет собой целое число, представляющее символ Юникода. К нему можно применить стандартную характеристику знака и назвать ее C2. Второй параметр `fromIndex` также является целым числом, которое должно быть меньше длины строки `n`. Чтобы распределить значения, необходимо ввести характеристику C3, объединяющую стандартную характеристику знака с отношением между `fromIndex` и `n`. В результате мы получим пять случаев:

- значение `fromIndex` отрицательно;
- значение `fromIndex` равно нулю, а строка пуста (недопустимый нуль);
- значение `fromIndex` равно нулю, а строка не пуста (допустимый нуль);
- значение `fromIndex` положительно и по крайней мере не меньше `n` (недопустимое положительное);
- значение `fromIndex` положительно и меньше `n` (допустимое положительное).

Наконец, характеристика C4 кодирует присутствие символа в заданной подстроке. В табл. 6.9 приведена сводка этих характеристик.

Таблица 6.9. Характеристики, выбранные для тестирования indexOf

Название	Характеристика	Блоки
C1	Признак пустой строки	{пустая, непустая}
C2	Знак `ch`	{отрицательное, нуль, положительное}
C3	Знак `fromIndex` и отношение с длиной строки	{отрицательное, допустимый нуль, недопустимый нуль, допустимое положительное, недопустимое положительное}
C4	Присутствие символа в подстроке	{присутствует, отсутствует}

Из 60 возможных комбинацией действительными являются следующие 27 (символ * используется как универсальный):

- (пустая, *, {отрицательное, недопустимый нуль, недопустимое положительное}, отсутствует) (9 комбинаций);
- (непустая, *, {отрицательное, допустимый нуль, недопустимое положительное, допустимое положительное}, отсутствует) (12 комбинаций);
- (непустая, {нуль, положительное}, {отрицательное, допустимый нуль, допустимое положительное}, присутствует) (6 комбинаций).

Если вы не хотите писать 27 тестов, то можете переключиться с полнокомбинаторного покрытия на одну из более ограниченных стратегий, представленных в этой главе. Здесь я остановлюсь на полновариантном покрытии и рассмотрю небольшую подборку комбинаций, в которой каждый блок из каждой характеристики встречается по крайней мере один раз. Обратите внимание: любое решение включает как минимум пять комбинаций, потому что CS поддерживает пять блоков. Одно из возможных решений:

1. (непустая, положительное, допустимое положительное, присутствует);
2. (непустая, положительное, отрицательное, присутствует);
3. (непустая, нуль, недопустимое положительное, отсутствует);
4. (непустая, отрицательное, допустимый нуль, отсутствует);
5. (пустая, положительное, недопустимый нуль, отсутствует).

Реализация первого теста на базе JUnit:

```
public class IndexOfTests {
   private final static String TESTME = "test me";

   @Test
   public void testNominal() {
      int result = TESTME.indexOf((int)'t', 2);
      assertEquals("test with nominal arguments", 3, result);
}
```

Другие тесты можно найти в сетевом репозитории (https://bitbucket.org/mfaella/exercisesinstyle).

Упражнение 3

1. Из предусловия следуют два свойства, включенные в характеристики: отличие списков от `null` и равенство их длин. Кроме того, особым случаем является пустая коллекция. В результате обобщения этих наблюдений мы получаем три характеристики, представленные в табл. 6.10.

Таблица 6.10. Характеристики, выбранные для тестирования interleaveLists

Название	Характеристика	Блоки
C1	Признак пустой строки	{пустая, непустая}
C2	Тип второго списка	{`null`, пустая, непустая}
C3	Списки имеют одинаковую длину	{`true`, `false`}

Характеристика C3 не является независимой от C1 и C2, поэтому некоторые комбинации не имеют смысла. Ниже перечислены осмысленные комбинации:

1. (`null`, непустая, `false`);
2. (непустая, `null`, `false`);
3. (пустая, пустая, `true`);
4. (пустая, непустая, `false`);
5. (непустая, пустая, `false`);
6. (непустая, непустая, `false`);
7. (непустая, непустая, `true`).

Вы спросите, почему я пропустил комбинацию (`null`, `null`, `false`)? Потому что если характеристика нарушает предусловие, достаточно объединить ее с номинальными (то есть нормальными) значениями других характеристик. Кстати говоря, включение этой комбинации — излишняя предосторожность, но ошибкой оно определенно не будет. Обратите внимание: только комбинации 3 и 7 удовлетворяют предусловию.

Так как есть только семь комбинаций, их все можно протестировать без особого труда. Ниже приведен код для первых трех комбинаций:

```
public class InterleaveTests {
   private List<Integer> a, b, result; ❶ Оснастки

   @Before
   public void setUp() {                ❷ Инициализация оснасток
      a = List.of(1, 2, 3);
      b = List.of(4, 5, 6);
      result = List.of(1, 4, 2, 5, 3, 6);
   }

   @Test(expected = NullPointerException.class)
   public void testFirstNull() { ❸ Тест 1: (null, непустая, false)
      InterleaveLists.interleaveLists(null, b);
   }
```

```
@Test(expected = NullPointerException.class)
public void testSecondNull() { ❹ Тест 2: (непустая, null, false)
   InterleaveLists.interleaveLists(a, null);
}

@Test
public void testBothEmpty() { ❺ Тест 3: (пустая, пустая, true)
   a = List.of();
   b = List.of();
   List<Integer> c = InterleaveLists.interleaveLists(a, b);
   assertTrue("should be empty", c.isEmpty());
}
```

Остальные тесты можно найти в репозитории (https://bitbucket.org/mfaella/exercisesinstyle).

2. Для начала отметим, что измерение покрытия вспомогательного метода, проверяющего постусловия, особого смысла не имеет. Выполнение каждой строки этого метода нецелесообразно, потому что мы надеемся, что постусловие будет выполняться, а это означает, что тесты всегда будут пропускать некоторые строки из `interleaveCheckPost`.

   Если ограничить анализ телом `interleaveLists`, описанные ранее семь тестов обеспечат 100 %-ное покрытие.

Упражнение 4

Данный интерфейс обладает хорошей управляемостью, но его наблюдаемость можно улучшить. В настоящее время `getMostVoted` остается единственной точкой доступа к внутреннему состоянию объекта, притом весьма ограниченной. Узнать можно только участника, за которого было отдано больше всего голосов, а счетчики голосов отдельных участников недоступны. Для улучшения ситуации можно начать с дополнения двух других методов возвращаемыми значениями. Например:

- `boolean addContestant(T contestant)` добавляет участника и возвращает `true`, если участник еще не включен в конкурс. В противном случае оставляет набор участников без изменений и возвращает `false`;
- `int voteFor(T contestant)` регистрирует голос за заданного участника и возвращает обновленное количество голосов. Если участник не входит в конкурс, выдается исключение `IllegalArgumentException`.

Новая версия `voteFor` становится мощным инструментом тестирования, но она сводит воедино голосование и получение количества голосов. Для тестирования будет полезно добавить метод чтения текущего количества голосов, доступный только для чтения:

- `int getVotes(T contestant)` возвращает текущее количество голосов для заданного участника. Если участник не участвует в конкурсе, выдается исключение `IllegalArgumentException`.

Кроме того, метод `getVotes` предоставляет возможность проверить, входит ли участник в число участников конкурса, без его изменения.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

- *Майерс Г., Баджетт Т., Сандлер К.* Искусство тестирования программ. — 3-е изд. — М. [и др.]: Диалектика: Вильямс, 2015.

  Разностороннее введение в тестирование и другие средства проверки, такие как рецензирование кода и инспекция. Книга сочетает вводное описание принципов, проверенных временем (первое издание опубликовано в 1979 году), с обновленным описанием гибких методологий тестирования.

- *Koskela L.* Effective Unit Testing. Manning Publications, 2013.

  Книга полна практических рекомендаций по проектированию эффективных тестов, включая каталог типичных недостатков тестов.

- *Freeman S., Pryce N.* Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, 2009.

  Процессно-ориентированная книга, демонстрирующая разработку через тестирование (TDD) и макетирование на реальном примере — от создателей популярной библиотеки макетирования jMock.

- *Ammann P., Offutt J.* Introduction to Software Testing. Cambridge University Press, 2010.

  Современное компактное изложение методов тестирования, с унифицированным описанием разновидностей критериев покрытия.

7 Удобочитаемость

В этой главе:

- ✓ Написание удобочитаемого кода.
- ✓ Документирование контрактов с использованием комментариев Javadoc.
- ✓ Замена комментариев к реализации самодокументируемым кодом.

Исходный код предназначен для двух совершенно разных категорий пользователей: программистов и компьютеров. Некрасивый, путаный код не смутит компьютер — формальную, четко структурированную систему. С другой стороны, мы, программисты, весьма чувствительны к тому, как написана программа. Даже пропуски и отступы, которые ничего не значат для компьютера, могут отличать для нас понятный код от непонятного. (Яркий пример такого рода приведен в приложении А.) В свою очередь, понятный код повышает надежность программы, потому что обычно в нем кроется меньше ошибок, а также улучшает сопровождаемость, потому что его проще изменять.

В этой главе приводятся некоторые современные рекомендации по написанию удобочитаемого кода. Как и в других главах, я не пытался написать исчерпывающее изложение всех рекомендаций и трюков по удобочитаемости. Я сосредоточусь на основных приемах, которые имеют смысл в небольших кодовых модулях, и продемонстрирую их на практике на нашем сквозном примере.

7.1. РАЗНЫЙ ВЗГЛЯД НА УДОБОЧИТАЕМОСТЬ

Написание удобочитаемого кода — недооцененное искусство. Его редко преподают в школах, но оно оказывает колоссальное влияние на надежность,

сопровождаемость и эволюцию программного продукта. Программисты учатся выражать нужную функциональность в коде, понятном для машины. Процесс кодирования занимает время и накладывает один уровень абстракции на другой для проведения декомпозиции этих функциональностей на меньшие единицы. В терминологии Java такими абстракциями являются пакеты, классы и методы. Если система достаточно велика, ни один программист не держит под контролем всю кодовую базу. Некоторые разработчики работают с вертикальным представлением функциональности: от требований до реализации через все уровни абстракции. Другие могут отвечать за один уровень и контролировать его API. Время от времени всем им приходится читать и разбираться в коде, написанном их коллегами.

Улучшение удобочитаемости означает, что компетентному программисту требуется меньше времени, чтобы разобраться в коде. Более конкретной характеристикой может быть *время, которое необходимо человеку, незнакомому с кодом, чтобы почувствовать достаточную уверенность для внесения изменений в код без нарушения его работоспособности*. Другие названия этого свойства кода — *изучаемость* и *ясность*.

НЕОЖИДАННЫЙ ВОПРОС 1

На какие другие свойства кода влияет удобочитаемость?

Как написать удобочитаемую программу? Еще в 1974 году, когда языку C было всего два года, эта проблема считалась достаточно важной для систематического изучения, что привело к появлению авторитетной книги «Элементы стиля программирования». В ней Керниган (известный как один из создателей C) и Плоджер разбирают несколько небольших программ, позаимствованных из опубликованных учебников, подводя итог своих здравых и на удивление современных наблюдений в списке афоризмов по стилю программирования. Первый афоризм хорошо обобщает всю проблему удобочитаемости:

> *Высказывайте то, что подразумеваете, просто и прямо.*

И действительно, суть удобочитаемости — четкое выражение предназначения кода. Грэди Буч, один из архитекторов UML, предлагает естественную аналогию:

> *Чистый код читается как хорошо написанная проза.*

Создание хорошо написанной прозы — не та задача, которую можно решить по фиксированному набору правил. Для этого нужны годы практики — и не только в написании, но и в чтении хорошо написанной прозы известных авторов. Выразительные способности компьютерного кода, бесспорно, ограниченны по сравнению с естественными языками, и, к счастью, процесс создания чистого кода немного проще (или по крайней мере лучше структурирован), чем сочинение

рассказа. И все же освоение этого процесса потребует многолетней практики, которую не заменит никакая книга (или глава книги!). В этой главе будут исследованы некоторые возможности улучшения удобочитаемости кода, в первую очередь методы, которые можно применить в текущем примере.

За два последних десятилетия удобочитаемость была выведена на передний план движением Agile, ориентированным на *рефакторинг* и *чистоту кода*. Рефакторинг — концепция реорганизации рабочей системы с целью улучшения ее структуры, чтобы будущие изменения стали более простыми и безопасными. Это один из ингредиентов облегченных процессов разработки, которые отдают предпочтение быстрым фазам разработки и итеративному совершенствованию продукта.

Даже если вы или ваша компания не принимаете всю философию Agile, не стоит недооценивать литературу, которая ее сопровождает. В книгах можно найти немало выдающихся идей о том, что такое плохо (код «с душком»), что такое хорошо (чистый код) и как превратить первое во второе (рефакторинг). За конкретными рекомендациями обращайтесь к разделу «Дополнительная литература» в конце этой главы.

Конечно, рекомендации по удобочитаемости, разработанные знаменитыми экспертами, было бы хорошо дополнить точными данными по их эффективности. К сожалению, удобочитаемость субъективна по своей природе, и будет в высшей степени тяжело разработать средства для ее объективного измерения. Это не помешало ученым предложить ряд формальных моделей, которые пытаются сформировать оценку удобочитаемости как комбинацию простых числовых метрик (длина идентификаторов, количество круглых скобок в выражениях и т. д.). Непрекращающаяся работа еще не привела к устойчивому консенсусу, поэтому я сосредоточусь на общепринятых отраслевых практиках. А начнем мы с краткого обзора стилевых политик крупнейших игроков в IT-сфере.

7.1.1. Корпоративные руководства по стилю программирования

Некоторые из крупнейших компаний-разработчиков ПО публикуют рекомендации по стилю программирования в интернете:

- Компания Sun когда-то опубликовала официальное руководство по стилю программирования на Java, которое не обновлялось с 1999 года. Архивная копия доступна по адресу http://mng.bz/adVx.
- У Google имеется руководство по стилю, действующее в масштабах компании: https://google.github.io/styleguide/javaguide.html.
- Twitter предоставляет библиотеку общих инструментов Java, к которой прилагается руководство по стилю: http://mng.bz/gVAZ. В нем содержатся явные ссылки на руководства Google и Oracle, послужившие источниками.

- Facebook также предоставляет руководство по стилю с библиотекой вспомогательных классов Java: http://mng.bz/eDyw.

Руководства в целом сходятся на общих принципах, изложенных в этой главе, и отличаются только по уровню подробностей и в небольших косметических аспектах. Для примера возьмем серию директив `import` в начале исходного файла. Одна из таких серий в формате Google выглядит так:

```
import static com.google.common.base.Strings.isNullOrEmpty;
import static java.lang.Math.PI;

import java.util.LinkedList;
import javax.crypto.Cypher;
import javax.crypto.SealedObject;
```

А вот как выглядят те же директивы в стиле Twitter:

```
import java.util.LinkedList;

import javax.crypto.Cypher;
import javax.crypto.SealedObject;

import static com.google.common.base.Strings.isNullOrEmpty;

import static java.lang.Math.PI;
```

Два варианта отличаются как порядком следования, так и использованием пустых строк. С другой стороны, Oracle и Facebook допускают произвольную структуру директив.

Руководства по стилю обеспечивают некоторое единообразие кодовой базы компании и являются приятным добавлением к «приветственному пакету» для новых работников, чтобы они могли с чем-то повозиться, до того как начнутся настоящие проблемы. (А когда они начнутся, они смогут сказать: «По крайней мере я соблюдал руководство по стилю!») Впрочем, для долгосрочного профессионального роста вам будет намного полезнее просмотреть эту главу, а затем обратиться к более подробным книгам по стилю, перечисленным в конце главы, особенно «Чистый код: создание, анализ и рефакторинг» и «Совершенный код: мастер-класс».

7.1.2. Составляющие удобочитаемости

Факторы, влияющие на удобочитаемость, можно разделить на две категории:

- *Структурные* — способные повлиять на выполнение программы: например, ее архитектура, выбор API, выбор управляющих команд и т. д. Эти факторы можно дополнительно разделить на три уровня:
  - *Архитектурный уровень* — факторы, в которых задействовано несколько классов.

  - *Уровень класса* — факторы, относящиеся к одному классу, но выходящие за границы одного метода.
  - *Уровень метода* — факторы, относящиеся к одному методу.
- *Внешние* — факторы, не влияющие на выполнение: комментарии, пропуски, выбор имен переменных и т. д.

В следующих разделах этой главы я кратко опишу основные рекомендации, относящиеся к каждой категории, а затем покажу, как применить их в примере с резервуарами.

7.2. СТРУКТУРНЫЕ ФАКТОРЫ УДОБОЧИТАЕМОСТИ

Факторы архитектурного уровня относятся к высокоуровневой структуре программы: способу ее разбиения на классы и отношениям, существующим между ними. Как правило, простая для понимания архитектура должна состоять из небольших классов с четко определенными зонами ответственности (*сильное сцепление*), которые объединяются незамысловатой сетью зависимостей (*слабая связанность*). Другой способ улучшения удобочитаемости заключается в использовании стандартных паттернов проектирования там, где это возможно. Так как паттерны знакомы большинству разработчиков, они вызывают ощущение осведомленности и передают читателю дополнительную контекстную информацию.

Каждая из этих кратких рекомендаций сопровождается множеством комментариев и предупреждений. Я не стану подробно рассматривать архитектурные факторы, но вы найдете дополнительную информацию в разделе «Дополнительная литература» в конце этой главы. В табл. 7.1 приведена сводка важнейших структурных факторов и соответствующих практических приемов.

Факторы уровня класса относятся к API конкретного класса и его структуре, отраженной в методах. Например, золотое правило гласит, что длинные методы труднее понять. В какой-то момент (наверное, свыше 200 строк) вы забываете о том, что происходило в начале метода, и начинаете переходить туда-сюда в редакторе, пытаясь держать в голове то, что не помещается на одном экране. Я включил этот принцип в число факторов уровня класса, потому, что, несмотря на то что проблема кроется в одном методе, ее *решение* влияет сразу на несколько методов: длинный метод сокращается разбиением на несколько более коротких методов, а это рекомендуется делать по правилу рефакторинга «Выделение метода», которое будет представлено позднее в этой главе.

А теперь займемся некоторыми *факторами уровня методов*, влияющими на удобочитаемость. К их числу принадлежат выбор управляющих конструкций, способ написания выражений и использование локальных переменных.

Таблица 7.1. Структурные факторы кода, влияющие на удобочитаемость

Структурная ясность		
Уровень	**Факторы**	**Возможности улучшения**
Архитектурный уровень	Зоны ответственности классов Отношения между классами	Сокращение связанности
		Повышение сцепления
		Архитектурные паттерны (модель — представление — контроллер и др.)
		Паттерны проектирования
		Рефакторинг (выделение классов и т. д.)
Уровень класса	Управление логикой выполнения	Использование самой конкретной разновидности циклов
	Выражения	Наглядное представление порядка вычисления
	Локальные переменные	Разбиение сложных выражений
	Длина методов	Рефакторинг (выделение методов и т. д.)

7.2.1. Управляющие команды

Интересной, хотя и узкой проблемой удобочитаемости является выбор наиболее подходящей конструкции цикла для заданного сценария. Java поддерживает четыре разновидности циклов: стандартный цикл `for`, `while`, `do-while` и расширенный цикл `for`. Очевидно, первые три конструкции эквивалентны в том смысле, что любую из них можно с минимальными усилиями преобразовать в любую другую. Например, цикл с выходной проверкой

```
do {
   тело
} while (условие);
```

преобразуется в следующий цикл с входной проверкой:

```
while (true) {
   тело
   if (!условие) break;
}
```

Какой из двух фрагментов лучше читается? Уверен, вы согласитесь, что первый вариант понятнее. Второй — всего лишь некрасивый трюк, сбивающий с толку

читателя, который остро ощутит, что у задачи должно быть *более естественное* решение. Занимаясь оптимизацией удобочитаемости, вы должны по возможности избегать этого ощущения, чтобы код читался гладко и без запинки. В этом и заключается смысл *четкого выражения намерений*.

Если вы должны реализовать цикл, условие которого будет проверяться после каждой итерации, как в цикле `do-while`, в вашем распоряжении три варианта. Сравним их по выразительности:

- Цикл `while` напоминает цикл `for` с удаленными блоками инициализации и обновления. Если вашему циклу необходимы эти возможности и они компактны в пределах разумного, используйте `for` — читателю будет проще распознать роль каждого компонента. Например, знакомая конструкция

  ```
  for (int i=0; i<n; i++) {
     ...
  }
  ```

  читается лучше эквивалентной конструкции

  ```
  int i=0;
  while (i<n) {
     ...
     i++;
  }
  ```

- Расширенный цикл `for` является специализированной формой стандартного цикла `for`, потому что он работает только с массивами и объектами, реализующими интерфейс `Iterable`. Более того, он не предоставляет телу цикла индекс или объект-итератор.

Чтобы выбрать конструкцию цикла, примените общее правило, известное как *принцип минимальных привилегий*, и подберите *наиболее конкретную* команду, подходящую для ваших целей. Цикл перебирает массив или коллекцию, реализующую `Iterable`? Используйте расширенный цикл `for`.

Кроме выигрыша в удобочитаемости, вы получите итерацию, которая гарантированно не выйдет за границы. Ваш цикл содержит компактную фазу инициализации и столь же компактную фазу обновления? Используйте стандартный цикл `for`. В противном случае — `while`.

Раз уж речь зашла о циклах, начиная с Java 8, у вас появилась возможность использования библиотеки потоков данных для создания конструкций циклов в функциональном стиле. Например, следующий фрагмент выводит каждый объект множества:

```
Set<T> set = ...
set.stream().forEach(obj -> System.out.println(obj));
```

Можно ли сказать, что он читается лучше традиционного расширенного цикла `for`?

```
for (T obj: set) {
   System.out.println(obj);
}
```

Пожалуй, нет. Хорошее практическое правило — используйте API в функциональном стиле, если помимо самого перебора у вас есть *другая* цель (например, фильтрация или некое преобразование содержимого потока). Особенно веская причина для использования потоков данных — разбиение задания на несколько потоков. В этом случае библиотека возьмет на себя много мелких неприятных подробностей.

СОВЕТ ПО УДОБОЧИТАЕМОСТИ

Выбирайте наиболее естественную и специфичную для конкретной задачи разновидность цикла.

НЕОЖИДАННЫЙ ВОПРОС 2

Какую разновидность цикла вы бы использовали для инициализации массива из n целых чисел значениями из интервала от 0 до $n - 1$?

7.2.2. Выражения и локальные переменные

Выражения являются основными структурными элементами любого языка программирования. Они могут быть исключительно сложными. Для улучшения удобочитаемости можно разбивать сложные выражения на более простые подвыражения и присваивать их значения локальным переменным, добавленным для этой цели. Естественно, новым локальным переменным следует присваивать содержательные имена, поясняющие смысл соответствующего подвыражения. (Вскоре я вернусь к теме имен переменных.)

В версии `Reference` уже применяется стратегия улучшения удобочитаемости, когда метод `connectTo` вычисляет объем воды, который должен находиться в каждом резервуаре при создании нового подключения. Самый короткий способ описания этих вычислений выглядит примерно так:

```
public void connectTo(Container other) {
   ...
   double newAmount = (amount * group.size() +
                       other.amount * other.group.size()) /
                      (group.size() + other.group.size());
   ...
}
```

Как видите, даже при разбивке на три строки с выравниванием выражение получается слишком длинным и невразумительным. Читателю придется потрудиться или хотя бы задержаться, чтобы найти парные круглые скобки, потому что закрывающая скобка расположена слишком далеко от открывающей. Громоздкие повторения `group.size()` и `other.group.size()` тоже делу не помогают.

Из-за этого в `Reference` вводится до четырех дополнительных переменных просто ради удобочитаемости:

```
public void connectTo(Container other) {
   ...
   int size1 = group.size(),
       size2 = other.group.size();
   double tot1 = amount * size1,
          tot2 = other.amount * size2,
                 newAmount = (tot1 + tot2) / (size1 + size2);
   ...
}
```

Не беспокойтесь о том, что вторая, более удобочитаемая версия будет менее эффективной. В общем случае затраты на использование двух лишних локальных переменных ничтожно малы, особенно если сравнить их с выигрышем по удобочитаемости. В этом конкретном случае лишние переменные экономят два вызова методов и даже могут ускорить выполнение[1].

Мартин Фаулер формализовал эту идею в одном из сформулированных им *правил рефакторинга* (за дополнительной информацией обращайтесь к разделу «Дополнительная литература»). Как и в случае с паттернами проектирования, каждому правилу рефакторинга для удобства присваивается стандартное имя. Данное правило называется *Выделением переменной*.

СОВЕТ ПО УДОБОЧИТАЕМОСТИ

Правило рефакторинга «Выделение переменной»: замените подвыражение новой локальной переменной с содержательным именем.

7.3. ВНЕШНИЕ ФАКТОРЫ УДОБОЧИТАЕМОСТИ

Для улучшения удобочитаемости можно воспользоваться тремя внешними факторами: комментариями, именами и пропусками. В табл. 7.2 приведена сводка соответствующих им практик, более подробно описанных в следующих подразделах.

---

[1] Более того, байт-код удобочитаемой версии на 3 байта короче старой версии.

Таблица 7.2. Сводка внешних факторов кода, влияющих на удобочитаемость

Внешние факторы удобочитаемости	
Факторы	**Пути усовершенствования**
Комментарии	Подробные комментарии в документации, лаконичные комментарии в реализации
Имена	Содержательные имена
Пропуски	Использование пропусков как знаков препинания
Отступы	Последовательное применение отступов

7.3.1. Комментарии

Сам по себе код не может качественно задокументировать себя. Иногда приходится использовать естественный язык, чтобы лучше раскрыть суть кода или представить некоторую функциональность в более глобальном контексте. Обычно различают два вида комментариев:

- *Документирующие комментарии* описывают контракт метода или целого класса. Они предназначены для объяснения правил класса его потенциальным клиентам. Их можно рассматривать как *публичные*, то есть общедоступные, комментарии. Обычно они извлекаются из класса и преобразуются в удобную форму (например, HTML). В Java для этой цели используется программа Javadoc, описанная далее в этой главе.
- *Комментарии реализации* предоставляют информацию о внутреннем устройстве класса. Они могут пояснять роль поля или предназначение фрагмента кода из хитроумного алгоритма. Их можно рассматривать как *приватные* комментарии.

Вопрос о том, когда и с какой частотой следует добавлять комментарии, открыт для обсуждения. Для современной разработки характерно обилие документирующих комментариев при относительно малом количестве комментариев реализации.

Это объясняется следующими рассуждениями: API появляется до реализации, обычно отличается большей стабильностью, и это единственная часть класса, которая должна быть известна клиентам для правильного его использования. А значит, для жизнеспособности системы в целом очень важно, чтобы зоны ответственности и контракты каждого класса и метода были полностью понятны клиентам. Как было показано в главе 5, выразить контракты в коде можно только до определенной степени, а какой именно, зависит от выбранного языка программирования. В остальном преобладают комментарии на естественном языке и другие формы документирования.

НЕОЖИДАННЫЙ ВОПРОС 3

К какой разновидности относится комментарий, описывающий поведение приватного метода: документирующий комментарий или комментарий реализации?

С другой стороны, тела методов часто изменяются и остаются скрытыми от клиентов. Из-за частого изменения вам придется с такой же частотой обновлять все содержащиеся в них комментарии, а программисты часто забывают обновлять комментарии (или выполнять другие действия, не имеющие прямых последствий для поведения программы). Вероятно, вы уже бывали в такой ситуации: вам поручают обновление фрагмента кода для исправления ошибки или включения новой функциональности в очень сжатые сроки. Вы направляете все усилия на функциональность — на написание кода, который работает и проходит тесты. Если только в вашей компании не приняты серьезные формы инспекции кода, ваш код не столкнется ни с каким фильтром, контролирующим качество комментариев. По этой причине будет естественно игнорировать комментарии и сосредоточиться на активных строках кода.

Если пройдет слух, что *некоторые* комментарии в кодовой базе ненадежны из-за устаревания, то *все* комментарии немедленно станут обычным шумом, даже если многие из них на самом деле вполне хороши и актуальны.

СОВЕТ ПО УДОБОЧИТАЕМОСТИ

Сократите количество комментариев реализации в пользу документирующих комментариев и следите за тем, чтобы все комментарии оставались актуальными (в этом вам поможет рецензирование кода).

7.3.2. Выбор имен

Согласно известной цитате Фила Карлтона, в информатике есть только две сложные вещи: недействительность кэша и выбор имен. Проблемы кэширования были кратко представлены в главе 4, теперь пора заняться второй сложностью. Языки высокого уровня позволяют назначать произвольные имена элементам программы. В Java такими элементами являются пакеты, классы, методы и всевозможные переменные, включая поля. Язык устанавливает некоторые ограничения для имен (например, запрет пробелов), а практические соображения наводят на мысль, что имена должны быть относительно короткими.

Полагаю, вы уже знакомы с основными лексическими соглашениями Java (которые также применяются во многих других языках, включая C# и C++), основанными на так называемом *верблюжьем регистре*. Несколько общих рекомендаций относительно имен, которые используются в разных обстоятельствах:

- Имена должны быть содержательными, чтобы читатель, не знакомый с вашим кодом, мог по крайней мере в общих чертах представлять роль соответствующего элемента. Это не обязательно означает, что имена должны быть *длинными*. Например, в некоторых случаях допустимы даже односимвольные имена:
  - `i` — хорошее имя для индекса массива, поскольку это традиционный (и поэтому понятный) выбор.
  - По тем же причинам `x` — хорошее имя для горизонтальной координаты на координатной плоскости.
  - `a` и `b` — хорошие имена для двух параметров простого компаратора:

    ```
    Comparator<String> stringComparatorByLength =
       (a,b) -> Integer.compare(a.length(), b.length());
    ```

    В таком контексте читателю не понадобятся более содержательные имена, чтобы понять ваши намерения. (Обратите внимание на длинное имя самого компаратора.)
  - `T` — хорошее имя для параметра-типа (как в классе class `LinkedList<T>`) из-за общепринятых соглашений.
- Имена классов должны представляться существительными, а имена методов — глаголами.
- В именах не следует применять нестандартные сокращения.

СОВЕТ ПО УДОБОЧИТАЕМОСТИ

Используйте содержательные имена, избегайте сокращений и следуйте общепринятым соглашениям.

НЕОЖИДАННЫЙ ВОПРОС 4

Какое имя будет наиболее подходящим для поля, предназначенного для хранения ежемесячного оклада в классе, представляющем работника: `salary`, `s`, `monthlySalary` или `employeeMonthlySalary`?

7.3.3. Пропуски и отступы

Наконец, многие языки, включая Java, предоставляют достаточную свободу в отношении визуального оформления кода. Вы можете разбивать строки почти в любых местах, свободно вставлять пропуски вокруг символов и добавлять пустые строки по своему усмотрению. Эту свободу следует использовать

не для художественного самовыражения (для этого есть ASCII-арт), а для снижения когнитивной нагрузки на коллегу-программиста, который позднее будет читать код.

Правильная расстановка отступов абсолютно необходима, но я полагаю, что вы уже это знаете и применяете на практике. Помимо базовых отступов, можно использовать пропуски для выравнивания двух частей разбиваемой строки. Типичный пример — методы с множеством параметров, как в следующем методе экземпляра `String`:

```
public boolean  regionMatches(int toffset,
                              String other,
                              int ooffset,
                              int len)
```

Что касается пустых строк в коде, рассматривайте их как знаки препинания. Если метод можно сравнить с абзацем текста (как по длине, так и по внутренней связности), пустая строка становится аналогом точки. Не используйте точку там, где достаточно запятой. Используйте пустые строки для визуального разделения частей кода, которые являются концептуально разнородными, включая разделение разных методов или разных частей одного метода. Пример второго типа встречается в методе `connectTo` как в `Reference` (листинг 7.3), так и в `Readable` (листинг 7.4).

СОВЕТ ПО УДОБОЧИТАЕМОСТИ

Используйте пустую строку как точку, завершающую предложение в абзаце текста.

В следующем разделе мы разработаем `Readable` — версию класса, представляющего резервуар с водой, оптимизированную для удобочитаемости.

7.4. КОД [READABLE]

Рассмотрим реализацию `Reference` и внесем следующие изменения для улучшения ее удобочитаемости:

- Добавить в класс в целом (а также в его открытые методы) комментарии в стандартном формате, который легко преобразуется в документацию HTML. Этот шаг станет единственным изменением в методах `addWater` и `getAmount`, потому что тело этих методов достаточно тривиально и прямолинейно.
- Применить правила рефакторинга к телу `connectTo` для улучшения его структуры.

Но сначала стоит поближе познакомиться со стандартным форматом документирующих комментариев Java: Javadoc.

7.4.1. Документирование заголовка класса комментариями Javadoc

Javadoc — служебная программа Java, которая извлекает из исходных файлов специально составленные комментарии (с использованием тегов из табл. 7.3 и 7.4) и генерирует из них удобно отформатированную разметку HTML, снабжая документацию удобными средствами навигации. Именно Javadoc генерирует знакомую электронную документацию для Java API, а также фрагменты документации, которую многие популярные IDE предоставляют по запросу.

Комментарии, предназначенные для Javadoc, должны начинаться с `/**`. Поддерживаются многие теги HTML, включая следующие:

- `<p>` для начала нового абзаца;
- `<i>...</i>` для оформления текста *курсивом*;
- `<code>...</code>` для оформления фрагментов кода.

Таблица 7.3. Часто используемые теги Javadoc

Тег	Описание
`@author`	Автор класса (обязательно)
`@version`	Версия класса (обязательно)
`@return`	Описание возвращаемого значения метода
`@param`	Описание параметра метода
`@throws` или `@exception`	Описание условий для выдачи заданного исключения
`{@link...}`	Ссылка на другой элемент программы (класс, метод и т. д.)
`{@code...}`	Оформление фрагмента кода

Таблица 7.4. Сводка основных Javadoc-совместимых тегов HTML

Тег	Описание
`<code>...</code>`	Оформление фрагмента кода
`<p>`	Начало нового абзаца
`<i>...</i>`	Курсив
`<b>...</b>`	Полужирный шрифт

Кроме того, Javadoc распознает различные дополнительные теги, начинающиеся со знака `@` (не путайте с аннотациями Java!). Например, в комментарии, описывающем целый класс, вставляются теги `@author` и `@version`, понятные без комментариев. Оба тега считаются обязательными в описаниях класса, но Javadoc не будет жаловаться при их отсутствии.

> **ДОКУМЕНТИРУЮЩИЕ КОММЕНТАРИИ В C#**
>
> В C# документирующие комментарии должны начинаться с последовательности /// (тройной слеш) и могут включать разнообразные теги XML. Компилятор извлекает эти комментарии из исходных файлов и сохраняет их в отдельном файле XML. Затем Visual Studio использует информацию из этого файла для расширения функциональности контекстной справки, а программист может воспользоваться внешней программой для размещения комментариев в удобочитаемом формате, например HTML. Популярное решение с открытым кодом — программа DocFX — наряду с C# поддерживает ряд других языков, включая Java.

Вместо того чтобы представлять каждый тег Javadoc по отдельности, применим их на практике для получения версии `Container` с улучшенной удобочитаемостью. В самом начале исходного файла `Container` добавьте вводный комментарий с общим описанием класса, приведенным в листинге 7.1. В таких комментариях также уместно вводить терминологию, специфичную для класса, — например, слово «группа» для обозначения набора резервуаров, соединенных с текущим.

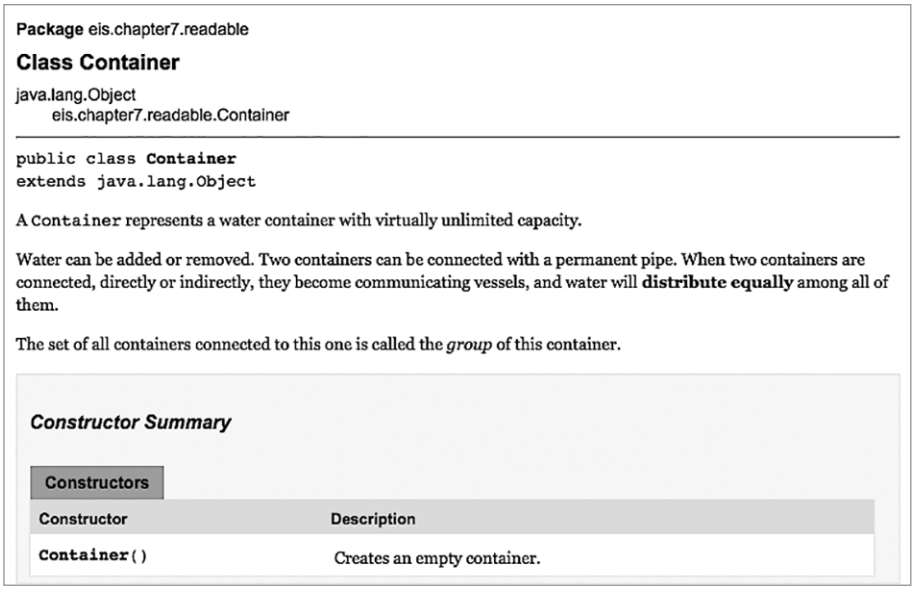
Тег HTML `<code>` и тег Javadoc `{@code...}` могут использоваться для оформления фрагментов кода. В табл. 7.3 и 7.4 перечислены теги Javadoc и HTML, которые чаще всего используются в комментариях.

Листинг 7.1. Readable: заголовок класса

```
/** ❶ Начало комментария Javadoc
 * A <code>Container</code> represents a water container
 * with virtually unlimited capacity.
 * <p> ❷ Поддерживается большинство тегов HTML
 * Water can be added or removed.
 * Two containers can be connected with a permanent pipe.
 * When two containers are connected, directly or indirectly,
 * they become communicating vessels, and water will distribute
 * equally among all of them.
 * <p>
 * The set of all containers connected to this one is called the
 * <i>group</i> of this container.
 *
 * @author Marco Faella ❸ Тег Javadoc
 * @version 1.0 ❹ Еще один тег Javadoc
 */
```

```
public class Container {
   private Set<Container> group;
   private double amount;
```

На рис. 7.1 изображена страница HTML, сгенерированная Javadoc для комментария в листинге 7.1.


Package eis.chapter7.readable

Class Container

java.lang.Object
eis.chapter7.readable.Container

public class Container
extends java.lang.Object

A Container represents a water container with virtually unlimited capacity.

Water can be added or removed. Two containers can be connected with a permanent pipe. When two containers are connected, directly or indirectly, they become communicating vessels, and water will distribute equally among all of them.

The set of all containers connected to this one is called the group of this container.

Constructor Summary

Constructors

Constructor	Description
Container()	Creates an empty container.

Рис. 7.1. Документация в формате HTML, сгенерированная Javadoc, с описанием класса и списком конструкторов

Конструктор и метод `getAmount` настолько просты, что в улучшении удобочитаемости они не нуждаются — достаточно коротких документирующих комментариев. Используйте тег `@return` для описания возвращаемого значения метода.

Листинг 7.2. Readable: конструктор и getAmount

```
/** Создает пустой контейнер. */
public Container() {
   group = new HashSet<Container>();
   group.add(this);
}

/** Возвращает количество воды, которое в данный момент находится в этом
контейнере.
  *
  * @return количество воды, которое в данный момент находится в этом контейнере
```

```
  */
public double getAmount() {
   return amount;
}
```

Избыточность в комментарии `getAmount` оправдана особенностями вывода информации Javadoc. Каждый метод представлен в HTML-странице класса дважды: сначала идет краткая сводка всех методов (рис. 7.2), а затем более подробный раздел с описанием каждого метода (рис. 7.3). Первое предложение в комментарии включается в сводку всех методов, и его нельзя опустить. Строка `@return` включается только в подробное описание метода.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	`addWater(double amount)`	Adds water to this container.
void	`connectTo(Container other)`	Connects this container with another.
double	`getAmount()`	Returns the amount of water currently held in this container.
double	`groupAmount()`	Returns the total amount of water in the group of this container.
int	`groupSize()`	Returns the number of containers in the group of this container.
boolean	`isConnectedTo(Container other)`	Checks whether this container is connected to another one.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Рис. 7.2. Документация в формате HTML, сгенерированная Javadoc: сводка открытых методов класса Readable

Method Detail

getAmount

`public double getAmount()`

Returns the amount of water currently held in this container.

Returns:

the amount of water currently held in this container

Рис. 7.3. Документация в формате HTML, сгенерированная Javadoc: подробное описание метода getAmount

7.4.2. Чистка метода connectTo

Теперь обратимся к методу `connectTo`, который может улучшить удобочитаемость с помощью рефакторинга. Для начала вспомним реализацию метода из версии `Reference`, которую я повторю здесь для удобства:

Листинг 7.3. Reference: метод connectTo

```
public void connectTo(Container other) {

   // Если резервуары уже соединены, ничего не делать
   if (group==other.group) return;

   int size1 = group.size(),
       size2 = other.group.size();
   double tot1 = amount * size1,
          tot2 = other.amount * size2,
          newAmount = (tot1 + tot2) / (size1 + size2);

   // Объединить две группы
   group.addAll(other.group);
   // Обновить группу резервуаров, соединенных с other
   ❶ Эти комментарии можно заменить вспомогательным методом с содержательным именем

   for (Container c: other.group) { c.group = group; }
   // Обновить amount для всех вновь присоединенных резервуаров
   for (Container c: group) { c.amount = newAmount; }
}
```

Я уже указывал на один из дефектов эталонной реализации из главы 3: избыток внутренних комментариев, объясняющих каждую строку. Добавление таких комментариев многим программистам кажется полезным. Но это не самый эффективный способ достижения благородной цели, лучше воспользоваться выделением метода.

СОВЕТ ПО УДОБОЧИТАЕМОСТИ

Правило рефакторинга «Выделение метода»: переместите связный блок кода в новый метод с содержательным именем.

Метод `connectTo` открывает широкие возможности для выделения метода. Его можно применить пять раз и получить столько же новых вспомогательных методов, а также новую версию `connectTo`, которая читается намного лучше (листинг 7.4).

Листинг 7.4. Readable: метод connectTo

```
/** Соединяет этот резервуар с другим.
  *
  * @param other Резервуар, который будет соединен с текущим.
```

```
  */
public void connectTo(Container other) {
   if (this.isConnectedTo(other))
      return;

   double newAmount = (groupAmount() + other.groupAmount()) /
                      (groupSize() + other.groupSize());
   mergeGroupWith(other.group);
   setAllAmountsTo(newAmount);
}
```

Тег Javadoc `@param` документирует параметр метода. За ним следует имя параметра и его краткое описание. По сравнению с `Reference` этот метод намного короче и лучше читается. Если я вас еще не убедил, попробуйте прочитать тело метода вслух; оно воспринимается почти как короткий абзац текста.

Для достижения этого результата были добавлены пять вспомогательных методов с разумно выбранными именами. *Длинные методы* — один из признаков кода «с душком», выявленных Фаулером, а *выделение метода* — прием рефакторинга, направленный на его устранение. В терминологии Agile новая версия `connectTo` в листинге 7.4 располагается на расстоянии пяти выделений методов от старой версии из `Reference`.

Если добавление комментария только поясняет смысл кода, выделение метода и объясняет, и *скрывает* код, перемещая его в отдельный метод. Таким образом абстракция в исходном методе поддерживается на более высоком уровне, избегая неуклюжих переключений между высокоуровневыми объяснениями и низкоуровневыми реализациями в листинге 7.3.

Замена временной переменной запросом — еще один прием рефакторинга, который можно использовать с `connectTo`.

СОВЕТ ПО УДОБОЧИТАЕМОСТИ

Правило рефакторинга «Замена временной переменной запросом»: локальная переменная заменяется вызовом нового метода, который вычисляет ее значение.

Этот прием можно применить к локальной переменной `newAmount`, которая инициализируется только один раз, а затем используется как аргумент `setAllAmountsTo`. В простейшем варианте применения переменная `newAmount` удаляется, а две последние строки `connectTo` заменяются следующими:

```
mergeGroupWith(other.group);
setAllAmountsTo(amountAfterMerge(other));
```

Здесь `amountAfterMerge` — новый метод, отвечающий за вычисление правильного объема воды в каждом резервуаре после объединения. Однако если немного

поразмыслить, становится ясно, что методу `amountAfterMerge` придется изрядно потрудиться для выполнения своей задачи, потому что группы *уже были объединены* при вызове метода. В частности, множество, на которое уже указывает `this.group`, содержит все элементы из `other.group`.

Хорошим компромиссом может стать инкапсуляция выражения нового объема в новом методе, но с сохранением локальной переменной, чтобы мы могли вычислить новый объем *до* объединения групп:

```
final double newAmount = amountAfterMerge(other);
mergeGroupWith(other.group);
setAllAmountsTo(newAmount);
```

В целом я бы не рекомендовал применять этот рефакторинг, потому что выражение, присвоенное `newAmount` в листинге 7.4, нормально читается и его незачем скрывать в отдельном методе. Правило «*Замена временной переменной запросом*» тем полезнее, чем сложнее заменяемое выражение и чем чаще оно встречается в классе.

А теперь рассмотрим пять новых методов, обеспечивающих удобочитаемость версии `connectTo`. Два из них лучше объявить приватными, потому что они могут оставить объект в противоречивом состоянии, а значит, не должны вызываться за пределами класса. Речь идет о методах `mergeGroupWith` и `setAllAmountsTo`.

Метод `mergeGroupWith` объединяет две группы резервуаров без обновления объемов воды. Если кто-то вызовет его отдельно, скорее всего, это приведет к тому, что в некоторых (или во всех) резервуарах будет храниться неправильный объем воды. Этот метод имеет смысл только в точном контексте его использования: в конце `connectTo`, за которым следует вызов `setAllAmountsTo`. Вопрос о том, должен ли он быть отдельным методом, остается спорным. С одной стороны, наличие отдельного метода позволяет документировать его намерения именем, а не комментарием, как в `Reference`. С другой стороны, отдельный метод может быть вызван в неправильном контексте. Так как в этой главе код оптимизируется для ясности, мы выберем вариант с отдельным методом и то же самое сделаем с `setAllAmountsTo`.

Код этих двух методов приведен в листинге 7.5.

Листинг 7.5. Readable: два новых приватных метода для connectTo

```
private void mergeGroupWith(Set<Container> otherGroup) {
   group.addAll(otherGroup);
   for (Container x: otherGroup) {
      x.group = group;
   }
}
```

```
private void setAllAmountsTo(double amount) {
   for (Container x: group) {
      x.amount = amount;
   }
}
```

Считается, что приватные методы не заслуживают комментариев Javadoc. Они используются только внутри класса, и мало кому когда-нибудь потребуется понимать их во всех подробностях. Таким образом, потенциальный выигрыш от комментариев не оправдает своих затрат.

Затраты на комментарий не ограничиваются временем на его написание. Как и любая другая строка исходного кода, комментарий нуждается в сопровождении и может устареть со временем, то есть потерять синхронизацию с кодом, который он должен прояснять. Помните: неактуальный комментарий хуже, чем отсутствие комментария!

Замена комментариев содержательными именами не исключает подобного риска. Без должной дисциплины и необходимых доработок в вашей программе могут появиться устаревшие имена, которые ничем не лучше устаревших комментариев.

Три остальных вспомогательных метода — безобидные блоки функциональности только для чтения, которые можно объявить открытыми. Это не означает, что к их открытости можно относиться легкомысленно. Будущие затраты на сопровождение при добавлении в класс любого открытого метода будут намного выше затрат на добавление того же метода с приватной видимостью. Дополнительные затраты открытых методов включают:

- документацию, описывающую контракт;
- проверки предусловия, которые бы устояли при взаимодействиях с некорректно работающими клиентами;
- набор тестов, подтверждающих правильность работы метода.

В нашем конкретном случае затраты ограниченны, потому что три рассматриваемых метода представляют простые блоки функциональности, доступные только для чтения, которые не имеют предусловий, заслуживающих упоминания[1]. Кроме того, эти три метода предоставляют клиентам информацию, которая не может быть получена иным образом. Как следствие, они значительно улучшают *тестируемость* класса, как объяснялось в главе 5.

---

[1] Точнее, `isConnectedTo` требует, чтобы аргумент был отличен от `null`. Это предусловие настолько тривиально, что его не нужно документировать или активно проверять. Его нарушение ожидаемо приведет к выдаче `NullPointerException`.

Листинг 7.6. Readable: три новых открытых метода, поддерживающих connectTo

```
/** Проверяет, соединен ли резервуар с другим.
 *
 * @param other резервуар, соединение которого с текущим будет проверяться
 * @return <code>true</code> признак соединения резервуара
 *                           с <code>other</code>
 */
public boolean isConnectedTo(Container other) {
   return group == other.group;
}

/** Возвращает количество резервуаров в группе этого резервуара.
 *
 * @return размер группы
 */
public int groupSize() {
   return group.size();
}

/** Возвращает общий объем воды в группе этого резервуара.
 *
 * @return объем воды в группе
 */
public double groupAmount() {
   return amount * group.size();
}
```

Кстати, метод `isConnectedTo` также улучшает тестируемость класса: он позволяет напрямую наблюдать то, что во всех предыдущих реализациях можно было только предполагать.

Все шесть методов, образующих функциональность `connectTo`, очень короткие; самый длинный из них, `connectTo`, занимает всего шесть строк. Краткость — один из канонов чистого кода.

7.4.3. Чистка addWater

Наконец, остается метод `addWater`. Его тело не изменилось по сравнению с `Reference`. Мы просто улучшим его документацию, чтобы она хорошо отражала его контракт с использованием синтаксиса Javadoc.

Листинг 7.7. Readable: метод addWater

```
/** Добавляет воду в этот контейнер.
 * Отрицательное значение <code> amount </code> указывает на удаление воды.
 * В таком случае в группе должно быть достаточно воды, чтобы удовлетворить
запрос.
```

```
 *
 * @param amount количество воды, которое нужно добавить
 */
public void addWater(double amount) {
   double amountPerContainer = amount / group.size();
   for (Container c: group) {
      c.amount += amountPerContainer;
   }
}
```

Сравните Javadoc-описание метода с контрактом `addWater` из главы 5:

- *Предусловие:* если аргумент отрицателен, то в группе достаточно воды.
- *Постусловие:* вода распределяется в равных пропорциях между всеми резервуарами в группе.
- *Штраф:* `IllegalArgumentException`.

Обратите внимание: комментарии в листинге не упоминают реакцию на нарушение предусловия клиентом, пытающимся удалить больше воды, чем есть в группе. Дело в том, что эта реализация (как и `Reference`) не проверяет условие и позволяет резервуарам хранить отрицательное количество воды. На рис. 7.2 показана страница HTML, сгенерированная Javadoc на основе этих комментариев.

А если бы реализация проверяла это условие и реализовала штраф, установленный контрактом, выдачей исключения `IllegalArgumentException`? И руководство по стилю Javadoc, и книга «Java. Эффективное программирование» рекомендуют документировать непроверяемые исключения при помощи тегов `@throws` или `@exception` (которые эквивалентны[1]). Для этого подойдет строка следующего вида, добавленная в комментарий метода:

```
@throws IllegalArgumentException
   при попытке добавить больше воды, чем реально присутствует.
```

Краткое обращение к официальной документации Java API показывает, что это действительно является стандартной практикой. Например, документация метода `get(int index)` из `ArrayList`, возвращающего элемент в позиции `index` в списке, сообщает, что метод выдает непроверяемое исключение `IndexOutOfBoundsException` при выходе индекса за пределы допустимого диапазона.

[1] Пункт 74 в «Java. Эффективное программирование».

НЕОЖИДАННЫЙ ВОПРОС 5

Допустим, открытый метод может выдавать `AssertionError` при обнаружении нарушения инварианта класса. Как бы вы документировали это обстоятельство в комментариях Javadoc этого метода?

7.5. НАПОСЛЕДОК ОБ УДОБОЧИТАЕМОСТИ

Эта глава несколько отличается от предыдущих тем, что ее рекомендации можно немедленно применить во многих, если не во всех, практических сценариях. И хотя в главе 1 я говорил, что удобочитаемость может противоречить другим целям (таким, как эффективность по затратам памяти или времени), в большинстве подобных конфликтов удобочитаемость должна побеждать. Ясность кода — огромное преимущество для кода, который неизбежно развивается из-за обнаружения ошибок или запросов на добавление новой функциональности.

Тем не менее ясность кода не следует путать с алгоритмической простотой. Я не предлагаю отказываться от эффективного алгоритма в пользу тривиального ради удобочитаемости, скорее следует выбрать лучший алгоритм для задачи, а затем постараться закодировать его самым чистым из возможных способов. Ясность вступает в конфликт с производительностью, а не с добросовестной разработкой.

Полноты ради я должен упомянуть пару сценариев, в которых удобочитаемость становится роскошью или даже излишеством. К первой категории относятся состязания для программистов с жесткими ограничениями по времени. В таких ситуациях участник должен быстро написать одноразовый код, который просто работает. Любая задержка оборачивается лишними затратами, и о стиле лучше забыть.

Другой особый случай: компания не желает, чтобы ее исходный код анализировался кем-то другим, включая законных пользователей их продукта. Скрывая или искажая свой исходный код, такие компании надеются скрыть свои алгоритмы или данные. В таких случаях вроде бы естественно забыть об удобочитаемости кода и выбрать самые запутанные строки кода, которые справятся со своей задачей. Существуют даже специальные программы (*обфускаторы*), предназначенные для преобразования программы в другую программу, которая функционально эквивалентна первой, но кажется в высшей степени запутанной для читателя. Такие преобразования возможны для всех языков программирования[1], от машинного кода до байт-кода или исходного кода Java. Любая

[1] Некоторые языки программирования проектировались так, чтобы написанные на них программы не читались и не нуждались в усложнении. А вы знаете какие-нибудь примеры такого рода? Подсказка: Brain****

поисковая система по запросу «Java obfuscator» выдает обширную подборку программ для решения этой задачи — как бесплатных, так и коммерческих. Даже самая скрытная компания только выиграет от ведения внутренней разработки с чистым, понятным кодом, который затем намеренно будет запутан перед выпуском.

7.6. А ТЕПЕРЬ СОВСЕМ ДРУГОЕ

В этом разделе рекомендации по написанию удобочитаемого кода будут применены в совершенно другом примере. В нем используется метод, который получает двумерный массив значений `double` и... что-то с ним делает. Я намеренно записал тело метода неаккуратно: он не запутан, но и не особенно понятен. Для тренировки попробуйте понять, что он делает, прежде чем читать дальше.

```
public static void f(double[][] a) {
   int i = 0, j = 0;
   while (i<a.length) {
      if (a[i].length != a.length)
         throw new IllegalArgumentException();
      i++;
   }
   i = 0;
   while (i<a.length) {
      j = 0;
      while (j<i) {
         double temp = a[i][j];
         a[i][j] = a[j][i];
         a[j][i] = temp;
         j++;
      }
      i++;
   }
}
```

Вы поморщились? Все эти циклы `while` и бессмысленные имена переменных создают лишнюю нагрузку на мозг. Представьте, что вся программа будет написана в том же стиле!

Как вы, возможно, догадались, загадочный метод *транспонирует* квадратную матрицу — эта стандартная операция меняет местами строки со столбцами. Первый цикл `while` убеждается в том, что переданная матрица является квадратной, то есть число строк равно числу столбцов. Так как матрицы Java могут быть *нерегулярными*, также необходимо проверить, что длина каждой строки равна количеству строк. Ниже приведена версия того же метода с аннотациями, которые помогают идентифицировать разные части кода:

```
public static void f(double[][] a) {
   int i = 0, j = 0;
   while (i<a.length) { ❶ Для каждой строки
      if (a[i].length != a.length) ❷ Если длина строки неправильная...
         throw new IllegalArgumentException();
      i++;
   }
   i = 0;
   while (i<a.length) { ❸ Для каждой строки
      j = 0;
      while (j<i) { ❹ Для каждого столбца меньше i
         double temp = a[i][j]; ❺ Поменять местами a[i][j] и a[j][i]
         a[i][j] = a[j][i];
         a[j][i] = temp;
         j++;
      }
      i++;
   }
}
```

Попробуем улучшить удобочитаемость этого метода по рекомендациям этой главы. Во-первых, исходная проверка квадратной формы идеально подходит для выделения метода: это связная операция с четко определенным контрактом. После выделения в отдельный метод он также может пригодиться в других контекстах. Из-за этого я объявил его открытым и дополнил полным комментарием Javadoc.

Так как проверка квадратной формы не изменяет матрицу, в главном цикле можно использовать расширенную конструкцию `for`:

```
/** Проверяет, имеет ли матрица квадратную форму
 *
 * @param matrix матрица
 * @return {@code true} если заданная матрица является квадратной
 */
public static boolean isSquare(double[][] matrix) {
   for (double[] row: matrix) {
      if (row.length != matrix.length) {
         return false;
      }
   }
   return true;
}
```

Затем метод `transpose` вызывает `isSquare`, а после выполняет свою работу в двух тривиальных циклах `for`. Расширенный цикл `for` здесь бесполезен, потому что для перестановки нужны индексы строки и столбца.

Заодно мы улучшим имена переменных и самого метода и сделаем их более содержательными. Индексам строки и столбца можно оставить имена `i` и `j`, стандартные для индексов массивов.

```
/** Транспонирует квадратную матрицу
  *
  * @param matrix матрица
  * @throws IllegalArgumentException если заданная матрица не является квадратной
  */
public static void transpose(double[][] matrix) {
   if (!isSquare(matrix)) {
      throw new IllegalArgumentException(
         "Can't transpose a nonsquare matrix.");
   }
   for (int i=0; i<matrix.length; i++) { ❶ Для каждой строки
      for (int j=0; j<i; j++) { ❷ Для каждого столбца меньше i
         double temp = matrix[i][j]; ❸ Поменять местами a[i][j] и a[j][i]
         matrix[i][j] = matrix[j][i];
         matrix[j][i] = temp;
      }
   }
}
```

7.7. РЕАЛЬНЫЕ СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

Мы рассмотрели и применили на практике некоторые очень важные принципы для улучшения удобочитаемости кода. Ниже представлены некоторые сценарии, которые помогут вам понять практическую важность удобочитаемости.

- Представьте, что вам — одному из соучредителей небольшого стартапа — удалось выиграть тендер на разработку программного обеспечения для компании, работающей в газовой промышленности, а целью проекта является реализация регламентирующего законодательства. Все выглядит хорошо: вам доверен престижный проект, а поскольку законы меняются нечасто, после завершения продукта вы сможете долго пользоваться плодами вашего труда. Вы и ваши коллеги принимаете стратегическое решение: как можно быстрее выдать готовый вариант, чтобы произвести впечатление на клиента. Для этого вы сэкономите на такой роскоши, как удобочитаемость, документация, модульные тесты и т. д. Через пару лет ваша компания выросла, половина исходной команды покинула ее, а контракт с газовой компанией остался. Однажды происходит невероятное: законодательство изменяется, и вам предлагают изменить свой продукт в соответствии с новыми требованиями. Вы на собственном опыте узнаете, что разобраться в том, как работает существующий код, бывает сложнее, чем создать новый. Удобочитаемость кода настолько важна, что она становится определяющим фактором в работе компаний-разработчиков (http://mng.bz/pyKE).
- Вы — одаренный и полный энтузиазма разработчик, желающий внести свой вклад в многообразие открытого кода. У вас есть идея (вы полагаете, отличная), и вы хотите поделиться своим кодом на github в надежде на то, что он

привлечет других участников и со временем будет использоваться в реальных проектах. Вы понимаете, что удобочитаемость сыграет ключевую роль для привлечения других разработчиков, которые изначально не знакомы с вашей кодовой базой и, скорее всего, не захотят задавать вопросы о ней.

Следующие примеры показывают, насколько серьезно в мире программирования относятся к удобочитаемости.

- Вы должны усердно трудиться над тем, чтобы сделать свой код удобочитаемым, независимо от того, каким языком программирования вы пользуетесь. Тем не менее в некоторых языках программирования стремление к удобочитаемости кода заложено еще на стадии проектирования. Python входит в число самых популярных языков, и, возможно, одной из причин его популярности является характерная удобочитаемость. Более того, удобочитаемость была сочтена настолько важной, что создатель языка написал знаменитый документ PEP8 (Python enhancement proposal) — руководство по стилю программирования, основной целью которого стало (сюрприз!) улучшение удобочитаемости.
- Я хочу еще немного поговорить о Python. (Да, в книге центральное место занимает Java, но описанные принципы универсальны.) Python является языком с динамической типизацией и не требует от вас задавать типы параметров функции и возвращаемых значений. Однако в PEP 484 были введены необязательные рекомендации для объявления типов в Python 3.5, которые абсолютно не влияют на производительность и не поддерживают автоматическое определение типов во время выполнения. Их целью является улучшение удобочитаемости и поддержка дополнительных статических проверок типов, которые также улучшают надежность программы.

7.8. ПРИМЕНИМ ИЗУЧЕННОЕ

Упражнение 1

Дано:

```
List<String> names;
double[] lengths;
```

Какую разновидность цикла вы бы выбрали для следующих задач?

1. Вывод всех имен в списке.
2. Удаление из списка всех имен, длина которых превышает 20 символов.
3. Вычисление суммы всех длин.
4. Установка логического флага, если массив имеет нулевую длину.

Упражнение 2

Как вам, возможно, известно, метод `charAt` из класса `String` возвращает символ переданной строки с указанным индексом:

```
public char charAt(int index)
```

Напишите комментарий Javadoc, который описывает контракт этого метода, и сравните его с аналогичным комментарием из официальной документации.

Упражнение 3

Проанализируйте следующий метод. Попробуйте определить, что он делает, и улучшить его удобочитаемость. (Не забудьте добавить комментарий Javadoc.) Исходный код этого и следующего управления можно найти в репозитории (https://bitbucket.org/mfaella/exercisesinstyle).

```
public static int f(String s, char c) {
   int i = 0, n = 0;
   boolean flag = true;
   while (flag) {
      if (s.charAt(i) == c)
         n++;
      if (i == s.length() -1)
         flag = false;
      else
         i++;
   }
   return n;
}
```

Упражнение 4

Следующий метод позаимствован из подборки алгоритмов, размещенной в репозитории github (которая получила звездный рейтинг от 10 000 человек и использовалась для ответвления 4000 раз). Метод выполняет обход в ширину графа, представленного матрицей смежности типа `byte`. Вам не нужно знать этот алгоритм для выполнения упражнения. Достаточно знать, что ячейка `a[i][j]` содержит 1, если существует ребро от узла `i` к узлу `j`, и 0 — в противном случае.

Улучшите удобочитаемость метода за два шага. Сначала внесите только внешние изменения в имена переменных и комментарии. Затем перейдите к структурным изменениям. Все изменения должны сохранить как API (типы параметров), так и наблюдаемое поведение (вывод на экран).

```
/**
 * Реализация поиска в ширину.
 *
 * @param a Структура для выполнения поиска в графе, матрица смежности и т. д.
 * @param vertices Используемые вершины
```

```
 * @param source Источник
 */
public static void bfsImplement(byte [][] a,int vertices,int source){
                          // Передается матрица смежности и количество вершин
   byte []b=new byte[vertices]; // Контейнер с флагами, содержащими
                                // статус каждой вершины
   Arrays.fill(b,(byte)-1); // Инициализация статуса
   /*       код   статус
            -1  = готово
             0  = ожидание
             1  = обработка      */
   Stack<Integer> st = new Stack<>(); // Рабочий стек
   st.push(source); // Присваивание источника
   while(!st.isEmpty()){
      b[st.peek()]=(byte)0; // Присваивание статуса ожидания
      System.out.println(st.peek());
      int pop=st.peek();
      b[pop]=(byte)1; // Присваивание статуса обработки
      st.pop();       // Удаление начала очереди
      for(int i=0;i<vertices;i++){
         if(a[pop][i]!=0 && b[i]!=(byte)0 && b[i]!=(byte)1 ){
            st.push(i);
            b[i]=(byte)0; // Присваивание статуса ожидания
         }}}
}
```

ИТОГИ

- Удобочитаемость — важный фактор, влияющий на надежность и сопровождаемость программы.
- Удобочитаемость может улучшаться как структурными, так и внешними средствами.
- Одной из целей популярных методов рефакторинга является улучшение производительности.
- Самодокументирующийся код предпочтительнее комментариев к реализации.
- Используйте стандартные формы записи и форматирования документирующих комментариев, чтобы упростить их просмотр.

ОТВЕТЫ НА ВОПРОСЫ И УПРАЖНЕНИЯ

Неожиданный вопрос 1

Удобочитаемость положительно влияет на сопровождаемость и надежность, потому что удобочитаемый код проще понять и изменить без нарушения работоспособности.

Неожиданный вопрос 2

Использовать расширенный цикл `for` не удастся, потому что нужно изменять элементы массива, а для этого необходим индекс. Лучше всего перебрать весь массив с использованием явного индекса в стандартном цикле `for`.

Неожиданный вопрос 3

Комментарий, описывающий поведение приватного метода, следует рассматривать как комментарий реализации. Приватные методы остаются скрытыми от клиентов.

Неожиданный вопрос 4

Пожалуй, самым подходящим будет имя `monthlySalary`. Альтернативы `s` и `salary` содержат слишком мало информации, а в `employeeMonthlySalary` без необходимости повторяется имя класса.

Неожиданный вопрос 5

Документировать `AssertionError` не следует, потому что эта разновидность исключения выдается только при возникновении внутренней ошибки.

Упражнение 1

1. Для первой задачи идеально подойдет расширенный цикл `for`:

```
for (String name: names) {
   System.out.println(name);
}
```

2. А эта задача для итератора:

```
Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
   if (iterator.next().length() > 20) {
      iterator.remove();
   }
}
```

3. И снова используйте расширенный цикл `for`:

```
double totalLength = 0;
for (double length: lengths) {
   totalLength += length;
}
```

   или следующую однострочную программу на базе потока данных:

```
double totalLength = Arrays.stream(lengths).sum();
```

4. Распространенное мнение гласит, что когда условие выхода определяется данными (содержимым массива), следует использовать цикл `while`. Я считаю, что расширенный цикл `for` с командой `break` по крайней мере не хуже, так как он автоматически обрабатывает случай, при котором необходимо просканировать весь массив.

```
boolean containsZero = false;
for (double length: lengths) {
   if (length == 0) {
      containsZero = true;
      break;
   }
}
```

Библиотека потоков данных предоставляет удобную альтернативу:

```
boolean containsZero = Arrays.stream(lengths).anyMatch(
                       length -> length == 0);
```

Упражнение 2

Слегка упрощенная версия документации Javadoc из OpenJDK 12:

```
/**
 * Возвращает значение {@code char} в позиции с заданным
 * индексом. Индекс лежит в диапазоне от {@code 0} до
 * {@code length() - 1}. Первое значение {@code char} последовательности
 * имеет индекс {@code 0}, следующее - индекс {@code 1} и т. д.,
 * как и при индексировании массивов.
 *
 * @param index индекс значения {@code char}.
 * @return значение {@code char} с заданным индексом в строке.
 * Первое значение {@code char} имеет индекс {@code 0}.
 *  @exception IndexOutOfBoundsException, если аргумент {@code index}
 * отрицателен или превышает длину строки.
 */
```

Упражнение 3

Нетрудно понять, что метод просто подсчитывает вхождения символа в строке. Цикл `while` и флаг — бесполезные отклонения, которые заменяются простым циклом `for` в следующем решении:

```
/** Подсчитывает количество вхождений символа в строке.
  *
  * @param s строка
  * @param c символ
  * @return Количество вхождений {@code c} в {@code s}
  */
public static int countOccurrences(String s, char c) {
   int count = 0;
```

```
   for (int i=0; i<s.length(); i++) {
      if (s.charAt(i) == c) {
         count++;
      }
   }
   return count;
}
```

Библиотека потоков данных также допускает альтернативные реализации, в которых тело метода представляет собой следующую однострочную программу:

```
return (int) s.chars().filter(character -> character == c).count();
```

Преобразование в `int` обусловлено тем фактом, что терминальная операция возвращает значение типа `long`. Более стабильная реализация установит меры защиты от переполнения.

Упражнение 4

Перейдем к финальной версии, включающей как внешние, так и структурные улучшения. Во-первых, обратите внимание, что алгоритм хранит *статус* для каждого узла, который может принять одно из трех значений: свежий (ранее не встречался), находящийся в очереди (помещен в стек, но еще не посещался) и обработанный (посещенный). В исходной реализации эта информация кодируется в массиве байтов `b`. Первое структурное улучшение — использование перечисления для этой цели. К сожалению, перечисление не может быть локальным для метода, поэтому вам придется добавить следующее объявление в область видимости класса (за пределами метода):

```
private enum Status { FRESH, ENQUEUED, PROCESSED };
```

Теперь можно провести рефакторинг основного метода, воспользовавшись этим перечислением, доработать имена переменных, удалить комментарии реализации, исправить пропуски и отступы. В итоге должен получиться примерно такой код:

```
/** Посещает узел в ориентированном графе в порядке ширины, распечатывая индекс
каждого посещенного узла
  *
  * @param adjacent матрица смежности
  * @param vertexCount количество вершин
  * @param sourceVertex исходная вершина
  */
public static void breadthFirst(
                   byte[][] adjacent, int vertexCount, int sourceVertex) {
   Status[] status = new Status[vertexCount];
   Arrays.fill(status, Status.FRESH);
```

```
   Stack<Integer> stack = new Stack<>();
   stack.push(sourceVertex);

   while (!stack.isEmpty()) {
      int currentVertex = stack.pop();
      System.out.println(currentVertex);
      status[currentVertex] = Status.PROCESSED;
      for (int i=0; i<vertexCount; i++) {
         if (adjacent[currentVertex][i] != 0 && status[i] == Status.FRESH)
         {
            stack.push(i);
            status[i] = Status.ENQUEUED;
         }
      }
   }
}
```

В этом методе я оставил класс `Stack`, потому что он не влияет на удобочитаемость, но вам нужно знать, что класс `Stack` был заменен `LinkedList` и `ArrayDeque`.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

- *Мартин Р. С.* Чистый код: создание, анализ и рефакторинг. — СПб. [и др.]: Питер, 2018. — (Серия «Библиотека программиста»).

  Подробное и исчерпывающее руководство по стилю, написанное одним из авторов «Agile-манифеста разработки программного обеспечения». Рекомендации более высокого уровня приводятся в его следующей книге «Чистая архитектура: искусство разработки программного обеспечения» (СПб. [и др.]: Питер, 2018).

- *Макконнелл С.* Совершенный код: мастер-класс. — СПб.: БХВ, 2020.

  Широкомасштабная, тщательно проработанная и качественно оформленная книга по практическим приемам программирования, от нюансов правильного выбора имен переменных до планирования проектов и управления группами.

- *Керниган Б. В., Плоджер Ф. Д.* Элементы стиля программирования. — М.: Радио и связь, 1984.

  Возможно, первая книга, в которой была систематически исследована проблема удобочитаемости кода. Примеры написаны на Fortran и PL/I. В 1978 году было издано обновленное второе издание. Керниган вернулся к теме через 20 лет вместе с Р. Пайком (R. Pike) в первой главе книги «The Practice of Programming» (Addison-Wesley, 1999).

- *Фаулер М.* Рефакторинг: улучшение проекта существующего кода. — М.: СПб.: Диалектика, 2019. — (Серия «Объективные технологии»).

Второе издание классической книги, в котором была популяризирована и стандартизирована концепция рефакторинга. На сайте автора книги по адресу https://martinfowler.com доступен каталог правил рефакторинга. Самые популярные IDE позволяют применить многие из этих правил одним-двумя щелчками мышью.

- *Knuth D. E.* Literate Programming. Center for the Study of Language and Information, 1995.

  Подборка очерков, представляющих программирование как форму искусства, сходную с литературой.

- How to Write Doc Comments for the Javadoc Tool.

  Официальное руководство по стилю Javadoc; на момент написания книги было доступно по адресу http://mng.bz/YeDe.

8 Потокобезопасность

В этой главе:

- ✓ Выявление и предотвращение взаимных блокировок и состояний гонки.
- ✓ Использование явных блокировок.
- ✓ Синхронизация без блокировок.
- ✓ Проектирование неизменяемых классов.

В этой главе мы будем делать нашу реализацию *потокобезопасной*. Если класс является потокобезопасным, сразу несколько потоков могут взаимодействовать с объектами этого класса без явной синхронизации. Другими словами, потокобезопасный класс берет на себя все нюансы, связанные с синхронизацией. Клиенты могут свободно вызывать любые методы класса, даже одновременно для одного объекта, без каких-либо нежелательных последствий. Методология контрактного проектирования, представленная в главе 5, позволяет точно назвать суть нежелательных последствий — это нарушение постусловия или инварианта.

Как я уже говорил, потокобезопасность не является свойством настолько общим, как эффективность или удобочитаемость. Тем не менее она начинает играть все более важную роль из-за повсеместного распространения оборудования для параллельного выполнения. По сравнению с другими функциональными дефектами отсутствие потокобезопасности может оставаться незамеченным гораздо дольше. Некоторые дефекты синхронизации проявляются только в особых обстоятельствах, когда из-за необычного сочетания временных характеристик и планирования состояние гонки искажает состояние объекта или программа зависает из-за взаимной блокировки. И это лишний повод для того, чтобы внимательно прочитать эту главу!

Предполагается, что читатель знаком с основными концепциями многопоточности в Java — например, созданием потоков и использованием блоков `synchronized` для реализации взаимоисключающего доступа. Если вы хотите проверить свои силы, попробуйте выполнить упражнение 1 в конце главы. Оно напомнит вам основные свойства ключевого слова `synchronized`.

8.1. ПРОБЛЕМЫ ПОТОКОБЕЗОПАСНОСТИ

Два главных врага потокобезопасности — *состояния гонки* (race conditions) и *взаимоблокировки* (deadlocks). Как правило, первые возникают из-за недостаточной синхронизации, а вторые — из-за ее избытка. Состояние гонки возникает тогда, когда две операции, запрашиваемые одновременно разными потоками, приводят к тому, что по крайней мере одна операция нарушает свое постусловие. Состояние гонки можно легко спровоцировать, работая с общими объектами без синхронизации.

Допустим, несколько потоков совместно работают с экземпляром следующего класса:

```
public class Counter {
   private int n;
   public void increment() { n++; }
   ...
}
```

Если два потока вызовут `increment` одновременно, счетчик может быть увеличен только один раз вместо двух[1]. Это объясняется тем, что операция `n++` не является *атомарной*. Она приблизительно эквивалентна следующей последовательности трех атомарных операций:

1. Скопировать текущее значение `n` в регистр (для регистровой машины) или в стек (для JVM).
2. Увеличить его на 1.
3. Сохранить обновленное значение `n` в объекте `Counter`, которому оно принадлежит.

Если два потока выполнят первый шаг одновременно (или до того, как у любого из них появится возможность сохранить обновленное значение на третьем шаге), оба потока прочитают одно и то же старое значение `n`, увеличат его, а потом сохранят одно и то же значение `n + 1`. Иначе говоря, одно значение `n + 1` будет сохранено дважды.

---

[1] Также может оказаться, что два потока «увидят» разные значения счетчика из-за проблем видимости, не связанных с состоянием гонки.

Убедитесь сами, выполнив класс `eis.chapter8.threads.Counter` из репозитория (https://bitbucket.org/mfaella/exercisesinstyle). Он запускает пять потоков, которые вызывают метод `increment` для одного объекта 1000 раз. В конце программа выводит значение счетчика. На моем ноутбуке при трех запусках были получены следующие результаты:

```
4831
4933
3699
```

Как видите, состояния гонки при таких условиях вполне типичны. При последнем запуске более 26 % инкрементов были потеряны из-за состояния гонки. Как вам, вероятно, известно, проблема гонки решается при помощи примитивов синхронизации (таких, как *мьютексы* или *мониторы*), которые делают все вызовы `increment` *взаимоисключающими*: если один такой вызов выполняется для заданного объекта `Counter`, любому другому вызову для того же объекта придется дождаться завершения текущего вызова перед тем, как он сможет войти в метод. В языке Java простейшая форма синхронизация представлена ключевым словом `synchronized`.

С другой стороны, нерегулируемая синхронизация может привести к взаимной блокировке — ситуации, в которой два и более потока навечно останавливаются в ожидании друг друга. Пример этого явления встречается в разделе 8.2.

В оставшейся части главы вы научитесь распознавать и обходить как состояния гонки, так и взаимные блокировки при помощи низкоуровневых примитивов синхронизации, таких как блоки `synchronized` и явные блокировки. Я буду придерживаться примера с резервуарами, который потребует интересной и нестандартной формы синхронизации.

Чтобы более основательно разобраться в проблемах и решениях многопоточности, вам стоит изучить фундаментальные правила *модели памяти* вашего языка. В Java лучшим источником остается книга *Java Concurrency in Practice*, упомянутая в разделе «Дополнительная литература». Кроме того, желательно изучить средства конкурентности более высокого уровня, предоставляемые выбранным языком.

С самых первых дней язык Java находился на передовых рубежах освоения многопоточности благодаря своей встроенной поддержке потоков. За последние годы эта поддержка непрерывно развивалась на трех последовательно возрастающих уровнях абстракции (рис. 8.1):

- *Исполнители* (Java 5). Небольшой набор классов и интерфейсов, обеспечивающих создание необходимого количества потоков для выполнения задач, определенных пользователем. Обратитесь к описанию интерфейса `ExecutorService` и классов `Executor` из пакета `java.util.concurrent`.

- Фреймворк `fork-join` (Java 7). Элегантный механизм разбиения сложных вычислений на несколько потоков (`fork`) и слияния их результатов в одно значение (`join`). Для начала обратитесь к описанию класса `ForkJoinPool`.
- *Параллельные потоки данных* (Java 8). Мощная библиотека для применения однотипных операций к последовательным поставщикам данных. Можно начать с описания класса `Stream`, но лучше выбрать книгу из раздела «Дополнительная литература», чтобы изучить многочисленные нюансы этой библиотеки.

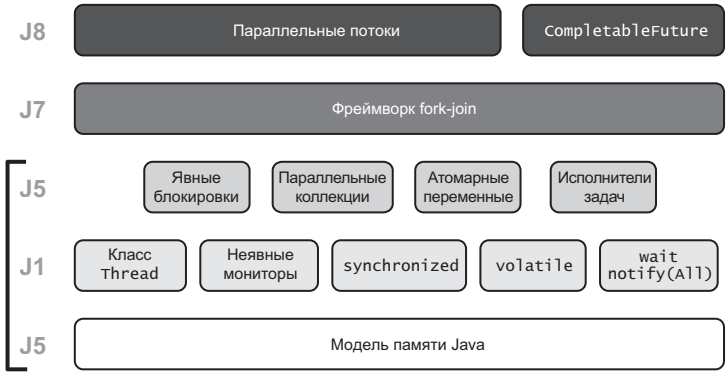


Рис. 8.1. Основные уровни поддержки многопоточности в Java с указанием номеров версий Java, в которых они впервые появились (или были исправлены в случае JMM). В этой главе рассмотрены только три нижних уровня

8.1.1. Уровни конкурентности

Если бы потокобезопасность была нашей единственной целью, можно было бы воспользоваться простым приемом, который работает во всех обстоятельствах и со всеми автономными классами: использовать глобальную блокировку для синхронизации всех методов. В Java блокировки неявно предоставляются с каждым объектом, поэтому в качестве глобальной блокировки для всех резервуаров можно воспользоваться блокировкой, присоединенной к объекту `Container.class`[1]. Затем тело всех методов класса `Container` можно упаковать в блок `synchronized`:

```
synchronized (Container.class) {
    ... ❶ Тело метода
}
```

[1] Это та же блокировка, которая будет использоваться любым синхронизированным статическим методом.

В этом случае все обращения к классу будут полностью *сериализованы*. Другими словами, даже если вызовы методов поступят из разных потоков к разным объектам, только один метод за раз сможет войти в тело блока. Этот подход оказывается в высшей степени жестким: он лишает нас любого выигрыша по производительности, которого можно достичь при конкурентности. Что еще хуже, операции захвата и освобождения блокировок могут замедлить даже однопоточную программу[1]. Этот прием можно назвать *конкурентностью уровня класса* и поместить его на конце шкалы. Случаи, заслуживающие упоминания, перечислены в табл. 8.1.

Таблица 8.1. Стандартные политики конкурентности для классов, упорядоченные по возрастанию степени конкурентности. Во втором столбце названы операции, которые могут выполняться одновременно. В третьем столбце обозначены блокировки, необходимые для реализации политики

Название	Что параллелизируется?	Сколько блокировок?
Уровень класса	Доступ к разным классам	Одна блокировка на класс
Уровень объекта	Доступ к разным объектам	Одна блокировка на объект
Уровень метода	Доступ к разным методам	Одна блокировка на метод каждого объекта
Анархия	Все	Без блокировок

Мы стремимся обеспечить потокобезопасность при сохранении максимально возможной степени конкурентности. Это делается в два этапа:

1. *Этап спецификации:* определите, какой уровень конкурентности может поддерживаться вашим классом(-ами), то есть какие методы или фрагменты кода могут выполняться одновременно без возникновения состояний гонки. На практике это фрагменты кода, которые работают с разными данными.
2. *Этап реализации:* добавьте примитивы синхронизации, которые разрешают допустимые случаи конкурентности, а для недопустимых случаев примените сериализацию.

В любой ситуации, когда объекты класса изолированы (то есть не содержат ссылок друг на друга или на совместно используемые объекты других типов), несколько методов могут выполняться одновременно — при условии, что они работают с разными объектами, а правильная потокобезопасная реализация сводится к простому добавлению `synchronized` ко всем методам экземпляров.

[1] Оптимизированные исполнительные среды могут применять специальные приемы для предотвращения этих затрат. Например, смещенная блокировка HotSpot распознает ситуацию, в которой блокировка большую часть времени принадлежит одному потоку, и оптимизирует этот случай.

Это стандартный случай *конкурентности уровня объектов*, который в традиционных классах демонстрируется следующим образом:

```
public class Employee {
   private String name;
   private Date hireDate;
   private int monthlySalary;
   ...
   public synchronized increaseSalary(int bonus) {
      monthlySalary += bonus;
   }
}
```

Кстати, даже этот простой случай можно улучшить: передовые практики *не* рекомендуют объявлять целые методы синхронизированными, потому что они станут конфликтовать с объектом `Employee`, используемым в качестве монитора клиентом. Чуть более стабильное, но также более громоздкое и неэффективное по памяти решение — использовать приватное поле в качестве монитора. В этом случае ключевое слово `synchronized` становится приватной подробностью реализации, как и должно быть:

```
public class Employee {
   private String name;
   private Date hireDate;
   private int monthlySalary;
   private Object monitor = new Object();
   ...
   public increaseSalary(int bonus) {
      synchronized (monitor) {
         monthlySalary += bonus;
      }
   }
}
```

Перейдем к третьей строке табл. 8.1: *конкурентность уровня методов* применяется относительно редко, и на то есть веские причины. Она оправданна только в тех ситуациях, когда все методы не зависят друг от друга. Чтобы два метода одного объекта были независимыми, они должны работать с разными частями состояния объекта. Если *все* методы класса взаимно независимы, это является признаком плохого сцепления: сведена информация, принадлежащая разным классам. Прежде чем размышлять над политикой конкурентности, разбейте этот класс на несколько классов.

НЕОЖИДАННЫЙ ВОПРОС 1

Кто должен заниматься политикой конкурентности класса — его пользователи или создатели реализации?

МОНИТОРЫ C#

Как и в Java, с объектами C# связываются мониторы, которые можно захватывать и освобождать с использованием следующего синтаксиса:

```
lock (object) {
   ...
}
```

Вы можете объявить синхронизированным целый метод, для чего он помечается следующим *атрибутом* (по аналогии с аннотацией Java):

```
[MethodImpl(
 MethodImplOptions.Synchronized)]
```

В отличие от Java, вы также можете вручную блокировать и разблокировать неявный монитор объекта вызовами `Monitor.Enter(object)` и `Monitor.Exit(object)`.

Наконец, *анархический уровень* обычно применяется к классам, которые либо не имеют состояния, либо являются неизменяемыми. В обоих случаях параллельное использование со стороны нескольких потоков не создает проблем. Например, компараторы (объекты, реализующие интерфейс `Comparator`) обычно не имеют состояния. Они могут свободно использоваться несколькими потоками без особых мер предосторожности. Тема неизменяемости будет рассматриваться в разделе 8.4.

Для резервуаров из нашего примера характерен особый уровень конкурентности, находящийся где-то на середине между уровнем класса и уровнем объекта, что потребует чуть больших усилий для описания и реализации такой задачи.

8.1.2. Политика конкурентности для резервуаров

Независимо от реализации, резервуары должны каким-то образом обращаться друг к другу; в противном случае они не выполнят свои контрактные обязательства. А именно, методы `connectTo` и `addWater` смогут изменить состояние нескольких резервуаров. В результате для достижения потокобезопасности недостаточно захватить блокировку текущего объекта.

Больше всего проблем возникает с методом `connectTo`, потому что он изменяет две группы резервуаров, в конечном итоге объединяя их в одну. Для предотвращения состояний гонки нужно запретить другим потокам обращаться к резервуару, принадлежащему любой из двух объединяемых групп.

Точнее, чтение состояния такого результата методом `getAmount` может быть разрешено, но изменение его `addWater` или `connectTo` определенно должно быть запрещено (вернее, отложено до завершения первого `connectTo`).

В итоге мы получаем следующую политику конкурентности для класса `Container`:

1. Класс должен быть потокобезопасным.
2. Если резервуары `a` и `b` не входят в одну группу, вызов любого метода для `a` может выполняться параллельно с вызовом любого метода для `b`.
3. Все остальные пары вызовов требуют синхронизации.

Только свойство 1 предназначено для пользователей класса `Container`. Оно сообщает клиентам, что они могут использовать класс из разных потоков одновременно, не беспокоясь о проблемах синхронизации.

С другой стороны, свойства 2 и 3 предназначены для разработчиков класса `Container` и задают целевой уровень поддерживаемой конкурентности. В табл. 8.1 этот уровень лежит где-то между конкурентностью уровня класса и уровня объекта, потому что он допускает конкурентность между разными группами объектов. В оставшейся части этой главы мы рассмотрим разные способы достижения этой цели с примитивами синхронизации Java, а именно `synchronized`, `volatile` и классом `ReentrantLock`.

8.2. ВЗАИМОБЛОКИРОВКИ

Вместо того чтобы изменять `Reference`, мы поработаем с примером `Speed1` из главы 3 — более эффективным и более подходящим для демонстрации потокобезопасности. Вспомните базовую структуру `Speed1`: каждый резервуар содержит ссылку на объект *группы*, который, в свою очередь, знает объем воды в каждом резервуаре и множество всех участников группы, как показано в следующем фрагменте.

```
public class Container {
   private Group group = new Group(this);

   private static class Group {
      double amountPerContainer;
      Set<Container> members;

      Group(Container c) {
         members = new HashSet<>();
         members.add(c);
      }
   }
```

Из политики спецификации видно, что группы являются единицами синхронизации класса. На практике метод `connectTo` должен захватить мониторы двух объединяемых групп. В любой момент, когда методу нужно более одного монитора, возникает риск взаимной блокировки — ситуации, при которой два

и более потока не могут продолжать выполнение, потому что каждый из них ожидает монитор, удерживаемый другим потоком. Взаимное ожидание длится до бесконечности.

Простейший сценарий со взаимной блокировкой возникает тогда, когда поток 1 пытается захватить монитор *A*, а затем монитор *B*, тогда как поток 2 запрашивает их в обратном порядке. Неудачное планирование может привести к тому, что поток 1 успешно захватит *A*, затем поток 2 успешно захватит *B* до того, как это сделает поток 1. В этот момент возникнет взаимная блокировка потоков. Сценарий может легко воспроизвести следующую естественную, но дефектную реализацию `connectTo`:

```
public void connectTo(Container other) {
   synchronized (group) {
      synchronized (other.group) {
         ... ❶ Операция
      }
   }
}
```

Если один поток вызывает `a.connectTo(b)`, а другой тогда же вызовет `b.connectTo(a)`, возникнет риск классического случая взаимной блокировки. Вообще говоря, существуют два способа предотвращения взаимных блокировок без каких-либо ограничений для клиентов класса: *атомарные* и *упорядоченные последовательности блокировок*.

НЕОЖИДАННЫЙ ВОПРОС 2

Возможно ли возникновение взаимной блокировки, если каждый поток гарантированно удерживает только одну блокировку?

8.2.1. Атомарные последовательности блокировок

Во-первых, можно преобразовать последовательность захвата блокировок, из-за которых возникает риск взаимной блокировки, в атомарную форму. Для этого используется дополнительная блокировка (назовем ее *глобальной* — `globalLock`), которая гарантирует, что две последовательности не смогут выполняться одновременно. При таком подходе последовательность запросов на блокировку может начаться только при отсутствии другой такой последовательности. Если последовательность блокируется из-за того, что одна из необходимых блокировок занята, она будет ожидать освобождения глобальной блокировки, так что ни одна другая последовательность не сможет запуститься и создать риск перехода во взаимную блокировку. Обратите внимание: даже последовательности, требующие совершенно другого набора блокировок, будут приостановлены до завершения текущей последовательности. Это чрезвычайно осторожный подход,

который предотвращает взаимные блокировки за счет ограничения допустимого уровня конкурентности.

В Java глобальная блокировка не может быть неявной, потому что неявные блокировки по своей природе должны освобождаться в порядке, обратном порядку их захвата. Таким образом, если блокировка `globalLock` захватывается до монитора *A*, она не может быть освобождена раньше последнего. Следующий фрагмент является дефектным:

```
synchronized (globalLock) {
   synchronized (group) {
      synchronized (other.group) {
} ❶ Мы хотим освободить globalLock здесь
         ... ❷ Операция
      }
   }
```

Несмотря на отступ, сбивающий с толку, первая закрывающая фигурная скобка освобождает `other.group`, а не `globalLock`, как предполагалось. *Явные* блокировки, предоставляемые в Java API классом `ReentrantLock`, преодолевают это ограничение. Класс `ReentrantLock` обладает большей гибкостью, чем неявная блокировка, в частности, его можно свободно захватывать и освобождать в любой момент при помощи методов `lock` и `unlock`. В этом решении в класс будет добавлена явная блокировка:

```
private static final ReentrantLock globalLock = new ReentrantLock();
```

Затем глобальная блокировка используется для защиты начала `connectTo`, пока не будут захвачены две другие неявные блокировки:

Листинг 8.1. AtomicSequence: предотвращение взаимных блокировок с использованием атомарной последовательности блокировок

```
public void connectTo(Container other) {
   globalLock.lock();
   synchronized (group) {
      synchronized (other.group) {
         globalLock.unlock();
         ... ❶ Вычисление нового значения amount
         group.members.addAll(other.group.members);
         group.amountPerContainer = newAmount;
         for (Container x: other.group.members)
            x.group = group;
      }
   }
}
```

Поскольку в любой момент времени только один поток может удерживать `globalLock`, это означает, что только один поток может находиться в середине последовательности двух строк `synchronized`, и взаимная блокировка не возникнет.

НЕОЖИДАННЫЙ ВОПРОС 3

Что произойдет, если исключение будет выдано из блока `synchronized`? Что, если вместо этого поток, выдавший исключение, является владельцем `ReentrantLock`?

8.2.2. Упорядоченные последовательности блокировок

Второй, более эффективный способ предотвращения взаимных блокировок — упорядочение мониторов в глобальную последовательность, известную всем потокам, и запрос их всеми потоками именно в таком порядке. Такую глобальную последовательность можно создать назначением уникального целочисленного идентификатора каждой группе. Для получения уникального идентификатора вводится глобальный (то есть статический) счетчик, который инкрементируется для каждого нового экземпляра и определяет идентификатор каждого нового объекта.

Доступ к общему счетчику необходимо должным образом синхронизировать, иначе из-за состояния гонки, в которой задействованы два одновременных инкремента, двум группам может быть назначено одно значение идентификатора. Самое простое решение — воспользоваться классом `AtomicInteger` (один из атомарных типов переменных на рис. 8.1). Объекты этого класса представляют собой потокобезопасные изменяемые целые числа. Как следует из названия, его метод экземпляра `incrementAndGet` идеально подходит для генерирования уникальных последовательных идентификаторов потокобезопасным способом.

В листинге 8.2 приведено начало класса `Container`, которое включает его поля и вложенный класс. Оно очень похоже на `Speed1`, если не считать добавления уникальных идентификаторов групп.

Листинг 8.2. OrderedSequence: предотвращение взаимных блокировок с использованием упорядоченной последовательности блокировок

```
public class Container {
   private Group group = new Group(this);

   private static class Group {
      static final AtomicInteger nGroups =
         new AtomicInteger(); ❶ Общее количество групп на данный момент
      double amount;
      Set<Container> elems = new HashSet<>();
      int id = nGroups.incrementAndGet(); ❷ Автоматически назначаемые
                                            последовательные идентификаторы
      Group(Container c) {
      elems.add(c);
   }
}
```

Каждому новому объекту `Group` теперь назначается уникальный идентификатор, начиная с 1, по аналогии с полями-счетчиками в базах данных. Как видно из листинга 8.3, метод `connectTo` запрашивает два монитора в порядке их идентификаторов, что предотвращает возможность взаимных блокировок.

ИДЕНТИФИКАЦИОННЫЙ ХЕШ-КОД

Похожий метод основан на упорядочении захвата блокировок по хеш-коду соответствующего объекта (в нашем случае группы), то есть по хеш-коду, возвращаемому методом `hashCode` класса `Object`. Если этот метод был переопределен, вы все еще можете использовать исходный хеш-код объекта, для получения которого следует вызвать статический метод `System.identityHashCode()`.

Такой подход экономит память и несколько строк кода, потому что хеш-код представляет собой встроенный идентификатор для любого объекта. С другой стороны, он не может считаться *уникальным* идентификатором, потому что два объекта могут иметь одинаковые хеш-коды (хотя это и маловероятно). В свою очередь, последовательные идентификаторы уникальны по определению — при условии, что количество объектов этого типа меньше 2^{32}. Впрочем, даже в этом случае можно переключиться на идентификатор `AtomicLong`.

Листинг 8.3. OrderedSequence: метод connectTo

```
public void connectTo(Container other) {
   if (group == other.group) return;
   Object firstMonitor, secondMonitor;
   if (group.id < other.group.id) {
      firstMonitor = group;
      secondMonitor = other.group;
   } else {
      firstMonitor = other.group;
      secondMonitor = group;
   }
   synchronized (firstMonitor) {
      synchronized (secondMonitor) {
         ... ❶ Вычисление нового значения amount
         group.members.addAll(other.group.members);
         group.amountPerContainer = newAmount;
         for (Container x: other.group.members)
            x.group = group;
      }
   }
}
```

Если у вас есть какой-либо способ назначения уникальных идентификаторов объектам, для которых необходимо захватить блокировку, действовать нужно

именно так. Если же объекты не имеют уникальных идентификаторов и вы не можете изменить код их класса, метод глобальной блокировки из предыдущего раздела может стать единственным вариантом.

НЕОЖИДАННЫЙ ВОПРОС 4

Почему упорядоченная блокировка предотвращает возникновение взаимных блокировок?

8.2.3. Скрытое состояние гонки

В разделах 8.2.1 и 8.2.2 описаны общие способы предотвращения взаимных блокировок, но в случае с резервуарами воды они подвержены неочевидным состояниям гонки. Проблема в том, что объекты групп, которые выполняют функции мониторов, *могут быть заменены* одновременной операцией соединения. В результате вызов `connectTo` может получить блокировку для устаревшей группы, которая больше не связана с другим резервуаром. В этом случае операции, выполняемые `connectTo`, не будут взаимоисключающими с другими операциями новой группы этого резервуара.

Эта проблема легко распознается при упорядоченной блокировке из раздела 8.2.2. Первые строки `connectTo`, сравнивающие идентификаторы групп и устанавливающие относительный порядок между мониторами, не защищены никакой синхронизацией. А значит, может случиться так, что любая из двух групп изменится до того, как у текущего потока будет возможность захватить соответствующий монитор. Естественным решением станет добавление глобальной блокировки, защищающей первую фазу — от начала метода до точки после захвата двух мониторов. Это приблизит код к другому решению с использованием атомарной последовательности блокировок. Однако глобальная блокировка первой фазы делает бессмысленным весь механизм упорядоченных блокировок, потому что глобальной блокировки достаточно для предотвращения взаимных блокировок! В конечном итоге получится *точно такая же* версия с атомарной последовательностью блокировок. Но избавлена ли она от состояний гонки?

Внимательное изучение или целенаправленное тестирование показывают, что нет. `connectTo` может захватить неправильный монитор и нарушить заявленную политику конкурентности (рис. 8.2). В самом деле, предположим, что поток 1 запустит `a.connectTo(b)`, но он вытесняется[1] перед обновлением группы `b`, то есть перед присваиванием `b.group = a.group`. Это может произойти по разным причинам, самая простая из которых заключается в том, что другой поток планируется для выполнения на том же аппаратном ядре. В конце концов, JVM не

[1] То есть его выполнение приостанавливается планировщиком ОС.

работает в изоляции. Она использует оборудование совместно с ОС и многими другими процессами.

Допустим, в этой точке поток 2 выполняет `b.connectTo(c)`. Второй поток застрянет в `synchronized (b.group)`, потому что первый поток удерживает этот монитор. Когда первый поток освободит его, он будет захвачен вторым потоком, даже несмотря на то что этот монитор не соответствует никакой группе и потому является монитором устаревшего объекта группы, готового к сбору мусора. У второго потока существует иллюзия, что он удерживает монитор для группы `b`, тогда как в действительности он удерживает устаревший монитор. Последующие операции не будут взаимоисключающими с другими операциями *текущей* группы `b`.

Этот сценарий изображен на рис. 8.2, а решение проблемы приводится в следующем разделе, где наконец-то будет представлена полностью потокобезопасная реализация резервуаров.

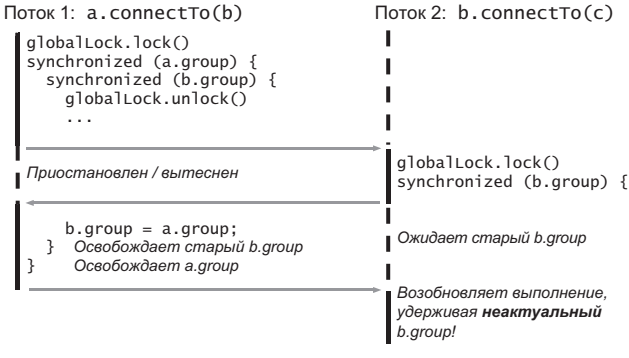


Рис. 8.2. Состояние гонки, влияющее на реализацию connectTo с атомарной последовательностью блокировок

8.3. ПОТОКОБЕЗОПАСНЫЕ РЕЗЕРВУАРЫ [THREADSAFE]

Чтобы получить потокобезопасную реализацию, мы возьмем версию `OrderedSequence` (листинги 8.2 и 8.3), свободную от взаимных блокировок и обеспечивающую конкурентность между вызовами методов, в которых задействованы разные группы резервуаров, и решим состояния гонки, описанные в предыдущем разделе. Новая реализация под названием `ThreadSafe` будет содержать те же поля и тот же вложенный класс `Group`, что и `OrderedSequence` и тоже не будет нуждаться в глобальной блокировке при соединении двух резервуаров. Но она будет попытаться захватить нужные мониторы несколько раз, что описано в следующем разделе.

8.3.1. Синхронизация connectTo

Чтобы исключить состояние гонки, необходимо убедиться, что `connectTo` захватывает мониторы текущих групп двух соединяемых резервуаров. Чтобы сделать это без особого ущерба для конкурентности, необходимо переключиться с классической синхронизации на основе блокировок на *синхронизацию, свободную от блокировок*. Если только вы не используете глобальную блокировку с потерей всей конкурентности, вы никогда не можете быть уверены, что нужные мониторы будут захвачены с первой попытки. Возможно, придется повторить несколько попыток (листинг 8.4), пока вы не поймете, что захваченные мониторы являются текущими. Вот почему код упорядоченной последовательности блокировок, позаимствованной из `OrderedSequence`, заключается в теоретически бесконечный цикл.

Листинг 8.4. ThreadSafe: метод connectTo

```
public void connectTo(Container other) {
   while (true) {
      if (group == other.group) return;
      Object firstMonitor, secondMonitor;
      if (group.id < other.group.id) {
         firstMonitor = group;
         secondMonitor = other.group;
      } else {
         firstMonitor = other.group;
         secondMonitor = group;
      }
      synchronized (firstMonitor) { ❶ Попытка захвата мониторов
         synchronized (secondMonitor) {
            if ((firstMonitor == group && secondMonitor == other.group) ||
               (secondMonitor == group && firstMonitor == other.group)) {
               ... ❷ Операция
               return;
            }
         }
      }
      ❸ По крайней мере один из двух мониторов неактуален – повторить.
   }
}
```

При каждой итерации вы захватываете два выбранных монитора только для того, чтобы немедленно проверить, являются ли они текущими, то есть ссылаются ли на них соответствующие контейнеры как на свои группы. Если проверка дает положительный результат, выполняются обычные операции слияния групп (в листинге 8.4 они опущены). В противном случае два монитора освобождаются и совершается повторная попытка с чтением полей групп двух объединяемых резервуаров. Такой подход к синхронизации можно назвать *оптимистичным*, ведь предполагается, что ни один другой поток не вмешивается в работу этих двух резервуаров. Если предположение нарушено, попытка повторяется.

СИНХРОНИЗАЦИЯ БЕЗ БЛОКИРОВОК

Паттерн с повторными попытками выполнения операции с общим объектом, до тех пор пока не будет обнаружено отсутствие конфликтов, напоминает стандартный цикл CAS (compare and swap, «сравнить и поменять местами») в синхронизации без блокировок. CAS — команда процессора с тремя аргументами (`src`, `dst` и `old`), которая меняет местами содержимое ячеек памяти `src` и `dst`, но только в том случае, если текущее содержимое `dst` равно содержимому `old`. Команда может использоваться для безопасного обновления общей переменной без использования мьютекса.

В этом случае мы сначала читаем общую переменную (`dst`) и помещаем ее значение в локальную переменную (`old`). Затем вычисляем новое значение общей переменной (обычно основанное на ее старом значении), которое сохраняем в другой локальной переменной. Наконец, CAS выполняется с указанными выше аргументами, чтобы общая переменная обновлялась только в том случае, если она за это время не была изменена другим потоком. Если CAS сообщает о неудаче, вся операция перезапускается до бесконечности, как показано в следующем псевдокоде:

```
do {
   old = dst
   src = новое значение, обычно основанное на old
} while (cas(src, dst, old) == неудача)
```

В данном случае задействован гибридный сценарий, который гарантирует, что правильные мониторы будут захвачены с использованием синхронизации без блокировок, тогда как остальные операции объединения выполняются в схеме классической защиты с блокировками.

8.3.2. Синхронизация addWater и getAmount

Перейдем к двум оставшимся методам: `addWater` и `getAmount`. `addWater` имеет структуру, сходную со структурой `connectTo`. В самом деле, даже при захвате монитора *одной* группы может оказаться, что тем временем другой поток изменил группу этого резервуара.

Дело в том, что вход даже в простейшую блокировку с синхронизацией не является атомарной операцией. Чтобы проанализировать ситуацию более подробно, необходимо заглянуть за кулисы кода Java и взглянуть на соответствующий байт-код.

АРХИТЕКТОР JVM

В отличие от большинства современных микропроцессоров, архитектура которых базируется на регистрах, JVM — абстрактная машина, предоставляющая каждому вызову метода стек операндов и серию локальных переменных. При входе в метод стек операндов пуст, а локальные переменные содержат аргументы текущего метода. При выполнении метода экземпляра первая локальная переменная содержит `this`. Арифметические и логические операции получают свои аргументы из стека операндов и возвращают результат туда же. Более того, JVM поддерживает объекты: обращения к полям, вызовы методов и другие ОО-операции напрямую соответствуют конкретным командам байт-кода.

Для наглядного представления содержимого файла класса в форме можно воспользоваться программой командной строки `javap`, включенной в JDK. Байт-код всех методов класса можно просмотреть командой `javap -c имя_класса`.

Например, допустим, что `addWater` начинается так:

```
public void addWater(double amount) {
   synchronized (group) { ❶
      ...
   }
}
```

Вторая строка преобразуется в следующий байт-код:

```
1: aload_0      Занести первую локальную переменную (this) в стек
2: getfield #5  Извлечь элемент из стека и занести в стек его поле group
3: dup          Продублировать элемент на вершине стека
4: astore_2     Сохранить вершину стека в локальной переменной #2
5: monitorenter Извлечь элемент из стека и получить его монитор
```

Как видите, то, что со стороны выглядит как атомарный захват блокировки, в действительности преобразуется в короткую последовательность байт-кода, в которой монитор запрашивается только в последней команде. Если другой поток изменит группу этого резервуара между строками байт-кода 2 и 5, текущий поток захватит монитор устаревшей группы, а его последующие операции не будут взаимоисключающими с другими операциями новой группы этого резервуара. В таком случае виновником станет параллельный вызов `connectTo`, потому что это единственный метод, который изменяет ссылки на группы.

Попытку необходимо повторить многократно, как показано в следующем листинге, пока вы не будете уверены, что захваченный монитор является текущим.

Листинг 8.5. ThreadSafe: метод addWater

```
public void addWater(double amount) {
   while (true) {
      Object monitor = group;
      synchronized (monitor) { ❶ Попытка захватить монитор
         if (monitor == group) { ❷ Монитор актуален
            double amountPerContainer = amount / group.elems.size();
            group.amount += amountPerContainer;
            return;
         }
      }
      ❸ Монитор был устаревшим - повторить попытку
   }
}
```

Наконец, как видно из листинга 8.6, `getAmount` является простым GET-методом, и непонятно, зачем применять к нему какую-либо синхронизацию. В конце концов, он просто читает примитивное значение. В худшем случае он может прочитать слегка устаревшее значение, которое только что было изменено. Верно? Нет, неверно. Модель памяти Java определяет, что даже одиночное чтение значения `double` не является атомарной операцией. Таким образом, 64-разрядная операция чтения может быть разделена на две 32-разрядных операции чтения, и эти две операции чтения могут перемежаться операцией записи со стороны другого потока. Без блока `synchronized` в итоге может быть прочитано некорректное значение, старшие 32 бита которого являются новыми, а младшие 32 бита — устаревшими, и наоборот. Кстати говоря, добавление модификатора `volatile` к полю `amount` также решает проблему, так как операция `read` становится атомарной.

Листинг 8.6. ThreadSafe: метод getAmount

```
public double getAmount() {
   synchronized (group) {
      return group.amount;
   }
}
```

Для метода `getAmount` не нужно беспокоиться, что группа, к которой вы обращаетесь, окажется устаревшей, потому что она даст только слегка устаревшее значение, а не полностью ошибочное. В многопоточной среде даже актуальное на данный момент значение может быть обновлено и оказаться устаревшим в любой момент. С учетом этого факта бессмысленно тратить лишние усилия на проверку того, что вы читаете `amount` из текущей группы.

Сравните ситуацию с `addWater`. Если обновить устаревшую группу методом `addWater`, произойдет нарушение целостности, потому что вы добавите воду в группу, на которую не ссылается ни один резервуар. Добавленная вода пропадет, а метод нарушит свое постусловие.

8.4. НЕИЗМЕНЯЕМОСТЬ

Потоковая небезопасность в конечном итоге обусловлена тем, что один поток записывает данные в общую память, в то время как другие потоки читают или записывают данные в ту же область данных. Если все ваши общие объекты будут неизменяемыми, никакой поток не сможет изменить объект после того, как он будет инициализирован и открыт для общего доступа. К сожалению, такой подход плохо сочетается с API, определенным в главе 1. Из того простого факта, что вы можете вызвать `getAmount` дважды для одного резервуара и получить два разных значения, следует, что резервуары обладают изменяемым состоянием. Объекты Java по умолчанию изменяемы, хотя в языке встречаются неизменяемые довольно важные классы, например `String` и `Integer`. Собственно, именно благодаря этим стандартным классам все Java-программисты умеют использовать неизменяемые объекты и знают, как строить программы на их основе.

НЕИЗМЕНЯЕМОСТЬ В C#

Чтобы создать неизменяемый класс в языке C#, объявите все его поля как доступные только для чтения, и убедитесь, что все объекты, на которые указывают ссылки, также принадлежат к неизменяемым классам.

Строки C# являются неизменяемыми, как и строки Java, но C# предоставляет возможность обойти неизменяемость при помощи ключевого слова `unsafe`. Другой пример неизменяемого класса — `System.DateTime`.

На всякий случай напомню: класс называется неизменяемым (immutable), если все его поля имеют пометку `final`, а все содержащиеся в нем ссылки указывают на другие неизменяемые классы[1]. Если метод изменяемого класса может изменить состояние текущего объекта, аналогичный метод неизменяемого класса создает и возвращает *новый* объект того же типа с нужным содержимым.

Посмотрим, как этот принцип работает в сочетании со стандартными классами `String` и `Integer`. Эти классы не предоставляют методов для изменения своего содержимого. Тем не менее для удобства программиста неизменяемость хитро маскируется механизмами стадии компиляции, которые позволяют использовать запись следующего вида (для `String s` и `Integer n`):

```
s += " and others";
n++;
```

[1] Точнее, класс может быть неизменяемым, даже если в нем нет ни одного поля с пометкой `final`, но ключевое слово `final` неизменяемость гарантирует. Эти два случая отличаются так же, как финальная и *фактически* финальная переменные (последние играют важную роль для видимости внутреннего класса).

Как вам, вероятно, известно, эти две строки не изменяют объекты, на которые указывают `s` и `n`, — они создают новые объекты, заменяющие старые. Например, компилятор преобразует безобидную конкатенацию строк в следующий кошмар, в котором даже задействован совершенно другой класс:

```
StringBuilder temp = StringBuilder();
temp.append(s);
temp.append(" and others");
s = temp.toString();
```

Аналогичным образом для инкремента `Integer` компилятор генерирует байт-код, который распаковывает значение, увеличивает его, а затем снова упаковывает в объект при помощи статического фабричного метода[1]. Процесс похож на следующий фрагмент на языке Java:

```
int value = n.intValue();          ❶ Распаковка
n = Integer.valueOf(value + 1); ❷ Повторная упаковка
```

В этой главе мы рассматриваем неизменяемые классы, потому что они по определению потокобезопасны. Несколько потоков могут вызывать методы для одного объекта, и вызовы не будут конфликтовать просто потому, что они не выполняют запись в одну и ту же область памяти. Если один из этих методов будет должен вернуть сложное значение, он создаст новый объект, который другие потоки не увидят до тех пор, пока этот объект не будет возвращен методом.

НЕОЖИДАННЫЙ ВОПРОС 5

Почему неизменяемые классы всегда потокобезопасны?

НЕИЗМЕНЯЕМОСТЬ И ФУНКЦИОНАЛЬНЫЕ ЯЗЫКИ

В других парадигмах программирования (например, в функциональной парадигме) неизменяемость используется по умолчанию или как единственный вариант. Например, в OCaml все переменные неизменяемы, если только они не снабжены модификатором `mutable`. Такой подход работает, потому что программа состоит из одного огромного (или маленького) выражения, а итерации заменяются рекурсией. Следует заметить, что рекурсия создает *видимость* изменяемости, поскольку параметры рекурсивной функции связываются с разными значениями на разных шагах рекурсии.

JVM-языки Scala и Kotlin также отдают предпочтение функциональному программированию. В них переменные неизменяемы по умолчанию, но можно создать изменяемые переменные с помощью ключевого слова `var`.

---

[1] В отличие от конструктора, фабричный метод не обязан возвращать новый объект. Собственно, `valueOf` кэширует все целые числа в диапазоне от –128 до 127.

8.4.1. API

Выйдем за пределы API, установленные в главе 1, и спроектируем открытый интерфейс для неизменяемой версии резервуаров с такой же функциональностью, как у изменяемой. Если предположить, что резервуары неизменяемы, метод `addWater` должен вернуть новый резервуар с обновленным объемом воды, но этого недостаточно. Если текущий резервуар соединен с другими, новые объекты с обновленным количеством воды должны заменить все эти резервуары. Представьте, насколько неудобно вызывать `addWater` и получать множество всех обновленных резервуаров, соединенных с текущим. Необходимо подвергнуть API более обширному рефакторингу.

Заложим в архитектуру API более масштабную перспективу, где основным объектом будет *система* резервуаров. Каждую систему мы создадим неизменяемой с фиксированным количеством резервуаров `n` с индексами от `0` до `n – 1`. Операции, изменяющие состояние даже одиночного резервуара, будут *создавать впечатление*, что они возвращают новую систему резервуаров. Станет ли новый объект во внутреннем представлении использовать одни данные совместно со старым объектом — вопрос реализации, к которому мы еще вернемся.

Для начала разработаем черновой вариант API, поддерживающего классы `ContainerSystem` и `Container`. Следующий фрагмент показывает, как создать систему из 10 резервуаров и добавить 42 единицы воды в шестой резервуар. Так как объекты неизменяемы, при добавлении воды будет возвращена новая система резервуаров.

```
ContainerSystem s1 = new ContainerSystem(10); ❶ Новая система из 10 резервуаров
Container c = s1.getContainer(5); ❷ Шестой резервуар
ContainerSystem s2 = s1.addWater(c, 42); ❸ Новая система, в которой резервуар c содержит
                                           42 единицы воды
```

Такой тип поведения, при котором изменяющая операция возвращает новый объект, называется *долгосрочной* структурой данных. Как следует из названия, такие структуры данных предоставляют клиентам всю свою историю. Например, в предыдущем фрагменте система `s1` все еще доступна после того, как вы получите созданную на ее основе систему `s2`. В обратной (классической) ситуации структура данных, которая изменяется «на месте» и не сохраняет свое прошлое состояние, называется *эфемерной*. Так как долгосрочные структуры данных предоставляют больше функциональности, неудивительно, что они обычно уступают эфемерным по затратам времени и памяти.

Возвращаясь к новому API, заметим, что резервуар `c` является неизменяемым и принадлежит системе `s1`. Здесь мы сталкиваемся с важным решением из области проектирования: является ли `c` также допустимым резервуаром для `s2`? То есть можно ли использовать вызов вида `s2.addWater(c, 7)`? Если нельзя, то пользоваться таким API будет крайне неудобно. При любом изменении в резервуаре будет генерироваться новая система, а все текущие объекты

резервуаров будут объявляться недействительными. Если можно, то мы будем ожидать, что `c` представляет «резервуар с индексом 5» в *любой* системе резервуаров. Другими словами, `c` становится почти явным псевдонимом для индекса 5. Ни один сценарий нельзя признать удовлетворительным. Вместо этого мы полностью избавимся от класса `Container` (как в версиях `Memory3` и `Memory4` из главы 4) и будем идентифицировать резервуары по простым целочисленным идентификаторам.

Первый фрагмент, который создает систему из 10 резервуаров и добавляет воду в шестой резервуар, преобразуется так:

```
ContainerSystem s1 = new ContainerSystem(10);
ContainerSystem s2 = s1.addWater(5, 42); ❶ Добавить 42 литра в резервуар 5
```

А если вы понимаете, что вам нужен одиннадцатый резервуар, но не хотите создавать новую систему с нуля? Метод экземпляра класса `ContainerSystem` может вернуть новую систему с дополнительным резервуаром:

```
ContainerSystem s3 = s2.addContainer(); ❷ Добавить 11-й резервуар
```

Естественно, метод `connectTo` (переименованный в `connect`) должен получать два идентификатора резервуаров и возвращать совершенно новую систему резервуаров:

```
s3 = s3.connect(5, 6); ❸ Соединяет резервуары 5 и 6
double amount = s3.getAmount(5); ❹ Содержит значение 21.0
```

Итак, мы получаем следующие методы:

```
public class ContainerSystem {
   public ContainerSystem(int containerCount)
   public ContainerSystem addContainer()
   public int containerCount() ❺ Количество резервуаров в системе

   public ContainerSystem connect(int containerID1, int containerID2)
   public double getAmount(int containerID)
   public ContainerSystem addWater(int containerID, double amount)
}
```

8.4.2. Реализация

Чтобы перейти от имеющейся изменяемой реализации к неизменяемой, можно применить следующий прием *копирования при записи*:

1. Система резервуаров содержит те же данные, которые были распределены между всеми резервуарами в изменяемой реализации.
2. Каждая изменяющая операция (`addWater` и `connectTo`) создает и возвращает новую систему, содержащую измененную копию всей структуры данных.

Это самый простой способ преобразования изменяемого класса в неизменяемый, но в общем случае не самый эффективный. Другой, более сложный способ заключается в том, чтобы по возможности использовать как можно больше данных старого объекта при применении к нему изменяющей операции вместо создания полного дубликата. В случае резервуаров с водой можно представить умную неизменяемую реализацию, которая дублирует резервуары по требованию, то есть копирует только группу резервуаров, на которую влияет заданная изменяющая операция (допустим, вызов `addWater`), и повторно использует все остальные резервуары, пока они не будут задействованы в вызове `addWater` или `connectTo`.

ДОЛГОСРОЧНЫЕ СТРУКТУРЫ ДАННЫХ

В области проектирования эффективных неизменяемых структур данных ведутся активные исследования. Их целью является оценка эффективности изменяемых структур данных при сохранении преимуществ неизменяемости, особенно в сочетании с функциональными языками.

Некоторые сторонние библиотеки предоставляют «умные» долгосрочные коллекции Java, в которых измененная копия совместно использует часть данных с исходной структурой, что экономит время и память по сравнению с простым копированием при записи. Среди примеров можно упомянуть PCollections (https://github.com/hrldcpr/pcollections) и Cyclops (https://github.com/aol/cyclops).

Теоретически метод копирования при записи может быть применен в любом из решений, рассмотренных в предыдущих главах. На практике большинство изменяемых реализаций, приводившихся ранее, не имеют особого смысла в виде неизменяемых классов копирования при записи. Рассмотрим самую эффективную изменяемую реализацию из главы 3: деревья указателей на родителей из версии `Speed3`. Ценность этой реализации неразрывно связана с ее изменяемостью: операции обновления и получения данных выполняются эффективно. Если каждая операция `connectTo` копировала бы весь лес (да, множество деревьев называется именно так), эффективность обновления полностью потерялась бы, потому что каждый вызов `connectTo` выполнялся бы за линейное время.

С таким же успехом можно с самого начала воспользоваться более простой структурой данных. Попробуем спроектировать неизменяемую версию `Memory3`. Эта реализация базируется на двух массивах, копирование которых выполняется быстро и эффективно. Метод `connectTo` все еще будет выполняться за линейное время, но по крайней мере все данные можно будет скопировать двумя простыми строками. Кроме того, копирование двух блоков непрерывной памяти выполняется быстрее, чем копирование связанного леса, хотя и с той же асимптотической сложностью.

Для начала вспомним, какие структуры данных использует `Memory3`:

- Массив `group` связывает идентификатор резервуара с идентификатором его группы.
- Массив `amount` связывает идентификатор группы с объемом воды в каждом резервуаре этой группы.

Каждый экземпляр `ContainerSystem` будет содержать эти два массива. Возможно, их стоит объявить с ключевым словом `final`, указывающим на их неизменяемость. Конечно, это не помешает изменить содержимое массива. В данном случае модификатор `final` всего лишь напоминает о том, что вы стремитесь к усилению неизменяемости.

Реализация `Memory3` старается поддерживать минимальную длину массива `amount`: если два резервуара соединяются, а их группы объединяются, то одна ячейка исключается из `amount`, так как групп становится на одну меньше. Здесь затраты памяти нас не особенно интересуют, так что можно выбрать более простой путь и хранить два массива с одинаковой длиной, равной общему количеству резервуаров.

Единственный открытый конструктор `ContainerSystem` создает систему с заданным количеством пустых и изолированных резервуаров. Для этого он выделяет каждому резервуару отдельную группу, а идентификатор `i`-й группы просто равен `i`.

Для заданного идентификатора резервуара метод `getAmount` обращается к массиву `group` для получения идентификатора группы этого резервуара, а затем к массиву `amount` — для получения объема воды в этой группе:

Листинг 8.7. Immutable: поля, конструктор и метод getAmount

```
public class ContainerSystem {
   private final int group[]; ❶ Связывает containerID с groupID
   private final double amount[]; ❷ Связывает groupID с объемом в резервуаре

   public ContainerSystem(int containerCount) {
      group = new int[containerCount];
      amount = new double[containerCount];
      for (int i=0; i<containerCount; i++) {
         group[i] = i; ❸ groupID i-го резервуара равен i
      }
   }

   public double getAmount(int containerID) {
      final int groupID = group[containerID];
      return amount[groupID];
   }
```

Метод `getAmount` прямолинеен (и очень похож на реализацию из `Memory3`), потому что он предоставляет функциональность только для чтения. Теперь рассмотрим

первый изменяющий метод `addContainer`, который возвращает новую систему с одним дополнительным резервуаром.

Так как два массива объявлены `final`, их необходимо инициализировать в конструкторе. Позднее тот же конструктор будет использовать другие изменяющие методы, `addWater` и `connect`, поэтому будет удобно передать ему два параметра:

- Копируемая существующая система.
- Новое количество резервуаров. Метод `addContainer` использует этот параметр для увеличения количества резервуаров на 1, тогда как другие изменяющие методы оставляют его неизменным.

В листинге 8.8 приведен метод `addContainer` и вспомогательный конструктор.

Листинг 8.8. Immutable: метод addContainer и вспомогательный конструктор

```
public ContainerSystem addContainer() {
   final int containerCount = group.length;
   ContainerSystem result =
      new ContainerSystem(this, containerCount + 1); ❶ Вызов приватного конструктора
   result.group[containerCount] = containerCount;
   return result;
}
private ContainerSystem(ContainerSystem old, int length) {
   group = Arrays.copyOf(old.group, length); ❷ Эффективный способ копирования массива
   amount = Arrays.copyOf(old.amount, length);
}
```

Затем метод `addWater` также должен создать совершенно новую систему резервуаров с обновленными количествами воды. Если есть вода для добавления, он вызовет приватный конструктор из предыдущего листинга, а затем обновит объем в предыдущей группе:

Листинг 8.9. Immutable: метод addWater

```
public ContainerSystem addWater(int containerID, double amount) {
   if (amount == 0) ❶ Новая система не нужна!
      return this;

   ContainerSystem result =
      new ContainerSystem(this, group.length); ❷ Вызов приватного конструктора
   int groupID = group[containerID],
      groupSize = groupSize(groupID);
   result.amount[groupID] += amount / groupSize;
   return result;
}
```

Наконец, метод `connect` также создает систему резервуаров, для чего он использует приватный конструктор, а затем соединяет два резервуара, объединяя их группы. Его исходный код находится в репозитории (https://bitbucket.org/mfaella/exercisesinstyle).

8.5. А ТЕПЕРЬ СОВСЕМ ДРУГОЕ

В этом разделе будет представлена другая ситуация, требующая методов, представленных в этой главе. Класс `Repository<T>` представляет *репозиторий* — контейнер фиксированного размера, который хранит свои элементы в индексируемых ячейках как массив. Его встроенная операция меняет местами содержимое двух ячеек. Естественно, в контексте этой главы мы стараемся сделать класс потокобезопасным, чтобы сразу несколько потоков могли легко одновременно обращаться к репозиториям и работать с ними.

Класс должен предоставлять следующий конструктор и методы:

- `public Repository(int n)` — создает репозиторий из `n` ячеек, изначально содержащих `null`;
- `public T set(int i, T elem)` — вставляет объект `elem` в `i`-ю ячейку и возвращает объект, который до этого находился в ней (или `null`);
- `public void swap(int i, int j)` — меняет местами содержимое ячеек `i` и `j`.

Как я уже говорил, прежде чем браться за реализацию класса, необходимо прояснить его политику конкурентности. Напомню, что такая политика определяет, какие операции могут выполняться параллельно, а какие должны быть взаимоисключающими.

Так как разные репозитории не содержат общих данных, простейшей политикой конкурентности, гарантирующей потокобезопасность, будет политика *уровня объекта*: одна блокировка на репозиторий, по которой синхронизируются все методы. Если несколько потоков используют репозиторий одновременно, такая политика может привести к замедлению выполнения, потому что все операции с одним репозиторием, даже операции с разными индексами, должны захватывать одну блокировку.

Более либеральная и эффективная политика конкурентности запрещает одновременный доступ по одному индексу, но разрешает всем остальным операциям выполняться одновременно. Ее можно описать следующим образом:

- Два вызова `set` для одного индекса должны быть сериализованы.
- Два вызова `swap`, которые используют хотя бы один общий индекс, должны быть сериализованы.
- Вызов `swap(i, j)` и вызов `set` для индекса `i` или `j` должны быть сериализованы.
- Все остальные операции могут выполняться параллельно.

Эта политика требует блокировки для каждой ячейки в репозитории, включая пустые ячейки (содержащие `null`). То есть классу понадобится один дополнительный объект для каждой ячейки, который будет использоваться в качестве

монитора этой ячейки. Элементы и мониторы могут храниться в двух списках `ArrayList`:

```
public class Repository<T> {
   private final List<T> elements;
   private final List<Object> monitors;

   public Repository(int size) {
      elements = new ArrayList<>(size);
      monitors = new ArrayList<>(size);
      for (int i=0; i<size; i++) {
         elements.add(null);          ❶ Списки должны быть заполнены до того, как вы
         monitors.add(new Object());    сможете вызывать get и set
      }
   }
```

Метод `set` просто захватывает монитор ячейки, в которую осуществляется запись:

```
public T set(int i, T elem) {
   synchronized (monitors.get(i)) {
      return elements.set(i, elem);
   }
}
```

Метод `swap` захватывает мониторы двух ячеек, участвующих в перестановке, в порядке возрастания индексов для предотвращения взаимных блокировок:

```
public void swap(int i, int j) {
   if (i == j) return;
   if (i > j) { ❷ Убеждается, что i — меньший индекс
      int temp = i;
      i = j;
      j = temp;
   }
   synchronized (monitors.get(i)) { { ❸ Захватывает мониторы в порядке индексов
      synchronized (monitors.get(j)) {
         elements.set(i, elements.set(j, elements.get(i)));
         ❹ В этой строке используется тот факт, что List.set
            возвращает значение, ранее находившееся в этой позиции
      }
   }
}
```

Отметим, что при таком подходе вы разрешаете разным потокам читать и даже изменять `ArrayList` параллельно — при условии, что они используют разные индексы. Но `ArrayList` не является потокобезопасным классом. Этот код неправилен? Если внимательно прочитать документацию `ArrayList`, вы поймете, что вызывающей стороне достаточно сериализовать только *структурные изменения* (такие, как вызов `add`), а параллельные вызовы `get` и `set` с разными индексами допустимы.

8.6. РЕАЛЬНЫЕ СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

В этой главе мы обсудили, как сделать резервуары потокобезопасными, чтобы множественные потоки могли взаимодействовать с ними, не требуя, чтобы клиентский код явно обрабатывал синхронизацию. Но почему мы решили пойти на все хлопоты по рефакторингу кода в целях сделать его потокобезопасным? Однопоточная версия работает вполне нормально. Для ответа на этот вопрос, рассмотрим несколько сценариев использования, в которых конкурентность не только полезна, но и критична.

- Вы любите шахматы, но в то же время являетесь талантливым программистом. Ради развлечения и практики вы решаете написать на Java программу для игры в шахматы с компьютером. Через несколько партий вы осознаете, что программа великолепна, и решаете поделиться ею с миром. Вы превращаете ее в сервис, чтобы компьютер мог состязаться с разными пользователями. В ситуации с множеством партий возможны два пути: либо ставить пользователей в очередь и обрабатывать их последовательно, либо использовать конкурентность и обрабатывать несколько игроков одновременно. Во втором варианте может потребоваться специальное оборудование (например, многоядерные машины).

- Приложения, операционные системы, сетевые устройства, базы данных — практически все постоянные сервисы в вычислительной системе — создают журналы. Такие журналы не генерируются для развлечения: в хорошо управляемых организациях их содержимое анализируется в пакетном режиме или в реальном времени для снижения рисков. Базовый процесс анализа состоит из разбора журнальных файлов, выявления важных закономерностей или аномалий, генерирования сводной статистики, отчетов и оповещений. Стандартная схема эффективной работы с большими файлами журналов — парадигма *отображения-свертки* (map-reduce). Как нетрудно догадаться, эта схема состоит из двух шагов: отображения и свертки. Шаг отображения позволяет системе анализа журналов обрабатывать независимые фрагменты журнальных данных одновременно (часто в распределенной сети) и генерировать промежуточные результаты. На шаге свертки собираются результаты и вычисляются итоговые данные. Фреймворк fork-join, упоминавшийся в начале этой главы, является разновидностью этой идеи, адаптированной для отдельных многоядерных архитектур.

- Каждый, кто жил в Великобритании, знает, насколько популярен в этой стране футбол. Воскресным днем можно разделить людей на две категории: тех, кто пьет пиво, и тех, кто его не пьет. Ко второй категории относятся футболисты и дети (хочется надеяться). Заметив эту страсть к спорту, вы решаете создать платформу для рассылки новостей со спортивных каналов среди подписчиков. Каналы генерируют потоки данных и помещают их в контейнерную структуру данных, а клиенты подписчиков запрашивают данные у контейнера. Потокобезопасный контейнер новостей позволя-

ет производителям и потребителям данных работать в разных потоках, чтобы ваши клиенты могли поднять пинту за любимую команду раньше соседей.

- Программа, которая способна выполнять только одну операцию, не обращая внимания на окружающий мир, редко приносит пользу. Напротив, реальные программы часто ожидают завершения операции ввода / вывода от внешнего источника (например, файла или сетевого подключения). Многопоточность позволяет программе, обращенной к пользователю, нормально реагировать на действия пользователя во время ожидания медленных периферийных операций. Например, представьте однопоточный браузер, который зависает на время загрузки файла из сети. Как вы думаете, сколько пользователей будет у такого браузера? В лучшем случае один — его создатель.

8.7. ПРИМЕНИМ ИЗУЧЕННОЕ

Упражнение 1

Следующий подкласс `Thread` увеличивает все элементы массива целых чисел на 1. Как видно из листинга, все экземпляры этого класса совместно используют массив.

```
class MyThread extends Thread {
   private static int[] array = ... ❶ Некоторое исходное значение

   public void run() {
      _____1_____                    ❷ Заполнитель
      for (int i=0; i<array.length; i++) {
         _____2_____
         array[i]++;
         _____3_____
      }
      _____4_____
   }
}
```

Программа создает два экземпляра `MyThread` и запускает их как два параллельных потока; предполагается, что каждый элемент массива будет увеличен на 2. Какая из следующих операций вставки обеспечит правильное выполнение программы, исключив все состояния гонки? (Правильных вариантов может быть несколько.)

```
a) 1 = "synchronized (this) {"     4 = "}"
b) 1 = "synchronized {"            4 = "}"
c) 1 = "synchronized (array) {"    4 = "}"
d) 2 = "synchronized (this) {"     3 = "}"
e) 2 = "synchronized (array) {"    3 = "}"
f) 2 = "synchronized (array[i]) {" 3 = "}"
```

Упражнение 2

Спроектируйте потокобезопасный класс `AtomicPair`, который содержит два объекта и предоставляет следующие методы:

```
public class AtomicPair<S,T> {
   public void setBoth(S first, T second);
   public S getFirst();
   public T getSecond();
}
```

При этом должна соблюдаться следующая политика конкурентности: вызов `setBoth` является атомарной операцией, а именно: если поток вызывает `setBoth(a,b)`, все последующие вызовы `getFirst` и `getSecond` должны видеть обновленные значения.

Упражнение 3

В простой социальной сети у каждого пользователя есть множество друзей, причем дружеские отношения симметричны. Реализация базируется на следующем классе:

```
public class SocialUser {
   private final String name;
   private final Set<SocialUser> friends = new HashSet<>();

   public SocialUser(String name) {
      this.name = name;
   }
   public synchronized void befriend(SocialUser other) {
      friends.add(other);
      synchronized (other) {
         other.friends.add(this);
      }
   }
   public synchronized boolean isFriend(SocialUser other) {
      return friends.contains(other);
   }
}
```

К сожалению, когда несколько потоков устанавливают дружеские отношения одновременно, система иногда виснет и ее приходится перезапускать. Почему это происходит? Как исправить проблему с помощью рефакторинга `SocialUser`?

Упражнение 4

Рассмотрим изменяемый класс `Time`, представляющий время суток в часах, минутах и секундах:

- `public void addNoWrapping(Time delta)` прибавляет задержку к текущему времени с достижением максимума за одну секунду до полуночи (23:59:59);

- `public void addAndWrapAround(Time delta)` добавляет задержку к текущему времени со сбросом значения в полночь;
- `public void subtractNoWrapping(Time delta)` вычитает задержку из текущего времени с достижением минимума в точке 00:00:00;
- `public void subtractAndWrapAround(Time delta)` вычитает задержку из текущего времени со сбросом значения по необходимости.

Преобразуйте этот API в неизменяемую версию и реализуйте ее.

ИТОГИ

- Хорошо продуманная *политика конкурентности* — важнейший фактор потокобезопасности.
- Главные враги потокобезопасности — состояния гонки и взаимные блокировки.
- Для предотвращения взаимных блокировок используется глобальная блокировка или политика упорядоченных блокировок.
- В отличие от неявных блокировок, явные блокировки можно захватывать и освобождать в любом порядке.
- Неизменяемость — альтернативный путь к потокобезопасности.

ОТВЕТЫ НА ВОПРОСЫ И УПРАЖНЕНИЯ

Неожиданный вопрос 1

Пользователи класса должны знать только, является ли класс потокобезопасным. Остальные составляющие политики конкурентности предназначены для создателей реализации класса. Однако на практике политика конкурентности может представлять интерес для пользователя, который хочет оценить скорость выполнения класса.

Неожиданный вопрос 2

Взаимная блокировка невозможна, если блокировки *реентерабельны* (то есть если поток может повторно захватить блокировку, которая ему уже принадлежит). В Java как явные, так и неявные блокировки реентерабельны. В других фреймворках (например, в мьютексах Posix) блокировки могут быть нереентерабельными и поток может создать взаимную блокировку при попытке повторно захватить уже принадлежащую ему блокировку.

Неожиданный вопрос 3

Если исключение выдается из блока `synchronized`, монитор автоматически освобождается. С другой стороны, необходимо явно освободить `ReentrantLock`. Именно по этой причине операция `unlock` обычно помещается в секцию `finally` блока `try...catch`, чтобы она гарантированно выполнялась при любых обстоятельствах.

Неожиданный вопрос 4

Метод упорядоченной блокировки предотвращает взаимные блокировки, потому что запрос блокировок в фиксированном глобальном порядке предотвращает возможность зацикливания.

Неожиданный вопрос 5

Неизменяемые классы всегда потокобезопасны, потому что их объекты можно только читать, а параллельные операции чтения со стороны нескольких потоков не создают проблем безопасности. Методы, создающие новые объекты, могут применять их изменяемые локальные переменные, потому что они находятся в стеке и не используются совместно с другими потоками.

Упражнение 1

Правильные варианты — (c) и (e). Оба они гарантируют, что если поток выполняет `array[i]++`, другой поток не сможет выполнить ту же команду даже с другим `i`. Более того, (c) полностью сериализует потоки: один цикл `for` отрабатывает полностью до того, как другой цикл начинает работу.

Варианты (a) и (d) не обеспечивают взаимоисключающего выполнения, потому что два потока будут синхронизироваться по двум разным мониторам. Варианты (b) и (f) приводят к ошибкам компиляции, потому что блок `synchronized` должен указать объект, предоставляющий монитор (а `array[i]` объектом не является).

Упражнение 2

Чтобы соблюдать политику конкурентности, достаточно использовать блоки `synchronized` во всех трех методах с блокировкой по одному монитору. Как объяснялось в этой главе, лучше использовать для синхронизации приватный объект вместо синхронизации по `this`, даже если последний вариант позволит заменить блоки `synchronized` более компактным модификатором метода.

```
public class AtomicPair<S,T> {
   private S first;
   private T second;
   private final Object lock = new Object(); ❶ Предоставляет приватный монитор

   public void setBoth(S first, T second) {
```

```
      synchronized (lock) {
         this.first = first;
         this.second = second;
      }
   }
   public S getFirst() {
      synchronized (lock) {
         return first;
      }
   }
   ... ❷ Метод getSecond аналогичен
}
```

Размещение одной команды `return` в блоке `synchronized` выглядит странно, но это необходимо по соображениям как *взаимоисключающего выполнения*, так и *видимости*. Во-первых, вызовы `getFirst` и `getSecond` не должны происходить на середине выполнения тела `setBoth`. Во-вторых, без блока `synchronized` потоки, вызывающие `getFirst`, не будут гарантированно видеть обновленное значение `first`. Кстати, объявление `first` и `second` с ключевым словом `volatile` решит вторую проблему (видимость), но не первую (взаимоисключающее выполнение).

Упражнение 3

Класс `SocialUser` может породить взаимную блокировку, если один поток вызывает `a.befriend(b)`, а другой поток одновременно вызовет `b.befriend(a)` для двух объектов `SocialUser` `a` и `b`. Чтобы избежать этого риска, можно воспользоваться методом упорядоченной блокировки, который основан на назначении каждому объекту уникального идентификатора:

```
public class SocialUserNoDeadlock {
   private final String name;
   private final Set<SocialUserNoDeadlock> friends = new HashSet<>();
   private final int id;
   private static final AtomicInteger instanceCounter = new AtomicInteger();

   public SocialUserNoDeadlock(String name) {
      this.name = name;
      this.id = instanceCounter.incrementAndGet();
}
```

Далее метод `befriend` избегает взаимных блокировок, запрашивая две блокировки в порядке увеличения идентификаторов:

```
public void befriend(SocialUserNoDeadlock other) {
   Object firstMonitor, secondMonitor;
   if (id < other.id) {
      firstMonitor = this;
      secondMonitor = other;
   } else {
      firstMonitor = other;
```

```
      secondMonitor = this;
   }
   synchronized (firstMonitor) {
      synchronized (secondMonitor) {
         friends.add(other);
         other.friends.add(this);
      }
   }
}
```

Упражнение 4

Чтобы преобразовать изменяемый API в неизменяемый, обеспечьте, чтобы каждый изменяющий метод возвращал новый объект класса. Также полезно объявить все поля с ключевым словом `final`. Все остальное — простая арифметика, необходимая для переноса переполнений из секунд в минуты, а из минут в часы.

```
public class Time {
   private final int hours, minutes, seconds;

   public Time addNoWrapping(Time delta) {
      int s = seconds, m = minutes, h = hours;
      s += delta.seconds;
      if (s > 59) ❶ Переполнение по секундам: перенос в минуты
         s -= 60;
         m++;
      }
      m += delta.minutes;
      if (m > 59) { ❷ Переполнение по минутам: перенос в часы
         m -= 60;
         h++;
      }
      h += delta.hours;
      if (h > 23) { ❸ Переполнение по часам: перенос до максимума
         h = 23;
         m = 59;
         s = 59;
      }
      return new Time(h, m, s); ❹ Возвращает новый объект
   }
```

Остальной код класса можно найти в репозитории (https://bitbucket.org/mfaella/exercisesinstyle). В Java есть стандартный класс `java.time.LocalTime`, который предоставляет функциональность, сходную с функциональностью класса `Time`.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

- *Брайан Г., Пайерлс Т., Блох Д., Боубер Д., Холмс Д., Ли Д.* Java Concurrency на практике. — СПб.: Питер, 2020. — 464 с.: ил.

То, что нужно обязательно прочитать по теме конкурентности в Java. В книге рассматриваются самые разнообразные вопросы конкурентности, при этом техническая строгость удачно сочетается с захватывающим изложением. К сожалению, на момент написания этой книги материал еще не был обновлен высокоуровневыми средствами конкурентности, добавленными в JDK начиная с версии 7. (См. следующую книгу.)

- *Urma R.-G., Fusco M., and Mycroft A.* Modern Java in Action. Manning Publications, 2019.

  Подробный вводный курс о потоках данных. Отдельная глава посвящена конкурентным вычислениям с использованием потоков данных и фреймворка fork-join.

- *Блох Д.* Java. Эффективное программирование — 3-е изд. — М.; СПб.: Диалектика, 2018.

  Обычно я стараюсь рекомендовать разные книги в каждой главе, но для этой книги я сделал исключение, потому что она содержит столько полезной информации по самым разным темам. Глава 11 полностью посвящена конкурентности, а пункт 17 — неизменяемости в частности.

- *Anderson R. J. and Woll H.* Wait-Free Parallel Algorithms for the Union-Find Problem. 1991.

  Потокобезопасный класс резервуара, разработанный в этой главе, основан на версии `Speed1` — не самом эффективном представлении. В этой статье продемонстрировано создание потокобезопасной реализации намного более производительных деревьев — указателей на родителей `Speed3`, которая к тому же избавлена от ожидания. Потокобезопасность достигается за счет использования команды CAS вместо блокировок.

- *Okasaki C.* Purely Functional Data Structures. Cambridge University Press, 1998.

  Автор этой книги доработал свою кандидатскую диссертацию и превратил ее в подробный учебник по структурам данных с примерами на ML и Haskell.

9 Повторное использование

В этой главе:

- ✓ Расширенное представление программного продукта в более широком контексте.
- ✓ Использование обобщений для написания классов, пригодных для повторного использования.
- ✓ Использование и модификация изменяемых коллекторов для потоков данных.

В предыдущих главах мы разработали конкретные классы, которые решали конкретные задачи. Теперь представьте, что ваше решение потребовалось обобщить для более широкого спектра задач. Вы стремитесь отделить важнейшие аспекты задачи от всего второстепенного и разработать решение для всех задач, имеющих в основе ту же структуру. К сожалению, отделение существенного от второстепенного — не самое простое дело. В общих чертах вы должны постараться сохранить ключевую структуру, то есть часть, которая может принести пользу в других контекстах.

Последняя глава книги предполагает, что вы знакомы с основными концепциями обобщений (generics), включая ограниченные параметры-типы.

9.1. ОПРЕДЕЛЕНИЕ ГРАНИЦ

В первые десятилетия существования ОО-парадигмы возможность повторного использования считалась одним из доводов в ее пользу. Предполагалось, что от программиста потребуется лишь написать небольшие компоненты, подходящие

для повторного использования, и объединить их с другими, уже готовыми компонентами. Приблизительно за 50 лет практического опыта (первый ОО-язык Simula появился в 1967 году) некоторые ожидания оправдались, а некоторые оказались ошибочными.

Программисты постоянно используют готовые компоненты: они работают с библиотеками и фреймворками. Современная разработка в значительной мере ориентируется на веб-приложения, которые могут извлечь огромную пользу от стандартных сервисов, упакованных во фреймворк. С другой стороны, стоит вам выйти за границы вашего фреймворка и погрузиться в специфический код приложения, возможность повторного использования быстро уходит на задний план под давлением более насущных функциональных и нефункциональных факторов (правильности, производительности, времени выхода на рынок и т. д.).

В этой главе мы разработаем библиотеку объектов, которые ведут себя как резервуары с водой в отношении повторного использования. Как это часто бывает с библиотеками, главный вопрос: насколько общей она должна быть? Будет ли она расширяться от резервуаров с водой до резервуаров с маслом или до межгалактической сети планет, соединенных торговыми маршрутами? Чтобы облегчить выбор, рассмотрим пару сценариев, которые вы, возможно, захотите представить обобщенной сетью, и еще один сценарий, который вы с большой вероятностью представлять *не* захотите, потому что это заведет обобщение слишком далеко.

СЦЕНАРИЙ 1

Муниципальная водопроводная компания, использующая вашу библиотеку `Container`, сообщает, что общие объемы воды имеют расхождения, достигающие 0,0000001 литра воды в год.

Анализ показывает, что расхождения обусловлены ошибками округления в вычислениях с плавающей точкой. Чтобы исправить проблему, необходимо представить объемы воды рациональными числами с произвольно большими числителями и знаменателями (допустим, двумя объектами `BigInteger`).

Поддержка этого изменения относительно тривиальна: бизнес-логика остается неизменной, тип поля заменяется переменной-типом `T`, а тип `T` должен реализовать интерфейс, предоставляющий соответствующие арифметические операции.

СЦЕНАРИЙ 2

Социальная сеть хочет отслеживать общее количество лайков, полученных всеми наборами взаимосвязанных сообщений. Два сообщения считаются взаимосвязанными, если оба получили комментарий от одного пользователя.

На первый взгляд кажется, что сценарий 2 не имеет отношения к резервуарам с водой, но потом вы осознаете, что каждое *сообщение* можно сравнить с резервуаром. Два сообщения, получившие комментарии от одного человека, становятся соединенными. Вместо `addWater` сценарию понадобится метод для добавления одного или нескольких лайков к сообщению. Наконец, вместо `getAmount` понадобится метод, возвращающий общее количество лайков, которые были получены всеми сообщениями, соединенными с текущим.

Описанный сценарий не слишком отличается от сценария с резервуарами с водой: в обоих случаях объекты могут соединяться друг с другом, но по-настоящему важно множество прямо или косвенно соединенных объектов. Более того, в обоих случаях объекты обладают свойством, которое можно читать или обновлять локально, но результат обновления зависит от группы соединенных элементов.

С другой стороны, конкретные способы обновления локального свойства и его влияние на глобальное свойство немного отличаются. В следующих разделах вы увидите, как согласовать их в одном контракте. Но сначала рассмотрим третий сценарий расширения.

СЦЕНАРИЙ 3

Оператор мобильной связи хочет автоматизировать управление своей сетью антенн. Компания может соединять антенны друг с другом и хочет знать, сколько соединений потребуется для двух антенн (длина кратчайшего пути).

В этом сценарии все еще остаются элементы, которые могут находиться в постоянном соединении друг с другом. Но главное свойство, представляющее интерес, — расстояние между антеннами — связывает *два* заданных элемента, а его значение зависит от того, какие *прямые* подключения существуют в системе. В частности, данное значение не используется всей группой соединенных антенн. Следовательно, в этом сценарии необходимы принципиально другие представления соединений и средства управления. Поддержка этого сценария сделает код настолько обобщенным, что его адаптация для конкретного сценария потребует больших усилий, чем написание конкретного решения с самого начала.

Мы разработаем обобщенную реализацию резервуаров с водой, которая может быть адаптирована для сценариев 1 и 2, но не для сценария 3.

9.2. ОБЩАЯ СТРУКТУРА

Сначала формализуем важнейшие аспекты обобщенного резервуара в интерфейсе:

1. Обобщенный резервуар обладает атрибутом некоторого типа `V` (`Value`). Клиенты могут прочитать атрибут или обновить его локально для резервуара, но

реальный результат обновления зависит от группы соединенных резервуаров. Для резервуаров с водой `V = Double`.

2. Клиенты могут соединять обобщенные резервуары друг с другом.

БИНАРНЫЙ МЕТОД

Бинарным методом называется метод класса, который получает в аргументе другой объект того же класса (как метод `connectTo` резервуаров). Типичные примеры — методы проверки равенства и сравнения двух объектов. В Java они соответствуют методу `equals` класса `Object` и методу `compareTo` интерфейса `Comparable`. Система типов популярных языков программирования (таких, как Java и C#) не позволяет отразить ограничение, согласно которому все подклассы заданного класса или интерфейса должны содержать бинарный метод заданной формы. Иначе говоря, не допускается использование записи следующего вида:

```
public interface Comparable {
   int compareTo(thisType other);
}
```

где `thisType` — вымышленное ключевое слово, которое представляет класс, реализующий этот интерфейс.

Как следствие, в Java для таких методов были приняты два решения:

- Объявлять параметр метода `equals` с типом `Object`. На стадии выполнения субклассы будут проверять, относится ли аргумент к правильному типу.
- Для случая `Comparable` разработчики языка решили проблему на уровне обобщений. Снабдить интерфейс параметром-типом `T` и объявить параметр `compareTo` с типом `T`. Это решение повышает безопасность типов, но допускает такие непредусмотренные варианты использования, как пресловутая конструкция

  ```
  class Apple implements Comparable<Orange> { ... }
  ```

В C# ситуация выглядит аналогично и решается похожим образом, не считая того, что равенство поддерживается по двум направлениям: методом `Equals` из класса `Object` с параметром `Object` и интерфейсом `IEquatable<T>`.

На концептуальном уровне эти два аспекта не зависят друг от друга, так что их можно представить двумя разными интерфейсами (допустим, `Attribute` и `Connectable`). Но поскольку они все время будут использоваться вместе, мы выделим их в один интерфейс с именем `ContainerLike`.

Наличие атрибута типа `V` (аспект 1) просто преобразуется в расширение интерфейса двумя методами следующего вида:

```
public interface ContainerLike<V> {
   V get() ❶ Обобщенное представление getAmount
   void update(V value) ❷ Обобщенное представление addWater
   ...
}
```

В API не отражено, что результат обновления зависит от группы соединенных резервуаров. Что касается одного резервуара, обобщенного с другим (аспект 2), выбрать правильную сигнатуру метода будет сложнее. Наше желание соединять обобщенные резервуары с другими обобщенными резервуарами *того же типа* не получится точно отразить в интерфейсе Java. В теории языков программирования это соответствует хорошо известной проблеме *бинарного метода*.

Рассмотрим пару альтернативных решений для сигнатуры `connectTo`:

- `void connectTo(Object other)` аналогична сигнатуре `Object::equals`. Вы просто отказываетесь от безопасности типов и не позволяете компилятору хоть как-то помочь вам. Тело `connectTo` должно проверить динамический тип аргумента, а затем выполнить понижающее преобразование, прежде чем что-то делать с ним.
- С `void connectTo(ContainerLike<V> other)` вы получите некоторую, но недостаточную помощь от компилятора. `connectTo` примет любой обобщенный резервуар, который содержит атрибут с таким же типом, как данный обобщенный резервуар. Чтобы метод `connectTo` смог выполнить свою работу, он все еще будет вынужден преобразовать свой аргумент в более конкретную форму, которая откроет свое представление для соединения резервуаров.

НЕОЖИДАННЫЙ ВОПРОС 1

Класс `Employee` представляет работника. Стоит ли включить в него метод `public boolean equals(Employee e)`? Почему?

Лучше добавить решение, выбранное Java для `Comparable`: ввести дополнительный параметр-тип `T`, представляющий тип объектов, с которыми могут соединяться обобщенные резервуары, и надеяться, что этот параметр будет использоваться правильно. Мы не можем потребовать, чтобы параметр `T` относился к тому же классу, который реализует интерфейс, но можем быть уверены, что это он относится к классу (возможно, другому), реализующему тот же интерфейс.

Листинг 9.1. ContainerLike: обобщенный интерфейс для резервуаров

```
public interface ContainerLike<V, T extends ContainerLike<V,T>{}> {
   V get();
```

```
    void update(V val);
    void connectTo(T other);
}
```

Предполагается, что `ContainerLike` будет использоваться примерно так:

```
class MyContainer implements ContainerLike<Something, MyContainer> { ... }
```

Предполагаемый вариант использования `Comparable`:

```
class Employee implements Comparable<Employee> { ... }
```

Если класс придерживается этой схемы (то есть использует в `T` самого себя), его методу `connectTo` не нужно выполнять понижающее преобразование, потому что он получит в аргументе объект, уже относящийся к одному типу с `this`. Именно это должен сделать метод для объединения групп.

РЕАЛИЗАЦИЯ ОБОБЩЕНИЙ В JAVA

В Java обобщения реализуются через механизм *стирания* (erasure). Это означает, что компилятор использует параметры-типы для выполнения более выразительной проверки типов, а затем отбрасывает их. Параметры-типы не включаются в байт-код и не поддерживаются JVM.

Такая стратегия реализации ограничивает действия с обобщениями. Например, вы не сможете создать экземпляр параметра-типа конструкцией `new T()` или сравнить тип выражения на стадии выполнения с параметром-типом конструкцией `exp instanceof T`.

РЕАЛИЗАЦИЯ ОБОБЩЕНИЙ В C++ И C#

В отличие от Java, в C++ и C# обобщения реализуются через механизм *воплощения* (reification), при котором каждая конкретная версия обобщенного класса (например, `List<String>`) преобразуется в конкретный класс на стадии компиляции (C++) или выполнения (C#).

Разные версии одного класса могут совместно использовать (или не использовать) общий код в зависимости от аргументов-типов, а также разумности компилятора и исполнительной среды.

Такой выбор реализации позволяет применять параметры-типы во многих местах, где уместен обычный тип, но это может создать лишние затраты в контексте дублирования кода (объектов) или ресурсов, необходимых для поддержания информации типа на стадии выполнения.

НЕОЖИДАННЫЙ ВОПРОС 2

Если T — параметр-тип, можно ли создать массив типа T в Java? А как насчет C#?

9.2.1. API атрибутов

Затем необходимо определить интерфейс, представляющий поведение атрибута при обновлении его значения методом `update` и особенно при соединении обобщенных резервуаров методом `connectTo`.

Чтобы ограничить уровень обобщения, который должен поддерживаться, сделаем некоторые предположения:

1. При локальном обновлении свойства можно вычислить новое значение группы, основанное только на текущем значении группы и новом локальном значении. Другими словами, значение группы должно содержать достаточно информации для выполнения необходимого обновления.
2. При объединении двух групп новое значение группы может быть получено на основании только двух старых значений группы.

Сравните эти предположения с двумя обобщенными сценариями, представленными в начале главы. Со сценарием 1 проблем нет, потому что это простая вариация на тему базовых резервуаров с водой. В сценарии 2 интерес представляет общее количество лайков, накопленных всеми соединенными сообщениями, — значение группы. Локальное обновление свойства означает добавление лайков к конкретному сообщению. В результате значение группы вырастет на одну величину в соответствии с предположением 1.

Проверим, выполняется ли предположение 2. Если две группы сообщений соединены (то есть когда пользователь, оставивший комментарий в первой группе, оставляет комментарий во второй группе), значения групп можно объединить суммированием. Никакая дополнительная информация для вычисления нового значения группы не нужна, что подтверждает предположение 2.

Руководствуясь этими предположениями, мы спроектируем API для определения поведения атрибута, присутствующего во всех резервуарах. Чтобы избежать путаницы между локальным значением и значением группы, назовем последнее *сводным значением* группы. Прежде всего, тип `V` локального значения следует отличать от типа `S` сводного значения. Иногда они будут совпадать: например, в сценарии 2 в обоих случаях будет использоваться `Integer`, потому что значения представляют счетчики лайков. В случае резервуаров с водой типы будут разными (подробнее в разделе 9.5).

Теперь введем интерфейс `Attribute<V,S>` для операций, необходимых резервуарам для выполнения их контрактных обязательств, описанных ранее как аспекты 1 и 2:

- Новому обобщенному резервуару необходимо инициализировать сводное значение группы (метод `seed`).
- Методу `get` обобщенного резервуара необходим метод для распаковки сводного значения в локальное значение типа `V` (метод `report`).
- Метод `update` обобщенных резервуаров должен обновить свое сводное значение (метод `update`).
- Методу `connectTo` необходим метод, объединяющий два сводных значения (метод `merge`).

В итоге будет получен интерфейс, сходный с листингом 9.2. В табл. 9.1 приведена сводка зависимостей между методами обобщенных резервуаров и методами интерфейса `Attribute`.

Листинг 9.2. Attribute: обобщенный интерфейс для свойства резервуаров

```
public interface Attribute<V,S> {
   S seed();                              ❶ Предоставляет исходное сводное значение
   void update(S summary, V value);       ❷ Обновляет сводное значение
   S merge(S summary1, S summary2);       ❸ Объединяет два сводных значения
   V report(S summary);                   ❹ Распаковывает сводное значение
}
```

Таблица 9.1. Отношения между методами обобщенных резервуаров и методами интерфейса Attribute

Метод обобщенного резервуара	Метод или свойство
Конструктор	`seed`
`get`	`report`
`update`	`update`
`connectTo`	`merge`

Обратите внимание, что сам объект `Attribute` не имеет состояния: он не содержит значения атрибута. Обобщенный резервуар сам может выбрать между хранением отдельного объекта типа `S` (для сводного значения) или `V` (для кэшированного локального значения).

Интерфейс `Attribute` безусловно напоминает интерфейс, введенный в главе 8 для получения итогов потоковой операции в одном результате. В следующем разделе мы воспользуемся им для краткого описания потоков данных и изменяемых коллекторов.

9.2.2. Изменяемые коллекторы

Потоки данных дополняют коллекции и предоставляют удобный компонуемый фреймворк для последовательных операций. Я кратко представлю его, а затем

сосредоточусь на конкретном аспекте, актуальном для примера с резервуарами, — *изменяемых коллекторах*. За более подробным описанием фреймворка обращайтесь к ресурсам, перечисленным в конце главы.

Стандартные коллекции преобразуются в потоки методом `stream`. В свою очередь, объекты потоков поддерживают разнообразные *промежуточные* и *терминальные* операции. Промежуточные операции преобразуют поток данных в другой поток данных того же или другого типа. Терминальные операции порождают результат, который не является потоком данных. Одна из простейших терминальных операций `forEach` выполняет фрагмент кода для каждого элемента потока данных. Например, следующий фрагмент выведет все строки из списка строк `listOfStrings`:

```
listOfStrings.stream().forEach(s -> System.out.println(s));
```

Аргумент `forEach` — это объект типа `Consumer`. Так как последний является функциональным интерфейсом, его экземпляр можно создать в удобном синтаксисе лямбда-выражений. Добавим промежуточную операцию для вывода только тех строк, длина которых превышает 10 символов:

```
listOfStrings.stream().filter(s -> s.length() > 10)
                      .forEach(s -> System.out.println(s));
```

Иногда нужно собрать результат серии операций потоков данных в новую коллекцию. Для этого можно воспользоваться терминальной операцией `collect`, которая получает изменяемый коллектор в виде объекта типа `Collector`. Статические фабричные методы класса `Collector` предоставляют типичные коллекторы. Например, следующий фрагмент собирает отфильтрованные строки в список:

```
List<String> longStrings =
             listOfStrings.stream().filter(s -> s.length() > 10)
                                   .collect(Collectors.toList());
```

Другие стандартные коллекторы позволяют поместить результат во множество или карту. Вы можете создавать собственные коллекторы, реализуя интерфейс `Collector`. Чтобы понять разные составляющие интерфейса `Collector`, подумайте, как бы вы поступили с обычной коллекцией, если бы потребовалось обобщить ее в один изменяемый результат. Вероятно, вы бы создали некий объект `summary`, инициализировали его неким значением по умолчанию, а затем обновляли для каждого элемента в коллекции. А после перебора всех элементов объект `summary` преобразовали бы к другому типу — назовем его `result`.

```
Collection<V> collection = ...
Summary summary = new Summary();      ❶ Исходное значение summary
for (V value: collection) {
   summary.update(value);             ❷ Обновляет summary значением value
}
Result result = summary.toResult();   ❸ Преобразует summary в result
```

Интерфейс `Collector` абстрагирует эти три шага, а также еще один шаг, необходимый для *параллельных* коллекторов. Если цикл, перебирающий все значения, закреплен за несколькими потоками (то есть каждый поток занимается своим подмножеством значений), каждый поток строит собственное сводное значение, и со временем понадобится *объединить* эти сводки, прежде чем они смогут построить окончательный результат. Операция объединения — четвертая и последняя составляющая коллектора.

Обозначим тип сводного значения `S`, а тип итогового результата `R`. Можно ожидать, что интерфейс `Collector` будет содержать методы следующего вида:

```
S supply();                                  ❶ Исходное значение summary
void accumulate(S summary, V value);         ❷ Обновляет summary значением value
S combine(S summary1, S summary2);           ❸ Объединяет два сводных значения
R finish(S summary);                         ❹ Преобразует summary в result
```

Обратите внимание на сходство между воображаемым коллектором и интерфейсом `Attribute`, введенным ранее для абстрагирования уровня воды в резервуарах. Однако реальный интерфейс `Collector` вводит еще один уровень абстракции: каждый метод возвращает *объект* для выполнения соответствующей функции. Такое поведение соответствует принципам фреймворка потока данных и функционального программирования, на основе которого он строился. Возвращаемые типы всех четырех методов представляют собой *функциональные интерфейсы*, каждый из которых содержит один абстрактный метод. В табл. 9.2 перечислены характеристики этих четырех интерфейсов.

Таблица 9.2. Функциональные интерфейсы, используемые изменяемыми коллекторами. Пакет java.util.function содержит более 40 функциональных интерфейсов

Интерфейс	Тип абстрактного метода	Роль
`Supplier<S>`	`void → S`	Предоставляет исходное сводное значение
`BiConsumer<S,V>`	`(S, V) → void`	Обновляет сводное значение заданным
`BinaryOperator<S>`	`(S, S) → S`	Объединяет два сводных значения
`Function<S,R>`	`S → R`	Преобразует сводное значение в результат

Пятый метод используется для обозначения того, обладает ли коллектор двумя стандартными характеристиками:

- *Конкурентность* — поддерживает ли коллектор одновременное выполнение из нескольких потоков?
- *Порядок* — сохраняет ли коллектор порядок элементов?

Внутри перечисление с именем `Characteristics` предоставляет флаги, соответствующие этим признакам. В итоге мы получаем следующий набор методов:

```
public interface Collector<V,S,R> {
   Supplier<S> supplier();          ❶ Исходное значение summary
   BiConsumer<S,V> accumulator();   ❷ Обновляет summary значением value
   BinaryOperator<S> combiner();    ❸ Объединяет два сводных значения
   Function<S,R> finisher();        ❹ Преобразует summary в result
   Set<Characteristics> characteristics(); ❺ Является ли конкурентным,
                                             упорядоченным и т. д.
}
```

Использование функциональных интерфейсов позволяет легко организовать взаимодействие коллекторов с лямбда-выражениями и ссылками на методы — двумя удобными способами реализации функциональных интерфейсов. В следующем разделе я расскажу о ссылках на методы и опишу реализацию конкретного коллектора строк.

НЕОЖИДАННЫЙ ВОПРОС 3

Какую роль играет метод коллектора `combiner`? Когда он используется?

Пример: конкатенация строк

Рассмотрим специализированный коллектор, который объединяет последовательность строк в одну строку, используя объект `StringBuilder` в качестве временного сводного значения. Так как класс `StringBuilder` не является потокобезопасным, коллектор не будет конкурентным[1]. С другой стороны, порядок строк сохраняется, потому что конкатенация выполняется в определенном порядке. Такая структура удобна, потому что коллектор обладает именно такими характеристиками по умолчанию, так что метод `characteristics` может вернуть пустое множество.

Если бы не лямбда-выражения и ссылки на методы, потребовалось бы несколько анонимных классов для определения коллектора, а именно *пять*: внешний класс для самого коллектора и четыре внутренних класса для создания экземпляров соответствующих функциональных интерфейсов. Возьмем первый метод:

```
Collector<String,StringBuilder,String> concatenator =
   new Collector<>() { ❶ Внешний анонимный класс
      @Override
      public Supplier<StringBuilder> supplier() { ❷ Предоставляет исходное сводное
                                                    значение
```

[1] Чтобы получить параллельный коллектор, можно использовать `StringBuffer` вместо `StringBuilder` или добавить явную синхронизацию.

```
        return new Supplier<>() { ❸ Первый внутренний анонимный класс
           @Override
           public StringBuilder get() {
              return new StringBuilder();
           }
        };
     }
     ... ❹ Переопределение остальных четырех методов Collector
  };
```

ССЫЛКИ НА МЕТОДЫ...

...были добавлены в Java 8 как новая разновидность выражений, которая преобразует существующий метод или конструктор в экземпляр функционального интерфейса. В них используется синтаксис с двумя двоеточиями `::`. В простейшей форме ссылка на метод превращает метод экземпляра в подходящий интерфейс. Пример:

```
ToIntFunction<Object> hasher = Object::hashCode;
```

где `ToIntFunction<T>` — функциональный интерфейс, содержащий единственный метод `int applyAsInt(T item)`.

Такая ссылка также может указывать на метод конкретного объекта:

```
Consumer<String> printer = System.out::println;
```

Ссылки на методы могут использоваться со статическими методами и конструкторами.

Со ссылками на методы предыдущий фрагмент значительно упрощается. Поставщика можно передать по ссылке конструктору `StringBuilder`. Компилятор позаботится об упаковке конструктора в объект типа `Supplier<StringBuilder>`.

```
Collector<String,StringBuilder,String> concatenator = new Collector<>() {
   @Override
   public Supplier<StringBuilder> supplier() {
      return StringBuilder::new; ❶ Ссылка на конструктор
   }
   ... ❷ Переопределение остальных четырех методов Collector
};
```

Что еще лучше, класс `Collector` предоставляет статический метод `of`, который позволяет избавиться даже от предоставления внешнего анонимного класса, в результате чего мы приходим к следующему удобному решению. Ниже все четыре основных метода интерфейса предоставлены в виде ссылок на методы:

```
Collector<String,StringBuilder,String> concatenator =
   Collector.of(StringBuilder::new,        ❶ Поставщик (ссылка на конструктор)
```

```
StringBuilder::append,      ❷ Функция обновления
StringBuilder::append,      ❸ Функция объединения (другой метод append)
StringBuilder::toString);   ❹ Завершитель
```

Ссылки на методы не позволяют задать сигнатуру метода, на который вы ссылаетесь, просто по имени. Компилятор вычисляет сигнатуру по *контексту*, в котором используется ссылка на метод. Такой контекст должен определять конкретный функциональный интерфейс. Например, в предыдущем фрагменте ссылка на функцию обновления преобразуется в следующий метод из `StringBuilder`:

```
public StringBuilder append(String s)
```

потому что контекст требует `BiConsumer<StringBuilder,String>`. Возможно, вы заметили несоответствие: `append` возвращает значение, тогда как `BiConsumer` возвращает `void`. Компилятор не имеет ничего против — все выглядит так, словно вы вызываете метод, возвращающий значение, и игнорируете это значение. Краткая сводка этого правила совместимости приведена в табл. 9.3.

Таблица 9.3. Сравнение сигнатур и типов метода StringBuilder::append и функционального интерфейса BiConsumer (SB, StringBuilder)

	Метод	Целевой функциональный интерфейс
Сигнатура	`SB append(String s)`	`BiConsumer<SB,String>`
Тип	`(SB,String) → SB`	`(SB,String) → void`

СОВЕТ

Ссылку на метод *не* `void` можно присвоить функциональному интерфейсу `void`.

Перейдем к ссылке на метод функции слияния в этом фрагменте. Ее контекст требует `BinaryOperator<StringBuilder>`, то есть метода, получающего два объекта `StringBuilder` (включая `this`) и возвращающего другой объект `StringBuilder`. На эту роль может подойти другой метод `append` из класса `StringBuilder`:

```
public StringBuilder append(CharSequence seq)
```

Этот случай также требует преобразования, потому что метод получает `CharSequence`, тогда как целевой функциональный интерфейс ожидает `StringBuilder`. Такое преобразование допустимо, потому что `CharSequence` является супертипом по отношению к `StringBuilder`. Ситуация описана в табл. 9.4.

СОВЕТ

Ссылку на метод, получающий аргумент типа `T`, можно присвоить функциональному интерфейсу, метод которого ожидает получить подтип `T`.

Таблица 9.4. Сравнение сигнатур и типов метода StringBuilder::append и функционального интерфейса BinaryOperator (SB, StringBuilder)

	Метод	Целевой функциональный интерфейс
Сигнатура	`SB append(CharSequence seq)`	`BinaryOperator<SB>`
Тип	`(SB,CharSequence) → SB`	`(SB,SB) → SB`

Кстати, коллектор, очень сходный с этим конкатенатором, включен в JDK как объект, возвращаемый статическим методом `Collectors.joining()`.

НЕОЖИДАННЫЙ ВОПРОС 4

Можно ли присвоить ссылку на метод переменной типа `Object`?

9.2.3. Адаптация Attribute для функциональных интерфейсов

`Attribute` можно оснастить адаптером, аналогичным включенному в `Collector`, — статическим методом, который получает четыре функциональных интерфейса и преобразует их в объект с типом `Attribute`. С таким методом клиенты смогут создавать конкретные реализации `Attribute`, используя четыре лямбда-выражения или ссылки на методы, как это делалось с конкатенатором строк.

Метод-адаптер имеет следующую форму:

```
public static <V,S> Attribute<V,S> of(Supplier<S> supplier,
                                      BiConsumer<S,V> updater,
                                      BinaryOperator<S> combiner,
                                      Function<S,V> finisher) {
   return new Attribute<>() { ❶ Анонимный класс
      @Override
      public S seed() {
         return supplier.get();
      }
      @Override
      public void update(S summary, V value) {
         updater.accept(summary, value);
      }
      @Override
      public S merge(S summary1, S summary2) {
         return combiner.apply(summary1, summary2);
      }
      @Override
      public V report(S summary) {
         return finisher.apply(summary);
      }
   }; ❷ Конец анонимного класса
}
```

9.3. ОБОБЩЕННАЯ РЕАЛИЗАЦИЯ РЕЗЕРВУАРА

Теперь мы можем разработать обобщенную реализацию `ContainerLike`, которая будет управлять соединениями и группами, делегируя поведение свойства объекту типа `Attribute`. Эту реализацию можно создать на основе версии `Speed3` из главы 3, потому что она обладает лучшей общей скоростью выполнения и она будет полезным упражнением.

Для начала вспомним базовую структуру `Speed3`, основанную на родительском дереве указателя. Каждый резервуар представлен узлом дерева, и только корневые резервуары знают объем воды и размер своей группы. Объекты резервуаров содержат три поля, два из которых актуальны только для корневых резервуаров:

- объем воды, находящейся в группе (для корневых резервуаров);
- размер группы (для корневых резервуаров);
- родительский резервуар (или циклическая ссылка для корневых резервуаров).

Начало `Speed3` выглядит так:

```
public class Container {
   private Container parent = this; ❶ Изначально каждый резервуар является корнем
   private double amount;             своего дерева
   private int size = 1;
```

В обобщенной версии `UnionFindNode` поле `amount` заменяется объектом типа `S`, содержащим сводное значение группы, и объектом типа `Attribute`, содержащим методы для работы со сводными и текущими значениями. Поля и конструктор `UnionFindNode` приведены в листинге 9.3.

Листинг 9.3. UnionFindNode: поля и конструктор

```
public class UnionFindNode<V,S>
    implements ContainerLike<V,UnionFindNode<V,S>{}> {

   private UnionFindNode<V,S> parent = this; ❶ Изначально каждый узел является корнем
   private int groupSize = 1;

   private final Attribute<V,S> attribute; ❷ Содержит методы для работы с атрибутом
   private S summary;

   public UnionFindNode(Attribute<V,S> dom) {
      attribute = dom;
      summary = dom.seed();
   }
```

Методы `get` и `update` идентифицируют корень своего дерева (как в `Speed3`), а затем вызывают соответствующий метод атрибута для распаковки сводного значения и его обновления с учетом нового значения (листинг 9.4). Приватный вспомогательный метод `findRootAndCompress` отвечает за нахождение корня и свертку пути, ведущего к корню, для ускорения будущих вызовов.

Листинг 9.4. UnionFindNode: методы get и update

```
public V get() { ❶ Возвращает текущее значение атрибута
   UnionFindNode<V,S> root = findRootAndCompress();
   return attribute.report(root.summary);
}
public void update(V value) { ❷ Обновляет атрибут
   UnionFindNode<V,S> root = findRootAndCompress();
   attribute.update(root.summary, value);
}
```

Наконец, метод `connectTo` обеспечивает политику связывания по размеру (глава 3) и вызывает метод `merge` класса `Attribute` для объединения сводных значений двух соединяемых групп. Как и ожидалось, `connectTo` не нужно выполнять никакие преобразования типа со своим аргументом благодаря выбранной ранее выразительной сигнатуре.

Листинг 9.5. UnionFindNode: метод connectTo

```
public void connectTo(UnionFindNode<V,S> other) {
   UnionFindNode<V,S> root1 = findRootAndCompress(),
                      root2 = other.findRootAndCompress();

   if (root1 == root2) return;
   int size1 = root1.groupSize, size2 = root2.groupSize;
      ❶ Объединяет два сводных значения
   S newSummary = attribute.merge(root1.summary, root2.summary);

   if (size1 <= size2) { ❷ Политика связывания по размеру
      root1.parent = root2;
      root2.summary = newSummary;
      root2.groupSize += size1;
   } else {
      root2.parent = root1;
      root1.summary = newSummary;
      root1.groupSize += size2;
   }
}
```

На рис. 9.1 изображены три класса, представленные ранее. В совокупности они образуют обобщенный фреймворк для генерирования поведений, похожих на поведение резервуаров.

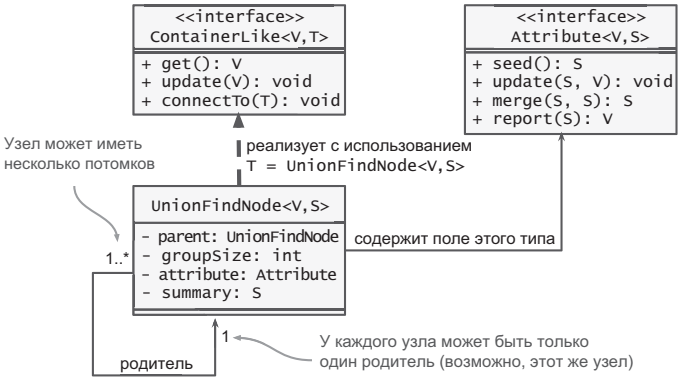


Рис. 9.1. Диаграмма классов UML обобщенного фреймворка для построения системы резервуаров с водой

9.4. ОБЩИЕ СООБРАЖЕНИЯ

Давайте ненадолго прервем поток кода и поразмыслим над процессом *обобщения* заданного набора функциональности в более широком контексте. Нужна четкая мотивация для обобщения кода или спецификаций. Возможно, вы представили превращение вашего решения в элегантный фреймворк... а может, вас охватил спортивный интерес! Если вы программируете для собственного удовольствия или для освоения нового языка, таких причин будет достаточно. Однако в профессиональной деятельности должны быть более веские причины для преобразования хорошего, но узкого решения в обобщенный фреймворк, который с большой вероятностью будет более медленным, запутанным и сложным в сопровождении. Хорошие бизнес-ориентированные причины сводятся к одному из пунктов следующего списка:

- Обобщенное решение может быть продуктом само по себе. Вы и ваши коллеги или руководители считают, что компания может выпустить обобщенное решение независимо как библиотеку или фреймворк, которые могут использоваться другими организациями.
- Обобщенное решение может относиться к другим функциям вашего продукта. Например, оно может заменить и унифицировать разные конкретные решения, являющиеся частью продукта.
- Обобщенное решение обеспечит эволюцию продукта. К этой мотивации следует относиться с осторожностью. Как упоминалось ранее, программисты и проектировщики склонны к излишнему техническому усложнению и обобщению. Девиз экстремального программирования YAGNI (раздел 1.7.2) отражает эту тенденцию и противостоит ей.

После того как мотивация будет четко сформулирована, следует определить один или несколько *сценариев использования*, которые текущая реализация или спецификация не обеспечивает, но должна обеспечивать в соответствии с вашей мотивацией. Именно это я сделал в начале главы, представив два целевых сценария и один сценарий, выходящий за границы обобщения.

Эти сценарии использования прокладывают путь к обобщенному API, часто в форме одного или нескольких интерфейсов. В случае резервуаров с водой этот анализ привел к появлению двух интерфейсов `ContainerLike` и `Attribute`.

Если вы начали с конкретной реализации, наступает момент для ее адаптации к обобщенным интерфейсам, спроектированным на предыдущем шаге. Именно это было сделано, когда мы начали с конкретного класса `Container` из версии `Speed3` (глава 3) и преобразовали его в обобщенный класс `UnionFindNode`. Теперь нужно написать дополнительный код (хочется надеяться, не слишком долгий) для восстановления исходной функциональности с использованием нового обобщенного фреймворка.

Об этом пойдет речь в следующем разделе.

9.5. ИСПОЛЬЗОВАНИЕ ГОТОВОГО КОДА [GENERIC]

В этом разделе я покажу, как задействовать конкретную реализацию резервуаров с их атрибутом уровня воды, используя обобщенную реализацию, разработанную в предыдущем разделе. Результат представит собой класс, который по поведению почти не будет отличаться от `Speed3`, не считая пары добавленных уровней абстракции. Такова цена обобщенной реализации, легко адаптируемой к широкому спектру условий.

9.5.1. Обновленный сценарий использования

Сценарий использования для конкретной реализации не будет полностью идентичен тому, который был задействован в остальных частях книги. Отличаться будут только имена двух методов: имена `getAmount` и `addWater` будут заменены общими именами `get` и `update`, предоставляемыми интерфейсом `ContainerLike`. В результате первые строки стандартного сценария использования примут следующий вид:

```
Container a = new Container();
Container b = new Container();
Container c = new Container();
Container d = new Container();

a.update(12.0); ❶ Метод update аналогичен addWater
d.update(8.0);
```

```
a.connectTo(b);
  ❷ Метод get аналогичен getAmount
System.out.println(a.get()+" "+b.get()+" "+c.get()+" "+d.get());
```

Желательный результат предыдущего фрагмента останется тем же, что и в исходном сценарии использования:

```
6.0 6.0 0.0 8.0
```

9.5.2. Проектирование конкретного атрибута

Каждый класс, основанный на `UnionFindNode`, должен скорректировать типы `V` и `S` и предоставить объект типа `Attribute<V,S>`. Для резервуаров с водой `V = Double`, потому что этот тип подходит для представления объемов воды. На первый взгляд может показаться, что сводное значение типа `S = Double` также сработает. В конце концов, разве сводное значение группы не должно быть общим объемом воды в группе? Казалось бы, объем воды на резервуар можно вычислить делением объема группы на ее размер, а результат сохранить в поле `groupSize` корневого узла. Но для объекта `Attribute` недоступен объект `UnionFindNode`, которому он принадлежит! Соответственно, он не может обратиться к полю `groupSize` этого объекта. Придется дублировать информацию размера группы и сохранять отдельную копию в сводном значении. Это еще одна цена, которую мы заплатим за обобщенный характер решения.

Вместо простого `S = Double` потребуется специальный класс, который будет представлять сводное значение группы. Назовем его `ContainerSummary`. Каждое сводное значение содержит общий объем воды в группе и размер группы. Кроме естественного конструктора с двумя параметрами, я добавлю конструктор по умолчанию (листинг 9.6). Это позволит позднее обращаться к нему по ссылке на метод (ладно, скажу: «ссылка на конструктор») и заполнить операцию инициализации интерфейса `Attribute`.

Листинг 9.6. ContainerSummary: поля и конструкторы

```
class ContainerSummary {
   private double amount;
   private int groupSize;

   public ContainerSummary(double amount, int groupSize) {
      this.amount = amount;
      this.groupSize = groupSize;
   }
   public ContainerSummary() { ❶ Конструктор по умолчанию
      this(0, 1); ❷ Вызывает другой конструктор с уровнем воды 0 и 1 резервуаром в группе
   }
```

Затем в листинге 9.7 приведены три метода, которые предоставляют остальные операции атрибутов.

Листинг 9.7. ContainerSummary: методы для работы со сводным значением

```
❶ Метод аналогичен addWater
public void update(double increment) {
   this.amount += increment;
}
❷ Используется при соединении двух резервуаров

public ContainerSummary merge(ContainerSummary other) {
   return new ContainerSummary(amount + other.amount,
                               groupSize + other.groupSize);
}
❸ Возвращает объем воды в резервуаре
   public double getAmount() {
   return amount / groupSize;
}
```

Наконец, статический метод `of` из интерфейса `Attribute` и четыре ссылки на методы могут использоваться для создания экземпляра объекта `Attribute`, который нужен `UnionFindNode`. Существует небольшое несоответствие между примитивным типом `double`, используемым методами `ContainerSummary`, и типом обертки `Double`, которую ожидает `Attribute`. Но не беспокойтесь: механизм автоупаковки(-распаковки) позаботится о том, чтобы вы использовали ссылки на методы, в которых задействованы примитивные типы, даже когда контекст потребует типов-оберток.

Теперь объект `Attribute` можно предоставить клиентам в качестве константы класса, то есть финального статического поля:

Листинг 9.8. ContainerSummary: поле Attribute

```
public static final Attribute<Double,ContainerSummary> ops =
   Attribute.of(ContainerSummary::new, ❶ Ссылка на конструктор по умолчанию
                ContainerSummary::update,
                ContainerSummary::merge,
                ContainerSummary::getAmount);
```

На рис. 9.2 изображена диаграмма классов UML для `ContainerSummary` и ее отношения с `Attribute`. Обратите внимание, как конструктор и три метода `ContainerSummary` соответствуют четырем методам интерфейса.

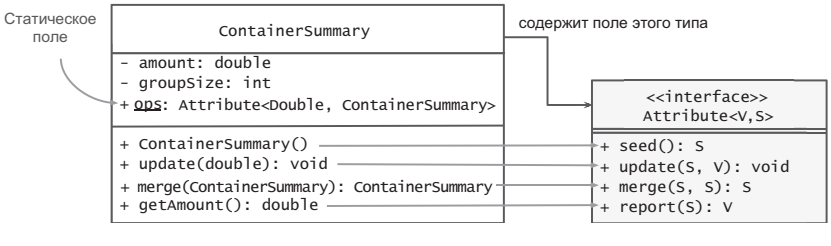


Рис. 9.2. Диаграмма классов UML для ContainerSummary и ее отношение с Attribute. Первый параметр методов update, merge и report связан с this в соответствующих методах ContainerSummary

9.5.3. Определение класса для резервуара

После определения сводного значения и его вспомогательных методов можно восстановить обычное поведение резервуаров всего в трех строках: достаточно расширить класс `UnionFindNode` и передать его конструктору объект `Attribute`:

Листинг 9.9. Generic: резервуары в трех строках

```
public class Container extends UnionFindNode<Double,ContainerSummary> {
   public Container() {
      super(ContainerSummary.ops);
   }
}
```

Неплохо, но мы сталкиваемся с некоторыми ограничениями обобщений Java. Если задуматься, необходимость хранения ссылки на один объект `Attribute` во всех `UnionFindNode` оборачивается напрасной тратой памяти. Если бы к обобщениям применялось воплощение вместо стирания типов, то эта ссылка могла бы стать *статическим* полем `UnionFindNode<Double,ContainerSummary>`. В этом случае все узлы этого типа будут совместно использовать одну ссылку на объект, ответственный за работу со сводными значениями.

Кстати, листинг 9.9 содержит самое короткое определение функционирующего класса резервуара в книге. Даже в приложении, оптимизированном для краткости, такое определение длиннее! Конечно, версия в этом разделе не является полной; вся функциональность была вынесена в обобщенный фреймворк. Если учитывать весь код (классы `UnionFindNode` и `ContainerSummary`, интерфейсы `ContainerLike` и `Attribute`), обобщенная версия окажется *самой длинной* в книге!

9.6. ПОСТЫ В СОЦСЕТЯХ

Чтобы убедиться в общем характере нашего решения, спроектируем еще одну версию конкретного резервуара; на этот раз будет рассматриваться второй сценарий, который был представлен в начале этой главы: сообщения в соцсети, соединенные общими комментаторами, с подсчетом лайков. Этот сценарий оказывается проще сценария с резервуарами. Достаточно, если в сводном значении группы будет храниться общее количество лайков, накопленных сообщениями в группе, а размер знать не нужно. В результате сводное значение будет представлять собой обычную обертку для целого числа.

Листинг 9.10. PostSummary: поля и конструкторы

```
class PostSummary {
   private int likeCount;

   public PostSummary(int likeCount) {
```

```
      this.likeCount = likeCount;
   }
   public PostSummary() {} ❶ Для использования ссылки на метод
```

Конструктор по умолчанию заполняет операцию «инициализации» `Attribute`. Методы из листинга 9.11 предоставляют три остальные операции. Как и прежде, вы можете воспользоваться статическим методом `of` для упаковки четырех операций в объект типа `Attribute`.

Листинг 9.11. PostSummary: методы и статическое поле

```
public void update(int likes) {
   likeCount += likes;
}
public PostSummary merge(PostSummary summary) {
   return new PostSummary(likeCount + summary.likeCount);
}
public int getCount() {
   return likeCount;
}
public static final Attribute<Integer,PostSummary> ops =
   Attribute.of(PostSummary::new, ❶ Ссылка на конструктор по умолчанию
                PostSummary::update,
                PostSummary::merge,
                PostSummary::getCount);
}
```

Как и с резервуарами из предыдущего раздела, экземпляр класса, представляющего сообщения в социальной сети, можно создать всего в трех строках:

Листинг 9.12. Post: подсчет лайков с использованием обобщенного фреймворка

```
public class Post extends UnionFindNode<Integer,PostSummary> {
   public Post() {
      super(PostSummary.ops);
   }
}
```

9.7. А ТЕПЕРЬ СОВСЕМ ДРУГОЕ

В последнем примере вместо отдельного класса представлено автономное приложение с графическим интерфейсом. Оно позволяет шире применить принципы, описанные в книге. В репозитории[1] хранится простое GUI-приложение, которое рисует параболу, то есть кривую, которая описывается формулой

$$y = ax^2 + bx + c.$$

[1] Базовая версия графического приложения находится в пакете eis.chapter9.plot, а обобщенная — в пакете eis.chapter9.generic.plot.

Скриншот вы можете увидеть на рис. 9.3. На верхней панели выведен график функции для фиксированного диапазона значений *x*. На средней панели выведены значения функции для пяти фиксированных значений *x*. На нижней панели пользователь может в интерактивном режиме изменять значения трех параметров *a*, *b* и *c*.

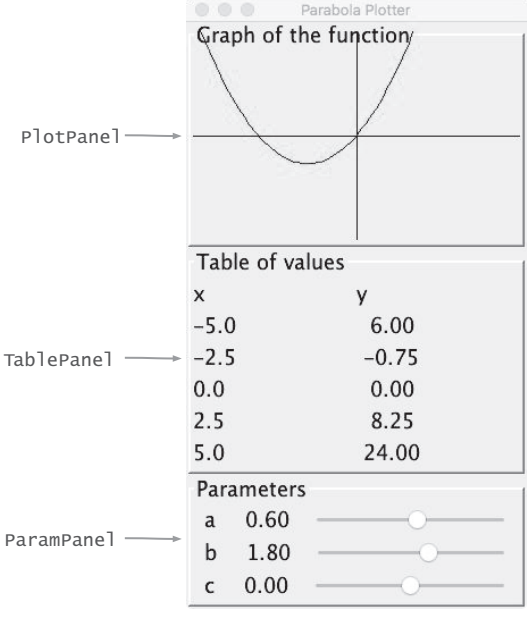


Рис. 9.3. Снимок экрана программы построения графика

Базовая реализация состоит из четырех классов — по одному для каждой панели и для класса `Main`, который связывает их воедино. Она может рисовать только параболы и имеет пару недостатков:

- *Дублирование кода:* и `TablePanel`, и `PlotPanel` содержат код для вычисления параболы в заданной точке. Было бы лучше, если бы этот код хранился только в одном месте.
- *Узкоспециализированная схема коммуникаций:* когда вы изменяете параметр, перемещая ползунок, код обработки этого события (*контроллер*) приказывает всем панелям выполнить перерисовку. И это не так плохо, но представьте полноценную версию этого приложения с десятками виджетов, которые могут по-разному изменять визуализацию. Если оставить такую схему коммуникаций, то все виджеты должны будут знать обо всех панелях, на которых осуществляется визуализация функции (то есть *представлениях*). На рис. 9.4 изображена типичная последовательность событий в этой архитектуре.

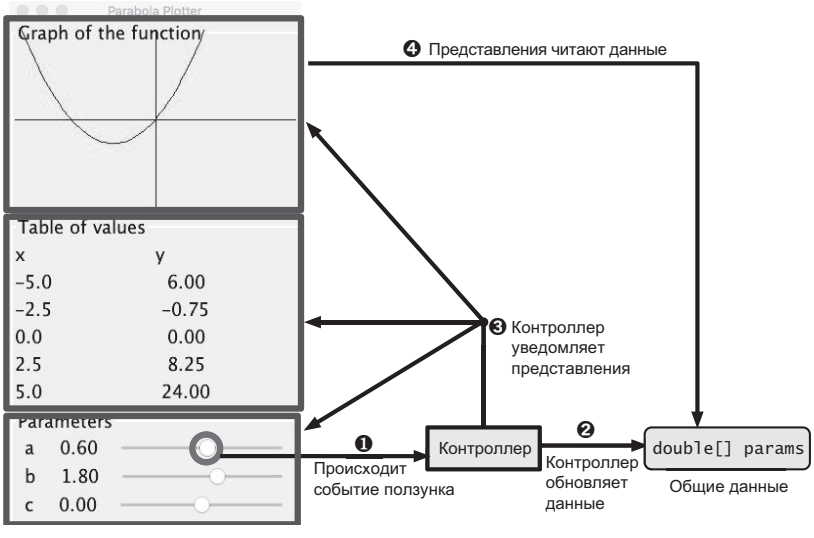


Рис. 9.4. Схема коммуникаций в базовой программе построения графика. Параметры хранятся в обычном массиве double, совместно используемом всеми компонентами. В этой схеме каждый контроллер должен располагать информацией обо всех представлениях

Обобщим приложение, чтобы оно могло строить графики произвольных параметрических функций с любым количеством параметров, то есть кривых, определяемых формулой

$$y = f(a_1, \ldots, a_n, x),$$

где $a_1, \ldots, a_n$ — параметры. Хочу уточнить, что речь идет не о передаче определения функции из текста, для чего потребовался бы парсер. Обобщенное приложение должно уметь переключаться на разные типы функций с минимальными усилиями со стороны программиста. Программа должна автоматически адаптировать графический интерфейс к типу функции. Например, количество ползунков на нижней панели должно соответствовать количеству параметров. Заодно так будут устранены два недостатка архитектуры, которые я указал выше.

9.7.1. Интерфейс для параметрических функций

Первым шагом в процессе обобщения станет определение интерфейса (назовем его `ParametricFunction`), представляющего параметрическую функцию. Чтобы приложение могло полностью адаптироваться к конкретной параметрической функции, интерфейс должен включать следующее:

- передачу количества параметров;
- передачу имен всех параметров (для настройки метки «a», «b» и «c» на панели параметров);
- чтение и присваивание значений всех параметров;
- вычисление функции для заданного значения с текущим значением его параметров (чтобы решить проблему дублирования кода, упоминавшуюся выше, параметрическая функция станет единственным местом, ответственным за вычисление значения функции).

Чтобы перевести эти функции на Java, необходимо индексировать параметры от `0` до `n – 1`; полученный интерфейс будет выглядеть примерно так:

```
public interface ParametricFunction {
   int getNParams(); ❶ Возвращает количество параметров

   String getParamName(int i); ❷ Возвращает имя параметра i

   double getParam(int i); ❸ Возвращает значение параметра i

   void setParam(int i, double val); ❹ Присваивает значение параметра i

   double eval(double x); ❺ Возвращает значение функции для x
}
```

В этой точке мы задействуем старое конкретное поведение в классе `Parabola`, реализующем этот интерфейс. (Я опускаю проверки предусловий для простоты.)

```
public class Parabola implements ParametricFunction {
   private final static int N = 3; ❶ Три параметра
   private final static String[] name = { "a", "b", "c" };
   private double[] a = new double[N];

   public int getNParams()           { return N; }
   public String getParamName(int i) { return name[i]; }
   public double getParam(int i)     { return a[i]; }
   public void setParam(int i, double val) { a[i] = val; }
   public double eval(double x)      { return a[0]*x*x + a[1]*x + a[2]; }
}
```

А теперь представьте, как легко определяется другая параметрическая функция. Допустим, вы хотите нарисовать график гиперболы по формуле

$$y = a/x = f(a,x)$$

Следующий класс справляется с этой задачей:

```
public class Hyperbola implements ParametricFunction {
   private final static int N = 1; ❶ Один параметр
   private final static String[] name = { "a" };
   private double[] a = new double[1];

   public int getNParams()           { return N; }
```

```
   public String getParamName(int i) { return name[i]; }
   public double getParam(int i)     { return a[i]; }
   public void setParam(int i, double val) { a[i] = val; }
   public double eval(double x)      { return a[0] / x; }
}
```

Если сравнить `Parabola` и `Hyperbola`, вы немедленно заметите, что они используют большой объем общего кода. Единственное серьезное различие заметно в реализации `eval`, где, собственно, определяется сама функция. Это наводит на мысль, что абстрактный класс, вставленный между интерфейсом и конкретными классами, сможет нести большую часть нагрузки этих классов.

Абстрактный класс — назовем его `AbstractFunction` — может отвечать за хранение параметров и управление ими и даже за предоставление стандартных имен параметров (буквы «a», «b» и т. д.). Фактически абстрактный класс может взять на себя все, кроме вычисления значения функции в методе `eval`, который остается абстрактным. Ниже приведена возможная реализация абстрактного класса (и снова некоторые проверки опущены для простоты):

```
public abstract class AbstractFunction implements ParametricFunction {
   private final int n;
   protected final double[] a; ❶ Доступен для подклассов ради эффективности

   public AbstractFunction(int n) { ❷ Конструктор для подклассов
      this.n = n;
      this.a = new double[n];
   }

   public int getNParams() { return n; }
   public String getParamName(int i) {
      final int firstLetter = 97; ❸ ASCII-код 'a'
      return Character.toString(firstLetter + i);
   }
   public double getParam(int i) { return a[i]; }
   public void setParam(int i, double val) { a[i] = val; }
}
```

Абстрактный класс упрощает определение конкретных функций. Например, вот как выглядит класс `Hyperbola` при использовании `AbstractFunction`:

```
public class Hyperbola extends AbstractFunction {
   public Hyperbola() { super(1); }
   public double eval(double x) { return a[0] / x; }
}
```

9.7.2. Схема коммуникаций

Раз уж мы занялись рефакторингом, заодно доработаем схему коммуникаций в программе. Сейчас имеется центральный объект, параметрическая функция, которая хранит важные данные (параметры) и предоставляет отображаемую

информацию (значения функции). Ситуация идеально подходит для применения хорошо известного архитектурного паттерна модель — представление — контроллер (MVC, model-view-controller).

МОДЕЛЬ — ПРЕДСТАВЛЕНИЕ — КОНТРОЛЛЕР

Архитектурный паттерн модель — представление — контроллер был предложен в 1970-е годы для настольных программ с графическими интерфейсами. В соответствии с этим паттерном, программные компоненты делятся на три категории:

- *модели* — компоненты для хранения данных, актуальных для приложения;
- *представления* — компоненты, обеспечивающие вывод данных для пользователя;
- *контроллеры* — компоненты, реагирующие на пользовательский ввод.

В исходном паттерне контроллеры не должны напрямую взаимодействовать с представлениями. При получении команды от пользователя — например, нажатии кнопки — контроллер информирует модель или изменяет ее. В свою очередь, модель отвечает за передачу уведомлений этим представлениям, нуждающимся в обновлении.

С момента своего появления паттерн был принят и адаптирован для разных сценариев, особенно для фреймворков веб-приложений. Он также породил такие вариации, как модель — представление — адаптер и модель — представление — презентатор.

В контексте приложения с графиками параметрическая функция является классом модели, три панели — представлениями, а обработчики событий, реагирующие на ползунки, — контроллерами. Спроектируем переработанное приложение так, чтобы оно соответствовало схеме коммуникаций, которая изначально подразумевалась паттерном модель — представление — контроллер:

- При запуске программы три представления регистрируются в качестве *наблюдателей* для модели. Модель (параметрическая функция) сохраняет ссылки на них. Чтобы не загромождать интерфейс `ParametricFunction` несвязанной функциональностью, вы можете выделить ответственность за хранение этих ссылок и отправку уведомлений в отдельный класс (`ObservableFunction` в репозитории), который упакует параметрическую функцию и добавит эти функции[1].

[1] Данный механизм является примером паттерна проектирования Декоратор.

- Когда пользователь перемещает ползунок на панели параметров в графическом интерфейсе, контроллер обновляет значение соответствующего параметра модели. Никаких других действий контроллер *не* выполняет.
- Каждый раз, когда модель получает вызов `setParam` для обновления значения параметра, она уведомляет все зарегистрированные представления о том, что в модели что-то изменилось.

Ниже приведены основные составляющие класса `ObservableFunction`. Он упаковывает объект `ParametricFunction` и в то же время реализует интерфейс. Список наблюдателей хранится в списке объектов `ActionListener` Это стандартный интерфейс из оконного инструментария Java AWT, содержащий всего один метод `void actionPerformed(ActionEvent e)`. В параметре `ActionEvent` передается информация о событии. Мы будем поддерживать единственный тип событий: изменение значения одного из параметров функции пользователем. Это позволит использовать один фиктивный объект события для всех уведомлений. Начало класса `ObservableFunction` будет таким:

```
public class ObservableFunction implements ParametricFunction {
   private final ParametricFunction f; ❶ Внутренняя параметрическая функция
   private final List<ActionListener> listeners = new ArrayList<>();
   private final ActionEvent dummyEvent =
      new ActionEvent(this, ActionEvent.ACTION_FIRST, "update");

   public ObservableFunction(ParametricFunction f) { this.f = f; }
```

Основная ответственность `ObservableFunction` — уведомление всех наблюдателей о вызове `setParam`:

```
public void setParam(int i, double val) {
   f.setParam(i, val);
   for (ActionListener listener: listeners) { ❶ Уведомляет наблюдателей
      listener.actionPerformed(dummyEvent); ❷ Фиктивное событие, не содержащее
   }                                          реальной информации
}
```

Все остальные методы будут переданы через внутренний объект `ParametricFunction`. Например, реализация `getParam` будет такой:

```
public double getParam(int i) { ❶ Передается внутренней функции
   return f.getParam(i);
}
```

Новая схема коммуникаций изображена на рис. 9.5. Поскольку за уведомление всех представлений отвечает одиночный объект (модель), мы можем позволить себе разбить три представления предыдущей версии на большее число представлений. Например, рассматривать как представление можно не весь объект `TablePanel`, а только пять меток в столбце «*y*». В конце концов, только эти части

панели будут должны перерисовываться программой при обновлении одного из параметров пользователем.

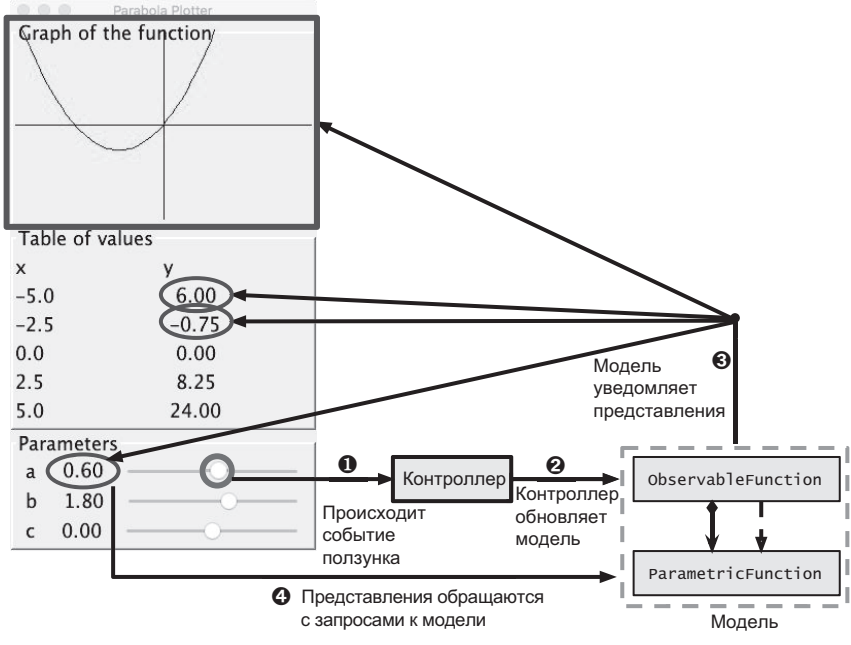


Рис. 9.5. Схема коммуникаций для переработанной программы построения графиков. Контроллер взаимодействует с представлениями только через модель (паттерн модель — представление — контроллер). Стрелки между ObservableFunction и ParametricFunction показывают, что первый реализует второй и вдобавок содержит ссылку на внутренний объект ParametricFunction

Такая схема коммуникаций лучше защищена от ошибок, чем специализированное решение, которое использовалось базовой программой построения графиков. В нее проще добавить новые представления или контроллеры. Чтобы активизировать новое представление, достаточно передать ему модель и зарегистрировать его как очередного наблюдателя модели. Никакие контроллеры изменять не придется. И наоборот, в GUI можно будет добавлять новые контроллеры (то есть новые интерактивные виджеты) без изменения компонентов представления.

9.8. РЕАЛЬНЫЕ СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ

Как уже было показано в этой главе, обобщения — чрезвычайно мощный механизм, позволяющий определять структуры данных, безопасные по отношению к типам и способные работать с разными типами данных. Типы становятся

параметрами (обобщения также называются *параметрическим полиморфизмом*), выбор которых откладывается до момента объявления. Параметризация типов способствует повторному использованию кода, потому что она позволяет избежать бесконечных повторений одного и того же алгоритма для разных типов данных. Чтобы сделать пример более конкретным, я приведу несколько дополнительных сценариев использования:

- Пожалуй, одним из важнейших сценариев использования обобщений становятся контейнеры данных: векторы, списки, множества, деревья, очереди, стеки и т. д. Сможете ли вы выделить важную характеристику, общую для всех контейнеров? Они не зависят от типов объектов, с которыми работают. Контейнерам важна только организация этих объектов: когда вы извлекаете элемент из стека, контейнер не интересуется типом извлекаемого объекта.

- Как было показано в предыдущей главе, многопоточность, как одна из важнейших особенностей языка Java, развивалась от версии к версии. В этом развитии особенно выделяются средства конкурентности, которые были добавлены разработчиками языка в Java 1.5 при добавлении обобщений. С первых дней Java существовала возможность представления потоковых задач посредством реализации интерфейса `Runnable`. Этот интерфейс содержит единственный метод `run`, который не получает параметров и не возвращает значения. Как следствие, он нужен, только если вы не ожидаете получить итоговое значение от потока. Новый же обобщенный интерфейс `Callable` возвращает параметризованный тип. Чтобы выполнить задачу, необходимо передать объект, реализующий `Callable`, объекту `ExecutorService` для запуска. Готовы предположить, какой тип вернет `ExecutorService`? Другой параметризованный тип `Future`. Тип `Future<T>` обладает семантикой ожидания, при которой вы ожидаете некоторый результат типа `T` после завершения вычислений.

- В первом сценарии использования мы говорили о том, как структуры данных используют обобщения для организации данных. Впрочем, нередко обобщения используются для контейнеров, содержащих *один* элемент параметризованного типа. `AtomicReference<T>` — пример одноэлементного контейнера, который может использоваться для выполнения атомарной потокобезопасной операции. Такой объект может совместно использоваться разными потоками без синхронизации. Другим примером служит недавно появившийся класс `Optional<T>`, который избавляет от необходимости возвращать значения `null` (упражнение 3 этой главы).

- В рабочей кодовой базе часто используются объекты доступа к данным (DAO, data access object), предоставляющие интерфейс к механизмам долгосрочного хранения данных (например, реляционным базам данных). Объекты DAO должны предоставлять операции с механизмом долгосрочного хранения, не раскрывая его внутреннее устройство клиенту. Представьте, что вы пишете объект DAO для выполнения CRUD-операций создания, удаления, обновления, поиска и т. д. Этот объект может использоваться для сохранения

разных типов сущностей, определенных в вашей модели предметной области. С обобщениями появляется возможность параметризовать DAO и применять типичные операции с разными типами сущностей.

9.9. ПРИМЕНИМ ИЗУЧЕННОЕ

Упражнение 1

Вспомните, что обобщения Java реализуются на основе *стирания* типов, а обобщения C# — на основе *воплощения*. В табл. 9.5 представлены три команды с параметром-типом `T`, допустимые в C#, но недопустимые в Java. Какое обходное решение можно использовать в языке Java в каждом из этих трех случаев? Другими словами, как добиться того же эффекта другим способом?

Таблица 9.5. Некоторые ограничения обобщений Java по сравнению с C# относительно возможностей использования параметра-типа T. Обратите внимание: в первом примере, чтобы код мог считаться правильным кодом C#, нужно ограничение типа where T: new()

Тип команды	Недопустимо в Java	Допустимо в C#
Новый объект	`new T()`	`new T()`
Новый массив	`new T[10]`	`new T[10]`
Проверка типа на стадии выполнения	`exp instanceof T`	`exp is T`

Упражнение 2

Используя обобщенную инфраструктуру `UnionFindNode`, спроектируйте решение для первого сценария, рассмотренного в начале главы: резервуары с уровнями воды, представленными рациональным числом с произвольной точностью (в математическом понимании рациональных чисел).

Подсказка: не изобретайте велосипед. Начните с существующего класса для представления рациональных чисел с произвольной точностью. Вы легко найдете такие классы в интернете.

Упражнение 3

Спроектируйте класс `Schedule<E>` для обработки обобщенных событий типа `E`, где `E` — подтип следующего интерфейса:

```
public interface Event {
   void start();
   void stop();
}
```

Класс `Schedule<E>` должен предоставлять следующие методы:

- `public void addEvent(Eevent, LocalTime startTime, LocalTime stop Time)` добавляет в график событие с заданным начальным и конечным временем. Если событие перекрывается с другим событием из графика, метод выдает исключение `IllegalArgumentException`. Если график уже был запущен (метод `launch`), метод выдает `IllegalStateException`;
- с момента вызова метода `public void launch()` график отвечает за вызов методов `start` и `stop` своих событий в правильное время. После того как график будет запущен, добавить в него новые события вы уже не сможете;
- `public Optional<E> currentEvent()` возвращает текущее активное событие (если таковое есть). Напомню, что `Optional` — современная альтернатива для возвращения `null`. `Optional<E>` может содержать объект типа `E` или быть пустым. Если клиент уже запустил этот график, но активного события на данный момент нет, этот метод возвращает пустой объект `Optional`. Если клиент еще не запустил график, метод выдает `IllegalStateException`.

Также реализуйте конкретный класс событий (скажем, `HTTPEvent`), у которого действия запуска и остановки генерируют сообщения HTTP GET к заданным URL.

Упражнение 4

Напишите метод, который получает коллекцию объектов и разбивает их в соответствии с предикатом эквивалентности.

```
public static <T> Set<Set<T>{}> partition(
   Collection<? extends T> c,
   BiPredicate<? super T, ? super T> equivalence)
```

`BiPredicate<U,V>` — стандартный функциональный интерфейс с единственным методом `boolean test(U u, V v)`. Можно предположить, что предикат эквивалентности удовлетворяет свойствам отношений эквивалентности (рефлексивность, симметричность и транзитивность), как и метод `equals` из `Object`.

Допустим, вы хотите группировать строки по длине. Соответствующая эквивалентность может быть определена в виде соответствующего предиката `BiPredicate`:

```
BiPredicate<String,String> sameLength = (a, b) -> a.length() == b.length();
```

После этого можно вызвать метод `partition` для множества строк:

```
Set<String> names = Set.of("Walter", "Skyler", "Hank", "Mike", "Saul");
Set<Set<String>{}> groups = partition(names, sameLength);
System.out.println(groups);
```

В результате мы получим следующие пять строк, объединенные в две группы по длине:

```
[[Walter, Skyler], [Saul, Mike, Hank]]
```

Подсказка: это упражнение имеет с резервуарами больше общего, чем может показаться на первый взгляд.

ИТОГИ

- В современном программировании мощные фреймворки, рассчитанные на повторное использование, объединяются с кодом, специфичным для приложения.
- Обобщения помогают писать компоненты, предназначенные для повторного использования.
- Компоненты, предназначенные для повторного использования, могут потребовать больше затрат, чем специализированные решения.
- Потоки данных Java 8 широко применяют обобщения для построения фреймворков обработки данных с гибко изменяемой конфигурацией.
- Обобщение программного объекта начинается с определения набора целевых сценариев.
- Программные компоненты, предназначенные для повторного использования, часто строятся на основе набора ключевых интерфейсов.
- Паттерн модель — представление — контроллер предписывает обязанности и коммуникационные протоколы для настольных приложений с графическим интерфейсом.

ОТВЕТЫ НА ВОПРОСЫ И УПРАЖНЕНИЯ

Неожиданный вопрос 1

Вставлять метод `public boolean equals(Employee e)` в класс `Employee` не стоит. Во-первых, отметим, что вы перегружаете, а не переопределяете метод `equals` из `Object`. Как следствие, у объектов появляются два разных метода проверки равенства: версия, унаследованная от `Object` (основанная на тождественности), и другая версия, с более конкретным параметром типа (скорее всего, основанная на содержащихся данных). При сравнении двух работников может быть вызван любой из двух методов в зависимости от статического типа второго работника:

```
Employee alex = ..., beth = ...;
alex.equals(beth); ❶ Сравнение на основании содержимого
alex.equals((Object) beth); ❷ Сравнение на основании тождественности
```

В такой ситуации повышается риск ошибок. Вряд ли это то, чего хотел добиться программист.

Неожиданный вопрос 2

Нет, в Java невозможно создать массив типа `T` (`new T[...]`), где `T` — параметр-тип. Дело в том, что массивы хранят свой тип и используют эту информацию во время выполнения для проверки правильности всех операций записи или преобразований типа. Из-за стирания типов байт-код не сохраняет фактическое значение `T` во время выполнения, так что этот механизм не сработает. Не путайте это ограничение с объявлением переменной типа `T[]` — оно абсолютно законно.

В C# возможно создать массив типа `T`, потому что параметры-типы воплощаются — их фактическое значение известно на стадии выполнения.

Неожиданный вопрос 3

Только параллельные коллекторы используют метод `combiner`. Он возвращает объект, который можно использовать для слияния частичных результатов, полученных разными потоками, сотрудничающими для выполнения операции потока данных.

Неожиданный вопрос 4

Ссылка на метод не может быть напрямую присвоена переменной типа `Object`, как в следующем примере:

```
Object x = String::length; ❶ Ошибка на стадии компиляции
```

потому что контекст не содержит достаточной информации для определения конкретного функционального интерфейса. Если вам потребуется сделать что-то подобное, может помочь преобразование типа:

```
Object x = (ToIntFunction<String>) String::length; ❷ Допустимый код Java
```

Упражнение 1

Если вам нужна информация типа на стадии выполнения, а обобщений оказывается недостаточно, обычно проблему удается решить при помощи отражения. Например, вместо `new T()` можно использовать объект `t` типа `Class<T>`, а затем динамически вызвать конструктор, как в следующем фрагменте:

```
Constructor<T> constructor = t.getConstructor(); ❶ Возвращает конструктор по умолчанию
T object = constructor.newInstance();
```

В зависимости от контекста возможно альтернативное решение: клиент должен предоставить `Supplier<T>` — функциональный интерфейс, который может упаковать конструктор или любой другой способ производства объектов типа `T`.

Рекомендуемое обходное решение для `new T[10]` основано на использовании коллекций вместо массивов:

```
List<T> list = new ArrayList<T>();
```

Как было показано в главе 4, с контейнером `List` вы также получаете множество разных дополнительных функций при минимальных затратах (но не можете использовать `list[i]`, что печально).

Наконец, проверку времени выполнения, аналогичную `exp instanceof T`, снова можно имитировать посредством отражения. Если у вас имеется объект `t` типа `Class`, вы можете проверить, относится ли заданное выражение к подтипу `t`, при помощи конструкции

```
t.isInstance(exp);
```

Упражнение 2

Для работы с рациональными числами произвольной точности я выбрал класс `BigRational`, созданный Робертом Седжвиком и Кевином Уэйном[1]. Это интуитивно понятная реализация неизменяемых рациональных чисел, которая может использоваться примерно так:

```
BigRational a = new BigRational(1, 3); ❶ Одна треть
BigRational b = new BigRational(1, 2); ❷ Одна вторая
BigRational c = a.plus(b);
System.out.println(c); ❸ Выводит 5/6
```

Чтобы в резервуарах уровни воды представлялись значениями типа `BigRational`, следует изменить класс `Container` из раздела 9.5. Сначала класс сводного значения группы переопределяется полем `amount` типа `BigRational` при том же целочисленном поле размера группы. Каждый раз, когда вам потребуется выполнить арифметические вычисления с уровнем воды, необходимо будет использовать методы `BigRational`, такие как `plus` или `divides`. Я приведу фрагмент класса сводного значения группы `RationalSummary`. Остальной код можно найти в репозитории.

```
class RationalSummary {
   private BigRational amount;
   private int groupSize;
```

[1] Копию этого класса можно найти в репозитории по адресу https://bitbucket.org/mfaella/exercisesinstyle.

```
   ...
   public void update(BigRational increment) {
      amount = amount.plus(increment); ❶ Значения BigRational неизменяемы
   }
   ...
   public static final Attribute<BigRational,RationalSummary> ops =
      Attribute.of(RationalSummary::new,
                   RationalSummary::update,
                   RationalSummary::merge,
                   RationalSummary::getAmount);
}
```

После того как у вас появится класс сводного значения группы, для получения класса резервуара достаточно будет расширить класс `UnionFindNode` и передать его конструктору объект `Attribute`:

```
public class Container extends UnionFindNode<BigRational,RationalSummary> {
   public Container() {
      super(RationalSummary.ops);
   }
}
```

Упражнение 3

Класс `Schedule` должен хранить отсортированную последовательность непересекрывающихся событий. Для этого определим вспомогательный класс (допустим, `TimedEvent`), который будет хранить событие вместе с временем запуска и остановки. Например, это может быть приватный внутренний класс `Schedule`.

Контейнер `TreeSet<TimedEvent>` с нестандартным порядком элементов может эффективно хранить события в отсортированном порядке и в то же время выявлять перекрытия. Вспомните, что все реализации интерфейса `Set` запрещают дубликаты элементов. Контейнер `TreeSet` реализует `Set`, а все его операции базируются на порядке элементов, включая обнаружение дубликатов (то есть он не вызывает `equals`). Чтобы отвергнуть событие, перекрывающееся с ранее вставленным событием, можно определить порядок, при котором перекрывающиеся события считаются эквивалентными (`compareTo` возвращает нуль). Другими словами, можно использовать следующий порядок:

- Если событие `a` *полностью происходит* до события `b`, то `a` «меньше» `b`, и наоборот.
- Если два события перекрываются, они являются эквивалентными (`compareTo` возвращает нуль).

Основная структура класса `TimedEvent`:

```
public class Schedule<E> {

   private class TimedEvent implements Comparable<TimedEvent> {
      E event; ❶ Приватный класс: его поля скрывать не нужно
```

```
      LocalTime startTime, stopTime;
      @Override
      public int compareTo(TimedEvent other) {
         if (stopTime.isBefore(other.startTime)) return -1;
         if (other.stopTime.isBefore(startTime)) return 1;
         return 0; ❷ Перекрывающиеся события считаются эквивалентными
      }
      ... ❸ Тривиальный конструктор опущен
   }
```

Каждый объект `Schedule` содержит следующие поля:

- `private volatile boolean active;` — значение задается в `launch` и сбрасывается в конце вспомогательного потока, выполняющего график. Модификатор `volatile` обеспечивает видимость между потоками;
- `private volatile Optional<E> currentEvent = Optional.empty();` — поддерживается вспомогательным потоком, выполняющим график. Метод `currentEvent` возвращает значение поля;
- `private final SortedSet<TimedEvent> events = new TreeSet<>();` — последовательность событий с привязкой ко времени.

Метод `addEvent` добавляет новое событие в `TreeSet` и проверяет три недопустимых случая.

```
public void addEvent(E event, LocalTime startTime, LocalTime stopTime)
{
   if (active)
      throw new IllegalStateException(
         "Cannot add event while active.");
   if (startTime.isAfter(stopTime))
      throw new IllegalArgumentException(
         "Stop time is earlier than start time.");
   TimedEvent timedEvent = new TimedEvent(event, startTime, stopTime);
   if (!events.add(timedEvent)) ❶ Вставка завершается неудачей при перекрытии событий
      throw new IllegalArgumentException("Overlapping event.");
}
```

Фактическое выполнение графика выделяется в другой поток, чтобы избежать блокировки метода `launch`. Код `launch` и двух примеров конкретных классов событий (`PrintEvent` и `HTTPEvent`) можно найти в репозитории.

Упражнение 4

Для решения этой задачи можно воспользоваться реализацией обобщенного фреймворка резервуаров, такой как `UnionFindNode`. Идея заключается в том, чтобы создать узел для каждого элемента заданной коллекции и соединять два узла, если их элементы эквивалентны, в соответствии с заданным предикатом. После того как все связи будут установлены, группы соединенных узлов дадут нужный результат.

Для этого каждый узел должен знать множество соединенных с ним узлов. Поместим эту информацию в сводное значение группы. Используем реализацию `Attribute<V,S>`, в которой `V` и `S` равны `Set<T>`. И снова пригодится метод адаптера `Attribute.of`:

```
public static <T> Set<Set<T>{}>
   partition(Collection<? extends T> collection,
             BiPredicate<? super T, ? super T> equivalent) {

   Attribute<Set<T>,Set<T>{}> groupProperty = Attribute.of(
             HashSet::new, ❶ Ссылка на конструктор
             Set::addAll, ❷ Ссылка на метод интерфейса

             (set1, set2) -> { ❸ Объединяет два множества
                Set<T> union = new HashSet<>(set1);
                union.addAll(set2);
                return union;
             },
             set -> set); ❹ Ничего распаковывать не нужно
```

Первая реальная операция включает создание узла для каждого элемента в коллекции. Также необходимо отслеживать, какому элементу принадлежит тот или иной узел. Для этой цели можно воспользоваться картой:

```
Map<T,UnionFindNode<Set<T>,Set<T>{}>{}> nodeMap = new HashMap<>();
for (T item: collection) {
   UnionFindNode<Set<T>,Set<T>{}> node =
      new UnionFindNode<>(groupProperty);
   node.update(Set.of(item)); ❶ Инициализирует группу
   nodeMap.put(item, node); ❷ Назначает узел для текущего элемента
}
```

Далее каждая эквивалентность преобразуется в соединение между двумя узлами:

```
for (T item1: collection) {
   for (T item2: collection) {
      if (equivalent.test(item1, item2))
         nodeMap.get(item1).connectTo(nodeMap.get(item2));
   }
}
```

Наконец, все группы собираются в множество, которое и будет искомым разбиением элементов:

```
   Set<Set<T>{}> result = new HashSet<>();
   for (T item: collection) {
      result.add(nodeMap.get(item).get());
   }
   return result;
}
```

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

- *NaftalinM., Wadler P.* Java Generics and Collections. O'Reilly, 2006.

  Книга написана для Java 5, так что вы не найдете в ней описания новейших возможностей. Тем не менее она содержит основательное изложение обобщений и тонкостей их использования.

- *Tulach J.* Practical API Design: Confessions of a Java Framework Architect. Apress, 2008.

  Написание эффективного повторно используемого кода тесно связано с определением правильных API. Это одна из немногих книг, посвященных исключительно этой теме.

- *Urma R.-G., Fusco M., Mycroft A.* Modern Java in Action. Manning Publications, 2019.

  Как упоминалось в главе 8, эта книга включает одно из лучших описаний библиотеки потоков данных.

- *Скит Д.* C# для профессионалов: тонкости программирования. — 3-е изд. — М. [и др.]: Вильямс, 2014.

  Современное представление эволюции C# между версиями, включая аргументированное сравнение реализаций обобщений в C#, C++ и Java.

Приложения

Программный гольф: компактность

Как известно, в гольфе игрок должен пройти поле за минимальное количество ударов. Программный гольф — игра, в которой требуется написать самую короткую программу для решения конкретной задачи. На некоторых сайтах проводятся турниры по программному гольфу, публикуются новые задачи и ведутся рейтинги игроков. Когда срок сдачи решений истекает, все заявки публикуются, и вы можете ознакомиться с приемами, которые использовались лучшими гольфистами.

Это приложение почти противоположно главе 7, в которой представлен самый непонятный код в книге, нарушающий все правила стиля. Помните — я вас предупреждал.

Кроме развлечения, программный гольф помогает исследовать темные закоулки языка и осваивать приемы, которые могут пригодиться и при повседневном программировании.

A.1. САМАЯ КОРОТКАЯ ПРОГРАММА, КОТОРАЯ У МЕНЯ ПОЛУЧИЛАСЬ [GOLFING]

В программном гольфе важно установить ограничения, которые вы должны соблюдать. Более свободная интерпретация правил может привести к более короткому решению, но у вас не должен получиться класс, который работает только в одном сценарии. Определим такие границы для своего упражнения:

- Нам нужен класс `Container`, который реализует сценарий использования, установленный в главе 1 и неоднократно повторяющийся в книге.
- Класс также должен выполнять функциональные спецификации, изложенные в главе 1.
- Другие требования *не* выдвигаются: ни к стабильности, ни к производительности, ни к удобочитаемости.

В своем решении я использовал для представления группы соединенных резервуаров циклический список, как и в `Speed2`. Поле экземпляра `n` (`next`) содержит указатель на следующий резервуар в группе.

Как и в `Speed2`, при добавлении воды методом `addWater` объем сохраняется локально в поле экземпляра `a` и никогда не распределяется между другими соединенными резервуарами. Как следствие, каждый вызов `getAmount` должен обходить всю группу, суммировать объемы в каждом резервуаре и, наконец, возвращать общий объем, разделенный на размер группы.

Прежде чем рассматривать сам код, перечислю пять полей экземпляров:

- `a` — общий объем, добавленный в резервуар;
- `s`, `t` — временные переменные, необходимые `getAmount`. В обычных обстоятельствах это были бы локальные переменные этого метода. Более того, `s` должно быть целым числом. Я объявляю их как поля, поэтому что это сэкономит несколько символов;
- `n` — указатель на следующий резервуар в списке, представляющем группу данного резервуара;
- `c` — временная переменная, используемая как `connectTo`, так и `getAmount`. Вне этих методов `c` равно `n`.

Код компактной реализации `Container` приведен в листинге А.1. Я оставил некоторые пропуски и отступы для удобочитаемости. Если удалить все пропуски, кроме необходимых, класс будет состоять из 223 байт и при этом работать как положено. Для сравнения: `Reference` занимает 1322 байта, включая пропуски.

Листинг А.1. Golfing: реализация резервуара (223 байта)

```
public class Container {
   float a,s,t; ❶ s,t используются как локальные переменные
   Container n=this,c=n;
   public float getAmount() {
      for(s=t=0;s<1||c!=n;c=c.n,s++) ❷ Обратите внимание на запятую
         t+=c.a;
      return t/s;
   }
   public void connectTo(Container o) {
      c=o.n; o.n=o.c=n; n=c; ❸ Переставляет указатели
```

```
    }
    public void addWater(float w) {
        a+=w; ❹ Локальное накопление
    }
}
```

Чтобы разобраться в этой загадочной реализации, начните с метода `addWater` — самого простого из всех. Добавленный объем суммируется с полем `a`, и никакие другие строки не изменяют это поле. Таким образом, поле `a` заданного резервуара обозначает суммарный объем воды, добавленной в этот резервуар.

Перейдем к методу `connectTo`. Вспомните, о чем говорилось в главе 3: объединение двух циклических списков, начинающихся с произвольных узлов, выполняется предельно просто — для этого нужно лишь поменять их указатели на следующий узел. Именно это и делает метод `connectTo`. Кроме того, значения вспомогательных переменных `c` и `o.c` обновляются равными новым (то есть переставленным) значениям `n` и `o.n` соответственно.

> **ВЛОЖЕННЫЕ ПРИСВАИВАНИЯ**
>
> Как и в языке C, в Java множественные присваивания могут объединяться в цепочки. Такая последовательность обрабатывается справа налево, так что всем переменным в списке присваивается значение крайнего правого выражения.

Наконец, обращает на себя внимание устрашающий цикл в `getAmount`. Он предназначен для вычисления общего объема во всех резервуарах группы одновременно с размером группы. После цикла общий объем хранится в переменной `t`, а размер группы — в `s`. Становится понятно, почему метод возвращает `t/s`.

> **ЗАПЯТЫЕ В ЦИКЛАХ FOR**
>
> В C выражение `exp1,exp2` вычисляет оба выражения в указанном порядке, а затем возвращает значение второго. В Java аналогичный синтаксис допускается только внутри первой и третьей секции цикла `for`. Он предназначен для элегантной поддержки циклов с множественными индексами:
>
> ```
> for (i=0, j=n; i<n; i++, j-{}-) ...
> ```

С учетом сказанного части инициализации и приращения цикла должны быть вполне понятны. И размер, и общий объем изначально равны нулю. Для каждой итерации размер увеличивается на 1, а указатель резервуара `c` перемещается к следующему резервуару в группе.

Условие продолжения требует некоторого пояснения. Цикл должен остановиться после обхода всего циклического списка, то есть при возвращении указателя на резервуар к исходному значению. В нашем случае указатель `c` изначально содержит значение указателя на следующий узел `n`. Таким образом, цикл должен продолжаться, пока `c!=n`. Однако здесь возникает одна проблема: цикл `for` проверяет свое условие *перед* каждой итерацией. Чтобы цикл гарантированно выполнил хотя бы одну итерацию, мне пришлось добавить условие `s<1`.

Вероятно, существуют более короткие решения. Удастся ли вам их найти? Если удастся — напишите, мне будет интересно посмотреть!

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Программный гольф — не та тема, по которой пишется множество книг. Пока Международный олимпийский комитет не признает его полноценной спортивной дисциплиной, за информацией лучше обращаться на специализированные веб-сайты.

- Anarchy Golf, http://golf.shinh.org — на этом сайте можно ознакомиться со скромными достижениями автора в языке AWK. Проведите поиск по строке marcof.
- Code Golf on StackExchange, https://codegolf.stackexchange.com — другой сайт, на котором проводятся конкурсы по программному гольфу.
- The International Obfuscated C Code Contest, http://www.de.ioccc.org/ — конкурс на самый запутанный и странный код C. Как и программный гольф, помогает исследовать темные закоулки языков программирования для развлечения.

Хорошим примером масштабного применения программного гольфа и экстремального программирования является игра .kkrieger (2004 года). Это 3D-шутер с видом от первого лица уровня Doom, упакованный в 96-килобайтный исполняемый файл (да, это не ошибка).

Б Финальная версия класса для резервуара с водой

Итак, мы рассмотрели 17 разных версий резервуаров. Возможно, вас интересует, какая версия и какой класс, представляющий резервуар с водой, следует признать лучшими? Ответ не прост. В каком-то смысле все версии (кроме `Novice`) могут считаться лучшими в разных обстоятельствах. Например, версия `Speed1` уместна, если вам однозначно нужно постоянное время `addWater` и `getAmount`. А версия `Memory4` подойдет, если вам нужно поместить как можно больше резервуаров в заданный объем памяти. Обе эти версии оптимальны, только если вас не интересуют другие характеристики программного продукта, а это предположение, мягко говоря, нереалистичное.

Рассмотрение свойств кода по отдельности, как в этой книге, — не более чем педагогический прием. Даже если на практике вы думаете о разных свойствах по отдельности, вы должны написать код, который учитывает их все одновременно. Если достижения двух целей противоречат друг другу, контекст (он же ваш руководитель) укажет вам, какое свойство должно доминировать в данном случае.

Как правило, в большинстве продуктов основными необходимыми свойствами являются удобочитаемость, надежность и эффективность по времени. Лишь в относительно небольшом подмножестве проектов важны такие свойства, как эффективность использования памяти, возможность повторного использования или потокобезопасность. Давайте создадим версию `Container`, которая оптимизирует первые три свойства. В ней самая быстрая реализация (`Speed3` из главы 3) будет объединена с самой удобочитаемой реализацией (`Readable` из главы 7) и некоторыми усовершенствованиями надежности, представленными в главах 5 и 6.

Точнее, мы возьмем версию `Speed3` и внесем следующие улучшения:

- (Удобочитаемость) Добавим комментарии Javadoc во все открытые методы.
- (Удобочитаемость) Применим основные приемы улучшения удобочитаемости (например, выделение метода).
- (Надежность) Добавим проверки предусловия во все открытые методы.
- (Надежность) Включим набор тестов, разработанных в главе 6 (раздел 6.2).

Основные части полученного класса будут представлены в следующих разделах, а полный исходный код можно найти в репозитории (https://bitbucket.org/mfaella/exercisesinstyle).

Б.1. УЛУЧШЕНИЯ УДОБОЧИТАЕМОСТИ

Вспомните, что `Speed3` достигает своей производительности за счет представления групп соединенных резервуаров в виде родительского дерева указателей. Корню каждого дерева известен размер своей группы и объем воды на резервуар. Соединение двух резервуаров означает, что меньшее из двух деревьев должно быть присоединено к большему — так называемая политика *связывания по размеру*.

Сосредоточимся на операции `connectTo`, потому что она в наибольшей степени выигрывает от улучшения удобочитаемости. Помимо добавления документирующих комментариев в формате Javadoc, вы можете применить выделение метода и делегировать операцию слияния деревьев новому вспомогательному методу `linkTo`. Так метод `connectTo` будет упрощен: он будет находить два корня групп, проверять, совпадают ли они (в этом случае никакая операция не выполнится), и, наконец, объединять два дерева в соответствии с политикой связывания по размеру. Метод также получит небольшое улучшение по надежности: при его вызове с аргументом `null` он выдаст `NullPointerException` с нестандартным сообщением об ошибке.

Листинг Б.1. Ultimate: метод connectTo

```
/** Соединяет текущий резервуар с другим. ❶ Комментарий Javadoc
 *
 * @param other резервуар, который будет соединен с текущим
 */
public void connectTo(Container other) {
   Objects.requireNonNull(other,
      "Cannot connect to a null container.");     ❷ Проверка предусловия
   Container root1 = findRootAndCompress(),       ❸ Вспомогательный метод такой же,
             root2 = other.findRootAndCompress();    как в главе 3
   if (root1==root2) return; ❹ Проверяет, соединены ли резервуары

   if (root1.size <= root2.size) { ❺ Политика связывания по размеру
      root1.linkTo(root2);
   } else {
      root2.linkTo(root1);
   }
}
```

Вспомогательный метод `linkTo` сделает все остальное. В свою очередь, `linkTo` выделит еще один вспомогательный метод `combinedAmount`, который вычислит объем воды на резервуар после объединения двух групп.

Листинг Б.2. Ultimate: приватные вспомогательные методы для connectTo

```
private void linkTo(Container otherRoot) {
   parent = otherRoot;
   otherRoot.amount = combinedAmount(otherRoot);
   otherRoot.size += size;
}
private double combinedAmount(Container otherRoot) {
   return ((amount * size) + (otherRoot.amount * otherRoot.size)) /
           (size + otherRoot.size);
}
```

Б.2. УЛУЧШЕНИЯ НАДЕЖНОСТИ

Добавление воды в резервуар — единственная операция с нетривиальным предусловием, согласно которому нельзя удалить больше воды, чем содержится в резервуаре. В листинге Б.3 приведена обновленная версия `addWater` с проверкой предусловия и документированием поведения комментариями Javadoc.

Листинг Б.3. Ultimate: метод addWater

```
/** Добавляет воду в резервуар. ❶ Комментарий Javadoc
 * Отрицательное значение <code>amount</code> означает удаление воды.
 * В этом случае в группе должно быть достаточно воды
 * для удовлетворения запроса.
 *
 * @param amount объем добавляемой воды
 * @throws IllegalArgumentException если значение <code>amount</code>
 * отрицательно, а воды недостаточно для удовлетворения запроса
 */
public void addWater(double amount) {
   Container root = findRootAndCompress();

   double amountPerContainer = amount / root.size;
   if (root.amount + amountPerContainer < 0) { ❷ Проверка предусловия
      throw new IllegalArgumentException(
         "Not enough water to match the addWater request.");
   }
   root.amount += amountPerContainer;
}
```

Наконец, вы можете выполнить модульные тесты, разработанные в главе 6, для этой версии `Container` без изменений, и они пройдут успешно.

Подведем итог: финальная версия обладает не только хорошей скоростью выполнения и удобочитаемостью, но и высокой степенью надежности. Проверки

предусловий защищают от внутренних злоупотреблений, а набор тестов укрепляет нашу уверенность во внутренней надежности класса. Если бы этот класс был частью системы, критичной по безопасности, вы смогли бы легко повысить его чувствительность к внутренним дефектам, воспользовавшись одним или несколькими из следующих приемов:

- *Добавление проверок предусловий в приватные методы как тестовых условий* — например, `linkTo` может проверить, действительно ли `this` и `otherRoot` являются двумя корневыми узлами.
- *Добавление проверок инвариантов (раздел 5.4)* — например, методы `addWater` и `connectTo` могут проверить, что объем воды в резервуаре всегда неотрицателен.
- *Добавление тестов для конкретной реализации (по принципу белого ящика).* Тесты, разработанные в главе 6, базируются только на контрактах методов, но не на их реализации — это идеальный пример тестирования по принципу черного ящика. Но реализация дерева указателей на родителей, используемая здесь и в версии `Speed3`, весьма нетривиальна. Возможно, стоит добавить тесты, предназначенные специально для этой реализации, чтобы убедиться, что вы правильно реализовали разные случаи. Например, протестировать политику связывания по размеру можно с помощью соединения резервуаров с разными размерами групп.

Список основных классов каждой главы

Глава	Имя	Класс	Описание
1	`UseCase`	`eis.chapter1.UseCase`	Сценарий использования
	`Novice`	`eis.chapter1.novice.Container`	Наивная реализация
2	`Reference`	`eis.chapter2.reference.Container`	Эталонная реализация
3	`Speed1`	`eis.chapter3.speed1.Container`	Первая версия `addWater` с централизованным множеством
	`Speed2`	`eis.chapter3.speed2.Container`	Быстрая версия `connectTo` с циклическим списком
	`Speed3`	`eis.chapter3.speed3.Container`	Оптимальный баланс на базе дерева указателей на родители

Глава	Имя	Класс	Описание
4	`Memory1`	`eis.chapter4.memory1.Container`	На базе `ArrayList`
	`Memory2`	`eis.chapter4.memory2.Container`	На базе простого массива
	`Memory3`	`eis.chapter4.memory3.Container`	API без объектов
	`Memory4`	`eis.chapter4.memory4.Container`	API без объектов, 4 байта на резервуар
5	`Contracts`	`eis.chapter5.contracts.Container`	Методы проверяют свой контракт
	`Invariants`	`eis.chapter5.invariants.Container`	Методы проверяют инварианты классов
6	`UnitTests`	`eis.chapter6.UnitTests`	Набор тестов для `Reference`
	`Testable`	`eis.chapter6.testable.Container`	Оптимизация для тестируемости
7	`Readable`	`eis.chapter7.readable.Container`	Оптимизация для удобочитаемости
8	`ThreadSave`	`eis.chapter8.threadsafe.Container`	Потокобезопасность
	`Immutable`	`eis.chapter8.threadsafe.ContainerSystem`	Неизменяемая и без объектов
9	–	`eis.chapter9.generic.ContainerLike`	Интерфейс резервуара
	–	`eis.chapter9.generic.Attribute`	Интерфейс атрибута
	–	`eis.chapter9.generic.UnionFindNode`	Реализация `ContainerLike`
	`Generic`	`eis.chapter9.generic.Container`	Конкретный контейнер на базе `UnionFindNode`
Приложение А	`Golfing`	`eis.appendixa.golfing.Container`	Компактная реализация
	–	`eis.appendixa.nowhitespace.Container`	Предельно компактная
Приложение Б	`Ultimate`	`eis.appendixb.ultimate.Container`	Быстрая, понятная, надежная