

O'REILLY®

Масштабируемый рефакторинг

Возвращаем контроль над кодом



Мод Лемер

Refactoring at Scale

Regaining Control of Your Codebase

Maude Lemaire

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Масштабируемый рефакторинг

Возвращаем контроль над кодом

Мод Лемер



Санкт-Петербург · Москва · Минск

2022

ББК 32.973.2-018-02

УДК 004.415

Л44

Лемер Мод

- Л44 Масштабируемый рефакторинг. Возвращаем контроль над кодом. — СПб.: Питер, 2022. — 256 с.: ил. — (Серия «Бестселлеры O'Reilly»).
ISBN 978-5-4461-3921-7

Поддерживать большие приложения сложно, а поддержка больших «неорганизованных» приложений превращается в непосильную задачу. Пришло время сделать паузу и задуматься о рефакторинге!

Внесение значительных изменений в крупную и сложную кодовую базу — нетривиальная задача, которую практически невозможно успешно выполнить без рабочей команды, инструментов и планирования. Мод Лемер раскрывает все тайны рефакторинга на примере двух исследований. Вы научитесь эффективно вносить важные изменения в кодовую базу, разберетесь, как деградирует код и почему иногда это неизбежно.

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

ББК 32.973.2-018-02

УДК 004.415

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492075530 англ.

Authorized Russian translation of the English edition of Refactoring At Scale
ISBN 9781492075530 © 2021 Maude Lemaire

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-3921-7

© Перевод на русский язык ООО «Прогресс книга», 2022

© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Бестселлеры O'Reilly», 2022

Краткое содержание

| | |
|-------------------|----|
| Предисловие | 13 |
|-------------------|----|

ЧАСТЬ I. ВВЕДЕНИЕ

| | |
|------------------------------------|----|
| Глава 1. Рефакторинг | 20 |
| Глава 2. Как деградирует код | 46 |

ЧАСТЬ II. ПЛАНИРОВАНИЕ

| | |
|--|-----|
| Глава 3. Количественная характеристика начального состояния..... | 64 |
| Глава 4. Составление плана | 91 |
| Глава 5. Получение одобрения..... | 118 |
| Глава 6. Подбор команды | 133 |

ЧАСТЬ III. ВЫПОЛНЕНИЕ

| | |
|--|-----|
| Глава 7. Коммуникация..... | 154 |
| Глава 8. Стратегии выполнения | 171 |
| Глава 9. Закрепление результатов рефакторинга..... | 183 |

ЧАСТЬ IV. РАЗБОРЫ ПРИМЕРОВ

| | |
|--|-----|
| Глава 10. Избыточные схемы базы данных | 196 |
| Глава 11. Переход к новой базе данных | 223 |
| Об авторе | 252 |
| Об обложке | 253 |

Оглавление

| | |
|---|-----------|
| Предисловие | 13 |
| Для кого предназначена книга | 13 |
| Почему я решила написать эту книгу..... | 14 |
| Структура издания..... | 15 |
| Условные обозначения..... | 16 |
| Примеры кода | 16 |
| Благодарности | 17 |
| От издательства..... | 18 |

ЧАСТЬ I. ВВЕДЕНИЕ

| | |
|--|-----------|
| Глава 1. Рефакторинг..... | 20 |
| Что такое рефакторинг | 21 |
| Что такое масштабируемый рефакторинг | 23 |
| Зачем нужен рефакторинг | 25 |
| Выгоды рефакторинга..... | 26 |
| Продуктивность разработчика | 26 |
| Обнаружение ошибок | 28 |
| Риски рефакторинга..... | 29 |
| Риск регрессии | 29 |
| «Спящие» ошибки..... | 30 |
| Неконтролируемый рост проекта | 30 |
| Ненужное усложнение..... | 31 |
| Когда начинать рефакторинг | 31 |
| Небольшой масштаб | 32 |
| Сложность кода мешает активному развитию | 32 |

| | |
|--|-----------|
| Изменение требований к продукту | 32 |
| Производительность | 33 |
| Переход к новой технологии | 34 |
| Когда не нужно делать рефакторинг | 35 |
| Для забавы или со скуки | 35 |
| Вы случайно проходили мимо | 35 |
| Чтобы обеспечить расширяемость кода | 37 |
| Когда не хватает времени | 37 |
| Первый пример рефакторинга | 38 |
| Упрощение условных операторов | 40 |
| Удаление волшебных чисел | 41 |
| Извлечение автономной логики | 42 |
| Глава 2. Как деградирует код | 46 |
| Почему важно понимать, что код деградирует | 47 |
| Изменение требований | 48 |
| Возможности масштабирования | 49 |
| Доступность | 49 |
| Совместимость с устройствами | 50 |
| Изменение среды | 50 |
| Внешние зависимости | 51 |
| Неиспользуемый код | 52 |
| Изменение требований к продукту | 53 |
| Технический долг | 56 |
| Обход выбранной технологии | 56 |
| Отсутствие привычки к систематизации | 59 |
| Слишком быстрое продвижение | 60 |
| Применение знаний | 62 |

ЧАСТЬ II. ПЛАНИРОВАНИЕ

| | |
|--|-----------|
| Глава 3. Количественная характеристика начального состояния | 64 |
| Почему сложно оценить последствия рефакторинга | 65 |
| Оценка сложности кода | 66 |
| Метрики Холстеда | 66 |
| Цикломатическая сложность | 69 |

| | |
|---|-----------|
| NPath-сложность..... | 72 |
| Строки кода | 74 |
| Метрики покрытия кода..... | 76 |
| Документация | 80 |
| Официальная документация..... | 80 |
| Неофициальная документация..... | 82 |
| Управление версиями..... | 84 |
| Комментарии к коммитам..... | 84 |
| Коммиты..... | 85 |
| Репутация..... | 86 |
| Составляем полную картину | 89 |
| Глава 4. Составление плана..... | 91 |
| Определение конечного состояния | 92 |
| В путешествии | 92 |
| На работе | 93 |
| Поиск кратчайшего расстояния | 94 |
| В путешествии | 94 |
| На работе | 95 |
| Промежуточные шаги | 97 |
| В путешествии | 97 |
| На работе | 98 |
| Выбор стратегии развертывания..... | 101 |
| Темный/светлый режим..... | 102 |
| Развертывание гипотетического кода | 107 |
| Очистка кода | 108 |
| Ссылка на метрики..... | 110 |
| Промежуточные этапы..... | 110 |
| Метрики промежуточных этапов..... | 111 |
| Оценка | 112 |
| Обсуждение планов с другими командами | 113 |
| Информационная открытость | 114 |
| Взгляд со стороны | 114 |
| Уточненный план..... | 116 |

| | |
|---|------------|
| Глава 5. Получение одобрения..... | 118 |
| Причины несогласия руководителей | 119 |
| Руководители не пишут код..... | 119 |
| Работа руководителя оценивается иначе..... | 120 |
| Руководители понимают риски | 121 |
| Необходимость координировать усилия..... | 121 |
| Поиск убедительной аргументации | 122 |
| Как убедить коллегу | 124 |
| Получение поддержки сверху и снизу | 125 |
| Опора на доказательства..... | 129 |
| Жесткие меры | 129 |
| Заинтересованность в рефакторинге..... | 131 |
| Глава 6. Подбор команды | 133 |
| Выбор экспертов..... | 134 |
| Подбор специалистов | 136 |
| Многопрофильные специалисты | 137 |
| Еще немного про активных участников..... | 138 |
| Необъективность при составлении списка..... | 139 |
| Типы команд, выполняющих рефакторинг..... | 140 |
| Владельцы..... | 140 |
| Рекомендуемый подход..... | 142 |
| Группы очистки..... | 143 |
| Предложение | 145 |
| Метрики | 146 |
| Великодушие | 147 |
| Возможности | 147 |
| Обмен..... | 148 |
| Повторные попытки | 149 |
| Возможные результаты..... | 150 |
| Реалистичный сценарий | 150 |
| Что делать в худшем случае..... | 150 |
| Создание сильных команд | 152 |

ЧАСТЬ III. ВЫПОЛНЕНИЕ

| | |
|---|------------|
| Глава 7. Коммуникация | 154 |
| Внутри команды | 155 |
| Стендапы..... | 156 |
| Еженедельная синхронизация | 158 |
| Ретроспективы..... | 160 |
| Вне команды | 161 |
| При запуске проекта | 162 |
| Во время выполнения проекта | 164 |
| Экспериментируйте | 169 |
| Глава 8. Стратегии выполнения | 171 |
| Формирование команды | 171 |
| Парное программирование | 172 |
| Сохранение мотивации | 174 |
| Учет результатов | 176 |
| Промежуточные измерения | 176 |
| Обнаруженные ошибки | 177 |
| Удаление артефактов | 178 |
| Элементы, выходящие за рамки проекта | 178 |
| Продуктивное программирование..... | 179 |
| Прототипирование | 179 |
| Движение вперед маленькими шагами | 180 |
| Тестирование..... | 181 |
| «Глупые» вопросы..... | 182 |
| Заключение | 182 |
| Глава 9. Закрепление результатов рефакторинга..... | 183 |
| Содействие принятию рефакторинга | 184 |
| Образование..... | 185 |
| Активное образование | 186 |
| Пассивное образование..... | 188 |
| Закрепление | 189 |
| Прогрессивный лингтинг..... | 189 |

| | |
|---|------------|
| Инструменты анализа кода..... | 190 |
| Ворота и ограждения..... | 190 |
| Интеграция улучшений в культуру..... | 192 |
| ЧАСТЬ IV. РАЗБОРЫ ПРИМЕРОВ | |
| Глава 10. Избыточные схемы базы данных | 196 |
| Slack 101 | 197 |
| Архитектура Slack 101 | 199 |
| Проблемы масштабирования..... | 202 |
| Загрузка клиента Slack | 203 |
| Видимость файла | 204 |
| Упоминания..... | 204 |
| Консолидация таблиц..... | 206 |
| Сбор разрозненных запросов | 207 |
| Разработка стратегии миграции..... | 209 |
| Количественная оценка нашего прогресса | 212 |
| Попытка получить помошь | 213 |
| Сообщения об успехах | 215 |
| Заключительные шаги..... | 216 |
| Извлеченные уроки..... | 218 |
| Разработка четкого плана выполнения | 218 |
| Анализ истории кода..... | 219 |
| Обеспечение адекватного тестового покрытия..... | 220 |
| Сохранение мотивации | 221 |
| Концентрация на стратегических этапах..... | 221 |
| Выбор метрик..... | 222 |
| Ключевые моменты | 222 |
| Глава 11. Переход к новой базе данных | 223 |
| Распределение по рабочим пространствам..... | 224 |
| Миграция таблицы channels_members на Vitess..... | 225 |
| Схема шардирования | 226 |
| Разработка новой схемы | 228 |

| | |
|--|------------|
| Разбиение запросов с оператором JOIN | 230 |
| Сложности развертывания | 235 |
| Режим Backfill..... | 236 |
| Режим Dark..... | 237 |
| Режим Light..... | 242 |
| Режим Sunset | 243 |
| Заключительные шаги..... | 244 |
| Извлеченные уроки..... | 247 |
| Реалистичные оценки..... | 247 |
| Коллеги, которые вам нужны | 248 |
| Планирование объема работ..... | 248 |
| Выбор места коммуникации..... | 249 |
| План развертывания | 250 |
| Основные моменты | 250 |
| Об авторе | 252 |
| Об обложке | 253 |

Предисловие

Рефакторингу посвящено уже много книг, но в большинстве из них рассматриваются детали улучшения небольших частей кода по строке за раз. Однако мне кажется, что самая сложная часть рефакторинга не в поиске точного способа улучшения кода, а скорее в происходящих вокруг него процессах. Можно пойти еще дальше и сказать, что для любого крупного программного проекта мелочи редко имеют значение. Самая большая проблема кроется в координации сложных изменений.

В книге я пытаюсь помочь вам разобраться в этих сложностях. Она представляет собой результат многолетнего опыта выполнения всевозможных вариантов рефакторинга. Благодаря многим проектам, которыми я руководила в Slack, компания смогла сильно расшириться. Изначально у нашего крупнейшего клиента было 25 000 сотрудников. Теперь мы поддерживаем приложение, где общается организация, насчитывающая 500 000 человек. Разработанные нами стратегии для эффективного рефакторинга позволяют приспособиться к взрывному росту организации. Наша собственная команда инженеров выросла почти в шесть раз. Успешно запланировать и выполнить проект, затрагивающий как значительную часть кодовой базы, так и сотрудников, очень непросто. Я надеюсь, что эта книга даст вам необходимые инструменты и знания.

Для кого предназначена книга

Если вы вместе с десятками других коллег работаете с большой сложной кодовой базой, эта книга для вас!

Если вы молодой разработчик, желающий улучшить свои навыки и изменить к лучшему ситуацию в компании, участие в серьезном рефакторинге даст вам такую возможность. Последствия подобных проектов выходят далеко за рамки отдельной команды. При этом они не настолько интересны,

чтобы за них сразу же хватались старшие инженеры. Но для молодых сотрудников участие в рефакторинге — отличная возможность приобрести новые профессиональные навыки (и укрепить уже имеющиеся). Эта книга расскажет, как органично выполнять такой проект от начала до конца.

Информация отсюда станет полезным ресурсом и для опытных специалистов, способных самостоятельно решить любую проблему, но разочаровавшихся из-за того, что другие не понимают ценности их работы. Тем, кто ищет способы подтянуть коллег до своего уровня, эта книга расскажет о стратегиях, с помощью которых можно стимулировать их посмотреть на технические проблемы с вашей точки зрения.

Техническим директорам, стремящимся помочь своим подчиненным в выполнении крупномасштабного рефакторинга, эта книга покажет, как поддерживать свою команду на каждом этапе проекта. Я предпочла не углубляться в технические детали, поэтому мои рекомендации будут полезны всем участникам рефакторинга независимо от специализации.

Почему я решила написать эту книгу

Участвуя в первом рефакторинге, я понимала, *зачем* менять код, и знала, *как* это выполнить. Но не имела представления о том, как сделать это безопасно, постепенно, не вызывая всеобщего недовольства. Мне хотелось, чтобы изменения затронули как можно больше сфер. Я не задумывалась о таких вещах, как влияние рефакторинга на работу других отделов или способы привлечения коллег к участию в проекте. Я просто шла вперед (все детали описаны в главе 10).

В последующие годы мне пришлось участвовать в переделке множества строк кода, в результате чего мы иногда получали плохо выполненные проекты. Но я извлекла из тех случаев несколько важных уроков, о которых стала рассказывать на конференциях. Эти выступления нашли отклик у сотен разработчиков, которые тоже сталкивались с проблемами рефакторинга больших фрагментов кода. Было очевидно, что в наших знаниях по этой теме есть пробелы. Особенно это касалось профессионального подхода к написанию программного обеспечения. Во многих смыслах эта книга пытается учить важным вещам, не входящим в стандартную учебную программу по информатике. Преподавать все это в классе слишком сложно. Возможно, по книге научить этому тоже не очень реально, но почему бы не попробовать.

Структура издания

Книга разделена на четыре части и разбита на главы в соответствии с хронологическим порядком этапов выполнения крупномасштабного рефакторинга.

- Часть I знакомит с важными концепциями рефакторинга.
 - В главе 1 обсуждаются основы и отличие масштабного рефакторинга от более мелких переделок кода.
 - Глава 2 описывает причины деградации кода и ее влияние на эффективность рефакторинга.
- Часть II посвящена всему, что нужно знать о планировании рефакторинга.
 - В главе 3 представлен обзор метрик для количественной оценки проблем, которые пытаются решить рефакторинг. Вы узнаете, как в цифрах описать исходное состояние системы.
 - Глава 4 объясняет, из каких компонентов состоит план выполнения рефакторинга и как его правильно составить.
 - В главе 5 обсуждаются способы получения поддержки рефакторинга от руководства всех уровней и коллег.
 - Глава 6 описывает, как определить лучших кандидатов для работы над рефакторингом, и дает советы, как привлечь их к участию в проекте.
- В части III вы узнаете, как убедиться в корректном протекании рефакторинга на каждом этапе его выполнения.
 - Глава 7 рассказывает, как общаться и взаимодействовать с членами команды и любыми внешними заинтересованными сторонами.
 - В главе 8 рассматриваются способы сохранения темпов выполнения рефакторинга.
 - Глава 9 советует, как лучше сохранить изменения после рефакторинга.
- В части IV рассматриваются примеры из проектов с моим участием в компании Slack. Эти проекты сильно повлияли на работу значительной части нашего основного приложения. Я надеюсь, что это поможет проиллюстрировать концепции из частей I–III.

Главы необязательно читать по порядку. Переход к новому этапу рефакторинга не означает, что мы не будем пересматривать сделанное ранее. Например, можно приступить к рефакторингу, имея четкое представление о команде, которая будет над ним работать, и, уже пройдя половину пути, обнаружить, что на самом деле нужно привлечь к проекту больше разработчиков, чем планировалось. Ничего страшного. Такие ситуации естественны!

Условные обозначения

В книге используются следующие условные обозначения:

Курсив

Курсивом выделены новые термины.

Моноширинный шрифт

Используется для листингов программ и внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширений.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.



Этот рисунок указывает на совет или предложение.



Этот рисунок указывает на примечание.

Примеры кода

Дополнительные материалы (примеры кода, упражнения и прочее) доступны по адресу <https://github.com/qcmaude/refactoring-at-scale>.

Эта книга предназначена помочь вам в работе. Поэтому все приведенные в тексте примеры кода вы можете использовать в своих программах и документации. Вам не нужно связываться с нами для получения разрешения, если вы не воспроизводите значительную часть кода. Если в вашей программе есть несколько фрагментов кода из этой книги, разрешения на использование не требуется, как и для ответов на вопросы с цитированием

примеров кода. А вот для продажи или распространения примеров из книг издательства O'Reilly получить разрешение понадобится, как и на включение большого количества примеров кода в документацию по вашему продукту.

Мы ценим указание авторства, хотя и не требуем его. Атрибуция обычно включает название, автора, издателя и ISBN. Например: «Масштабируемый рефакторинг», Мод Лемер («Питер»). Copyright 2021 Maude Lemaire, 978-5-4461-3921-7».

Если вы не уверены в том, что ваш вариант использования примеров кода не выходит за описанные выше рамки, не стесняйтесь обращаться за уточнениями по адресу permissions@oreilly.com.

Благодарности

Писать книгу непросто, и эта работа не стала исключением. Издание не появилось бы на свет без вклада множества людей.

Прежде всего я хочу поблагодарить своего редактора Джеффа Блейла. Именно он превратил меня из неопытного писателя в автора книги. Его отзывы всегда были точны и помогали мне более связно излагать мои мысли. Я просто не могу представить лучшего редактора для своей книги.

Хочу поблагодарить друзей и коллег, которые читали ранние версии нескольких глав: Моргана Джонса, Райана Гринберга и Джейсона Лишку. Их отзывы убедили меня в правильности моих идей и ценности информации для читателей. За поддержку и беседы, заставляющие задуматься, спасибо Джоанн, Кевину, Чейзу и Бену.

Я хотела бы поблагодарить Мэгги Чжоу за помощь в написании второго примера рефакторинга (глава 11). Она одна из самых вдумчивых, умных и энергичных коллег, с которыми мне когда-либо приходилось работать. Я очень рада, что мир сможет узнать о наших совместных приключениях!

Огромное спасибо моим научным редакторам Дэвиду Коттреллу и Генри Робинсону. Дэвид — мой близкий друг со времен учебы в университете. За годы работы в Google он руководил рядом крупномасштабных рефакторингов, а потом основал собственную компанию. Генри — мой коллега в Slack. Именно он множество раз вносил свой вклад в разработку программного обеспечения с открытым исходным кодом и лично видел стремительный

рост компаний Кремниевой долины. Они оба невероятно добросовестные разработчики, чье руководство и мудрость очень пригодились мне в работе над книгой. Я бесконечно благодарна им за часы, потраченные на проверку текста. Любые неточности в окончательной рукописи — только мои собственные ошибки.

Спасибо всем, кто когда-либо участвовал со мной в рефакторинге. Вас слишком много, чтобы перечислять поименно, но вы знаете, кого я имею в виду. Вы все помогли в формировании идей, изложенных мной в книге.

Спасибо за поддержку моей семьи (Саймону, Мари-Жозе, Франсуа-Реми, Софи, Сильвии, Джерри, Стефани и Селии).

Наконец, спасибо моему мужу Эйвери. За терпение и за то, что давал мне время, место и поддержку в процессе работы. Спасибо, что позволял мне днями обсуждать возникающие идеи. Спасибо, что верил в меня. Эта книга настолько же твоя, как и моя. Я люблю тебя.

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

Введение

ГЛАВА 1

Рефакторинг

Однажды меня спросили, чем мне так нравится рефакторинг. Почему на работе я то и дело возвращаюсь к подобным проектам? Я ответила, что такие вещи затягивают. Может, это просто любовь к наведению порядка. Все равно что аккуратно рассортировать специи в кухонном шкафу. Или радость избавления от беспорядка. Как возможность отнести мешок со старой одеждой в секонд-хенд. Или даже голос в голове, напоминающий, что крошечные постепенные изменения улучшат рабочие будни моих коллег. Но, скорее всего, все сразу.

В рефакторинге есть нечто, что нравится всем. Это может быть работа над новым функционалом или над масштабированием инфраструктуры. Всегда нужно искать баланс между написанием большего или меньшего количества кода. Важно думать о том, к чему приведут намеренно или случайно внесенные изменения. Код — это живой организм. Когда я думаю, что написанный мной код существует еще пять или десять лет, то невольно вздрогиваю. Надеюсь, что к тому времени кто-то либо полностью удалит его, либо заменит более приличной версией, подходящей для актуальных нужд приложения. Это и есть суть рефакторинга.

В этой главе мы определим несколько концепций. Начнем с базового определения рефакторинга. Взяв его за основу, определим масштабируемый рефакторинг. Чтобы развить некоторые положения из этой книги, я расскажу, почему важно серьезно относиться к рефакторингу и какие преимущества дает отработка этого навыка. Мы поговорим о том, чего можно ожидать от рефакторинга и какие риски следует учитывать, принимая решение о его необходимости. Мы составим баланс преимуществ и недостатков рефакторинга. На его основе рассмотрим сценарии, где к нему можно прибегать и где от него лучше воздержаться. Завершит главу небольшой пример практического применения этих концепций.

Что такое рефакторинг

Простыми словами, *рефакторинг* — это процесс, с помощью которого мы реструктурируем код (*factoring*) без изменения его внешнего поведения. Если вы думаете, что это очень общее и размытое определение, не волнуйтесь. Я умышленно даю его в таком виде! Рефакторинг может принимать разные формы в зависимости от кода. Для примера определим «систему» как код, принимающий набор выходных данных и на выходе дающий какие-то другие данные.

На рис. 1.1 показана конкретная реализация системы S. Ее создавали в сжатые сроки, так что на качестве авторы сэкономили. Со временем образовалась груда запутанного кода. К счастью, пользователей системы этот внутренний беспорядок никак не затрагивает. Они взаимодействуют с S через интерфейс и получают приемлемые результаты.

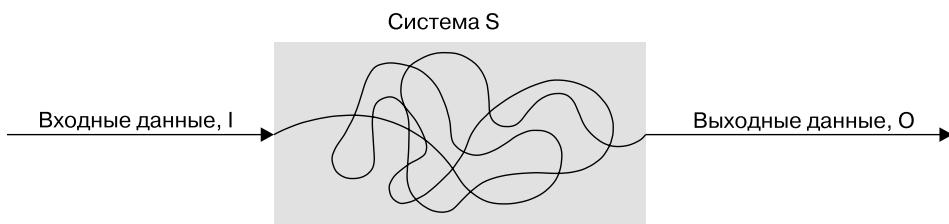


Рис. 1.1. Простая система с входными и выходными данными

Несколько смелых разработчиков очистили внутреннюю часть системы, получив вариант S' (рис. 1.2). Хотя код стал более аккуратным и упорядоченным, для пользователей S' абсолютно ничего не изменилось.

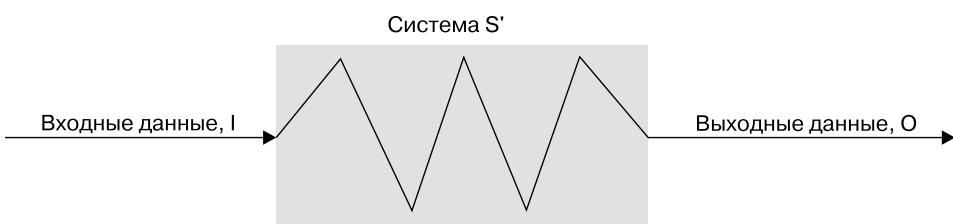


Рис. 1.2. Простая перепроектированная система с входными и выходными данными

Система S может выглядеть как угодно. Это может быть один оператор `if`, функция из десятка строк, популярная библиотека с открытым исходным кодом, огромное приложение или что-то среднее между ними. Такие же разнообразные формы способны принимать входные и выходные данные. Система может работать с записями базы данных, наборами файлов или потоками данных. На выходе она не просто возвращает значения, но может дополнительно выполнять такие действия, как вывод на консоль или сетевой запрос. На рис. 1.3 мы видим пример системы в виде RESTful-службы, отвечающей за работу с пользовательскими сущностями.



Рис. 1.3. Система на примере простого приложения

Постепенно мы будем развивать определение рефакторинга и перейдем к рассмотрению разных аспектов этого процесса. Убедитесь, что мы одинаково понимаем происходящее, будет проще, связывая каждую идею с конкретным примером.

К сожалению, привести реальные примеры из сферы программирования сложно. Из-за многообразия вариантов выбор одного даст преимущество только определенной группе читателей. Причем тех, кто хорошо знаком с этой областью, может разочаровать упрощение концепций для краткости или игнорирование некоторых нюансов. И чтобы все были в равных условиях, покажем проблему в общем виде на примере знакомого всем бизнеса — химчистки.

Расположенная на оживленной улице Спрингфилда химчистка Саймона открыта с понедельника по субботу в рабочие часы и принимает вещи как в стирку, так и для химической чистки. Срок выполнения заказа зависит от количества вещей, срочности и сложности. Это может занять от двух до шести рабочих дней.

Как это связано с нашим определением системы? В данном случае система — это набор операций по чистке вещей. Входные данные — грязная одежда, которую сдают клиенты. Возвращаемая чистая одежда — результат работы системы. Все тонкости обработки скрыты от потребителя. Люди сдают грязную одежду и надеются на добросовестность служащих химчистки. При этом система довольно сложна. В зависимости от типа вводимых данных (кожаная куртка, груда носков, шелковая юбка) выполняются одна или несколько операций для обеспечения нужного результата (чистой одежды). В промежутке между сдачей и получением вещей может произойти много сбоев. Например, может потеряться ремень, останется незамеченным пятно, рубашку случайно отдадут другому клиенту. Но если сотрудники работают слаженно, машины исправны, а квитанции оформляются как положено, система будет бесперебойно работать и легко выполнять заказы.

Допустим, химчистка Саймона до сих пор работает с бумажными квитанциями. Каждый клиент указывает на бланке свое имя и номер телефона, а приемщик присваивает их заказу номер. Если клиент потеряет квитанцию, в химчистке могут легко найти копию, пролистав последние заказы, отсортированные по фамилиям в алфавитном порядке. Если клиент не только потерял квитанцию, но и вовремя не пришел за своим заказом, служащему приходится искать копию в архиве. Но даже в этом случае все еще можно получить чистые вещи, просто процедура выдачи займет больше времени. Бумажные квитанции неудобны и при подсчете дохода в конце каждого месяца, ведь приходится вручную сопоставлять все транзакции (как по кредитной карте, так и наличными) с выполненными заказами. Поэтому команда решает упростить рабочий процесс, перейдя от бумажного документооборота к системе кассовых терминалов. По завершении рефакторинга клиенты все еще будут оставлять грязные вещи и забирать их чистыми через несколько дней. Для них ничего не изменится, а вот для сотрудников химчистки процесс пойдет более гладко.

Что такое масштабируемый рефакторинг

В конце 2013 года все основные американские новостные агентства объявили, что запуск сайта Healthcare.gov, который был символом реформы здравоохранения, завершился полным фиаско. Множество дыр в безопасности, многочасовые отключения и огромное количество серьезных ошибок. Перед запуском оказалось, что стоимость выросла почти до двух миллиардов долларов, а кодовая база разрослась до более пяти миллионов строк кода.

Провал Healthcare.gov был обусловлен неудачной разработкой, связанной с бюрократической политикой правительства, когда администрация Обамы объявила о планах вложить в улучшение сервиса значительные средства. Но основной проблемой стали неоспоримые трудности, связанные с изменением архитектуры огромного программного комплекса. В последующие месяцы команды, которым было поручено переписать Healthcare.gov, с головой погрузились в почти полную переработку кодовой базы — в масштабируемый рефакторинг.

Рефакторинг называется масштабируемым, если влияет на значительную часть системы. Обычно (но бывают и исключения) это означает большую кодовую базу (миллион или более строк) приложения с множеством пользователей. Пока есть старые системы, останется потребность в таких реорганизациях, когда разработчикам приходится переосмыслять всю структуру кода и искать способы ее эффективного улучшения.

В чем разница между рефакторингом огромной кодовой базы и рефакторингом небольшого, более четко определенного приложения? Для небольшой системы проще обнаружить итеративные способы улучшения (рассматривая отдельные функции или классы), но почти невозможно предсказать, как это влияет произвольное изменение на огромную сложную систему. Есть много инструментов для выявления проблем в структуре кода или автоматического обнаружения улучшений в отдельных фрагментах. Но как автоматизировать рассуждения людей на тему реструктуризации быстрорастущих приложений с огромной кодовой базой?

Можно возразить, что для совершенствования таких систем подойдут постоянные небольшие кумулятивные преобразования. Конечно, так мы пойдем в верном направлении, но их эффективность сильно упадет после выполнения простейших действий, и вносить изменения осторожно и постепенно станет сложнее.

Масштабируемый рефакторинг сводится к определению системной проблемы в кодовой базе, разработке лучшего решения и его стратегическому и упорядоченному внедрению. Для определения системных проблем и поиска их решений нужно хорошо понимать один или несколько основных разделов приложения. Много сил уйдет и на корректное распределение решения на всю пораженную область.

Масштабируемый рефакторинг тесно связан с рефакторингом работающих систем. Многие из нас работают над приложениями с частыми циклами

развертывания. Новый код отправляется в Slack десять раз в день. Нужно корректно вписываться в эти циклы, чтобы минимизировать риски и недостатки пользователей. Без понимания того, как в процессе рефакторинга осуществлять стратегическое развертывание в разных точках, мы можем прийти к полному отключению сервиса.

Как будет выглядеть этот процесс на примере химчистки Саймона? Представим, что внедрение системы кассовых терминалов позволило всего за два года открыть пять новых точек в соседних городах! Теперь компания работает в нескольких местах, и у нее появились другие проблемы. Для снижения расходов оборудованием для химчистки снабдили всего два приемных пункта из шести. Поэтому из четырех пунктов между нужна отвезти в ближайший с оборудованием. Машина компании объезжает четыре точки и сбрасывает собранные вещи в большие ящики на погрузочных площадках двух химчисток. Сотрудники Саймона сортируют вещи, чистят их и возвращают в нужный пункт приема. Это непростой процесс. Обе химчистки обрабатывают вещи из собственного пункта приема и из четырех других. Нередко во время перекладывания из машины в ящики вещи спутываются. Из получившейся кучи приходится вручную доставать срочные заказы.

Как улучшить функционирование химчистки Саймона? Может, стоит купить оборудование для химчистки еще в один пункт, чтобы дополнительные заказы приезжали всего с трех? Как тогда должны поменяться маршруты машин? Сократит ли открытие еще одной химчистки время обработки заказов? Как сделать так, чтобы при перегрузке вещи не цеплялись друг за друга? Можно ли научить водителей сортировать вещи по срочности заказа? Может быть, лучше везти вещи в химчистку сразу после обеда и вскоре после закрытия, чтобы высвободить дополнительное время на их прием? Вариантов масса, и многие из них можно сочетать. Вопрос лишь в том, как выбрать самую выгодную стратегию. Это нелегкий выбор. Примерно так же выглядит рефакторинг больших приложений.

Зачем нужен рефакторинг

В теории рефакторинг выглядит интересно. Но как понять, не станет ли чтение этой книги пустой тратой времени? Я надеюсь, что каждый читатель сможет пополнить свой арсенал новыми инструментами, но есть и другие причины не откладывать книгу в сторону.

Уверенность в способности произвести рефакторинг позволяет приступить к созданию системы до того, как станут понятны все ее составляющие, подводные камни и пограничные случаи. Возможность эффективно улучшать компоненты во время разработки, которую используют по мере усложнения системы, избавляет от больших временных трат на предварительное планирование архитектуры программы. Отточенные навыки управления кодом позволяют не ограничивать себя одним проектным решением. В процессе программирования лучше написать простой код, работающий в данных обстоятельствах, а не планировать на десяток шагов вперед.

Вы поймете, что всегда можно найти решение лучше, но не всегда это будет просто. Программирование не игра в шахматы, где при заданной конфигурации доски хорошие игроки могут за считанные минуты ловко разыграть несколько партий. К сожалению, в этой области никто не даст вам полный набор возможных ходов. Конечное состояние тоже не определено. Я не хочу сказать, что искать надежное решение поставленной задачи с учетом всех требований бессмысленно. Но не тратьте много времени на идеальную шлифовку последних 10–20 процентов. Имея навыки рефакторинга, вы сможете без проблем доработать решение с учетом окончательных спецификаций.

Выгоды рефакторинга

Рефакторинг не только позволяет решать задачи увереннее. У него есть и другие преимущества. Пусть этот инструмент подходит не для всех задач, в будущем он может положительно повлиять на приложение, команду разработчиков и организацию в целом. Рассмотрим два основных преимущества: повышение производительности труда разработчиков и упрощение поиска ошибок. Некоторые считают, что рефакторинг дает и другие преимущества. Но я утверждаю, что все они могут быть отнесены к одной из двух этих групп.

Продуктивность разработчика

Одна из основных целей рефакторинга — получение более простого кода. Это упрощение помогает понять, что делает код, вам и всем, кто будет иметь с ним дело потом. Новая версия кода повысит уровень абсолютно всех членов команды независимо от их опыта.

Обычно инженеры-разработчики хорошо знакомы с некоторыми частями кодовой базы. Но по мере ее роста незнакомых фрагментов становится больше. Кроме того, велика вероятность появления зависимостей.

Представьте, что в процессе внедрения новой функции вы переходите от хорошо знакомого кода на неизвестную территорию. Если эта кодовая база в хорошем состоянии и регулярно обновляется с учетом исправлений ошибок и меняющихся требований, идеальное место для внесения изменений вы найдете гораздо быстрее и быстрее разработаете простое решение. Может быть и так, что этот незнакомый вам код со временем разросся и стал хуже из-за частичного исправления ошибок. В таком случае придется просмотреть каждую его строку, чтобы сначала понять принцип его действия, и только после этого появится возможность перейти к поиску подходящего решения. Нередко бывает, когда к работе над этим истерзанным кодом привлекают кого-то еще. Например, коллегу или того, кто хорошо знаком с кодом и может отвечать на возникающие вопросы.

ЭВОЛЮЦИЯ ЗНАКОМСТВА С КОДОВОЙ БАЗОЙ

Если кодовая база невелика и с ней работает небольшая команда, ее участники будут хорошо знакомы со всеми фрагментами кода. По мере добавления и изменения новых модулей степень знакомства будет уменьшаться. Постепенно члены команды начнут специализироваться на отдельных частях. В итоге наступит момент, когда знать весь код не сможет ни один сотрудник (даже если он работает с начала проекта!).

Представьте, что коллеге из другой команды придется читать ваш код. Сможет ли он легко понять, как все работает? Или вы ожидаете недоуменных взглядов и просьбы проанализировать код?

Новый человек в команде не знает, как выглядит кодовая база, с которой ему предстоит работать. Способность разобраться в любой части кода напрямую зависит от его читабельности. Чем он аккуратнее, тем проще новичку определить взаимоотношения между модулями и понять, что делает код, не беспокоя своими вопросами остальных.

Впрочем, нужно понимать, когда и как можно задавать вопросы своим коллегам. Это невероятно важный навык, который тоже следует оттачивать. Научиться оценивать, сколько времени вам нужно, чтобы сформировать свое понимание, прежде чем обращаться за помощью, сложно, но важно для

вашего роста как разработчика. В вопросах менее опытных коллег нет ничего плохого. Но если вы штатный инженер-разработчик и вас просто засыпают вопросами, возможно, пришло время написать документацию и провести рефакторинг кода.

При разработке чего-то нового люди склонны пользоваться базовыми шаблонами. Если они будут ясными и краткими, более вероятно, что новый код получится четким и сжатым. Верно и обратное: если вы ориентируетесь на громоздкие решения, вы будете и дальше плодить громоздкий код. Нужно обеспечить положительную обратную связь для начинающих разработчиков. Если код, с которым они регулярно взаимодействуют, будет простым для понимания, они будут придерживаться такого же в собственных решениях.

Обнаружение ошибок

Выявление и устранение ошибок — важная (и увлекательная!) часть нашей работы. Эффективным инструментом для решения обеих задач может стать рефакторинг. В процессе разбиения сложных операторов на более мелкие фрагменты и превращения алгоритмов в новые функции можно не только лучше понять, что делает код, но и найти ошибку. Рефакторинг, проводимый в процессе активного написания кода, упрощает выявление ошибок на ранних этапах разработки. Это позволяет полностью их избежать.

Представим, что несколько часов назад введен в эксплуатацию новый код. Некоторые внесенные изменения находятся в файлах, которые все опасаются редактировать. Код в этих файлах очень сложно читается и имеет множество подводных камней. К сожалению, в процессе тестирования обновлений не рассматривался ни один из пограничных случаев. Из службы поддержки клиентов сообщают об ошибке, с которой начинают сталкиваться пользователи. Рабочая группа быстро понимает, что ошибка, как это часто бывает, скрывается в той самой «страшной» части кода. К счастью, одному из членов группы удается последовательно воспроизвести ошибку. Вместе вы пишете тест, подтверждающий, что при таких обстоятельствах она действительно возникает. Теперь ошибку нужно локализовать. Вы систематически разбиваете сложный код на более мелкие фрагменты. Конвертируете длинные однострочные выражения в сжатые многострочные операторы и переносите содержимое нескольких условных блоков кода в отдельные функции. Наконец ошибка обнаружена. Теперь код

стал проще, и ее можно быстро исправить, протестировать и отправить исправление клиентам.

Иногда ошибки становятся лишь досадной неприятностью, но бывают случаи, когда они блокируют возможность пользоваться приложением. Чем серьезнее ошибка, тем более оперативно на нее нужно реагировать. Но для удобства пользователей команда должна уметь быстро исправлять любые ошибки. Хорошее состояние кодовой базы сильно сократит время поиска и исправления ошибки, радуя пользователя выпуском обновления в рекордно короткий срок.

Риски рефакторинга

Несмотря на очевидные преимущества рефакторинга, есть серьезные риски и подводные камни, которые нужно учитывать до начала работы по улучшению кодовой базы. Хотелось бы повториться: рефакторинг требует гарантированного сохранения одинакового поведения на каждой итерации. Удовостериться в отсутствии изменений можно, написав набор тестов (для отдельных модулей, для сопряжений, сквозных). Поэтому бессмысленно прилагать какие-либо усилия к началу рефакторинга, пока не достигнут достаточный тестовый охват. Но даже при тщательном тестировании всегда остается вероятность того, что мы что-то не заметим. Кроме того, всегда нужно помнить о конечной цели: улучшить код так, чтобы он стал понятным и вам, и всем, кто будет с ним работать.

Риск регрессии

Рефакторинг непротестированного кода очень опасен, поэтому крайне не рекомендуется его проводить. Почему команды разработчиков, оснащенные самыми высококачественными и продуманными пакетами для тестирования, все еще отправляют в эксплуатацию код с ошибками? Любое изменение нарушает равновесие системы. Несмотря на стремление минимизировать это нарушение, риск регрессии все равно есть. Особенно об этом нужно беспокоиться при рефакторинге запутанных фрагментов кодовой базы. Часто их состояние такое из-за того, что ими долго никто не занимался. Они сплошь и рядом попадаются в приложениях быстрорастущих компаний и реже всего тестируются. Попытка как-то причесать такие файлы или функции напоминает прогулку по минному полю. Пройти его можно, но это очень опасно.

«Спящие» ошибки

Рефакторинг позволяет выявлять не только существующие, но и «спящие» ошибки. К этой категории я причисляю и регрессию, возникающую при реструктуризации кода. Рассмотрим это явление на примере химчистки Саймона. Для получения скидки поставщика компания начала заказывать чистящие средства большими партиями. Но в офисе мало места для хранения, поэтому их начали складывать рядом с погрузочной платформой. Однажды несколько дней шел дождь, и оказалось, что ближайшие к двери коробки разваливаются от сырости, так как задняя дверь закрывается неплотно и пропускает воду. С такой проблемой в химчистке никогда не сталкивались, потому что раньше рядом с погрузочной платформой ничего не было. Изменение схемы хранения выявило критический недостаток в инфраструктуре, который в иных обстоятельствах никогда бы не обнаружили.

Неконтролируемый рост проекта

Чем-то рефакторинг похож на поедание пирожных: их вкус так восхитителен, что легко увлечься и съесть целую дюжину. Но проглотив последний кусок, вы начинаете испытывать сожаление. Наблюдать, как вашими усилиями улучшается код, очень приятно! Легко увлечься и не заметить, как число изменений выходит за *разумные пределы*. Разумность этих пределов зависит от кодовой базы. Это может быть как одна функция, так и небольшой взаимозависимый набор библиотек. В идеале рефакторинг кода желательно ограничить редактированием, которое другой разработчик может легко просмотреть в рамках одного набора изменений.

При планировании более крупного рефакторинга, особенно такого, который может занять несколько месяцев, необходимы жесткие рамки. Все мы сталкивались с неожиданностями при рефакторинге небольших фрагментов (несколько строк кода, отдельные функции). Можно внести ряд улучшений и эффективно справиться с этими странностями, но при работе с большим массивом кода такой подход становится опасным. Чем больше площадь планируемого рефакторинга, тем с большим количеством неожиданных проблем вы столкнетесь. Это не значит, что вы плохой программист. Просто человеческие возможности не безграничны. Поэтому для уменьшения

возможности серьезного регресса или столкновения со «спящими» ошибками и повышения производительности нужно придерживаться четкого плана. Долговременный систематический рефакторинг сложен сам по себе. А если в процессе еще начать менять правила игры, его проведение станет невозможным.

Ненужное усложнение

Не тратьте много времени на проектирование вначале. Более того, нужно быть готовым менять планы. Основная цель рефакторинга — создание удобного для человека кода, пусть даже за счет исходного проекта. Если сосредоточиться не на процессе, а на желаемом результате, велика вероятность, что приложение окажется более надуманным и сложным, чем было изначально. Рефакторинг должен быть итеративным на всех уровнях. Небольшие осознанные шаги в одном направлении и поддержка существующего поведения на каждой итерации позволяют лучше сосредоточиться на цели. Этого намного проще достичь, если брать на себя столько кода, сколько умещается на экране, а не три десятка библиотек за раз. При планировании нового проекта многие стараются разработать подробную спецификацию и план работы. И даже если для рефакторинга нужно много усилий, важно понимать, как должен выглядеть код по его завершении.

Когда начинать рефакторинг

На этот вопрос можно ответить так: когда выгода от него перевешивает риски. Но такой ответ совершенно неинформативен — непонятно, как просчитать выгоды и риски для множества частей сложной системы. Как же понять, что наступил критический момент, когда без рефакторинга уже не обойтись?

По моему опыту, «критический момент» на самом деле — это *временной интервал*, индивидуальный для каждого приложения. Определение верхней и нижней границ этого интервала вносит в вопрос о рефакторинге толику субъективности. Формулы, которой можно было бы воспользоваться и однозначно узнать «да» или «нет», не существует. К счастью, принять решение можно, опираясь на уже имеющиеся эмпирические данные.

Небольшой масштаб

Почти никто не мешает провести рефакторинг небольшого несложного фрагмента хорошо протестированного кода. Если вы не сомневаетесь, что новое решение будет лучше старого, и не боитесь, что изменение затронет слишком большую часть кода, то стоит попробовать. Тщательно выполните несколько коммитов — и пусть новый код работает! Далее в главе вы найдете пример, попадающий под эту категорию.

Сложность кода мешает активному развитию

Встречается код, при чтении которого портится настроение и поднимается давление. Но рано или поздно наступает момент, когда, скав зубы, приходится вносить в него запланированные изменения. Такое вынужденное редактирование только прибавит вам проблем. Когда человек концентрируется на том, чтобы следовать плану, одновременно удерживая в голове множество аспектов проблемы, есть риск упустить из виду *настоящую* цель. Как достичь реальной цели, если ум занят другим?

Если конкретный фрагмент кода еще не успел причинить вреда, мы часто идем на риск и проводим его рефакторинг. Но когда вред уже причинен (и, возможно, неоднократно), редактировать такой код для предотвращения будущих ошибок может быть более рискованно, чем оставить его в нынешнем состоянии. Если вы не уверены, обсудите ситуацию с коллегами и соберите данные о количестве ошибок за последние полгода, которые можно связать с этой частью кодовой базы.

Изменение требований к продукту

Радикальные сдвиги в требованиях к продукту часто становятся причиной кардинального редактирования кода. Даже если приложить массу усилий к написанию абстрактных расширяемых решений для любой функциональности приложения, будущее предсказать невозможно. Код, легко поддающийся адаптации в случае небольших отклонений, редко идеально адаптируется к более крупным. Зато мы получаем редкую возможность начать все с чистого листа и пересмотреть проект.

Вы можете подумать, что такого рода сдвиги никак не могут сохранить поведение. Ведь с теми же входными данными теперь нужен совсем другой результат! Разве это подходящее время для рефакторинга? Если текущий код не соответствует новым требованиям, нужно предложить решение, которое продолжит поддерживать имеющуюся функциональность и позволит легко добавлять новые. Можно сначала провести рефакторинг кода, а после (и только после!) реализовать новые функции поверх него. В итоге вы внедрите стандарт высококачественного кода, извлечете выгоду из рефакторинга и поддержите коммерческие цели. Что нам и требуется.

Производительность

Повысить производительность непросто. Сначала нужно детально разобрать существующее поведение, а после определить, с помощью каких рычагов можно склонить чашу весов в нужном направлении. Такие вещи проще всего проделывать с чистого листа (или создав такой лист в качестве первого шага). Определенные вами рычаги важно правильно изолировать, чтобы манипуляция ими происходила без побочных эффектов.



Не все разработчики считают повышение производительности причиной для рефакторинга. Некоторые утверждают, что производительность системы неразрывно связана с ее поведением. Иначе говоря, ее изменение меняет поведение. Я с этим не согласна. Если мы продолжим определять рефакторинг через нашу обобщенную систему, для которой мы предоставляем входные данные, и в результате все равно получаем ожидаемый набор выходных данных, то повышение скорости их генерации — допустимая форма рефакторинга.

Уникальность рефакторинга для повышения производительности в том, что код в результате не становится понятнее. Иногда в кодовой базе можно наткнуться на длинный блок комментариев с предупреждением. По моему опыту, в большинстве случаев это предупреждение о подводных камнях: странном поведении приложения, временном обходном пути и своеобразном способе повышения производительности. Большая часть повышающего производительность кода написана грамотно и с глубоким пониманием кодовой базы. Но сделано это так, чтобы минимизировать затрагиваемую поверхность. Такие «улучшения» деградируют гораздо быстрее, так что речи об устойчивости, которой призван способствовать рефакторинг, тут не идет. Примеры увеличения производительности, которые попадают под

определение рефакторинга, всегда глубоки и разветвлены. Это примеры эффективного масштабируемого рефакторинга. Подробно мы рассмотрим их во второй части.

Переход к новой технологии

В мире разработки программного обеспечения регулярно внедряются новые технологии. Неважно, что нами движет — желание идти в ногу с новейшими тенденциями в отрасли, масштабировать решение для множества пользователей или усовершенствовать продукт новым способом, — мы постоянно оцениваем новые библиотеки с открытым исходным кодом, протоколы, языки программирования, поставщиков услуг и т. д. Решения воспользоваться чем-то новым принимаются нелегко. Отчасти из-за стоимости интеграции в существующую базу кода. Если мы предпочтем полностью *заменить* существующее решение, придется разработать план депрекации, идентифицировав все задействованные места вызовов и перенеся их (иногда по одному). Если же новую технологию *планируется внедрить* в будущем, нужно определить перспективные технологии, составив план расширения их применения для всех возможных сценариев использования.

Я не буду перечислять все варианты влияния новой технологии на систему (их очень много). Из написанного выше и так ясно, что каждый раз нужен тщательный анализ состояния существующей системы. В процессе могут появиться отличные возможности для рефакторинга! Хотя на этот счет есть и другое мнение. Многие разработчики считают, что внедрение новой технологии уже достаточно рискованно и добавление еще каких-то изменений будет лишним. В этом есть рациональное зерно. Нет ничего хуже, чем добавлять новшества в запутанную систему. Переход к новой технологии будет проще, если сначала привести код в порядок.

Этот подход легко применить к химчисткам Саймона. Допустим, туда было заказано современное экологичное оборудование. При составлении плана его установки выясняется, что уже имеющееся оборудование размещено нерационально. Чтобы забрать отсортированную одежду со стеллажей, сотрудникам приходится пройти мимо ряда машин почти 10 метров. Перестановка оборудования поможет сократить каждый цикл на несколько минут. В результате новые машины расставляются иначе. Это позволяет не только снизить воздействие на окружающую среду, но и повысить производительность сотрудников.

Когда не нужно делать рефакторинг

Рефакторинг может стать удивительно полезным. Поэтому многие разработчики считают, что время, потраченное на него, потрачено не зря. На самом деле все не так просто. Для рефакторинга важно правильно выбрать место и время. В этом разделе вы узнаете, когда его делать *не стоит*.

Для забавы или со скуки

Закройте на минуту глаза и представьте, что смотрите на очень сложную функцию. Она слишком длинная, пытается одновременно делать слишком много, ее название никак не отражает ее суть.

Вероятно, вы захотите ее исправить. Разбить на четко определенные лаконичные фрагменты, дать переменным осмысленные имена. Это было бы так увлекательно! Но разве у вас сейчас нет дел поважнее? Возможно, коллега уже несколько дней ждет от вас обзора кода или все это время вы откладывали написание тестов? Копаясь для развлечения в старом, выведенном из эксплуатации коде и редактируя его, вы вредите себе (и товарищам по команде).

Обычно, когда рефакторинг проводится для развлечения, можно не обращать особого внимания на то, как вносимые изменения повлияют на остальной код, систему в целом и на работу коллег. Скорее всего, будут добавляться надуманный функционал или испытываться новые шаблоны. Пробовать что-то новое и отрабатывать навыки программирования все-таки лучше отдельно от рефакторинга. Он должен быть осознанной процедурой, где нужно стараться внести больше улучшений с минимальным количеством изменений.

Вы случайно проходили мимо

Представьте, что через несколько месяцев вы вернулись к своему коду, чтобы расширить функционал. Но становится ясно, что он совсем не похож на то, что вы писали. Возможно, вы стали жертвой реформатора. Обычно это достаточно опытный сотрудник, хорошо понимающий, как писать код. Инженеры-разработчики консультируются с ним по поводу проектных решений. Но у них есть склонность переписывать чужой код, попадающий им в руки. Такие сотрудники думают, что так будет лучше.

Возможно, вы захотите согласиться с внесенными изменениями, но есть один нюанс. Чужие правки часто снижают производительность ответственных за код. Хорошее знакомство с кодом обеспечивает наибольшую продуктивность. Когда нужно быстро решить проблему, мы используем мысленную модель кода, чтобы сузить набор файлов, классов или функций, в которые могла вкрасться ошибка. А потом открываем редактор и видим там совершенно другую картину. Понятно, что об оперативном устранении проблемы уже не может быть и речи. За чужой рефакторинг приходится платить увеличением времени работы технического персонала и службы поддержки, а иногда и потерей бизнеса.

Не уведомить автора кода о проведенном рефакторинге — значит повести себя непорядочно по отношению к нему. Во-первых, это открытое выражение недоверия. Я бы предпочла, чтобы мне перечислили недостатки моего кода и подсказали, как их исправить, а не уведомляли постфактум, что кто-то за моей спиной устранил возникшие проблемы. Особенно обидно в такой ситуации начинающим инженерам. Представьте себе вчерашнего выпускника, который неделю трудился над кодом, а потом внезапно обнаружил, что кто-то из старших коллег, ни слова не сказав, внес туда правки.

Во-вторых, вряд ли такому стороннему любителю рефакторинга известны исходные обстоятельства появления кода. Почему их нужно знать? Программирование всегда связано с компромиссами. Например, можно получить решение, которое работает быстрее за счет структуры данных с большим объемом памяти. Или наоборот, уменьшить потребление памяти, заменив точные вычисления приблизительными. Я хочу сказать, что каждая строчка «плохого» кода направлена на решение какой-то задачи. Проведя рефакторинг наобум, можно пасть жертвой ошибки или уязвимости, которых авторы исходного кода тщательно пытались избежать.

ОБ АВТОРАХ КОДА

К сожалению, не каждый может проработать в одной компании всю жизнь. Бывает, что с автором исходного кода невозможно поговорить, потому что он здесь больше не работает. К счастью, прежде чем уйти, люди обычно передают информацию своим коллегам. Так что узнать о коде подробнее, вероятно, вы сможете у коллег создателя. Если за вашим желанием провести рефакторинг стоят *благие намерения*, как можно реже прикасайтесь к коду, в поддержке которого вы не принимаете активного участия. Но если вы все же решили это сделать, обязательно привлеките к процессу ответственных за этот код.

Чтобы обеспечить расширяемость кода

Многие специалисты по рефакторингу советуют эту процедуру как способ получения более расширяемого кода. Такой код действительно можно получить с помощью рефакторинга. Но переписывать его ради гибкости в будущем неразумно. Без четкого понимания дальнейшего результата вы можете зря потратить время на рефакторинг, и внесенные изменения не окупятся за все время эксплуатации кода.

Если можно внести в блок изменения для продвижения проекта, вряд ли нужно проводить его рефакторинг. В большинстве компаний есть список новых функций, которые планируется разрабатывать, и список ошибок для устранения. Почти всегда задачи из этих списков будут приоритетными. Так что без набора конкретных целей и убедительных аргументов, показывающих, как рефакторинг напрямую повлияет на прибыль компании, у вас вряд ли что-то получится. Но не пугайтесь! Позднее я расскажу, как обосновать необходимость рефакторинга.

Когда не хватает времени

Хуже кода, остро нуждающегося в рефакторинге, только код с рефакторингом, сделанным наполовину. Код в подвешенном состоянии сбивает разработчиков с толку. Если не указан конкретный момент завершения реорганизации, код приходит в состояние хаоса. Тому, кто читает код в процессе рефакторинга, обычно трудно определить направление, которому нужно следовать. Особенно если нет никаких комментариев. В предположениях о том, какой вариант будет в итоге принят, легко ошибиться. Можно отредактировать блок, который на самом деле собираются подвергнуть депрекации. Такие ошибки быстро накапливаются, что ухудшает состояние кода, который вы хотели улучшить.

Перед проведением рефакторинга убедитесь, что у вас достаточно времени для его завершения. Если времени нет, постарайтесь ограничить объем вносимых изменений. Никакие временные выгоды, полученные от незавершенного рефакторинга, не перевешивают замешательство и разочарование разработчиков, взаимодействующих с кодом после вас.

Первый пример рефакторинга

Мы заложили прочный фундамент для понимания целей рефакторинга и того, как при определенных обстоятельствах он позволяет улучшать код. Теперь рассмотрим небольшой пример. Ему далеко до масштабов того рефакторинга, который мы будем обсуждать дальше, но он отлично отображает некоторые базовые концепции.

Представим, что в университете для выставления оценок разработана и поддерживается простая программа. С ее помощью ассистенты проверяют, попадают ли оценки за задания в заданный диапазон. Дело в том, что преподаватели по-разному структурируют задания и не все наборы задач оцениваются по шкале от 0 до 100. Например, в задании из десяти вопросов, каждый из которых оценивается до 6 баллов, максимальная оценка — 60 из 60, а минимальная — 0 баллов.

Преподаватели используют эту программу, чтобы обеспечить попадание среднего балла в ожидаемый диапазон. Допустим, нужно, чтобы среднее значение для набора задач оказалось в диапазоне от 42 до 48 баллов (или 70–80 %). Этот диапазон можно задать программе, которая после обработки итоговых оценок определит, попадает ли среднее значение в заданные границы.

Функция `checkValid` реализует эту логическую схему (пример 1.1).

Пример 1.1. Небольшой запутанный фрагмент кода

```
function checkValid(
    minimum,
    maximum,
    values,
    useAverage = false)
{
    let result = false;
    let min = Math.min(...values);
    let max = Math.max(...values);
    if (useAverage) {
        min = max = values.reduce((acc, curr) => acc + curr, 0)/values.length;
    }

    if (minimum < 0 || maximum > 100) {
        result = false;
    } else if (!(minimum <= min) || !(maximum >= max)) {
        result = false;
    } else if (maximum >= max && minimum <= min) {
```

```
    result = true;  
}  
return result;  
}
```

В глаза сразу же бросаются некоторые изъяны. Во-первых, имя функции не полностью отражает ее суть. Неясно, чего можно ожидать от функции с общим именем вроде `checkValid` (особенно если перед ее объявлением нет комментариев). Во-вторых, непонятно, что представляют собой встроенные значения (0, 100).

Мы знаем принцип работы функции, поэтому можно сделать вывод, что это абсолютные минимум и максимум баллов для любого задания. В контексте минимального значения 0 имеет смысл, но почему в качестве верхней границы указано значение 100? В-третьих, следовать за логической схемой непросто. Помимо того что в коде много условных операторов, сложно воспринимается и их внутреннее содержимое. Если в функции есть ошибка, сразу заметить это почти невозможно. Есть и другие недочеты, но для простоты мы пока остановимся на этом.

Как получилось, что такой небольшой код оказался таким непонятным? Код в активной разработке регулярно понемногу редактируется (исправляются ошибки, добавляется новый функционал, выполняются настройки производительности и т. п.). Именно накопление этих небольших модификаций часто порождает более длинный и запутанный код. По структуре рассматриваемого кода можно выделить два изменения, которые были внесены в исходную функцию.

- Добавлена возможность проверки диапазона по среднему значению, а не по их сумме из предоставленного набора. Вывод: эта функция была введена позже. Я сделала это по двум причинам. Логическая переменная `useAverage` — это необязательный аргумент со значением по умолчанию `false`. Это значит, что не при всех вызовах функции ожидается четыре аргумента. Такие переменные (их еще называют переменными-флагами) указывают на проблемы в структуре кода. Скоро мы поговорим об этом подробнее. Кроме того, для удобства отображения нового среднего перезаписываются переменные `min` и `max`. Фактически автор искал самый простой способ добавить проверку диапазона, изменив при этом наименьшее количество кода.
- Было гарантировано, что значения, задающие границы диапазона, будут не меньше 0 и не больше 100. Запрет на задания с оценкой выше

100 баллов кажется странным. Но предположим, что так и было задумано. Условное выражение для проверки границ диапазона находится внизу. Скорее всего, его попросили добавить в уже готовый код, иначе эта проверка была бы выполнена сразу. А при редактировании кода под новое требование автор решил, что проще добавить новый условный оператор в самый конец. Чтобы подтвердить это предположение, можно посмотреть историю версий и найти исходный коммит.

Упрощение условных операторов

Начнем с упрощения логической схемы оператора `if`. Перенесем в начало проверку минимального и максимального значений (пример 1.2). Если хоть одно из этих значений выходит за пределы диапазона от 0 до 100, происходит возврат из функции.

Пример 1.2. Небольшой пример с ранним возвратом из функции

```
function checkValid(
  minimum,
  maximum,
  values,
  useAverage = false
) {
  if (minimum < 0 || maximum > 100) return false; ①

  let min = Math.min(...values);
  let max = Math.max(...values);

  if (useAverage) {
    min = max = values.reduce((acc, curr) => acc + curr, 0)/values.length;
  }

  if (!(minimum <= min) || !(maximum >= max)) return false; ②
  if (maximum >= max && minimum <= min) return true; ②

  return false;
}
```

- ① Если минимальное или максимальное значение вне допустимого диапазона, происходит выход из функции.
- ② Упрощаем логическую схему, завершая работу функции, когда это возможно.

Уже кое-что! Посмотрим, нельзя ли еще упростить логическую схему. Для этого посмотрим, в каких еще случаях функция возвращает значение `false`. Например, когда вычисленный минимум или максимум выходят за пределы установленных для них пороговых значений. Имеющиеся условные операторы можно переписать так, чтобы результат `true` возвращался только при соблюдении этих условий, как показано в примере 1.3.

Пример 1.3. Небольшой пример с упрощенной логической схемой

```
function checkValid(
    minimum,
    maximum
    values,
    useAverage = false
) {

    if (minimum < 0 || maximum > 100) return false;

    let min = Math.min(...values);
    let max = Math.max(...values);

    if (useAverage) {
        min = max = values.reduce((acc, curr) => acc + curr, 0)/values.length;
    }

    if (min < minimum) return false; ①
    if (max > maximum) return false; ①
    return true; ②
}
```

- ① Упрощаем логическую схему, рассматривая каждый случай отдельно. В результате каждый оператор `if` содержит только одно условие.
- ② Значение `true` функция возвращает, только убедившись в попадании обоих значений в заданный диапазон.

Удаление волшебных чисел

Следующий шаг — превращение числовых констант в коде (их еще называют волшебными числами) в переменные с информативными именами. Для наглядности мы переименуем и переменную `values` (значения) в `grades` (оценки). В принципе их можно было бы определить как константы в той же области видимости, что и объявление функции, но пока не будем усложнять. Итоговый код мы видим в примере 1.4.

Пример 1.4. Небольшой пример с более наглядными именами переменных

```
function checkValid(
    minimumBound, ❶
    maximumBound, ❶
    grades, ❶
    useAverage = false
) {

    // Корректные оценки не могут быть меньше 0
    var absoluteMinimum = 0; ❷

    // Корректные оценки не могут быть больше 100
    var absoluteMaximum = 100; ❸

    if (minimumBound < absoluteMinimum) return false; ❹
    if (maximumBound > absoluteMaximum) return false; ❹

    let min = Math.min(...grades);
    let max = Math.max(...grades);

    if (useAverage) {
        min = max = grades.reduce((acc, curr) => acc + curr, 0)/grades.length;
    }

    if (min < minimumBound) return false;
    if (max > maximumBound) return false;
    return true;
}
```

- ❶ Переименовываем параметры в соответствии с их назначением.
- ❷ Волшебные числа получили имена. Это добавило в программу дополнительный контекст.
- ❸ Упрощение логической схемы путем разбиения одного оператора `if` со сложным условием на два с простыми.

Извлечение автономной логики

Вычисление среднего можно извлечь в отдельную функцию (пример 1.5).

Пример 1.5. Небольшой пример с дополнительными функциями понятного назначения

```
function checkValid(
    minimum,
    maximum,
    grades,
    useAverage = false
```

```
){
  // Корректные оценки не могут быть меньше 0
  var absoluteMinimum = 0;

  // Корректные оценки не могут быть больше 100
  var absoluteMaximum = 100;

  if (minimumBound < absoluteMinimum) return false;
  if (maximumBound > absoluteMaximum) return false;

  let min = Math.min(...grades);
  let max = Math.max(...grades);

  if (useAverage) {
    min = max = calculateAverage(grades);
  }

  if (min < minimumBound) return false;
  if (max > maximumBound) return false;
  return true;
}

function calculateAverage(grades) { ❶
  return grades.reduce((acc, curr) => acc + curr, 0)/grades.length;
}
```

❶ Вычисление среднего помещено в отдельную функцию.

По мере каждого нового пересмотра решения все более очевидна неуместность логики вычисления среднего для набора оценок. Поэтому на текущей итерации мы создадим внутри нашей функции две функции: одна будет проверять попадание среднего в заданный диапазон, а вторая — попадание в этот диапазон всех оценок из набора. На этом этапе добавить в код более специализированные функции можно несколькими способами. Пока нет способа эффективно разделить логику для двух случаев, правильного или неправильного варианта это сделать не существует. Мой вариант показан в примере 1.6.

Листинг 1.6. Небольшой пример с лучше определенными функциями

```
function checkValid(
  minimum,
  maximum,
  grades,
  useAverage = false
){

  // Корректные оценки не могут быть меньше 0
  var absoluteMinimum = 0;
```

```
// Корректные оценки не могут быть больше 100
var absoluteMaximum = 100;

if (minimumBound < absoluteMinimum) return false;
if (maximumBound > absoluteMaximum) return false;

let min = Math.min(...grades);
let max = Math.max(...grades);

if (useAverage) {
    return checkAverageInBounds(minimumBound, maximumBound, grades); ❶
}

return checkAllGradesInBounds(minimumBound, maximumBound, grades); ❷
}

function calculateAverage(grades) {
    return grades.reduce((acc, curr) => acc + curr, 0)/grades.length;
}

function checkAverageInBounds(
    minimumBound,
    maximumBound,
    grades
){ ❶
    var avg = calculateAverage(grades);
    if (avg < minimumBound) return false;
    if (avg > maximumBound) return false;
    return true;
}

function checkAllGradesInBounds(
    minimumBound,
    maximumBound,
    grades
){ ❷
    var min = Math.min(...grades);
    var max = Math.max(...grades);

    if (min < minimumBound) return false;
    if (max > maximumBound) return false;
    return true;
}
```

- ❶ Добавлена функция, проверяющая, попадает ли средняя арифметическая оценка в заданный диапазон.
- ❷ Добавлена функция, проверяющая, попадает ли каждая оценка в заданный диапазон.

Ура! За шесть несложных шагов мы провели рефакторинг функции `checkValid`.

ЕЩЕ БОЛЕЕ ПРОРАБОТАННОЕ РЕШЕНИЕ

Но для действительно полного рефакторинга нашей функции нужно еще несколько изменений. Код с 7-й по 14-ю строку лучше поместить в отдельную функцию, которая в качестве первого шага будет вызываться функциями `checkAverageInBounds` и `AllGradesInBounds`. Кроме того, нужно было бы идентифицировать все места вызова в функции `checkValid` и в случае, когда переменная `useAverage` имеет значение `true`, заменить их вызовом функции `checkAverageInBounds`. В случае же, когда эта переменная имеет значение `false` или оно опущено, должна вызываться функция `checkAllGradesInBounds`. После этого исчезла бы необходимость изучать определение функции `checkValid`, чтобы понять назначение необязательного логического параметра. Не потребовалось бы и читать код, чтобы узнать, что подразумевается под «допустимым» набором оценок. Наконец-то функцию `checkValid` можно полностью удалить из кодовой базы.

У новой версии этой функции есть очевидные преимущества. Теперь сразу понятно, что делает код. В результате упрощения условных операторов он стал чуть более производительным и меньше подверженным ошибкам. По большому счету мы увеличили вероятность того, что следующий разработчик сможет без проблем расширить это решение. Это только маленький пример положительного влияния рефакторинга на приложение. Теперь представьте, каких результатов можно достичь при *масштабном* применении.

Но прежде чем приступить к работе, нужно правильно сориентироваться в ситуации. Мы должны основательно изучить историю кода перед его рефакторингом, чтобы понять, как он деградирует.

ГЛАВА 2

Как деградирует код

Пробежать марафон — впечатляющее достижение. Я на такие подвиги никогда не решалась, а мои друзья — да. Удивительно, что большинство из них до записи на полумарафон или на полную марафонскую дистанцию не были заядлыми бегунами. Но регулярный график тренировок всего за несколько месяцев помог им выработать необходимую выносливость.

Эти люди уже были в хорошей форме. Но если цель пробежать марафон поставит человек, большую часть времени проводящий на диване с пачкой чипсов, ему будет гораздо труднее. Придется нарабатывать выносливость сердечно-сосудистой системы с помощью регулярных тренировок и здорового питания (как бы ни хотелось сидеть в удобном кресле с большим куском пиццы).

Причиной неудачи могут стать даже небольшие изменения в привычном режиме. Не получилось выспаться или выдался внезапно жаркий день — усталость наступит быстрее, понижая способность пробежать заданную дистанцию. Даже пребывая в день забега в лучшей форме, нужно быть готовым к любым неожиданностям. Может пойти дождь, порваться шнурок, можно застрять в тесной толпе бегунов. Нужно не только уметь управлять подконтрольными факторами, но и учиться реагировать спонтанно.

Программист в чем-то похож на марафонца. Его работа тоже требует постоянного приложения усилий. Она тоже основана на прошлых успехах, коммит за коммитом, миля за милем. Поддержание здоровых привычек сильно влияет на возможность быстро вернуться в форму для марафонского забега или для разработки. Сохранение высокого уровня активного внимания к окружению и соответствующая корректировка — ключи к успешному и плавному пути к финишу.

В этой главе я расскажу, почему понимание процесса деградации кода играет центральную роль в успехе рефакторинга. Мы рассмотрим код в состоянии стагнации, код в активной разработке и опишем способы деградации кода в каждом случае на примерах из ранней и современной истории информатики. В конце поговорим о способах раннего обнаружения деградации и о ее предотвращении.

Почему важно понимать, что код деградирует

Признак деградации кода — уменьшение его воспринимаемой полезности. Это значит, что ранее хороший код больше не ведет себя так, как хотелось бы, или его стало сложнее читать и использовать для разработки. Поэтому такой код становится отличным кандидатом на рефакторинг. Но я твердо убеждена, что приступать к внесению улучшений нельзя, пока вы не начнете хорошо ориентироваться в истории этого кода.

Код пишется не на пустом месте. Тот, который сегодня могут счесть плохим, сразу после его написания, скорее всего, был хорошим. Потратив время на изучение обстоятельств его появления и причин его деградации, мы получим представление о главной проблеме и о подводных камнях. Это увеличит шансы вернуть этот код в хорошее состояние.

Можно выделить две причины деградации кода. Первая — это изменение требований к задаче кода или к его поведению. Вторая — когда фирма идет по пути наименьшего сопротивления, пытаясь за короткий период сделать как можно больше. Первую причину мы будем называть изменением требований, а вторую — техническим долгом.

Я считаю, лучше не предполагать сразу, что ухудшение кода связано с техническим долгом. Поэтому сначала рассмотрим первую причину. Каждому хоть раз попадался ужасный код, при виде которого возникало множество вопросов: как могли допустить его появление? Почему его до сих пор никто не исправил? Но немедленно приступив к его рефакторингу, мы рискуем ограничиться устраниением того, что нам не нравится, а не убрать истинные болевые точки. Важно прочувствовать этот код. Первым делом следует определить, что изменилось с момента его написания. Когда мы поймем, почему изначально он считался хорошим, мы увидим, каких подводных камней избегал автор исходного решения

и как он справлялся с набором ограничений, и используем эти сведения в рефакторинге.

К сожалению, бывают случаи, когда работать нужно в условиях крайне ограниченных ресурсов, и мы просто выжимаем из имеющихся возможностей максимум. Нехватка времени или денег на разработку лучшего решения порождает технический долг. Его первоначальное влияние на состояние кодовой базы может быть минимальным, но со временем оно сильно возрастает. Проще всего записать такой код в плохое решение, но я бы советовала переосмыслить ситуацию. Иногда самое бесполезное решение — то, которое быстрее всего выводит ваш продукт или функцию на рынок. Если это важно для выживания компании, технический долг можно считать вполне допустимым компромиссом.

В процессе чтения этой главы рекомендую искать примеры каждого из вариантов деградации в коде, с которым вы чаще всего работаете. Скорее всего, это будет получаться не всегда. Но сам процесс поиска признаков деградации может сформировать у вас новый взгляд на фрагменты приложения, с которыми вам было особенно неприятно работать.



Узнать, чем было обусловлено именно такое исходное решение, можно у авторов кода. Поэтому по возможности просто поговорите с ними. Зачастую они сразу смогут назвать причину деградации кода. Если вы услышите «мы не знали, что...» или «тогда мы думали...», то речь идет о деградации кода из-за изменений требований. А если авторы говорят «этот код изначально был плохим» или «мы просто пытались уложить-ся в срок», здесь имеет место стандартный случай технического долга.

Изменение требований

Написание каждого нового куска кода в идеале должно сопровождаться документированием четкого определения его назначения. Конечно, можно постараться и разработать гибкие и адаптивные системы. Но маловероятно, что мы сможем полностью предугадать будущее. Среда, где работает приложение, со временем непредсказуемо меняется, воздействуя и на код в активной разработке, и на нетронутый код. Давайте на примерах посмотрим, как предъявляемые к коду требования начинают превосходить его возможности.

Возможности масштабирования

Зачастую мы не можем заранее оценить направление и степень масштабирования разрабатываемого продукта. Получается список требований, иногда довольно длинный, с множеством разных параметров. Рассмотрим запрос к программному интерфейсу приложения (API) на создание в системе новой записи пользователя. При этом можно дать рекомендации относительно ожидаемой задержки запроса, числа обращений к базе данных в рамках запроса, общего количества разрешенных в секунду запросов на создание новых пользователей и т. п.

При запуске нового продукта сначала оценивается предполагаемое количество его пользователей. Создается решение, которое должно легко справиться с этим количеством (плюс-минус допустимая погрешность). Но в случае успеха продукта у него может оказаться больше пользователей, чем предполагалось. В итоге первоначальная реализация просто не справится с непредвиденной нагрузкой. Сам код остался без изменений, но фактически он деградировал из-за резкого изменения требований к числу пользователей.

Доступность

Каждое приложение должно стремиться к обеспечению максимальной доступности. Нужны цветовые схемы для пользователей со слабым зрением, изображения и значки лучше снабдить альтернативным текстом, а еще нужно обеспечить доступ к любым интерактивным элементам с клавиатуры. К сожалению, команды, спешащие выпустить новый продукт или функцию, часто пренебрегают доступностью в пользу ускорения работы. Добавление новых функций помогает сохранить текущих пользователей и привлечь новых. Но если для части предполагаемой аудитории некоторые функции будут недоступны, вы рискуете оттолкнуть ее, ведь воспринимаемая полезность продукта сильно уменьшится.

С 1999 года в рамках технологического стандарта WAI (Web Accessibility Initiative), разрабатываемого Консорциумом Всемирной паутины, выходило несколько версий методических рекомендаций по обеспечению доступности в интернете (<https://oreil.ly/r0376>). Но некоторые важные поправки были стандартизированы. Каждая новая версия заставляла разработчиков сайтов и приложений пересматривать код, который долгое

время не трогали, и вносить в него изменения в соответствии с новейшими стандартами. Появление новых стандартов доступности тоже может стать причиной снижения качества приложения.

Совместимость с устройствами

Ежегодно производители оборудования выпускают новые версии, а иногда даже представляют совершенно новый класс своих устройств. Множество смартфонов, умных часов, автомобилей и телевизоров постоянно подталкивает к изменению приложений, чтобы ими можно было пользоваться на этом оборудовании. Пользователи привыкли к тому, что их любимые приложения работают на разных платформах. Когда крупная компания выпускает новое устройство с более высоким разрешением экрана, разработчики популярных мобильных игр рисуют потерять значительную часть пользовательской базы, если быстро не появится новая сборка игры для экрана крупнее.

Изменение среды

Изменения программного окружения иногда приводят к неожиданным последствиям. До появления современных игровых компьютеров с мощными графическими процессорами (GPU) и десятками гигабайт оперативной памяти (RAM) использовались скромные игровые консоли сначала в корпусах игровых автоматов, а потом и для домашнего применения. Для создания таких классических игр, как *Space Invaders* и *Super Mario Bros*, разработчикам пришлось искать хитрые способы обхода ограничений, которые накладывало доступное им оборудование.

Стандартной практикой было использование центрального процессора как таймера тактовой частоты. Это был устойчивый и надежный измеритель времени. Но после перехода к ПК появилась проблема. По мере увеличения тактовой частоты увеличивалась скорость игрового процесса. Представьте, что вы складываете тетрис или пытаетесь избежать потока злых грибов Гумба на скорости, вдвое выше нормальной. В итоге наступает момент, когда играть становится физически невозможно. Код игры при этом не изменился. Просто он должен был выполняться на определенном оборудовании, и изменение аппаратной платформы стало причиной деградации кода.

Такие перемены до сих пор представляют серьезную проблему. В январе 2018 года исследователи безопасности из Google Project Zero, Cyberus

Technology и Грацкого технического университета опубликовали детали двух серьезных уязвимостей, затрагивавших все микропроцессоры x86 от Intel, процессоры POWER от IBM и некоторые микропроцессоры на базе архитектуры ARM. Первая уязвимость (Meltdown) давала несанкционированный доступ на чтение к привилегированной памяти. Вторая (Spectre) – доступ к содержимому виртуальной памяти текущего приложения или других программ, то есть нарушала изоляцию памяти между программами. Подробности можно узнать на официальном сайте (<https://meltdownattack.com/>).

Эти уязвимости на момент раскрытия затрагивали все устройства, работающие под любыми версиями iOS, Linux, macOS и Windows, кроме самых последних. Пострадал ряд серверов, облачных сервисов и большинство умных и встроенных устройств. За несколько дней появились программные варианты решения проблемы, но они вызывали снижение производительности на 5–30 % в зависимости от рабочей нагрузки. Позже Intel сообщила, что ищет способы защиты от уязвимостей Meltdown и Spectre для следующей линейки процессоров. Как видите, изменения затрагивают даже те вещи, которые считаются наиболее стабильными (операционные системы, микропрограммы). Это неминуемо отражается на функционировании разработанных нами приложений.

Внешние зависимости

У каждого ПО есть внешние зависимости, например библиотеки, интерпретаторы, пакеты программ и т. п. Степень связи с ними может быть разной. В этом подходе нет ничего нового. С момента начала исследований искусственного интеллекта много важных программ разрабатывалось на языке Lisp и егоialectах, которые активно развивались в 1960-х и начале 1970-х годов. Программа понимания естественного языка SHRDLU, написанная на языке Micro Planner для компьютера PDP-6, использовала нестандартные макросы и программные библиотеки, которых сегодня уже нет. Это стало причиной деградации этого программного обеспечения.

Современные разработчики прилагают все усилия для своевременного обновления внешних зависимостей, чтобы добавить в программы новейшие функции и устранить дыры в безопасности. Но иногда приоритетность обновлений снижается или про них вообще забывают. Особенно если речь идет о коде, активная поддержка которого была прекращена. В случае зависимостей вполне допустимо отставание на несколько версий, хотя это

и связано с риском увеличения вероятности появления уязвимостей. Кроме того, в будущем могут возникнуть сложности при обновлении ПО.

Допустим, наша программа зависит от версии 1.8 библиотеки с открытым исходным кодом Super Timezone. Через несколько недель после выхода версии 4.0 разработчики этой библиотеки объявляют о прекращении активной поддержки версий ниже 3.0. Чтобы продолжить получение исправлений уязвимостей, нужно обновить программу минимум до версии 3.0. Но, к сожалению, в версии 2.5 появились изменения без обратной совместимости. А функционал версии 2.8, широко используемый в нашем приложении, устарел. Теперь вместо небольших регулярных обновлений для поддержания актуального состояния библиотеки придется срочно решать более сложную задачу.

Неиспользуемый код

Иногда изменения требований порождают неиспользуемый код. Допустим, было принято решение отказаться от одного из открытых API. Сторонние разработчики предупреждаются об изменениях. Удостоверившись в отсутствии внешних систем, пользующихся этой конечной точкой, рабочая группа удаляет документацию с сайта. Но по каким-то причинам забывает удалить код. Через несколько месяцев новый инженер-разработчик натыкается на него и решает, что код до сих пор используется. Он хочет перепрофилировать его под свои нужды. Но быстро обнаруживается, что код не работает как положено, потому что его не адаптировали к остальной кодовой базе и многочисленным изменениям требований.

Неиспользуемый код может снизить производительность разработчика. Ведь при каждом столкновении с таким кодом нужно понять, можно ли его безопасно удалить. Без надежных инструментов для правильного определения объема мертвого кода сложно распознать его точные границы. Обычный разработчик, если не уверен в том, что код можно без проблем удалить, просто оставляет его, надеясь, что позже им займется кто-то другой. Кто знает, сколько существует этот код, пока не найдется смельчак, который его удалит!

Накапливающийся неиспользуемый код постепенно начинает плохо влиять на производительность. Например, при работе с клиентской частью веб-сайта время первой загрузки страницы обратно пропорционально размеру запрашиваемых файлов. Чем больше размер, тем медленнее отклик. Большое

количество запросов к приложению с раздутым кодом сильно ухудшает пользовательский опыт.

ЗАКОММЕНТИРОВАННЫЙ КОД

Очевидно, что код, превращенный в комментарии, не используется. При наличии системы контроля версий я всегда рекомендую разработчикам просто удалять его. Если когда-нибудь он вдруг понадобится, его легко восстановить, просмотрев историю коммитов.

Изменение требований к продукту

Часто решение под актуальные и будущие требования к продукту с учетом имеющихся ограничений написать проще, чем решение на следующий год. Потому что интуиции обычного человека недостаточно для прогнозирования ситуации, с которыми когда-нибудь придется столкнуться.

Отличный пример того, к чему приводят трудности прогнозирования будущих требований к продукту, — переменные-флаги. В основном они применяются для изменения поведения функции. Вы уже сталкивались с ними в первом примере рефакторинга. Такой флаг показывал, хотим ли мы знать, попадает ли каждая из оценок или их среднее в заданный диапазон. К логическим переменным часто прибегают, когда функция делает почти то, что нужно, за небольшим исключением. К сожалению, в будущем эти переменные могут стать причиной разных проблем. Рассмотрим вариант одной из них на примере небольшой функции для загрузки изображений в формате PNG (пример 2.1).

Пример 2.1. Функция с переменной-флагом

```
function uploadImage(filename, isPNG) {  
    // детали реализации  
    if (isPNG) {  
        // операции с изображением в формате PNG  
    }  
    // выполняем какие-то операции  
}
```

Как поступить, если через несколько месяцев понадобится добавить загрузку изображений в формате GIF? Наверное, добавить еще один логический аргумент, как показано в примере 2.2.

Пример 2.2. Функция с двумя переменными-флагами

```
function uploadImage(filename, isPNG, isGIF) { ❶
    // какие-то детали реализации
    if (isPNG) {
        // какие-то операции с изображением в формате PNG
    } else if (isGIF) { ❷
        // какие-то операции с изображением в формате GIF
    }
    // выполняем какие-то операции
}
```

- ❶ Новый флаг, допускающий загрузку только изображения в формате GIF.
- ❷ Изображение не может быть одновременно в двух форматах, поэтому добавлен оператор `else if`.

Чтобы функция загружала изображения GIF, нужно установить для второго логического аргумента значение `true`. Те, кто увидит код, вызывающий функцию `uploadImage`, будут сбиты с толку. Им нужно будет посмотреть определение функции, чтобы понять назначение двух логических аргументов.



В языке с именованными аргументами куда меньше нужды смотреть определение функции, чтобы понять роль и порядок аргументов. Вне зависимости от выбранного языка запись `uploadImage (filename = имя_файла, isPNG = true, isGIF = true)` может показаться бессмысленной, но это вполне допустимый вызов функции (который может привести к ошибкам). Пример 2.3 демонстрирует код, где непросто понять назначение функции `uploadImage` в заданном контексте.

Пример 2.3. Функция, загружающая изображение в формате GIF

```
function changeProfilePicture(filename) {
    // детали реализации
    if (isAnimated) {
        uploadImage(filename, false, true); ❶
    } else {
        uploadImage(filename, true, false); ❷
    }
    // выполняем какие-то операции
}
```

- ❶ Загружаем изображение в формате GIF.
- ❷ В противном случае загружаем изображение в формате PNG.

Разработчикам не стоит поддерживать такой шаблон только потому, что, читая функцию `changeProfilePicture`, сложно понять, как работает `uploadImage`.

Это полбеды. Но в будущем может появиться необходимость увеличить количество форматов. Разработчик, который изначально использовал переменную-флаг `isPNG`, решал только актуальную задачу и совсем не думал о будущем. Гораздо продуктивнее было бы разделить логику на отдельные функции: `uploadJPEG`, `uploadPNG` и `uploadGIF`, как в примере 2.4.

Пример 2.4. Отдельные функции для загрузки изображений в разных форматах

```
function uploadImagePreprocessing(filename) {
    // детали реализации
}

function uploadImagePostprocessing(filename) {
    // выполняем какие-то операции
}

function uploadJPEG(filename) {
    uploadImagePreprocessing();
    // что-то делаем с изображением JPG
    uploadImagePostprocessing();
}

function uploadPNG(filename) {
    uploadImagePreprocessing();
    // что-то делаем с изображением PNG
    uploadImagePostprocessing();
}

function uploadGIF(filename) {
    uploadImagePreprocessing();
    // что-то делаем с изображением GIF
    uploadImagePostprocessing();
}
```

Вы могли задаться вопросом, чем плоха переменная-флаг `isPNG`, если позже всегда можно реорганизовать код. Но для корректной замены всех вхождений функции `uploadImage` нужно отдельно проверить все места ее вызова, заменяя на `uploadJPEG` или `uploadPNG`, в зависимости от логического аргумента значения `true`. Эти изменения выполняются вручную, поэтому вероятность ошибки высока и может стать причиной серьезной регрессии кода. Конечно, в каждом случае все зависит от масштаба проблемы и от того, насколько тесно она будет связана с важной бизнес-логикой. Но рефакторинг кода с простым логическим аргументом может стать очень трудной задачей.

Технический долг

Технический долг чаще всего появляется из-за недостатка времени, рабочих рук и финансов. Все технологические компании рано или поздно сталкиваются с проблемой ограниченных ресурсов. Поэтому можно утверждать, что технический долг есть всегда. И у крошечных молодых стартапов, и у гигантских, существующих десятилетиями конгломератов можно найти изрядное количество неработоспособного кода. Сейчас мы разберемся в том, что вызывает накопление технического долга. Конечно, проще всего ткнуть пальцем в авторов такого кода, свалив на них всю вину за принятие неоптимальных решений. Но важно помнить, что, скорее всего, на них давили серьезные ограничения. А в таких условиях очень сложно написать хороший код.

Обход выбранной технологии

Перед внедрением новшества нужно решить, какие технологии будут использоваться для реализации задуманного. Придется выбрать язык, диспетчер зависимостей, базу данных и т. п. Прежде чем приложение станет доступно пользователям, приходится выбирать множество вещей. Часто этот выбор обусловливается опытом инженеров-разработчиков. Если для быстрого начала работы и запуска проекта удобнее использовать привычную технологию, предпочтение могут отдать ей.

После первых положительных сдвигов в проекте начинается проверка правильности выбранных технологических решений. Если проблема обнаруживается на раннем этапе жизненного цикла приложения, поиск альтернативы может быть быстрым и недорогим. Но часто связанные с технологией ограничения становятся очевидными гораздо позже. Одним из таких камней преткновения может стать выбор между языком программирования со статической типизацией и с динамической.

Сторонники последнего утверждают, что такие языки упрощают чтение и понимание кода благодаря меньшему числу косвенных обращений к строго определенным структурам и объявлениям типов. Многие отмечают и ускоренный цикл разработки, так как не нужно тратить время на компиляцию.

Но, несмотря на многочисленные плюсы вышеупомянутого языка, при росте приложения кодом в какой-то момент становится трудно управлять. Типы

проверяются только во время выполнения, поэтому за обеспечение их правильности ответствен разработчик. Именно ему приходится писать набор модульных тестов для проверки всех путей выполнения и подтверждения ожидаемого поведения. Если имена переменных не содержат указания на их тип, стороннему разработчику будет сложно понять принцип взаимодействия разных структур. Часто возникает необходимость в безопасном программировании (пример 2.5), где проверяется, что передаваемое в функцию значение не равно `null` и обладает определенными свойствами.

Пример 2.5. Безопасное программирование в действии

```
function addUserToGroup(group, user) {
    if (!user) {
        throw 'user не может иметь значение null';
    }

    // подтверждаем заполнение обязательных полей
    if (!user.name) {
        throw 'требуется указать имя';
    }

    if (!user.email) {
        throw 'требуется указать email';
    }

    if (!user.dateCreated) {
        throw 'требуется дата создания';
    }

    // подтверждаем отсутствие пустых строк и других нелегитимных значений
    if (user.name === "") {
        throw 'поле name не может быть пустым';
    }
    if (user.email === "") {
        throw 'поле email не может быть пустым';
    }
    if (user.dateCreated === 0) {
        throw 'дата создания не может быть 0';
    }

    group.push(user);
    return group;
}
```

Вероятно, автор этого кода регулярно сталкивается с тем, что из-за динамической природы JavaScript недействительные пользователи проходят через стек вызовов во время выполнения. Поэтому он пытается гарантировать

добавление в группу только допустимых пользователей. К сожалению, в итоге функция `addUserToGroup` сначала занимается проверкой легитимности. При появлении нового условия, которому должен соответствовать действительный пользователь, придется обновлять каждую из функций проверки, разбросанных по всей кодовой базе. Это увеличивает вероятность ошибки, достаточно лишь при обновлении пропустить одну из функций. В итоге мы остаемся с длинным, запутанным и предрасположенным к появлению ошибок кодом.

Можно добавить вспомогательную функцию `validateUser` из примера 2.6. Она поможет уменьшить деградацию кода. В ней мы инкапсулируем логическую схему проверки объекта `user`.

Пример 2.6. Вспомогательная функция, проверяющая легитимность пользователя

```
function validateUser(user) {
    if (!user) {
        throw 'user не может иметь значение null';
    }

    // подтверждаем заполнение обязательных полей
    if (!user.name) {
        throw 'требуется указать имя';
    }

    if (!user.email) {
        throw 'требуется указать email';
    }

    if (!user.dateCreated) {
        throw 'требуется дата создания';
    }

    // подтверждаем отсутствие пустых строк и других нелегитимных значений
    if (user.name === "") {
        throw 'поле name не может быть пустым';
    }
    if (user.email === "") {
        throw 'поле email не может быть пустым';
    }
    if (user.dateCreated === 0) {
        throw 'дата создания не может быть 0';
    }

    return;
}
```

Обновим функцию `addUserToGroup`, резко упростив ее логику, как в примере 2.7.

Пример 2.7. Функция `addUserToGroup` без встроенной логики проверки

```
function addUserToGroup(group, user) {  
    validateUser(user);  
    group.push(user);  
    return group;  
}
```

Теперь для проверки действительности пользователя достаточно вызвать функцию `validateUser`. Но заменить ею все включения кода проверки непросто. Во-первых, все эти включения нужно идентифицировать. Во-вторых, при ревизии этих фрагментов можно обнаружить, что кое-где нет одного-двух проверяемых условий. Если это следствие ошибки или небрежности, вместо этого кода можно смело вставлять функцию `validateUser`. Но не исключена ситуация, когда проверка каких-то условий опущена преднамеренно. В этом случае механическое замещение кода вспомогательной функцией может вызвать регрессию. Как видите, результат безопасного программирования требует тщательно спланированного рефакторинга.

Отсутствие привычки к систематизации

Сохранение структурированного состояния кодовой базы можно сравнить с поддержанием порядка в доме. Часто оказывается, что у нас есть куда более важные дела, чем разбор скапливающейся на журнальном столике почты и развешивание одежды по шкафам. Но чем дольше копится беспорядок, тем дольше его нужно будет устранять. Мои родители хорошо понимали, что справиться с мелким беспорядком намного легче, чем с большим. Они учили меня сразу класть вещи на свои места и каждый день делать небольшую уборку.

Проблемы с систематизацией кодовой базы часто возникают из-за нежелания людей менять привычный шаблон. Представим приложение, кодовая база которого состоит из десятков файлов плюс папка для тестов. Такая вот относительно плоская файловая структура. Приложение стабильно растет, и каждый месяц добавляется несколько новых файлов. Заниматься созданием папок и распределением по ним связанных файлов никому не хочется. Поэтому разработчики просто учатся ориентироваться в разрастающемся

коде. Новичок в команде обеспокоен растущим хаосом и предлагает заняться систематизацией кода. Но менеджеры напоминают о сроках сдачи проекта, а штатные сотрудники заверяют, что без проблем учатся ориентироваться в этом хаосе и на продуктивность он не влияет. Но в какой-то момент кодовая база становится огромной, и производительность рабочей группы все-таки снижается. Только после этого находится время составить план по приведению ее в порядок. Но теперь приходится учитывать массу переменных, которых не было бы, начни группа согласованно заниматься систематизацией кода раньше.

СЛИШКОМ МНОГО ПОВАРОВ НА КУХНЕ

Код деградирует еще быстрее, когда плохая систематизация сочетается с постоянным набором новых сотрудников. В растущие компании ежемесячно могут приходить десятки новых инженеров. Каждый из них хочет оперативно вникнуть в код и приступить к работе. Но без четко определенной структуры и стиля новички могут начать поддерживать существующие проблемные шаблоны, глубоко укоренившиеся в кодовой базе.

Когда с одной кодовой базой работает слишком много инженеров, условия труда будут оптимизироваться исходя не из долгосрочного здоровья кодовой базы, а из удобства работы в большом коллективе. Это приводит к появлению длинного безопасного кода или к его неоптимальному размещению во избежание возможных конфликтов слияния.

Слишком быстрое продвижение

При отсутствии должного контроля слишком быстрая разработка продукта стремительно ухудшает качество ПО. В попытке добавить новую функциональность в сжатые сроки начинается срезание углов: пропускаются некоторые тесты, переменным присваиваются обобщенные имена или вместо новой функции добавляются несколько операторов `if`. Если не записывать все эти компромиссы и после сдачи приложения в срок не выделить время на их исправление, они накапливаются. Вскоре появляются очень длинные функции и заваленные логическими схемами ветки, разбросанные по кодовой базе. Покрытия модульными тестами тоже почти нет. Если над разными функциями одновременно работает несколько команд, все эти вещи быстро усугубляются. Без эффективного информирования других команд количество мусорного кода начинает увеличиваться. Это видно на рис. 2.1.

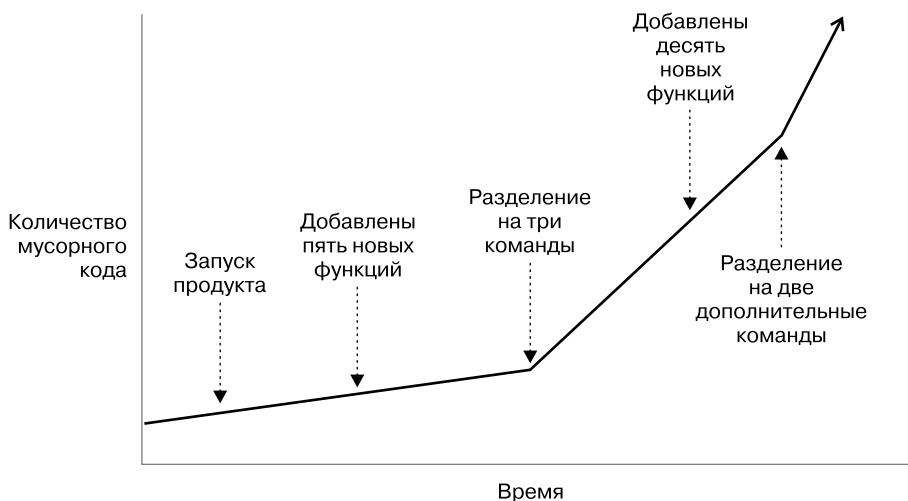


Рис. 2.1. Накопление мусорного кода с течением времени

Сейчас при создании приложений часто практикуется непрерывная интеграция и доставка. Все изменения как можно чаще сливаются в основную ветку. Там они проверяются автоматическими тестами для новой сборки приложения. Чтобы незавершенный функционал и частично исправленные ошибки не увидели пользователи, изменения блокируются с помощью переключателя функциональности (feature toggle). Он обеспечивает гибкость активной разработки, но после предоставления изменений пользователям о нем часто забывают.

В каждой компании, где я работала, по завершении производства оставались десятки (если не сотни) переключателей функциональности. Это может казаться безобидным, но создает определенные неудобства.

Во-первых, при чтении такого кода на разработчиков падает дополнительная когнитивная нагрузка. Если не тратить время на проверку статуса функции, важные изменения могут быть внесены в код, уже выведенный из активной разработки. Но если же этим заняться, можно обнаружить, что рассматриваемый функционал уже несколько недель доступен всем пользователям. Во-вторых, если переключателей функциональности больше сотни, это точно повлияет на производительность приложения. Только представьте, сколько времени уходит на проверку всех связанных с функционалом условных операторов при каждом запросе. Убрав устаревшие переключатели, можно улучшить производительность.

Применение знаний

Деградация кода неизбежна. Вне зависимости от принятых мер приложения должны приспосабливаться к меняющимся требованиям. Можно попытаться минимизировать разработку в сжатые сроки. Но все равно иногда нам придется идти на компромиссы для быстрого предоставления нового функционала, обеспечив себе конкурентное преимущество. Поэтому деградация кода действительно неизбежна, а значит, неизбежен и масштабируемый рефакторинг. Рано или поздно наступает момент, когда нужно решать сложные системные проблемы в кодовой базе. Если вам кажется, что деградация кода стала мешать развитию, значит, пришло время выяснить, почему и как вы оказались в этой ситуации.

Научившись не только искать проблемы кода, но и понимать, как они возникли, мы поймем, что он не так и плох. Это умение смотреть под другим углом позволяет выявить настоящие проблемы кода и разработать оптимальный план его улучшения.

Теперь мы знаем, как и почему деградирует код. Нужно научиться давать этому процессу количественную оценку. Свои ощущения того, что деградация дошла до критической точки, и понимание причины происходящего нужно использовать для описания проблемы через набор показателей. Они помогут нам убедить других в серьезности проблемы. Далее я познакомлю вас с методами измерения проблем в кодовой базе, которые станут прочной базой для обоснования необходимости рефакторинга.

ЧАСТЬ II

Планирование

ГЛАВА 3

Количественная характеристика начального состояния

Каждую весну я перебираю всю свою одежду. Некоторые делают это по методике Мари Кондо, пытаясь понять, «вызывает ли конкретная вещь радость». Я же предпочитаю более систематический подход. Каждый раз после сортировки вещей у меня накапливается стопка тех, которые я отдаю на благотворительность. Я каждый раз не знаю, что именно это будет, так как в первую очередь все зависит от сочетаемости предметов моего гардероба.

Вся одежда распределяется по категориям: свитеры, платья и т. п. При этом учитывается практичность каждой вещи. Я спрашиваю себя: для каких сезонов подходит это платье, насколько оно удобное, как часто я надевала его за последний год. Дальше я прикидываю, с какими вещами в моем гардеробе оно сочетается. И только после того, как у меня появится четкое представление о роли каждой вещи в моем шкафу, я начинаю их сортировать.

Эта логика применима и к рефакторингу. Только получив основательную характеристику поверхности для улучшения, можно начинать поиск подходящего способа решения этой задачи. К сожалению, найти значимые метрики для болевых точек внутри кода сложнее, чем распределить по категориям висящую в шкафу одежду. В этой главе мы поговорим о методах количественной оценки состояния кода. Я покажу несколько давно известных приемов и парочку новых, более творческих подходов. Надеюсь, что к концу главы вы найдете один (или больше) способ количественного описания проблем кода, который вы хотите улучшить.

Почему сложно оценить последствия рефакторинга

Оценить состояние кодовой базы можно по-разному. Но проведенный рефакторинг не всегда сдвигает эти метрики в положительном направлении просто потому, что они не связаны с подлежащими устраниению болевыми точками. Поэтому для измерения начального состояния кодовой базы важно выбрать метрику, дающую наглядное представление о проблеме и подчеркивающую воздействие рефакторинга.

Измерить последствия рефакторинга сложно. В первую очередь потому, что успешно выполненный рефакторинг обязан быть невидим для пользователей и никак не должен менять поведение кода. Это же не новый функционал, который повысит привлекательность приложения для пользователей. Для обеспечения надежности работы приложений в них часто добавляется мониторинг за критически важными частями. Но в этом случае отслеживаются показатели, фиксирующие заметное пользователям поведение. И корректно проведенный рефакторинг никак на них не влияет. Значит, нужно найти метрики, точно характеризующие те нюансы кода, которые мы хотим улучшить, и определить точку отсчета. Только после этого можно будет двигаться дальше.

РЕФАКТОРИНГ ДЛЯ УЛУЧШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

Представим небольшое приложение для отслеживания заказов. Чтобы обеспечить его бесперебойную работу, мы наблюдаем за таким показателем, как время получения информации о статусе заказа, по его идентификатору. Через несколько месяцев этот показатель увеличивается, и принимается решение провести рефакторинг. Здесь у нас уже есть начальная метрика — среднее время получения ответа на запрос. Если после внедрения новой версии кода оно уменьшится, то усилия были не напрасны!

Часто простейшая метрика для оценки эффективности рефакторинга — это производительность. В таком случае у нас сразу есть надежный набор исходных показателей. Еще стоит отметить, что усилия для повышения производительности кода, в отличие от тех, что вызваны желанием сделать эффективнее труд разработчиков, относятся к одному из немногих видов рефакторинга, дающему отчетливые улучшения, видимые пользователям.

Особенно трудно поддаются количественной оценке результаты крупного рефакторинга. Он редко ограничивается несколькими неделями. Чаще всего этот процесс выходит далеко за рамки типичного цикла разработки функций. И если на время рефакторинга разработка продукта не была полностью приостановлена, будет сложно определить, какие именно действия влияют на показатели. Разумнее всего пользоваться набором разных метрик для получения более целостной картины прогресса после рефакторинга и отделить эти изменения от параллельно вносимых разработкой.

Оценка сложности кода

Для многих рефакторинг — это средство облегчения поддержки приложений и создания нового функционала и, как следствие, повышения продуктивности труда разработчиков. На практике это часто означает упрощение сложных, запутанных фрагментов кода. А раз мы ставим цель *уменьшить* сложность кода, нужно найти эффективный способ ее измерения. Количественная оценка сложности кода станет отправной точкой для оценки будущего прогресса.

Измерение сложности упрощают две вещи. Во-первых, если есть история версий, можно легко путешествовать в прошлое и вычислять сложность в любом временном интервале. Во-вторых, на многих языках программирования существует огромное число библиотек и инструментов с открытым исходным кодом, позволяющих получить отчет для всего приложения.

Давайте рассмотрим три распространенных метода расчета сложности кода.

Метрики Холстеда

В 1975 году Морис Холстед впервые предложил измерять сложность программного обеспечения путем подсчета операторов и operandов в компьютерной программе. По его мнению, поскольку программы в основном состоят из этих двух компонентов, подсчет их уникальных экземпляров может дать реальное представление о размере программы и, следовательно, о ее сложности.

Операторы — это конструкции, которые ведут себя как функции, но синтаксически или семантически отличаются от них. Можно выделить ариф-

метические и логические операторы, операторы сравнения и присваивания. Рассмотрим простую функцию, складывающую два числа.

Пример 3.1. Короткая функция сложения

```
function add(x, y) {  
    return x+y;  
}
```

Она содержит единственный оператор `+` и два операнда. Операнды — это любые объекты, с которыми мы совершаем разные действия при помощи операторов. Операнды в нашем примере — переменные `x` и `y`.

Холстед предложил использовать информацию о количестве операторов и операндов для расчета следующих характеристик.

1. Объем программы или нужное количество информации для понимания ее назначения.
2. Сложность программы или умственные затраты программиста на создание кода.
3. Количество ошибок, которые, скорее всего, будут обнаружены в системе.

Для примера возьмем функцию, вычисляющую простые множители целых чисел. Уникальные операторы и операнды и количество раз, которое они встречаются в программе, перечислены в табл. 3.1.

Пример 3.2. Факторизация целых чисел

```
function primeFactors(number) {  
    function isPrime(number) {  
        for (let i = 2; i <= Math.sqrt(number); i++) {  
            if (number % i === 0) return false;  
        }  
        return true;  
    }  
  
    const result = [];  
    for (let i = 2; i <= number; i++) {  
        while (isPrime(i) && number % i === 0) {  
            if (!result.includes(i)) result.push(i);  
            number /= i;  
        }  
    }  
    return result;  
}
```

Таблица 3.1. Уникальные операторы, операнды и количество их появлений в программе

| Оператор | Количество появлений | Операнд | Количество появлений |
|------------------------|----------------------|-----------------------|----------------------|
| function | 2 | 0 | 2 |
| for | 2 | 2 | 2 |
| let | 2 | primeFactors | 1 |
| = | 3 | number | 7 |
| <= | 2 | isPrime | 2 |
| () | 4 | i | 12 |
| . | 3 | Math | 1 |
| ++ (постфикс) | 2 | sqrt | 1 |
| if | 2 | FALSE | 1 |
| ==== | 2 | TRUE | 1 |
| % | 2 | result | 4 |
| return | 3 | <anonymous> | 1 |
| const | 1 | includes | 1 |
| [] | 1 | push | 1 |
| while | 1 | | |
| && | 1 | | |
| ! (префикс) | 1 | | |
| /= | 1 | | |
| Уникальных операторов: | Всего операторов: 35 | Уникальных operandов: | Всего operandов: 37 |
| 18 | | 14 | |

Приведенный код содержит 18 уникальных операторов (n_1), 14 уникальных operandов (n_2) и всего operandов 37 (N_2). Вот предложенная Холстедом формула вычисления относительной сложности чтения программы:

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$$

Подстановка значений даст следующий результат:

$$D = \frac{18}{2} \cdot \frac{37}{14}$$

$$D = 23,78$$

Сама по себе эта цифра особого значения не имеет. Но работая с отдельными фрагментами кода, мы постепенно сможем понять, как эта оценка соотносится с нашим опытом. Со временем, сопоставляя вычисляемые значения с реализациями кода, мы научимся интерпретировать их в контексте приложения.



Эта метрика, как и две другие, о которых я расскажу ниже, может применяться в разных масштабах — для оценки сложности отдельной функции или целого модуля. Метрика сложности Холстеда рассчитывается даже для всего файла, например, как сумма сложностей всех входящих в него функций.

Цикломатическая сложность

Предложенная в 1976 году Томасом Маккейбом цикломатическая сложность — количество линейно независимых маршрутов через программный код. Это подсчет операторов потока управления в программе. Сюда входят операторы `if`, циклы `while` и `for` и блоки `case` в конструкциях `switch`.

Для начала возьмем простую программу без управляющих конструкций (пример 3.3). Для вычисления цикломатической сложности присвоим 1 объявлению функции. Этот параметр должен увеличиваться с каждой встречной точкой принятия решения. В нашем примере через функцию есть только один путь. Соответственно, ее цикломатическая сложность равна 1.

Пример 3.3. Функция пересчета температуры

```
function convertToFahrenheit(celsius) {  
    return celsius * (9/5) + 32;  
}
```

Для более сложного примера возьмем уже знакомую функцию разложения на простые множители. В примере 3.4 мы нумеруем каждую точку потока управления, получив в итоге цикломатическую сложность 6.

Пример 3.4. Факторизация целых чисел

```
function primeFactors(number) { ❶  
    function isPrime(number) {  
        for (let i = 2; i <= Math.sqrt(number); i++) { ❷  
            if (number % i === 0) return false; ❸  
        }  
    }
```

```
    return true;
}

const result = [];
for (let i = 2; i <= number; i++) { ④
  while (isPrime(i) && number % i === 0) { ⑤
    if (!result.includes(i)) result.push(i); ⑥
    number /= i;
  }
}
return result;
}
```

- ❶ Первая управляющая точка — объявление функции.
- ❷ Вторая — первый цикл `for`.
- ❸ Третья — первый оператор `if`.
- ❹ Четвертая — второй цикл `for`.
- ❺ Пятая — цикл `while`.
- ❻ Шестая — второй оператор `if`.

Каждый раз, когда при чтении кода появляется ветвление (оператор `if`, цикл `for` и т. п.), возможны несколько путей выполнения. Для понимания того, что делает код, нужно уметь удерживать в голове больше информации. Цикломатическая сложность 6 позволяет сделать вывод, что функция `primeFactors`, вероятно, не так уж сложна для чтения и понимания.

Подсчет числа точек принятия решения — это упрощение предложенного Маккейбом метода расчета сложности программы. Математически мы можем вычислить цикломатическую сложность структурированной программы, создав ориентированный граф, описывающий ее поток управления. Каждый узел — это базовый блок (прямолинейная кодовая последовательность без ветвей), связывающие узлы ребра показывают способ перехода от одного блока к другому. Сложность M определяется по формуле:

$$M = E - N + 2P,$$

где E — количество ребер, N — количество узлов, а P — количество компонентов связности. Компонентом связности называется подграф, в котором все узлы достижимы друг для друга.

На рис. 3.1 показан поток управления для функции primeFactors.

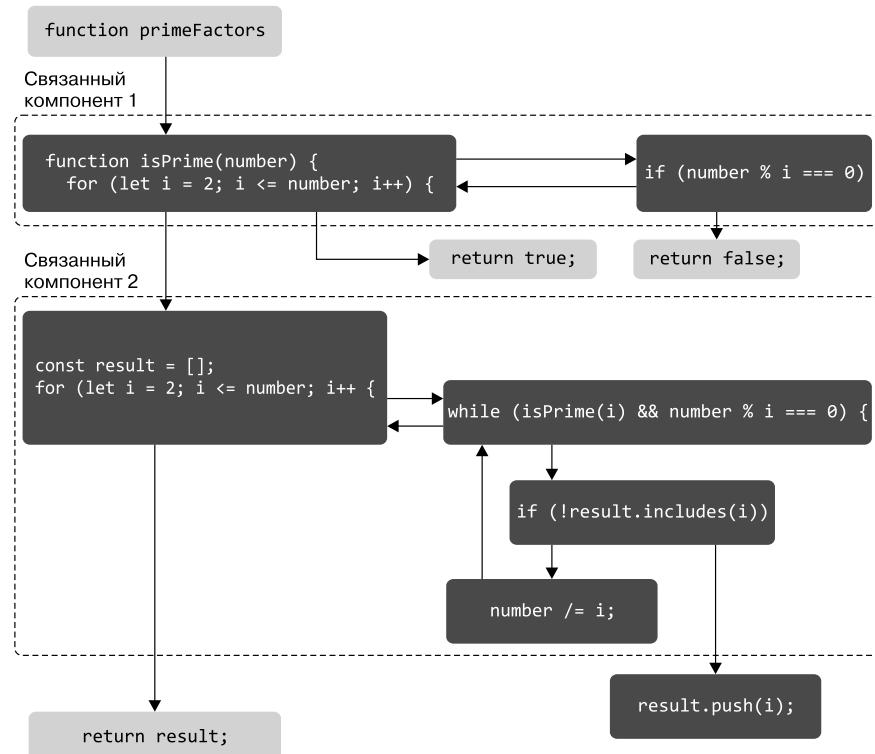


Рис. 3.1. Граф потока управления функции primeFactors, на котором синие узлы обозначают нетерминальные, а красные — терминальные состояния. Здесь 13 ребер, 11 узлов и 2 компонента связности

ДРУГИЕ ВАРИАНТЫ ПРИМЕНЕНИЯ ГРАФОВ ПОТОКА УПРАВЛЕНИЯ

Графы потока управления (CFG) помогают не только в расчете сложности программного обеспечения. Столкнувшись с особо запутанными потоками управления, я часто рисовала эти графы вручную, чтобы выделить точки принятия решений. Конечно, есть инструменты автоматической генерации CFG. Но чтобы начертить их самостоятельно, нужно внимательно читать код, дающий представление о потоке управления.

Еще CFG позволяют эффективно определить недоступный код. Наличие подграфа, не связанного ни с одной точкой входа, позволяет предположить, что соответствующий код недоступен, и его можно удалить. С другой стороны, недоступность блока выхода из точки входа может указывать на бесконечный цикл.

NPath-сложность

В 1988 году Брайан Неджми предложил NPath-сложность — альтернативу существующим метрикам. По его утверждению, фокусировка внимания на ациклических путях выполнения не позволяет адекватно смоделировать соотношение между конечными подмножествами и множеством всех *возможных* путей.

Дело в том, что цикломатическая сложность не учитывает вложенность структур потока управления. Метрика функции с тремя последовательными циклами `for` и функции с тремя вложенными циклами `for` будет иметь одинаковое значение. Вложенность влияет на сложность восприятия функции, а значит, и на способность разработчика поддерживать качество ПО.

Метрика Маккейба легка в вычислении, но не позволяет различать типы структур потока управления, трактуя операторы `if` и циклы `while` и `for` одинаково. Неймех утверждает, что разница структур — в сложности их понимания. Например, цикл `while` оказывается для разработчика сложнее конструкции `switch`. Этот фактор и призвана учесть NPath-сложность.

К сожалению, такой подход затрудняет вычисление метрики даже для небольших программ. Это рекурсивный процесс, который может быстро увеличиться в размерах. Для лучшего понимания того, как это все работает, рассмотрим несколько примеров с операторами `if`. Тем, кто хочет узнать, как вычисляется NPath-сложность для большего диапазона операторов потока управления (включая вложенные потоки), рекомендую прочесть статьи Брайана Неджми.



Метрики потока управления позволяют определить количество необходимых тестов. Цикломатическая сложность показывает нижнюю границу, а NPath-сложность — верхнюю. Например, как мы посчитали выше, функция `primeFactors` требует минимум шести тестов для проверки каждой из точек принятия решения.

В случае программы без точек принятия решения, как для предыдущей функции преобразователя температуры в примере 3.3, NPath-сложность совпадает с цикломатической сложностью и равна 1. Для иллюстрации мультиплексивного компонента метрики рассмотрим функцию с несколькими операторами `if`.

Это короткая функция, возвращающая вероятность штрафа за превышение скорости. В качестве входных данных мы указываем нашу скорость (`currentSpeed`) и ограничение скорости (`limit`). Первый оператор `if` пред-

лагает нам два возможных пути. Если скорость превышает 45 км/ч, начинает исполняться код внутри блока `if`. В противном случае мы двигаемся дальше. Затем нужно проверить, превышает ли скорость заданное ограничение больше чем на 10 км/ч. Здесь у нас снова появляются *два* возможных варианта прохождения кода. В конце функция возвращает рассчитанный коэффициент риска.

Пример 3.5. Функция с двумя последовательными операторами `if`, разбитая на блоки A, B, C, D, E и F

```
function likelihoodOfSpeedingTicket(currentSpeed, limit){
    risk = 0; // A

    if (currentSpeed < 45) {
        risk = 1; // B
    } // C

    if (currentSpeed > (limit + 10)) {
        risk = 2; // D
    } // E

    return risk; // F
}
```

Метрика NPath-сложность показывает количество разных путей через функцию. Для подсчета можно вызывать функцию `likelihoodOfSpeedingTicket` с различными входными данными и смотреть, как выполняются наборы условий. Я рассмотрю одно прохождение, а остальные варианты вы найдете в табл. 3.2.

Таблица 3.2. Все уникальные пути через функцию `likelihoodOfSpeedingTicket`

| Входные данные | Пути |
|-------------------|------------|
| 30, 10 | A, B, D, F |
| 30, 50 | A, B, E, F |
| 90, 50 | A, C, D, F |
| 90, 110 | A, C, E, F |
| Уникальных путей: | 4 |

Передадим в функцию `likelihoodOfSpeedingTicket` параметры `currentSpeed=30` и `limit=0`. Условие в первом операторе `if` при таких параметрах получит значение `true`, и мы зайдем в ветку B. Второй оператор `if` тоже оценит условие как `true`, что приведет нас в ветку D. Затем будет достигнут оператор `return`

в ветке F. Повторяя этот шаблон для разных комбинаций входных данных, мы увидим, что есть четыре уникальных пути через функцию. Следовательно, метрика NPath равна 4.

НЕДОСТАТКИ МЕТРИКИ NPATH

Метрика NPath всегда дает завышенную оценку количества путей выполнения. Как изменится NPath функции `likelihoodOfSpeedingTicket`, если добавить еще один оператор `if`, проверяющий, превышает ли `currentSpeed` значение 135 км/ч? Мы получаем три оператора `if`, каждый с двумя возможными исходами. Это $2 \times 2 \times 2$, или 8 путей через функцию. Но параметр `currentSpeed` не может одновременно иметь значение ниже 45 и выше 135 км/ч. То есть один из путей просто невозможен. Как видите, фактическая сложность оказалась меньше рассчитанной. Важно помнить, что NPath-сложность дает нам лишь оценку верхней границы.

Метрика NPath больше подходит для иллюстрации поведения отдельных типов управляющих операторов и оценки психологической нагрузки, связанной с вложенными точками принятия решений. Для больших, давно существующих баз кода она может дать *огромные* значения (сотни тысяч). Это обуславливается ее экспоненциальным характером. При огромных значениях метрика теряет свой смысл — ее становится сложно использовать для отслеживания небольших улучшений. Я советую применять NPath-сложность для работы с ограниченными фрагментами кода, взяв за отправную точку ее среднее значение для отдельных разделов.



Некоторые простые формы рефакторинга никак не меняют показатели CFG. Иногда сложность неизбежна из-за лежащей в его основе запутанной логической схемы. Убедитесь, что приложение делает то, что должно. Если подлежащий рефакторингу код допускает упрощение слишком сложной логики, для оценки подойдут NPath или цикломатическая сложность. В противном случае лучше использовать другой набор показателей. Более того, даже если вы распутываете спагетти-код, не опирайтесь только на значения NPath или цикломатическую сложность. Корректно и целостно охарактеризовать результативность рефакторинга с помощью одного типа показателей невозможно.

Строки кода

К сожалению, для очень больших кодовых баз (которые обычно и нуждаются в рефакторинге) рассчитать цикломатическую сложность непросто. Здесь в игру вступает такой показатель, как количество строк кода. Он менее научно обоснован, чем алгоритмы Холстеда, цикломатическая сложность или NPath. Но в сочетании с другими показателями размер программы по-

может в определении вероятных болевых точек приложения. Когда нужен pragматичный и простой подход к количественной оценке сложности кода, лучше всего подойдут размерно-ориентированные метрики.

Измерить длину кода можно несколькими способами. Разработчики предпочтуют подсчитывать логические строки кода, полностью игнорируя пустые строки и комментарии. Как и с метриками потока управления, проводить оценку можно с разным разрешением. Опереться можно на следующие параметры.

- *Количество строк кода (lines of code, LOC) в файле.* В каждой кодовой базе есть будто бы бесконечные файлы. Можно сказать, что подсчет строк в них дает представление о психологической нагрузке на разработчика, попытавшегося разобраться в структуре и назначении кода в редакторе.
- *Длина функции.* В каждом бесконечном файле будет хоть одна бесконечная функция (чаще всего они обнаруживаются именно в бесконечных файлах). Измерение длины функций или методов в приложении позволяет оценить их индивидуальную сложность.
- *Средняя длина функции внутри файла, модуля или класса.* Еще можно воспользоваться таким показателем, как средняя длина функции/метода на логическую единицу. Это средняя длина каждого метода внутри класса или пакета в объектно-ориентированных базах кода. В коде на императивных языках измеряют среднюю длину каждой функции в файле или более крупном модуле. Вне зависимости от использованных организационных единиц знание средней длины входящих в нее более мелких логических компонентов может дать представление об относительной сложности этой единицы как целого.

ПОДСЧЕТ СТРОК КОММЕНТАРИЕВ

Игнорирование комментариев при подсчете строк кода — хорошая практика. Блоки документации и встроенные пометки TODO не влияют на поведение программ. Поэтому добавление их в метрику не поможет лучше охарактеризовать сложность программы. Но практика показала, что подсчет числа комментариев на уровне функций позволяет легко выделять сложные фрагменты кода. Обычно разработчики добавляют комментарии, когда понять логику кода затруднительно. Неважно, обусловлена ли запутанная логическая схема сложным исходным алгоритмом или она такой стала со временем. У разработчиков есть привычка оставлять поясняющие комментарии для тех, кто придет после них. Поэтому число комментариев внутри функции может служить предупреждающим знаком.

Параметр LOC может сильно зависеть от языка программы или стиля программирования. Но если сравнение происходит в рамках одной программы, беспокоиться не о чем. Масштабированный рефакторинг обычно призван улучшить код в рамках одной *большой* кодовой базы. Опыт показывает, что большинство разработчиков старается писать руководства по стилю оформления кода и по набору лучших практик, обеспечивая выполнение этих рекомендаций автоматическими средствами форматирования. Конечно, если над приложением работает несколько команд, неизбежно будут появляться различия. Но в целом приложения обычно выглядят достаточно однородно, чтобы можно было сопоставлять друг с другом два набора метрик LOC из разных разделов базы.

Метрики покрытия кода

При разработке нового функционала есть разные подходы к тестированию. Например, при разработке через тестирование (test-driven development, TDD) сначала пишется полный набор тестов, а после реализации функционала повторяется, пока тесты не будут пройдены. Можно, наоборот, сначала написать решение, а затем соответствующие тесты. Допустимо чередовать эти два метода. В каждом случае мы хотим получить один результат: новый функционал, полностью подкрепленный набором высококачественных тестов.

Рефакторинг — совсем другое дело. Какие бы усилия ни прилагались к улучшению существующей реализации, нужна гарантия неизменности ее поведения. Уверенно утверждать, что новое решение работает как старое, можно только тогда, когда оно успешно проходит весь набор тестов исходной реализации. Для предупреждения возможных регрессий перед началом рефакторинга важно проверить две вещи. Убедиться, во-первых, в том, что исходная реализация имеет тестовое покрытие, во-вторых, в полноте и адекватности этого покрытия.

Допустим, мы решили реорганизовать функцию `primeFactors` из примера 3.2. Прежде чем рассматривать возможность внесения изменений, нужно определить, есть ли у этой функции тестовое покрытие и достаточное ли оно. Первое проверить несложно. Откроем соответствующий тестовый файл и посмотрим, что в нем. В нашем случае нашелся всего один тест (пример 3.6).

Пример 3.6. Простой тест для функции primeFactors

```
describe('базовые случаи', () => {
  test('0', () => {
    expect(primeFactors(0)).toStrictEqual([]);
  });
});
```

Куда сложнее определить адекватность тестового покрытия. Ее можно оценить количественно и качественно. В первом случае вычисляется процент кода, который выполняется при запуске набора тестов. Еще можно посчитать, сколько функциональных строк кода и путей выполнения проверяет наш простой модульный тест. Мы получим 40 и 35,71 % соответственно. На примере 3.7 мы видим результаты, полученные с помощью фреймворка модульного тестирования Jest.

Пример 3.7. Полученный в фреймворке Jest результат проверки тестового покрытия функции primeFactors на основе одного теста

| File | %Stmts | %Branch | %Funcs | %Lines | Uncovered Line #s |
|-----------------|--------|---------|--------|--------|-------------------|
| All files | 35.71 | 0 | 50 | 40 | |
| primeFactors.js | 35.71 | 0 | 50 | 40 | 3-6, 11-13 |

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total

Теперь нужно понять, адекватно ли это тестовое покрытие. Ни одна из метрик не позволяет считать функцию `primeFactors` хорошо протестированной. Цифры показывают, что текущий набор тестов не выполняет более трех четвертей функции. Тестовое покрытие используется преимущественно двумя способами:

- для определения не прошедших тестирование путей через программу;
- приблизительно показывает, достаточно ли оказалось тестирование.



Тем, кого интересуют стратегии тестирования унаследованного ПО, рекомендую книгу *Working Effectively with Legacy Code* Майкла Физерса (Michael Feathers). Он обсуждает разные варианты ввода модульных тестов задним числом, используя для этого швы кода — стратегические места, где можно менять поведение программы, не внося правки в код.

Для улучшения покрытия добавим еще один тест (пример 3.8). Новые метрики, которые даст фреймворк модульного тестирования Jest (пример 3.9), покажут, что, добавив лишь один дополнительный тест, мы получили почти идеальное покрытие. Можно ли считать его адекватным? Количественно оно кажется вполне достаточным, а вот качественно не совсем. Если еще раз посмотреть на реализацию функции `primeFactors`, легко заметить несколько отсутствующих вариантов теста. Например, для отрицательных чисел или для числа 2.

Пример 3.8. Простой тест для функции `primeFactors`

```
describe('базовые случаи', () => {
  test('0', () => {
    expect(primeFactors(0)).toStrictEqual([]);
  });
});

describe('небольшие составные числа', () => {
  test('20', () => {
    expect(primeFactors(20)).toStrictEqual([2, 5]);
  });
});
```

Пример 3.9. Полученный в фреймворке Jest результат проверки тестового покрытия функции `primeFactors` на основе двух тестов

| File | % Stmt | % Branch | % Funcs | % Lines | Uncovered Line #s |
|-----------------|--------|----------|---------|---------|-------------------|
| All files | 100 | 83.33 | 100 | 100 | |
| primeFactors.js | 100 | 83.33 | 100 | 100 | 12 |

Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total

По моему опыту, тестовое покрытие тщательно написанного кода обычно составляет 80–90 %. Это значит, что большая часть кода будет проверена. Но это не связано с качеством тестирования. Идеальное или почти идеальное тестовое покрытие легко достигается с помощью низкокачественных модульных тестов. Если руководство поощряет высокое тестовое покрытие, можно обнаружить, что большинство модульных тестов не прилагает особых усилий для подтверждения важных поведений кода.

С определением качества тестового покрытия все не так просто. На эту тему написано много работ, так что я не буду подробно на этом останавливаться.

ся. Скажу только, что, с моей точки зрения, качество тестирования можно считать приемлемым при выполнении следующих условий.

- Тесты *надежны*. При любом запуске они стабильно дают удовлетворительные результаты, если код не подвергался изменениям, и выявляют ошибки, если код в процессе разработки.
- Тесты *отказоустойчивы*. Они не так тесно связаны с реализацией, чтобы мешать изменениям.
- Для проверки кода используются *разные типы* тестов. Модульные, интеграционные и сквозные тесты помогут убедиться в правильности работы кода.

Если вы убедились в достаточном объеме и качестве тестового покрытия, будьте уверены — процесс рефакторинга будет двигаться вперед. Иначе лучше начать с написания дополнительных и более качественных тестов. Измерение размеров и качества тестового покрытия каждого из разделов кода, который мы собираемся подвергнуть рефакторингу, — важный шаг. Он помогает определить, сколько дополнительной работы понадобится выполнить предварительно.

ПОКРЫТИЕ ТИПОВ

В главе 2 мы кратко коснулись плюсов и минусов языков программирования с динамической типизацией. Разработчикам, имеющим дело с большими базами кода на таком языке, лучше подумать о постепенном внедрении типизированного языка. Это позволит получить возможность явного статического назначения типов. Статические типы позволяют обнаруживать ошибки на ранних этапах разработки — мы получаем предупреждения об их несовпадении. Автоматическое отслеживание информации, которую иначе пришлось бы запоминать, снижает нагрузку на программиста. Примеры языков с последовательной типизацией: TypeScript для JavaScript, Cython или тьюру для Python, Hacklang и PHP (начиная с версии 7.0).

Если вы решили добавить в кодовую базу типы, скорее всего, вам захочется измерить прогресс этой операции, отследив их покрытие. Эта метрика рассчитывается как процент кода с информацией о типе. Определение расплывчатое, но это сделано намеренно. Все зависит от того, как в конкретном языке реализована статическая типизация. Эта метрика может зависеть даже от версии языка. Как и в случае с тестовым покрытием, низкие значения покрытия типов можно применять для локализации кода, которому нужно повышенное внимание. Достичь 100%-ного покрытия типов невозможно. Но по своему опыту работы с последовательно типизированной кодовой базой могу сказать, что увереннее всего я себя чувствую при работе с максимально типизированным кодом. Если ваше приложение сможет набрать более 95 % баллов, значит, у вас все в порядке.

Документация

Перед началом рефакторинга проанализируйте всю документацию к нему. Это позволяет получить ценный дополнительный контекст. Конечно, документация не может дать числовые показатели для характеристики исходного состояния кода, но это важный источник информации о существующих проблемах. Им можно воспользоваться, чтобы доказать необходимость рефакторинга. Чтобы понять и количественно оценить отправную точку, желательно познакомиться как с официальной, так и с неофициальной документацией.

Официальная документация

Обычно под документацией подразумевают официальную. И речь идет даже не о соответствии каким-то стандартам (таким, как язык UML). Скорее это любая информация, специально написанная (и часто активно поддерживающаяся), чтобы дать представление о системе. Сюда относятся технические характеристики, схемы архитектуры, руководства по стилю оформления, инструкции для новых сотрудников и отчеты, которые пишутся после инцидентов (так называемые *postmortem*).

Технические спецификации можно использовать как доказательство необходимости или пользы рефакторинга, ссылаясь на проектные решения, предположения и другие рассмотренные или отклоненные проекты. Допустим, ведется работа над подразделом приложения, отвечающим за обработку всех действий пользователя. Разработчики пишут новый функционал, который должен запоминать и перечислять все виды событий, запускаемых и распространяемых на дочерние подсистемы при редактировании пользователем своего профиля. Если рабочая группа ранее писала технические требования к проекту для каждого варианта функционала, значит, можно найти исходную спецификацию для распространения событий. Этот документ описывает текущую реализацию, ее ограничения и любые другие подходы.

В разделе ограничений указано, что каждое событие удобно запускать индивидуально в каждом месте. Но при появлении большого числа новых событий такой подход становится неуклюжим и обременительным. Рассматриваемая система страдает именно от этой проблемы. Она обрабатывает более десятка типов событий, и разработчики стараются отслеживать их бесконтрольный рост. При появлении очередного нового функционала команда боится, что забудет запустить какой-нибудь важный тип события.

Это провоцирует досадную ошибку. Вы сделали все возможное, чтобы подтвердить желаемое поведение с помощью тестов, но посчитали, что рефакторинг обработки этих событий — лучшее решение для укрощения хаоса повторяющейся логики.

Технические характеристики могут подтвердить предположения о том, что и как нужно улучшить. Иногда в этой документации описываются другие подходы, которые когда-то рассматривались, но были отброшены. Возможно, с ними придется поработать в процессе рефакторинга.

Те, кто занимается сопровождением *руководства по стилю оформления и инструкций для новых сотрудников*, иногда делятся в этой документации своим опытом. Например, туда вносится информация о неожиданных открытиях в функционале или жирным шрифтом выделяются предупреждения о том, чего лучше избегать. Часто попадаются многословные заметки для особенно сложных фрагментов кодовой базы. Люди искренне хотят помочь выбрать верное направление и уберечь от подводных камней, с которыми они сталкивались. Если код, который нужно реорганизовать, подробно документирован и следует описанным шаблонам, перспективы его рефакторинга будут хорошими.

Отличные доказательства необходимости рефакторинга можно найти в *postmortem-отчетах*. Если рабочая группа следует процедуре реагирования на инциденты PagerDuty (<https://oreil.ly/T966e>), то у вас будет доступ к десяткам отчетов с подробным описанием, как, где, когда и почему приложение работало не так, как ожидалось.

При поиске аргументов в поддержку рефакторинга я ищу отчеты и обобщающие инциденты, напрямую связанные с этим кодом. Особое внимание я уделяю разделам Contributing Factors (Факторы, способствовавшие случившемуся) и What Didn't Go So Well? (Что пошло не так?). При подозрении на то, что сложность кода напрямую повлияла на время разрешения проблемы или даже стала ее причиной, подтверждение нужно искать в первую очередь в этих двух разделах. Ценным показателем будет и число инцидентов, где фигурировала подлежащая реорганизации область.



Не забывайте про стороннюю и общедоступную документацию. Хотя рефакторинг не подразумевает заметного пользователям изменения поведения приложения, документация поможет лучше понять код, который вы собираетесь переписать.

Неофициальная документация

Наряду с официальной документацией все разработчики генерируют множество неофициальной. Это могут быть разные записи, которые мы не причисляем к документации просто потому, что они не оформляются как требуется. Я в неофициальных источниках находила куда больше полезных сведений, чем в любых официальных бумагах.

Для поиска таких источников нужен творческий подход. Я сейчас дам несколько рекомендаций, но главное — внимательно смотреть вокруг. Иногда информацию можно найти в самом неожиданном месте!

Познавательную информацию о коде, подлежащем рефакторингу, можно получить из переписок в *чате* и по *электронной почте*. Лучшее в этой информации то, что часто она идет вместе с историческим и организационным контекстом. Скажем, вы хотите реорганизовать структуру асинхронных заданий в приложении. Для обеспечения максимальной гибкости очередь заданий сейчас принимает динамический набор аргументов произвольного размера. К сожалению, при таком подходе можно столкнуться с нехваткой памяти при обработке заданий с огромным числом аргументов или с внезапным сбоем, если система не сможет проанализировать некорректные входные данные.

Но нужно убедиться, что замеченные вами проблемы — системные. Для этого в Slack (или в другом рабочем мессенджере) проводится поиск по ключевым словам, связанным с аргументами очереди заданий. Скорее всего, будут обнаружены сообщения, где кто-то был удивлен или обеспокоен тем, что созданные им задания работают неверно. Разработчики интересуются, какие идентификаторы использовать, обычные или непрозрачные? Чем одни лучше других? Регистрируются ли аргументы заданий? Если да, то нужно ли проявлять осторожность при добавлении персональных данных? Сколько данных можно переслать с помощью этих аргументов? Можно ли сериализовать объекты целиком и отправлять их в очередь заданий?

Нужно создать документ со всеми скопированными сообщениями и кратким описанием контекста каждого из них (это легко сделать, прокрутив беседу в обратном направлении). На этот документ можно ссылаться при разговоре о трудностях, с которыми сталкиваются разработчики.

Из истории чата можно узнать, что обсуждалось в компании задолго до текущего момента. Иногда можно с удивлением обнаружить, что проблемы,

которые должен устраниТЬ рефакторинг, обсуждаются сотрудниками из разных команд уже месяцы или годы. Может оказаться, что одни и те же вопросы по поводу кода возникали регулярно. Это не только подтверждает необходимость рефакторинга, но и позволяет заполучить ценных союзников. Достаточно расспросить людей об опыте работы с кодом, который вы хотите улучшить. Эти беседы можно использовать, чтобы оценить, сколько инженерных часов потеряно из-за путаницы в коде.

Инструменты управления проектами путем поиска в *системе отслеживания ошибок* сходных проблем позволяют собрать важные метрики, связанные с кодом. Можно оценить и время, затраченное на связанное с целевым кодом изучение и исправление ошибок или внедрение изменений.

Скажем, код, связанный с определенной функцией или набором функций, со временем усложнялся. И нужно привести его в порядок, чтобы дальнейшая разработка пошла быстрее. Подозрения на снижение скорости работы можно подтвердить с помощью *ПО для управления проектами*. Хотя это очень грубая метрика, которая, как и остальные, количественно оценивает только один аспект общей проблемы. Для получения более точных данных необходимо хорошо разбираться в том, как команда проводит циклы разработки, и уверенно удалять из данных все резко отклоняющиеся значения. Но иногда вполне достаточно и того, что уже есть.



Руководители технических программ могут помочь со сбором, фильтрацией и донесением полученных метрик до руководства. Часто они отлично разбираются в инструментах управления проектами и находят труднодоступные документы. А возможно, у вас даже появится новый друг!

На данный момент все это может показаться чрезмерным объемом исследовательской работы для количественной оценки проблемы. Но это нормально! Только вам решать, какие показатели смогут лучше всего убедить в серьезности проблемы и выгоде от ее решения. Возможно, вы не хотите или вам не нужно тратить время на то, чтобы рыться в сотнях задач или читать postmortem-отчеты. Но если выяснится, что эту информацию легко найти и использовать, она может оказаться очень полезной. Любые метрики будут полезны для того, чтобы убедить в целесообразности рефакторинга руководство, сильно удаленное от кода и связанных с ним проблем.

Управление версиями

В первую очередь системы управления версиями воспринимаются как инструмент для контроля изменений кода. Они используются для пошагового продвижения вперед, позволяя разрабатывать сразу несколько функций, которые постепенно доставляются пользователям. Обращение к прошлым версиям позволяет отследить ошибку или найти того, кто знаком с каким-то разделом кода. При общем анализе кода мы редко воспринимаем системы управления версиями как источник информации о принятых в команде шаблонах разработки. Но взглянув на коммиты под другим углом, можно немного узнать о проблемах, с которыми сталкивается команда разработчиков.

Комментарии к коммитам

Короткие комментарии к коммитам могут дать остальным представление о проблемах, с которыми они могут столкнуться. Распознать шаблоны можно либо поиском по ключевым словам, либо локализацией *комментариев к коммитам* (*commit messages*), связанных с изменениями в интересующем нас наборе файлов.

Снова рассмотрим проблему с очередями задач. Известно, что инженеры перед постановкой задач в очередь все время забывают про очистку аргументов. Это приводит к регистрации личных данных (personally identifiable information, РП). Нужно провести поиск по комментариям к коммитам с помощью ключевых слов *job*, *job handler* или *РП*. Это даст набор коммитов, где добавлялось задание, ответственное за утечку РП, либо исправлялось задание с такой ошибкой. Если обработчики заданий удобно распределены по отдельным файлам, можно сузить поиск, включив в него только коммиты с модификациями этих файлов, а затем проверить результаты поиска на предмет похожих шаблонов.

Некоторые команды связывают свои коммиты или наборы изменений с инструментами управления проектами. Они выделяют в комментариях к коммитам или в имени ветки номера ошибок или тикетов. Если у вас есть эта информация, набор изменений можно будет связать с такими ранее полученными метриками, как скорость разработки и число ошибок. Круг замкнулся!

Коммиты

В своей книге *Software Design X-Rays* Адам Торнхилл (Adam Tornhill) предлагает методы извлечения из истории версий важных шаблонов разработки. По его мнению, эти шаблоны помогут в определении разделов приложения, которым нужно уделять приоритетное внимание при рефакторинге. Кроме того, они показывают, как со временем менялась сложность определенных функций, и указывают на все тесно связанные файлы или модули. Я рекомендую прочитать эту книгу, чтобы полностью понять причину информативности этих показателей. А в этом разделе я кратко опишу базовые методы.

Частота изменений — это число коммитов файла за всю историю версий приложения. Эти данные легко получить, извлекая из истории коммитов имена файлов, объединяя их и упорядочивая от наиболее к наименее частому. Торнхилл заметил, что в небольшом подмножестве основных файлов происходит непропорционально много изменений. Именно к этим файлам, которые редактируются чаще всего, нужно приложить больше всего усилий. Следует максимально упростить их понимание и навигацию — это положительно скажется на производительности труда разработчиков.

Такой подход применим и к функциям. Изучая коммиты, можно определить частоту изменения функций в отдельных файлах и общую для каждой из них. Объединив эти данные с метрикой количества строк кода, можно отобразить изменения сложности во времени для всей кодовой базы. Эта информация показывает возможные проблемные места. По завершении рефакторинга лучше снова изучить эти метрики, чтобы подтвердить, что уменьшилась не только сложность этих фрагментов кода, но и частота их изменений.

Торнхилл описывает, как при изучении наборов файлов, измененных в рамках одного коммита, обнаружить тесно связанные модули программы. Допустим, у нас есть три файла: `superheroes.js`, `supervillains.js` и `sidekicks.js`. В подмножестве коммитов мы видим, что первый коммит изменяет файлы `superheroes.js` и `sidekicks.js`, второй — все три файла, третий — снова `superheroes.js` и `sidekicks.js`, а четвертый — только `superheroes.js`. Для наглядности представим это в виде таблицы (табл. 3.3).

Легко заметить, что три коммита из четырех модифицировали как файл `superheroes.js`, так и файл `sidekicks.js`, что наводит на мысль о связи между этими файлами. Не всегда это плохо (как в случае изменений в исходном

коде и соответствующих файлах модульных тестов). Но иногда такие вещи указывают на ошибочную абстракцию, скопированный код или на все вместе. Как только проблема становится понятной, можно начинать ее устранение. После нужно повторно провести анализ, чтобы убедиться в успехе операции.

Таблица 3.3. Файлы, которые менялись при каждом коммите

| Коммит # | superheroes.js | supervillains.js | sidekicks.js |
|----------|----------------|------------------|--------------|
| 1 | x | | x |
| 2 | x | x | x |
| 3 | x | | x |
| 4 | x | | |

Как и метрики, описанные выше, эти показатели нужно использовать с оговорками. У разных разработчиков свои методы фиксации изменений. Одни делают много крошечных коммитов, другие предпочитают большие, включая в один набор десятки изменений, внесенных в несколько файлов. Помимо этого, в численных данных возможны резкие выбросы (часто изменяемые конфигурационные файлы или проблемные места в автоматически сгенерированном коде). Будьте бдительны с такими аномалиями, чтобы не обнаружить проблему там, где ее нет.

АВТОРСТВО КОДА

Еще из системы управления версиями Торнхилл извлекает информацию об авторстве кода. Чем больше у файла или модуля авторов, тем вероятнее обнаружение в них дефектов. Исследование, на которое он ссылается, не объясняет природу этой взаимосвязи. Сам Торнхилл утверждает, что это связано с возросшей потребностью в координации между авторами в сочетании с неопытностью в реализации. Возможно, вам будет интересно узнать, сколько человек работает над вашим кодом, и получить дополнительную метрику.

Репутация

Осознаем мы это или нет, но каждый раздел наших программных систем имеет свою репутацию. Иногда она положительная, иногда — глубоко отрицательная. Но какой бы ни была репутация, она упрочивается со временем. По мере того как все больше разработчиков взаимодействует с кодом, она

распространяется по всей организации. Слухи о самых неудачных кодовых базах иногда выходят за пределы компании и даже обсуждаются на интернет-форумах. Любая репутация может многое рассказать о самых проблемных частях приложения и о том, насколько они нуждаются в нашем внимании.

Проще всего узнать репутацию кода, *опросив* коллег-разработчиков. Представим, что перед нами стоит задача улучшить биллинг в приложении, взимающем с клиентов ежемесячную плату за обслуживание. На беседу приглашаются как разработчики, взаимодействующие с кодом биллинга на регулярной основе, так и те, кто иногда имеет с ним дело. В каждой из этих категорий будут люди с разным опытом. Ясно, что впечатления тех, кто много лет работал с кодом биллинга, будут сильно отличаться от впечатлений новичка.

Далее нужно написать список вопросов, которые должны дать представление о квалификации опрашиваемого, и узнать, что он думает о коде. Примерные варианты вопросов даны в табл. 3.4.

Таблица 3.4. Предлагаемые вопросы для собеседования с разработчиками

| Формат интервью | Формат опроса | Примечания |
|--|---|---|
| Как долго вы работаете с кодом X? | Выберите, как долго вы работаете с кодом X: больше 6 месяцев; от 6 месяцев до года; больше года | Временные диапазоны в опросе выбираются в зависимости от компании. В быстрорастущих и молодых компаниях это будут месяцы, а в более крупных, давно существующих компаниях это могут быть уже годы |
| Если бы вы могли поменять одну вещь в работе с кодом X, что бы это было? Почему? | Выберите из предлагаемого списка пункт, который улучшит ваш опыт работы с кодом X | Для опроса выбирайте варианты, на ваш взгляд, оказывающие наибольшее влияние. Можно добавить поле для своего варианта ответа. Если код не тестируется, добавьте вариант «полностью протестированный код». Если большая часть кода содержится в нескольких функциях в сотни строк, добавьте вариант «код разбивается на небольшие модульные функции» |

Продолжение ↗

Таблица 3.4 (продолжение)

| Формат интервью | Формат опроса | Примечания |
|--|---|------------|
| Расскажите о связанной с кодом X ошибке, которую вам недавно пришлось исправлять. Что могло бы тогда облегчить процесс исправления? | Что из перечисленного сильнее понижает эффективность исправления ошибок в коде X? | |
| Приходилось ли вам сознательно избегать работы с кодом X (то есть исправлять ошибку на уровне выше или ниже проблемной области)? Если да, расскажите об этом | Оцените по шкале от 1 до 5, где 1 — маловероятно, а 5 — очень вероятно, вероятность того, что вы найдете способ избежать редактирования кода X? | |
| Как, на ваш взгляд, сложность кода X мешает разрабатывать новые функции? | Оцените по 5-балльной шкале, где 1 — категорически не согласен, а 5 — полностью согласен, утверждение: сложность кода X сильно влияет на время, необходимое мне для разработки нового функционала | |
| Как, на ваш взгляд, сложность кода X мешает тестированию и/или отладке кода? | Оцените по 5-балльной шкале, где 1 — категорически не согласен, а 5 — полностью согласен, утверждение: сложность кода X значительно усложняет мне тестирование и/или отладку кода | |
| По вашему мнению, как сложность кода X мешает знакомиться с изменениями, которые вносят в код другие разработчики? | Оцените по 5-балльной шкале, где 1 — категорически не согласен, а 5 — полностью согласен, утверждение: сложность кода X сильно влияет на время и сложность проверки изменений, внесенных в код другими разработчиками | |

Допустим, предшествующий опыт работы с кодом биллинга показал, что выгоднее всего тщательный рефакторинг файла `chargeCardCard.js`. Но если собеседник, независимо от степени его знакомства с этим кодом, негативно реагирует в ответ на вопрос о файле, становится ясно, что этому файлу нужно уделить повышенное внимание.

Если хочется получить обратную связь от большой группы разработчиков или опрос нужно провести в сжатые сроки, вопросы интервью можно перефразировать. Можно дать им выбор из стандартного набора ответов. Это упростит статистическую обработку результатов и позволит быстрее сделать

выводы. Но важно помнить, что в таком случае будут упущены некоторые нюансы, которые можно было бы узнать при собеседовании.

По моему опыту, личный контакт дает больше возможностей для анализа впечатлений и проблем. Это живой диалог, помогающий выявлять лучшие идеи и озарения. Конечно, можно разослать разработчикам вопросы, которые задаются на интервью. Но вы потеряете возможность попросить у респондентов более детальную информацию, если вас что-то заинтересует. Кроме того, ответов будет меньше. Я прекрасно помню, как почти сразу откладывала на потом опросники с огромным числом открытых вопросов. Так что краткость опроса очень важна для получения от разработчиков максимального отклика. Это увеличит процент ответивших.

Иногда репутация мешает команде нанимать и удерживать сотрудников. Представьте, что в компании прекрасно известно, как ужасен код биллинга. Конечно, всегда могут найтись энтузиасты, готовые работать с чем угодно. Но удручающее сложная кодовая база не может не оказаться на общем моральном духе. Организации не любят признавать, что сотрудники уходят из-за качества кода и принятых практик разработки, но это происходит сплошь и рядом. Если удастся собрать информацию о причинах ухода людей из команды и показать, что это связано со сложностью кода, это может стать невероятно убедительным аргументом в пользу выделения ресурсов на рефакторинг.

Составляем полную картину

Теперь, когда мы ознакомились с широким спектром возможных метрик, нам нужно выбрать, какие из них использовать. Чтобы составить наиболее полное представление о текущем состоянии дел, нужно определить показатели, лучше всего отображающие конкретные проблемы, которые вы хотите решить. Ни одна из этих метрик по отдельности не может дать количественной оценки многих уникальных аспектов масштабируемого рефакторинга. Но вместе они позволяют получить многогранную характеристику проблемы.

Советую выбирать из каждой категории по одной метрике. Способ примерной оценки *сложности кода* ищется исходя из природы поставленной задачи и имеющегося инструментария. Далее нужно сгенерировать несколько *показателей тестового покрытия*, чтобы убедиться в правильности выбора.

Важно найти источник *официальной документации*, которой можно будет воспользоваться для иллюстрации проблем, требующих рефакторинга. Для подкрепления аргументов пригодится и *неофициальная документация*. Соберите как можно больше информации о проблемных фрагментах кода и шаблонах программирования, тщательно *проанализировав данные в системе управления версиями*. Не забывайте и о *репутации* кода, для выяснения которой достаточно поговорить с членами рабочей группы.

После сбора показателей для количественной оценки текущего состояния кода и его влияния на компанию выберите подмножества, которые больше всего подойдут для демонстрации улучшений после рефакторинга. Именно они убедительнее всего покажут членам рабочей группы и руководству, что время и трудозатраты на рефакторинг окупятся.

Успешный сбор данных, которые помогут правильно охарактеризовать стоящую перед нами проблему, — только часть головоломки. Дальше на основе этих данных нужно составить четкий план.

ГЛАВА 4

Составление плана

Когда-нибудь я проеду от Монреяля до Ванкувера. Эти города разделяет 4500 километров, которые можно проехать примерно за 48 часов, а самый быстрый маршрут идет вдоль границы Канады и США. Но самый быстрый маршрут не всегда самый интересный. Если добавить остановки для посещения Парламентского холма в Оттаве, культовой башни Си-Эн Тауэр в Торонто и парка Спящего великана, поездка станет дольше на несколько часов и длиннее на 600 километров.

Любой, кто путешествует на машине на большие расстояния, знает, что ехать без остановки от начала до конца непрактично и опасно. Поэтому прежде чем отправиться в путь, мне нужно было составить приблизительный план поездки. Прикинуть, сколько времени я могу провести за рулем, не испытывая дискомфорта, и куда заглянуть по пути. Оказалось, что поездка может занять от семи до десяти дней в зависимости от времени, потраченного на осмотр достопримечательностей. Мой план допускает и разные неожиданности. Например, решение чуть задержаться для знакомства еще с одним городом или поломка, которая заставит стоять на обочине в ожидании помощи.

Но как оценить, действительно ли удалось путешествие? Если на поездку был выделен определенный бюджет, цель можно считать достигнутой, когда следующая выписка по кредитной карте покажет попадание в рассчитанный диапазон трат. Иногда хочется вкусно поесть на каждой остановке. Иногда возникает желание увидеть что-то новое, провести время с друзьями и родственниками из другого города или как-нибудь иначе получить новые впечатления. Как бы банально это ни звучало, путешествие — не только прибытие в пункт назначения, но еще и сама дорога к нему.

Любую крупную разработку ПО можно сравнить с долгим путешествием. Определяются этапы выполнения, ставится примерный набор задач для каждого этапа, оценивается приблизительное время достижения цели. В процессе работы отслеживается прогресс для гарантии выполнения задач за отведенное время. Ведь нам нужен ощутимый и стабильный положительный результат. Мы потратили время, чтобы разобраться в истории кода и понять, как он деградировал, а после охарактеризовать эту деградацию. Сейчас настало время строить планы на будущее.

В этой главе я расскажу, как разделить рефакторинг на части, составив продуманный и точный план. Мы поговорим о том, когда и как ссылаться на метрики, тщательно собранные в предыдущей главе, чтобы охарактеризовать состояние проблемы. Вы узнаете о важности представления готового плана другим командам и о ценности его постоянного обновления в процессе рефакторинга.

У каждого свой подход к составлению плана действий. Документы, входящие в него, могут называть техническими спецификациями, описаниями продуктов или запросами на комментарии и предложения. Но все они выполняют одну функцию: фиксируют, что и как вы собираетесь делать. Четкий и лаконичный план — ключ к успеху любого программного проекта. Он заставляет всех сосредоточиться на важных задачах и обеспечить возможность наблюдать за прогрессом их выполнения.

Определение конечного состояния

Для начала определите конечное состояние. У нас уже есть четкое представление о том, где мы находимся. В главе 3 немало времени было потрачено на измерение и формулировку задачи. Теперь, когда под ногами появилась твердая почва, нужно решить, куда мы хотим попасть.

В путешествии

Я начинаю поездку из Монреяля — города, где я живу. Из множества населенных пунктов на Западном побережье мне нужно выбрать всего один. После небольшого исследования я выбираю Ванкувер.

Теперь нужно узнать, какие автомагистрали ведут туда и где я смогу остановиться по прибытии. Я прошу совета у друзей, которые либо жили в Ван-

кувере, либо часто бывают там. В итоге я решаю остановиться в Йельтауне — районе, известном своими старыми складскими зданиями у воды. Итак, пункт назначения четко определен, и можно начать думать над тем, как туда добраться.

На работе

Чтобы показать основные идеи, о которых пойдет речь в этой главе, я использую вымышленную компанию с 15-летней историей Smart DNA, Inc., где нужно провести большой рефакторинг. Большинство ее сотрудников — ученые-исследователи, вносящие свой вклад в сложный конвейер данных, включающий сотни скриптов Python в нескольких репозиториях. Развертка и выполнение сценариев происходит в пяти разных средах, основанных на версии Python 2.6. К сожалению, она давно устарела. Это привело к появлению дыр в системе безопасности и не дает обновлять важные зависимости. Несмотря на эти неудобства, в компании не хотят выполнить обновление до более новой версии Python. Тем более что ограниченные возможности тестирования делают это масштабное мероприятие очень рискованным. Это самый большой технический долг компании, который копился много лет.

В последнее время исследовательскую группу беспокоит отсутствие возможности пользоваться новыми версиями основных библиотек. Проблема обновления назрела как никогда остро. Поэтому наша цель — выяснить, как перевести все репозитории и среды на версию Python 2.7.

Управление зависимостями осуществляется с помощью диспетчера пакетов `pip`. У каждого репозитория есть свой список зависимостей в файле `requirements.txt`. Таких файлов несколько, и, переключаясь между проектами, сложно вспомнить, какие именно зависимости есть в конкретном. По итогу команда разработчиков должна провести аудит каждого файла и по отдельности обновить их все для обеспечения совместимости с Python 2.7. В результате принимается решение объединить репозитории для унификации зависимостей. Это упростит обновление до Python 2.7 (и последующую эксплуатацию ПО).

В план выполнения нужно внести начальные показатели и целевые конечные показатели, добавив необязательный, но полезный столбец для фактического наблюдаемого конечного состояния. В нашем случае начальный набор показателей определяется легко: у каждого репозитория есть отдельный список зависимостей, а в каждой среде запущен Python 2.6. Определить целевые

показатели тоже просто: каждая среда должна работать на Python 2.7 и иметь один четкий и емкий набор необходимых библиотек. Всю эту информацию можно увидеть в табл. 4.1.

Таблица 4.1. Сравнение показателей в начале проекта с целевыми показателями и с реальным результатом по завершении проекта

| Метрика | Начало | Цель | Наблюдается |
|---|--------------|--------------|-------------|
| Среда 1 | Python 2.6.5 | Python 2.7.1 | — |
| Среда 2 | Python 2.6.1 | Python 2.7.1 | — |
| Среда 3 | Python 2.6.5 | Python 2.7.1 | — |
| Среда 4 | Python 2.6.6 | Python 2.7.1 | — |
| Среда 5 | Python 2.6.6 | Python 2.7.1 | — |
| Количество отдельных списков зависимостей | 3 | 1 | — |



В таблице можно указывать как идеальное, так и приемлемое конечное состояние. Иногда 80 % работы дают 99 % тех преимуществ, ради которых затевался рефакторинг. Не нужно прилагать усилия ради достижения 100 % — это просто нецелесообразно.

Поиск кратчайшего расстояния

Теперь нужно рассмотреть самый короткий путь из начального состояния в конечное. Так мы узнаем минимальное количество времени для выполнения проекта. Знание кратчайшего пути дает курс, которого нужно будет придерживаться на промежуточных этапах.

В путешествии

Любая поисковая система легко покажет самый короткий маршрут из Монреяля в Ванкувер (рис. 4.1). Если дороги будут свободными, поездка без остановок займет около 47 часов.

Чтобы найти более разумную нижнюю границу времени поездки, я прикидываю, сколько часов в день смогу комфортно провести за рулем, и делю 47 часов на это число. Если каждый день ехать по восемь часов, дорога займет около шести дней.



Рис. 4.1. Кратчайший маршрут от моего дома в Монреале до района Йельтаун в Ванкувере

Теперь мы знаем кратчайший путь. Время подумать о том, какие препятствия могут встретиться по дороге, и отредактировать стратегию. Особенность прямого маршрута в том, что он в основном проходит через Соединенные Штаты, а не Канаду. Мне не хотелось бы ехать по территории другого государства, а это добавит к поездке еще пару часов. Но общая сложность пути при этом понижается (мне не нужно брать с собой паспорт и беспокоиться о том, сколько времени займет пересечение границы), так что в приоритете теперь другой маршрут (рис. 4.2).



Рис. 4.2. Более медленный маршрут, но полностью проходящий по канадской территории

На работе

К сожалению, для программных проектов еще не придумали Google Maps. Так что возникает вопрос: как определить кратчайший путь к завершению проекта? Это можно сделать двумя способами.

- Откройте пустой документ и в течение 15–20 минут (или пока не закончатся идеи) записывайте каждый шаг, который сможете придумать. Отложите документ на несколько часов (а лучше на день-два) и после попробуйте последовательно расположить шаги. В процессе спрашивайте

себя, действительно ли каждый шаг необходим для достижения цели. Удалите те, что удостоились отрицательного ответа. Перечитайте то, что получилось, и заполните любые явные пробелы. Не беспокойтесь об угловатости вашего плана. Сейчас важно получить минимальный набор шагов для завершения проекта. Это еще не конечный продукт.

- Соберите сотрудников, либо заинтересованных в проекте, либо способных внести в него свой вклад. Раздайте стикеры и ручки. Попросите в течение 15–20 минут (или пока не иссякнут идеи) записывать каждый шаг, который покажется человеку важным для достижения цели, на отдельный стикер. После попросите кого-нибудь выложить записанные им шаги в хронологическом порядке. Остальные при этом просматривают собственные и, если обнаруживают какие-то не озвученные варианты, вставляют их в соответствующее место на временной шкале. По завершении процедуры начинается коллективное обсуждение каждого шага, чтобы выяснить необходимость каждого шага для достижения цели и отбросить лишнее. В результате должен остаться минимально разумный набор шагов. Этот метод легко адаптировать для распределенных команд. Результаты мозгового штурма нужно просто оформить в общедоступный документ. Но в любом случае конечным результатом этого упражнения должен стать письменный документ, который легко распространять и совместно улучшать.

Ничего страшного, если эти варианты вам не подходят! Используйте тот метод, который считаете эффективнее. Главное — итоговый список шагов, на ваш взгляд, моделирующих прямой путь к цели, независимо от того, как они сформулированы.

Команда Smart DNA несколько часов занималась мозговым штурмом, нарисовав на доске временную шкалу, где слева была отправная точка, а справа — поставленная цель. Вот что получилось в результате.

- Вручную создать список всех пакетов в каждом из репозиториев.
- Сузить список до необходимых пакетов.
- Определить версию каждого пакета, который нужно обновить до Python 2.7.
- Создать контейнер Docker со всеми необходимыми пакетами.
- Протестировать контейнер Docker в каждой из сред.
- Найти тесты для каждого репозитория. Определить надежность каждого из них.
- Объединить все репозитории в один.

- Выбрать линтер и его конфигурацию.
- Включить линтер в систему непрерывной интеграции.
- С помощью линтера найти проблемы в коде (неопределенные переменные, синтаксические ошибки и т. п.).
- Устранить выявленные линтером проблемы.
- Во всех средах установить и протестировать Python 2.7.1.
- Использовать Python 2.7 для набора неопасных сценариев.
- Развернуть Python 2.7 для всех сценариев.

Некоторые из этих шагов можно распараллелить или переупорядочить, некоторые допускают разбиение на более подробные этапы. Пока нам важно получить общее представление о необходимых шагах. Полученный список будет уточняться на протяжении всей главы.

Промежуточные шаги

Теперь воспользуемся описанной выше процедурой, чтобы составить упорядоченный список промежуточных шагов. Они могут быть разного размера и распределяться неравномерно. Главное — достичь контрольной точки за приемлемое время. Нужно сосредоточиться на определении шагов, которые значимы сами по себе. То есть или достижение контрольной точки можно считать победой, или в этот момент при необходимости можно без проблем остановиться (или и то и другое). Если вы сможете определить контрольные точки, которые не просто значимы, но и демонстрируют вероятный результат усилий, прилагаемых к рефакторингу, можно сказать, что вы отлично продвигаетесь вперед!

В путешествии

Я спросила у друзей и родственников, какие достопримечательности можно посмотреть и чем заняться по дороге между Виннипегом и Ванкувером. Выбрав из полученного списка рекомендаций наиболее интересные, я составила примерный маршрут, включавший все — от кемпинга до посещения музеев, симпатичных кафе и нескольких визитов к родственникам (рис. 4.3). Ни одна из перечисленных вещей не сбивала меня с основного курса.



Рис. 4.3. Мой приблизительный маршрут

На работе

Эту тактику можно применить и для определения основных шагов рефакторинга. Для каждого этапа из списка задайте себе следующие вопросы.

1. Можно ли завершить это шаг в разумные сроки?

Для примера возьмем из списка выше такой шаг, как объединение репозиториев для удобства дальнейшей работы. Команда разработчиков ПО Smart DNA ожидает, что для правильного объединения репозиториев без нарушения процесса работы исследовательской группы нужно шесть недель. Но разработчики привыкли выполнять такие операции быстрее, а сотрудники обеспокоены возможными помехами в работе, которые могут возникнуть при слишком раннем объединении репозиториев в процессе миграции. Поэтому решено начать с более простого первого шага: создать файл `requirements.txt` для всех зависимостей пакетов из каждого репозитория. В результате набор зависимостей сократится. Это упростит работу сотрудников и станет существенным шагом к слиянию репозиториев, и все это произойдет задолго до завершения миграции на Python 2.7.

2. Полезны ли действия этого этапа сами по себе?

Основные шаги всегда должны отражать преимущества рефакторинга. Поэтому в процессе их определения нужно сосредоточиться на вещах, которые сразу же приносят пользу другим сотрудникам. Вносимые изменения должны поднимать моральный дух команды разработчиков и других инженеров.

Например, при оценке процесса миграции на новую версию Python было обнаружено, что ни один из репозиториев не использовал непрерывную интеграцию для поиска общих проблем в предлагаемых изменениях кода. При этом проверка соблюдения стандартов оформления кода может показать возможные проблемы после миграции на Python 2.7. А добавление автоматической проверки в дальнейшем может помочь улучшить стиль программирования. Дополнение настолько полезное, что в других обстоятельствах могло бы стать самостоятельным проектом. Это ощутимый и значительный промежуточный шаг.

3. Если что-то произойдет, получится ли остановиться на этом шаге и легко вернуться к нему позже?

В идеальном мире не нужно учитывать изменения в приоритетах бизнеса, разные процессы или реорганизации. В реальности же от этих вещей никуда не деться. Вот почему лучшие планы строятся с учетом возможных неожиданностей. Этого можно достичь, разделив проект на части, которые могут выполняться изолированно друг от друга. В примере с миграцией на новую версию Python рефакторинг можно спокойно приостановить после исправления всех ошибок и предупреждений, выделенных линтером, перед запуском подмножества сценариев с использованием новой версии. Здесь остановка процесса на полпути не помешает работе сотрудников компании, активно использующих репозиторий. При этом мы достигли прогресса в движении к цели. И когда появится возможность продолжить рефакторинг, мы сможем легко это сделать.

После составления списка стратегических шагов нужно реорганизовать план выполнения для выделения этих шагов и группировки подзадач соответствующим образом. Вот итоговый вариант плана.

- Создать файл `requirements.txt`.
 - Подсчитать все пакеты из всех репозиториев.
 - Проанализировать полученный перечень и оставить там только важные пакеты.
 - Определить текущую версию каждого пакета.
- Объединить все репозитории.
 - Создать новый репозиторий.
 - Добавить все существующие репозитории новой командой `git submodule`.

- Создать образ Docker со всеми нужными пакетами.
 - Протестировать образ Docker во всех средах.
- Включить в процесс непрерывной интеграции лингтинг для монорепозитория.
 - Выбрать линтер и необходимую конфигурацию.
 - Добавить линтер в процесс непрерывной интеграции.
 - Воспользоваться линтером для выявления логических проблем в коде (неопределенных переменных, синтаксических ошибок и т. п.).
- Во всех средах установить и развернуть Python 2.7.1.
 - Локализовать тесты для каждого репозитория и определить надежность каждого из них.
 - Использовать Python 2.7 для подмножества безопасных сценариев.
 - Развернуть Python 2.7 для всех сценариев.

Надеюсь, определение основных шагов поможет вам составлять сбалансированные, достижимые и оправдывающие себя планы рефакторинга. Конечно, это непросто, ведь на практике при сравнении шагов часто сложно сказать, какой из них требует больших усилий, а какой сильнее влияет на общее состояние кода. Мы рассмотрим этот процесс на примерах в главах 10 и 11, когда речь пойдет о стратегическом планировании крупномасштабного рефакторинга.

ПОВТОРЯЕМЫЕ ШАГИ

Один из способов разбиения рефакторинга на значимые этапы — выбор отдельных фрагментов кода, где его можно применить. Получается своего рода мини-рефакторинг, отображающий основные задачи рефакторинга в небольшом масштабе. Выбирать для этого лучше или часть кодовой базы, которая остро нуждается в рефакторинге, или часть, реорганизация которой требует небольших усилий, но хорошо моделирует улучшения.

Этот процесс можно повторять на разных фрагментах кода по всей целевой поверхности. Так можно по очереди сосредоточиваться на определенных частях кодовой базы и координировать свои действия с командами, на которые могут повлиять изменения. Работа с каждым таким фрагментом приближает нас к цели, ограниченно затрагивая общую кодовую базу. Хорошо распределенная кодовая база минимизирует вероятность того, что рефакторинг какой-то из ее частей займет слишком долгое время.

К примеру, процесс слияния репозиториев в Smart DNA можно было бы разделить на несколько отдельных повторяемых шагов для каждого. Сначала файл `requirements.txt` репозитория объединялся бы с глобальным файлом `requirements.txt`. Затем командой `git submodule` этот репозиторий добавлялся бы в монорепозиторий. После оставалось бы только удостовериться, что сценарии запускаются, и процедуру можно было бы повторять для остальных репозиториев.



Если это целесообразно, поищите способ перенести изменения на отдельный уровень. Так вся подлежащая редактированию логика прячется за какой-то абстракцией. Это сводит на нет риск того, что другие разработчики могут пострадать из-за многократных реализаций (и их деталей). После создания абстракции можно сосредоточиться на внесении нужных изменений в логику.

Еще раз повторю важные моменты этого раздела. После определения конечного состояния и основных шагов к нему начинается попытка интерполировать переходы от одной контрольной точки к другой. В процессе мы сосредоточиваемся на самых важных моментах и составляем подробный план выполнения.

Будет нeliшним выяснить, все ли части рефакторинга обязаны выполняться в том порядке, в котором они фигурируют в плане, или некоторые можно проводить в любой момент с небольшим числом предварительных условий или вообще без них. Допустим, мы определили основные этапы проекта А, В, С и D. Оказалось, что прежде чем приступить к этапам В или С, обязательно нужно выполнить этап А. Кроме того, до завершения этапа В нельзя начинать D. По итогу для этапа С есть три варианта: его можно выполнять одновременно с D, завершить С и перейти к D или завершить D, а потом перейти к С.

Если есть подозрение, что этап В будет сложным и длительным, да и этап D выглядит не лучше, то имеет смысл поместить этап С между В и D. Это поможет поднять моральный дух, который падает из-за длительной работы над каким-то этапом. С другой стороны, если есть вероятность, что распараллеливание работы над этапами С и D поможет немного раньше завершить проект, этот вариант тоже можно рассмотреть.

Перед нами стоит задача уравновесить время и усилия, связанные с каждым обязательным шагом, учитывая их влияние на кодовую базу и настроения в команде.

Выбор стратегии развертывания

От наличия продуманной стратегии развертывания зависит, закончится рефакторинг успехом или полным провалом. Поэтому крайне важно добавить этот пункт в план выполнения. Если процесс рефакторинга разбит на несколько этапов и на каждом применяется своя стратегия развертывания,

обязательно выделяйте ее среди завершающих шагов каждого этапа. Есть много методов развертывания, но мы обсудим только стратегии для непрерывного развертывания.

При таком развертывании разработка нового функционала сопровождается его ручным и автоматическим тестированием на протяжении всего процесса. Готовый функционал постепенно развертывается для реальных пользователей. Перед этим многие команды практикуют развертывание во внутренней сборке продукта, давая себе еще одну возможность устраниить оставшиеся проблемы. Провести оценку успешности при этом очень легко. Если функционал работает как следует, замечательно! Если нашлись какие-то ошибки, разрабатывается исправление и в зависимости от его результатов либо повторяется инкрементальное развертывание, либо функционал разворачивается для пользователей.



В средах непрерывного развертывания обычная практика — постоянное использование переключателей функций для скрытия, включения/отключения разного набора функций или путей выполнения кода. Это позволяет гибко разграничивать доступ пользователей к определенному функционалу (иногда в соответствии с набором атрибутов). Например, работая над приложением для социальных сетей, вы можете захотеть предоставить функцию всем пользователям в пределах одной географической области, случайному 1 % пользователей по всему миру или всем пользователям старше 40 лет.

Конечно, при рефакторинге хотелось бы тестировать внесенные изменения часто, на ранней стадии и очень аккуратно развертывать их для пользователей. Но часто определить, все ли работает как задумано, непросто. Тем более что один из базовых показателей успешного рефакторинга — *сохранение поведения*. Определить *отсутствие* изменений труднее, чем обнаружить даже самое маленькое изменение. Убедиться в том, что рефакторинг не породил новых ошибок, проще всего, сравнив поведение кода до рефакторинга и после него.

Темный/светлый режим

В компании Slack для сравнения поведения до и после рефакторинга придумали технику светлого/темного режима. Вот как она работает.

Сначала отредактированная версия логики реализуется отдельно от исходной, как в примере 4.1.

Пример 4.1. Новая и старая реализация (могут находиться в разных файлах)

```
// Линейный поиск; это старая реализация
function search(name, alphabeticalNames) {
    for(let i = 0; i < alphabeticalNames.length; i++) {
        if (alphabeticalNames[i] == name) return i;
    }
    return -1;
}

// Двоичный поиск; это новая реализация
function searchFaster(name, alphabeticalNames) {
    let startIndex = 0;
    let endIndex = alphabeticalNames.length - 1;

    while (startIndex <= endIndex) {
        let middleIndex = Math.floor((startIndex+endIndex)/2);
        if (alphabeticalNames[middleIndex] == name) return middleIndex;

        if (alphabeticalNames[middleIndex] > name) {
            endIndex = middleIndex - 1;
        } else if (alphabeticalNames[middleIndex] < name) {
            startIndex = middleIndex + 1;
        }
    }

    return -1;
}
```

Далее, из текущей реализации логика перемещается в отдельную функцию (пример 4.2).

Пример 4.2. Старая реализация перенесена в отдельную функцию

```
// Существующая функция теперь вызывается в перемещенной реализации
function search(name, alphabeticalNames) {
    return searchOld(name, alphabeticalNames);
}

// Логика линейного поиска перенесена в новую функцию
function searchOld(name, alphabeticalNames) {
    for(let i = 0; i < alphabeticalNames.length; i++) {
        if (alphabeticalNames[i] == name) return i;
    }
    return -1;
}

// Двоичный поиск; это новая реализация
function searchFaster(name, alphabeticalNames) {
    let startIndex = 0;
    let endIndex = alphabeticalNames.length - 1;
```

```
while (startIndex <= endIndex) {  
    let middleIndex = Math.floor((startIndex+endIndex)/2);  
    if (alphabeticalNames[middleIndex] == name) return middleIndex;  
  
    if (alphabeticalNames[middleIndex] > name) {  
        endIndex = middleIndex - 1;  
    } else if (alphabeticalNames[middleIndex] < name) {  
        startIndex = middleIndex + 1;  
    }  
}  
  
return -1;  
}
```

Теперь предыдущая функция преобразовывается в абстракцию, из которой можно вызывать обе реализации. Разница темного и светлого режима в том, что после вызова обеих реализаций и сравнения результатов первый возвращает результаты старой, а второй — новой. На примере 4.3 видно, что изменение существующего определения функции позволяет оставить большую часть кода незатронутой (из приведенного примера это непонятно, но для предотвращения снижения производительности обе реализации должны выполняться одновременно).

Пример 4.3. Существующий интерфейс используется как абстракция для вызова старой и новой реализаций

```
// Функция преображена в абстракцию для вызова обеих реализаций  
function search(name, alphabeticalNames) {  
    // В темном режиме (darkMode) возвращаем старый результат.  
    if (darkMode) {  
        const oldResult = searchOld(name, alphabeticalNames);  
        const newResult = searchFaster(name, alphabeticalNames);  
  
        compareAndLog(oldResult, newResult);  
  
        return oldResult;  
    }  
  
    // В светлом режиме (lightMode) возвращаем новый результат.  
    if (lightMode) {  
        const oldResult = searchOld(name, alphabeticalNames);  
        const newResult = searchFaster(name, alphabeticalNames);  
  
        compareAndLog(oldResult, newResult);  
  
        return newResult;  
    }  
}
```

```
    return search(name, alphabeticalNames);
}

// Логика линейного поиска перенесена в новую функцию.
function searchOld(name, alphabeticalNames) {
    for(let i = 0; i < alphabeticalNames.length; i++) {
        if (alphabeticalNames[i] == name) return i;
    }
    return -1;
}

// Двоичный поиск; это новая реализация.
function searchFaster(name, alphabeticalNames) {
    let startIndex = 0;
    let endIndex = alphabeticalNames.length - 1;

    while (startIndex <= endIndex) {
        let middleIndex = Math.floor((startIndex+endIndex)/2);
        if (alphabeticalNames[middleIndex] == name) return middleIndex;

        if (alphabeticalNames[middleIndex] > name) {
            endIndex = middleIndex - 1;
        } else if (alphabeticalNames[middleIndex] < name) {
            startIndex = middleIndex + 1;
        }
    }

    return -1;
}

function compareAndLog(oldResult, newResult) {
    if (oldResult != newResult) {
        console.log(`Diff found; old result: ${oldResult}, new result:
${newResult}`);
    }
}
```

После корректного расположения абстракции включается темный режим (выполнение двойного кода с возвращением результатов старой реализации). Мы ищем различия между двумя наборами результатов, а после отслеживаем и исправляем в новой реализации любые ошибки, которые могут быть причиной этих расхождений. Процесс повторяется до полного устранения всех несоответствий включением темного режима для более широких групп пользователей.

После того как мы переберем всех пользователей в темном режиме, начиная со сред с наименьшим риском, для небольших групп пользователей будет

включаться светлый режим (то есть возврат будет происходить из новой реализации). При этом должны фиксироваться любые различия в наборах результатов. Это может пригодиться, если активная разработка связанного кода вызовет какие-то изменения в старой реализации. Постепенно нужно переводить в светлый режим более широкие группы пользователей, пока новая реализация не начнет для всех случаев давать те же результаты, что и старая.

После этого можно отключить обе ветки кода, продолжая отслеживать любые ошибки, и удалить абстракцию, переключатели функций и логику условного выполнения. Как только результат рефакторинга через какое-то время (каждый определяет его сам в зависимости от случая) станет доступен пользователям, старая логика полностью удаляется. На месте старой реализации должна появиться новая (пример 4.4).

Пример 4.4. Новая реализация внутри старого определения функции

```
// Двоичный поиск; это новая реализация
function search(name, alphabeticalNames) {
    let startIndex = 0;
    let endIndex = alphabeticalNames.length - 1;

    while (startIndex <= endIndex) {
        let middleIndex = Math.floor((startIndex+endIndex)/2);
        if (alphabeticalNames[middleIndex] == name) return middleIndex;

        if (alphabeticalNames[middleIndex] > name) {
            endIndex = middleIndex - 1;
        } else if (alphabeticalNames[middleIndex] < name) {
            startIndex = middleIndex + 1;
        }
    }

    return -1;
}
```

Конечно, у такого подхода есть и минусы. Желательно следить за производительностью реорганизуемого кода. Особенно когда работа происходит в среде, где не поддерживается истинная многопоточность (PHP, Python или Node). В этом случае лучше исключить одновременное выполнение двух версий одной логики. К примеру, вдвое возрастет число посылаемых сетевых запросов, если в процессе рефакторинга не уменьшится количество зависимостей. В итоге за возможность с высокой точностью проверять вносимые изменения приходится платить увеличением задержки. Решением может стать выборочное сравнение двух реализаций. Иногда сравнение всего 5 % случаев позволяет собрать достаточно данных, чтобы определить,

работает ли новая реализация как надо, не нанося слишком большого ущерба производительности.

Но нужно помнить о дополнительной нагрузке на связанные ресурсы. К ним может относиться что угодно — от базы данных до очереди сообщений и систем для регистрации различий между сравниваемыми реализациями. Если при реорганизации кода с высоким трафиком мы хотим часто сравнивать старую и новую версии, нужно следить за тем, чтобы случайно не перегрузить базовую инфраструктуру.

Опыт показывает, что сравнения выявляют много неожиданных различий (особенно при рефакторинге давно существующего сложного кода). Лучше медленно и постепенно увеличивать двойное выполнение и сравнения, чем перегрузить систему протоколирования. Начальная частота сравнений должна быть небольшой. Потом мы начинаем устранять все возникающие различия и постепенно увеличивать частоту сравнений до 100 % или стабильного состояния, при котором гарантированно не возникнет дальнейших расхождений.

Развертывание гипотетического кода

В процедуре рефакторинга в фирме Smart DNA переход каждой зависимости репозиториев на совместимые с Python 2.7 версии был рискованнее, чем запуск существующего кода с использованием более новой версии Python. Команда разработчиков решила сначала провести предварительные тесты. Она настроила в изолированной среде уменьшенную версию конвейера данных, установив обе версии Python и выполнив в среде 2.7 несколько заданий с новым файлом зависимостей. После появления уверенности в результатах предварительных тестов начался медленный и осторожный ввод в эксплуатацию нового набора зависимостей.

Для уменьшения риска команда проверила задания из конвейера передачи данных и сгруппировала их по степени важности. Затем для миграции было выбрано задание с самым низким уровнем риска и с наименьшим числом нисходящих зависимостей. Вместе с сотрудниками фирмы было выбрано подходящее время для ввода новой конфигурации с новым файлом `requirements.txt` и новой версией Python. После внесения изменений разработчики начали следить за журналом регистрации, чтобы вовремя обнаруживать любое странное поведение, связанное с заданием. В случае возникновения проблем конфигурация возвращалась к исходной версии, а разработчики начинали их устранять. Далее эксперимент повторялся

с исправленной версией. В рамках плана развертывания эксплуатация измененной конфигурации велась несколько дней. После многократного успешного выполнения задания разработчики переходили к следующему.

После успешного переноса второго задания в новую конфигурацию были включены все задания с низким уровнем риска. Затем процесс повторялся для заданий со средним уровнем риска. Задания с самым высоким уровнем риска переносились по одному с выделением нескольких дней на проверку каждого из них. По оценкам разработчиков, полная миграция конвейера данных в новую среду должна была занять почти два месяца. На такой сложный процесс разработчики и сотрудники фирмы согласились для снижения риска. Благодаря этому подходу можно было быстро устранять проблемы, возникающие на каждой итерации, гарантируя максимальную исправность конвейера на протяжении всего процесса.

Очистка кода

Я уже говорила, что не стоит приступать к рефакторингу, если нет времени для его завершения. Ни один рефакторинг не будет завершен, пока оставшиеся переходные артефакты не будут как следует очищены. Ниже приведен краткий список видов артефактов, генерируемых в процессе рефакторинга.

- *Переключатели функций.* Большинство из нас оставляло их в коде. Забытый на несколько дней (или несколько недель) переключатель не создаст особых проблем. Но в финальной версии забытые переключатели могут повлечь неприятности. Необходимость проверки состояния каждого переключателя усложняет код. Читающим этот код разработчикам приходится постоянно учитывать изменение поведения в зависимости от состояния переключателя. При разработке функций в среде непрерывного развертывания это неизбежная необходимость. Но переключатели нужно удалять сразу, как появляется такая возможность, ведь они имеют свойство накапливаться. Один переключатель не утяжелит приложение, а вот сотни вполне могут. Хорошим тоном считается снабжение каждого переключателя информацией об его авторе и сроке действия. После указанной даты можно связаться с автором для решения судьбы переключателя.
- *Абстракции.* Экранировать рефакторинг от других разработчиков можно с помощью абстракции. Вы уже знакомы с этим приемом. Абстракция использовалась в процедуре развертывания в подразделе «Темный/светлый режим» выше. Но по завершении рефакторинга эти абстракции теряют

смысл и создают путаницу. Если же абстракции содержат значимую логику, стремитесь к их упрощению. Это нужно, чтобы у разработчиков, которые позже будут читать код, не возникло впечатления, что они написаны для плавного рефакторинга.

- *Неиспользуемый код.* Последствием масштабных реорганизаций обычно становится большое количество неиспользуемого кода. Сам по себе он не опасен, но в будущем разработчикам придется тратить время на попытки определить, применяется ли этот код. Но мы уже обсуждали это в подразделе «Неиспользуемый код» главы 2.
- *Комментарии.* Рефакторинг сопровождается добавлением множества комментариев. Мы предупреждаем других разработчиков о нестабильном коде, иногда оставляем несколько TODO или отмечаем неиспользуемый код, который нужно удалить после завершения работы. Эти комментарии тоже нужно удалить, чтобы никого не вводить в заблуждение. Если мы столкнемся с какими-то незавершенными TODO, мы будем еще больше удовлетворены тем, что нашли время, чтобы привести в порядок нашу работу.
- *Модульные тесты.* Иногда во время рефакторинга для проверки правильности внесенных изменений пишутся повторные модульные тесты. Чтобы не путать разработчиков, которые позже будут к ним обращаться, все дубликаты нужно удалить (кроме того, избыточные модульные тесты помешают, если команда решит поддерживать быстрый набор модульных тестов).



Несколько лет назад мой коллега провел эксперимент, чтобы определить, сколько времени тратится на вычисление переключателей функций. Для среднестатистического запроса к нашим серверам он составил почти 5 % времени выполнения. При этом большинство переключателей, на вычисление которых тратилось время, применялось для уже введенного в эксплуатацию кода, соответственно, они были не нужны. После того как разработчики начали убирать свои устаревшие переключатели, всего за несколько недель времени, затрачиваемое на их обработку, значительно сократилось.

По сути, причина, по которой нужно убирать все артефакты переходного периода, одна — минимизация путаницы и раздражения, которое она вызывает у разработчиков. Артефакты усложняют код и заставляют тратить много времени на попытку понять их назначение. Этого легко избежать, просто вовремя удалив их!



Можно выбрать тег для обозначения любых артефактов, подлежащих удалению по завершении рефакторинга. Это может быть простой комментарий *TODO: имя-проекта, удалить после развертывания*. Вне зависимости от формы тегов, максимально упростите их поиск, чтобы на завершающей стадии проекта оперативно обнаружить все места, где можно навести порядок.

Ссылка на метрики

В главе 3 я показала вам много способов охарактеризовать начальное состояние системы. Напомню, что именно количественные показатели должны стать убедительным аргументом в пользу рефакторинга как для коллег, так и для руководства. Глава началась с использования этих же показателей для характеристики конечного состояния. Теперь эти метрики помогут нам в описании промежуточных шагов. Они позволят определить наличие ожидаемого прогресса и на ранней стадии скорректировать направление работ при выборе неверного направления.

Руководителям предстаиваются и планы выполнения. Чтобы они поддержали инициативу, нужна не только убедительно поставленная задача с четкими критериями успеха, но и точные показатели прогресса. Тщательно прописанный сценарий должен уменьшить опасения, мешающие дать разрешение на длительный рефакторинг.

Промежуточные этапы

Вспомните табл. 4.1 со значениями метрик в начале рефакторинга и по его завершении. Для каждого промежуточного этапа можно добавить запись, указывающую, изменения каких показателей мы ожидаем и насколько они должны измениться.

Для характеристики промежуточных измерений лучше всего подойдут метрики конечной цели — сложность, временные показатели, величина тестового покрытия и число строк кода. Но имейте в виду возможное временное ухудшение показателей! Например, прием, описанный в подразделе «Темный/светлый режим», предполагает появление второго пути кода. Это неизбежно приведет к ощутимому росту таких показателей, как сложность и количество строк.

К сожалению, в примере с миграцией на более новую версию Python все, что связано с кодом, большую часть времени остается без изменений. Соответственно, показатели начнут меняться только после начала развертывания новой версии в каждой из сред. Получается, что в таких случаях нужен другой набор метрик для отслеживания прогресса на всех этапах рефакторинга.

Метрики промежуточных этапов

Итак, нам нужна хотя бы одна метрика для отслеживания прогресса на промежуточных этапах. Она может не соотноситься напрямую с конечной целью, главное, чтобы она могла быть ориентиром на пути к ней.

Здесь возможны несколько вариантов. В примере с фирмой Smart DNA была настроена непрерывная интеграция, и линтер выводил предупреждения о неопределенных переменных. На рассматриваемом промежуточном этапе метрикой, позволяющей следить за прогрессом, может быть число остающихся предупреждений. В табл. 4.2 есть все основные этапы, определенные в ходе мозгового штурма, с соответствующими показателями прогресса. Начальное значение метрики, используемой на этапе линтинга, приблизительное.

Для получения этой оценки в трех репозиториях по очереди была запущена программа `pylint` с конфигурацией по умолчанию. Далее число сгенерированных предупреждений суммировалось.

Таблица 4.2. Метрики этапов миграции на новую версию Python в фирме Smart DNA

| Описание этапа | Описание метрики | Начальное значение | Цель | Наблюдаемое значение |
|---|---|--------------------|------|----------------------|
| Создать единый файл <code>requirements.txt</code> | Количество списков зависимостей | 3 | 1 | — |
| Слить все репозитории в один | Количество репозиториев | 3 | 1 | — |
| Построить образ Docker для всех требуемых пакетов | Количество сред, использующих новый образ | 0 | 5 | — |
| Включить линтинг в непрерывную интеграцию для монорепозитория | Количество предупреждений линтера | Примерно 15 000 | 0 | — |
| Установить и внедрить Python 2.7.1 во всех средах | Количество заданий, запущенных на Python 2.7.1 с новым файлом <code>requirements.txt</code> | 0 | 158 | — |

Оценка

После выбора метрик для самых важных этапов можно приступать к оценкам. До завершения плана далеко, поэтому оценки не должны быть конкретными (лучше использовать порядок недель или месяцев, а не дней), и главное, должны быть завышенными.

Вернемся к нашему путешествию. Для поездки через всю Канаду были установлены обобщенные правила со временем и местом остановок для еды и отдыха. Самый длинный из запланированных перегонов — между городом Реджайна в провинции Саскачеван и городом Калгари в провинции Альберта. Это чуть менее 800 км, которые можно преодолеть примерно за 7,5 часа. Утром я даю себе время на сборы и планирование дня. Но выезжаю достаточно рано, ведь согласно плану в день за рулем нужно проводить не больше восьми часов. Важно соблюсти баланс: ежедневно преодолевать большой участок пути, но так, чтобы не выгореть по прибытии в Ванкувер.

У большинства команд есть правила и процедуры оценки. Но с теми, у кого их нет (или кто не совсем понимает, как оценивать крупные программные проекты), я поделюсь простым методом. Оцените каждый этап от 1 до 10, где 1 — относительно короткая задача, а 10 — длительная. Подумайте, сколько времени может занять самый длинный этап.

Теперь представьте, что на этом этапе что-то пошло не так, и обновите оценку, чтобы учесть это. Здесь важно не переусердствовать! Если на возможные форс-мажоры будет выделено слишком много времени, руководство решит, что проводить рефакторинг будет долго и хлопотно. После все более короткие этапы сравниваются с самым длинным. Если длинный этап займет около десяти недель, примерно столько же должен длиться и второй по величине этап. Его можно оценить в девять недель. Так вы постепенно дадите всем этапам, от наибольшего к наименьшему, приблизительную оценку.

Для рефакторинга завышать оценки нужно по двум причинам. Во-первых, это дает пространство для маневра при столкновении с препятствиями. Чем крупнее программный проект, тем вероятнее, что что-то пойдет не по плану. Рефакторинг не исключение. Встраивание в оценки разумного буфера позволит уложиться в запланированные сроки даже в случае ошибок и инцидентов.

Масштабный рефакторинг обычно затрагивает несколько команд. Поэтому вероятность пересечения с проектом другой команды довольно велика. Запас времени позволяет проще ориентироваться в таких ситуациях и спокойнее себя чувствовать на переговорах. Увеличиваются шансы придумать творческие решения для выхода из тупика. При приостановке работы над текущим этапом будет проще переключить внимание на другую стадию рефакторинга, чтобы вернуться к этой работе позже.

Во-вторых, оценки помогают понять ожидания заинтересованных сторон (менеджеров по продукту, директоров, технических директоров) и команд, которые могут пострадать от рефакторинга. В следующем разделе мы узнаем, как обсуждать с ними планы на рефакторинг.

Помните, что лучше указать большее предполагаемое время для рефакторинга, чем сумму времени на выполнение его отдельных частей. Если в фирме нет строгих требований к оценке программных проектов, ни одно правило не предполагает точное совпадение ожидаемой даты завершения проекта с завершением его отдельных компонентов.

Обсуждение планов с другими командами

Масштабный рефакторинг обычно затрагивает много групп с разной специализацией. Всю эту информацию можно найти в плане выполнения, а после подумать, кого сильнее затрагивает каждый из этапов. Лучше всего для этого подойдет метод мозгового штурма. В небольших компаниях лучше подумать, какой вклад в рефакторинг могут внести все остальные отделы. Если в компании есть группа, ответственная за техническое проектирование, проект рефакторинга можно отдать туда для тщательного анализа инженерами разных специальностей. Так вы получите много полезной информации еще до первой организационной встречи.

Есть две основные причины поделиться планом выполнения рефакторинга с другими командами. Первая и самая важная — обеспечение информационной открытости. Вторая — будет полезно ознакомиться с мнением других сотрудников компаний, прежде чем добиваться согласия руководства на рефакторинг.

Информационная открытость

Информационная открытость укрепляет доверие между командами. Если другие рабочие группы узнают о ваших планах, вероятнее всего, они смогут помочь с их реализацией. Очевидно и обратное. Если без предупреждения начать рефакторинг, затрагивающий другие рабочие группы, отношения с сотрудниками могут испортиться.

Помните, что предлагаемые в рамках рефакторинга изменения могут резко поменять код, с которым работают другие группы, или повлиять на важные процессы, которые они поддерживают. Например, при миграции на новую версию Python в Smart DNA три репозитория превращаются в один. Это весомое изменение для любого, кто работает с каждым из трех репозиториев. Всех этих людей нужно предупредить о грядущих изменениях.

Рефакторинг влияет и на производительность других команд. Если будет принято решение объединить все пакеты в один большой файл `requirements.txt`, другим командам придется согласовывать и утверждать изменения. Иногда хорошим решением будет привлечь инженеров из других команд для помощи с рефакторингом. Подробнее мы поговорим об этом в главе 6.

Обязательно убедитесь, что ваши планы согласованы с другими группами. Например, если нужно изменить код другой команды, а они в это время планируют приступить к разработке основного функционала (или провести собственный рефакторинг), нужно будет согласовать свои действия, чтобы не путаться друг у друга под ногами.

Взгляд со стороны

Вторая причина поделиться планом рефакторинга — узнать их мнение на этот счет. Вы потратили усилия, чтобы определить проблему и составить комплексный план ее решения. Но поддержат ли этот план те, кого могут затронуть возможные преобразования? Если они считают, что преимущества рефакторинга меньше рисков и неудобств от него, планы придется пересмотреть. Можно, конечно, попытаться убедительнее рассказать о плюсах или найти способ снизить уровень риска. Обсудите ситуацию с коллегами и попытайтесь выяснить, как вы можете смягчить для них неудобства от рефакторинга. Для этого пригодятся приемы из следующей главы.

При рефакторинге сложного продукта почти всегда остаются неучтенные пограничные случаи. Здесь может помочь привлечение второй (и третьей, и четвертой) пары глаз. Представим, что сотрудники фирмы Smart DNA на одной из машин обновляли файл `requirements.txt` вручную, а не вносили изменения в историю версий, и развертывали новый код. Это невозможно обнаружить при ревизии списка пакетов. Только при обсуждении плана рефакторинга с сотрудниками можно узнать, что они обновляют зависимости на самой машине, поэтому проверять текущую версию нужно там, а не в репозитории. Знание этих подробностей позволяет избежать больших затруднений, которые возникли бы, начинись рефакторинг без предварительной консультации.

Но нужно понимать, что даже скорректированный с учетом мнений заинтересованных сторон план — не окончательный. Он будет меняться на протяжении всего рефакторинга. Вы столкнетесь с парой неожиданных пограничных случаев, возможно, потратите на устранение надоедливой ошибки больше времени, чем ожидалось, или поймете, что часть первоначального подхода просто не работает. Мы собираем разные точки зрения, чтобы обеспечить информационную открытость с другими командами и на ранней стадии устраниć явные проблемы. В главе 7 я расскажу, как поддерживать заинтересованность коллег и информировать их о стадиях реализации плана.

ИЗБЕГАЙТЕ РАСШИРЕНИЯ ОБЛАСТИ ОПРЕДЕЛЕНИЯ ПРОЕКТА

Идеи и наблюдения других команд помогают доработать план. Но важно концентрироваться на конечной цели, чтобы случайно не ввести дополнительные варианты. Возможно, в план нужно добавить несколько небольших новых этапов, чтобы корректно обработать один-два граничных случая, которые мы пропустили. Тем не менее важно добавлять только вещи, необходимые для достижения желаемого конечного состояния при условии сохранения основных этапов.

Состорожностью относитесь к таким предложениям коллег, как: «Пока мы этим занимаемся, можно было бы...» или «Мне всегда хотелось, чтобы X заодно обрабатывал...». Если вы не умеете отказывать, то можете согласиться сделать больше, чем запланировано. Конечно, хотелось бы, чтобы после рефакторинга получилось решить как можно больше проблем и угодить множеству инженеров. К сожалению, такой подход к масштабному рефакторингу почти гарантирует неустойчивый результат. Всегда будет появляться следующая проблема, требующая решения, или следующий инженер со своими пожеланиями. Нам же нужно стремиться к планированию и проведению рефакторинга, реализуемого в разумные сроки. Конечно, он не устранит всех ошибок, но, по крайней мере, будет исправлено *то, что нужно*.

Уточненный план

Вот план, который получился у разработчиков, обеспечивавших в Smart DNA переход с Python 2.6 на 2.7, после выполнения всех процедур из этой главы (определения целевого состояния и основных этапов выполнения, выбора стратегии развертывания и т. д.).

- Создать единый файл `requirements.txt`.
 - **Метрика:** количество списков зависимостей; **начальное значение:** 3; **цель:** 1.
 - **Оценка:** 2–3 недели.
 - **Промежуточные задачи:**
 - сформировать список всех пакетов в каждом из репозиториев;
 - провести ревизию всех пакетов и оставить в списке только действительно необходимые с соответствующими версиями;
 - определить, с какой версии каждый пакет будет обновляться до Python 2.7.
- Слиять все репозитории в один.
 - **Метрика:** количество репозиториев; **начальное значение:** 3; **цель:** 1.
 - **Оценка:** 2–3 недели.
 - **Промежуточные задачи:**
 - создать новый репозиторий;
 - добавить каждый из существующих репозиториев в новый с помощью команды `git submodule`.
- Создать образ Docker со всеми необходимыми пакетами.
 - **Метрика:** число сред, где используется новый образ Docker; **начальное значение:** 0; **цель:** 5.
 - **Оценка:** 1–2 недели.
 - **Промежуточная задача:**
 - протестировать образ Docker в каждой из сред.
- Включить линтинг в процесс непрерывной интеграции для монорепозитория.
 - **Метрика:** количество предупреждений линтера; **начальное значение:** примерно 15 000; **цель:** 0.

- **Оценка:** 1–1,5 месяца.
- **Промежуточные задачи:**
 - выбрать линтер и его конфигурацию;
 - добавить линтер в процесс непрерывной интеграции;
 - выявить с помощью линтера логические проблемы кода (неопределенные переменные, синтаксические ошибки и прочее).
- Во всех средах установить и развернуть Python 2.7.1.
 - **Метрика:** количество задач, работающих на Python 2.7.1 с новым файлом `requirements.txt`; **начальное значение:** 0; **цель:** 158.
 - **Оценка:** 2–2,5 месяца.
 - **Промежуточные задачи:**
 - определить, где находятся тесты для репозитория, выбрать из них надежные;
 - использовать Python 2.7 для подмножества сценариев с низким риском;
 - развернуть Python 2.7 для всех сценариев.



Если для управления проектами используется программное обеспечение (например, Trello или JIRA), создайте для крупных этапов записи верхнего уровня. Детали рефакторинга при разработке могут меняться, но вряд ли сильно изменятся стратегические этапы из этой главы.

Записи для подзадач обычно создаются для первого или второго этапа. Дело в том, что при разработке обычно появляются дополнительные мелкие задачи, а на более поздние этапы скорее повлияет предыдущая работа, и специфика их отдельных подзадач может измениться. Поэтому записи для подзадач следующих этапов лучше создавать только после начала этих этапов.

Итак, предварительная подготовка для понимания и всесторонней характеристики рефакторинга завершена. Мы составили план, который, надеюсь, благополучно приведет нас к финишу. Теперь нужно заручиться поддержкой руководства (и других заинтересованных сторон). Без этого невозможно двигаться вперед.

ГЛАВА 5

Получение одобрения

Учась в средней школе, я решила, что мне нужен мобильный телефон. Почти у каждого из моих друзей он был, но им надоело делиться телефоном со мной каждый раз, когда мне нужно было сказать родителям, где я. Одно текстовое сообщение стоило около 10 центов, а каждый звонок расходовал драгоценные минуты, поэтому несколько месяцев я раздавала друзьям монетки. Куда бы я ни пошла, приходилось таскать с собой полный карман мелочи в надежде одолжить чей-нибудь телефон.

Но мои родители считали, что мобильный мне не нужен, и убедить их в обратном было непросто. Фраза «у всех он уже есть» не работала. Им были нужны веские и обоснованные аргументы. Я постаралась их собрать. Во-первых, я мотивировала необходимость телефона соображениями безопасности. Я совсем недавно получила водительские права, и в случае какой-нибудь непредвиденной ситуации мне бы очень пригодилась возможность кому-нибудь позвонить. Чтобы придать этому аргументу больше веса, я подсчитала, сколько примерно часов в неделю я провожу за рулем. Затем я сравнила расходы на аппарат и тарифный план с суммой, розданной друзьям за последние полгода. Тем более недавно я начала подрабатывать созданием веб-сайтов и знала, что могу позволить себе купить простой раскладной телефон и ежемесячно оплачивать мобильную связь.

Сначала родители ответили, что я могу пользоваться телефоном матери. Тогда я сказала, что трачу от трех до четырех часов в неделю, возя своего младшего брата, и они все же согласились, что мобильный телефон не роскошь. Я доказала, что удобство, которое он дает, перевешивает его стоимость. Несколько дней спустя я получила свой первый мобильный телефон.

Этот опыт мне очень пригодился, когда на работе мне нужно было убедить других в преимуществах рефакторинга. Одна из самых частых жалоб от

коллег-инженеров состоит в том, что они хотят улучшить положение дел, но просто не знают, как получить на это разрешение. Можно потратить время на определение обстоятельств возникновения проблемы, найти доказательства и данные, характеризующие ситуацию, тщательно разработать план действий и столкнуться со скептицизмом, представив этот план руководству или менеджеру службы технической поддержки.

Я начну эту главу с объяснения причин такого отношения со стороны руководства. Понимание их позиции поможет в поиске более убедительных аргументов. Далее мы рассмотрим приемы для получения поддержки руководящего состава и конкретные стратегии, позволяющие вести людей за собой. В завершение вы узнаете, какие формы может принимать участие других людей в проекте и как это влияет на план и собранную для его реализации команду.

Причины несогласия руководителей

Колебаться (или быть категорически против большого рефакторинга) руководство может по нескольким причинам. Во-первых, руководители обычно находятся далеко от кода и вряд ли глубоко понимают его болевые точки. Во-вторых, их работа оценивается по способности возглавляемой команды вовремя выпускать новый эффективный функционал. В-третьих, самые плохие последствия крупного рефакторинга обычно вызывают гораздо больше опасений, чем плохие последствия от нового функционала. Наконец, крупномасштабный рефакторинг обычно требует координации с заинтересованными сторонами за пределами рабочей группы.

Руководители не пишут код

Руководители инженерных групп редко занимаются программированием и почти не участвуют в проверке кода. Человек, нанятый на руководящую должность, может даже не увидеть кода, над которым работают его подчиненные. Естественно, он не очень хорошо знаком с проблемами при разработке. Поэтому не удивляйтесь скептическому отношению к идее рефакторинга. Представьте, что вы пытаетесь объяснить приглашенному на обед гостю, почему нужно заменить все разболтавшиеся дверные ручки в доме. Человек сможет понять ваше недовольство, но он и представить себе не может, до какой степени ими неудобно пользоваться.

Поэтому, возможно, руководитель группы и понимает трудности, от которых призван избавить рефакторинг, но ему сложно понять, почему этим нужно заниматься прямо сейчас. И если эти проблемы появились давно, компания, скорее всего, научилась с ними нормально жить. Несложно понять, что он выберет: возможность создания чего-то нового или решение давно существующих проблем.

Работа руководителя оценивается иначе

Руководителей обычно оценивают по способности их команды уложиться в сроки и помочь в достижении бизнес-целей. Часто это подразумевает создание функционала, помогающего удерживать и привлекать пользователей или открывающего новые каналы поступления доходов. Исходя из этого у руководителей, вероятнее всего, будет в приоритете работа, требующая небольших усилий, но обеспечивающая высокую отдачу. И для этой работы они будут устанавливать более жесткие сроки в надежде быстрее донести изменения до пользователей.

При этом инженеры-разработчики предпочитают проекты, в которых можно решить интересные проблемы, а создание более надежного решения считают важнее его быстрой отправки пользователям. Конечно, этот шаблон описывает далеко не *всех*. Но опыт показывает, что таких специалистов довольно много. Каким бы значимым и стоящим для команды не был масштабный рефакторинг, в списке возможных проектов руководителя группы он будет на последнем месте. Это очень длительный процесс, который невидим для пользователей, а значит, не оказывает на бизнес немедленного положительного влияния. Если руководитель группы хочет получить повышение (или обеспокоен предстоящей аттестацией), вряд ли поддержит план рефакторинга.



Даже если ваш руководитель тоже считает проведение рефакторинга разумным, дав на него добро, он может подставить под удар свою репутацию. Ведь точно так же, как он сам оценивается по способности команды выполнять работу в срок, его руководитель оценивается по влиянию, которое его организационные усилия могут оказать на бизнес. Вашему руководителю может быть сложно убедить *своего* в том, что рефакторинг — это ценное вложение времени и трудовых ресурсов.

Руководители понимают риски

Есть несколько причин неудачи при разработке функционала. Рабочая группа может столкнуться с препятствиями, мешающими выпустить продукт в срок. Или в переданном пользователем функционале обнаружится много досадных ошибок. Но вероятность катастрофического сбоя при его разработке невысока, потому что у нового функционала обычно довольно хорошо очерченные границы.

При крупномасштабном рефакторинге ставки *намного* выше. Есть риск вызвать регрессию на большой площади, и вероятность катастрофического сбоя довольно высока. Распутывая старый и непонятный код, команда может обнаружить неожиданные ошибки. Попытка их исправить приведет к дополнительным сложностям, которые увеличат сроки завершения рефакторинга. Обо всех этих рисках мы говорили в первой главе. И все они очевидны для вашего руководителя.

Необходимость координировать усилия

В большинстве компаний для работы над отдельными частями продукта (или продуктов) создаются отдельные рабочие группы. Представим приложение для потоковой передачи музыки RadTunes. Одна группа разработчиков отвечает за создание списка воспроизведения, а вторая — за управление поиском. После решения добавить новый функционал каждая группа работает со своей частью кодовой базы. Было бы странно, если бы команда, отвечающая за поиск, начала разрабатывать функционал, позволяющий создавать совместные списки песен.

Теперь представим, что первая команда испытывает сложности с объектной моделью музыкальных композиций. В план по ее улучшению входит изменение кода, с которым регулярно работает почти каждая команда в компании. Поэтому автор предложения и руководитель группы сначала должны будут обсудить свои действия с каждой командой, а затем согласовывать их на протяжении всего рефакторинга. Именно этот колossalный объем работы видит ваш руководитель, когда вы подходите к нему с планом рефакторинга. Естественно, ему непросто решиться на такое.

КАК МОЖЕТ ВЫГЛЯДЕТЬ ОДОБРЕНИЕ

Прежде чем мы перейдем к стратегиям убеждения руководства, нужно понять, как на практике выглядит одобрение. Проект может быть целиком одобрен, целиком отвергнут или что-то среднее. Часто это как раз третий вариант. Решение о принятии большого рефакторинга обычно основано на двух вопросах.

- *Подходящее ли сейчас время для рефакторинга?* Крупномасштабный рефакторинг — это длительный процесс. Поэтому нужно быть уверенными, что команде хватит времени на его завершение и что работа начнется в подходящий момент. Важно учитывать как состояние текущих проектов, так и проекты, которые компания (в том числе и группа, которая будет проводить рефакторинг) планировала реализовать в дальнейшем.

Перед подачей проекта руководству выберите момент начала рефакторинга и обоснуйте, почему это оптимальное время. Например, потому что проблемы, которые стремится решить рефакторинг, еще не достигли критической точки, и это дает время на реализацию идеального решения. Или потому что выполненный сейчас рефакторинг может помочь команде в будущих проектах. Такие соображения будут полезны при поиске поддержки со стороны руководства.

- *Сколько ресурсов (чаще всего это трудовые ресурсы) на это нужно?* На рефакторинг тратится много трудовых ресурсов. И заручившись поддержкой руководства, вы сможете сформировать идеальную команду. Но эту тему мы подробно рассмотрим в главе 6. Помните, что трудовые ресурсы, которые могут быть выделены на рефакторинг, напрямую зависят от времени его запуска, и наоборот.

В этой главе мы рассмотрим ситуацию, когда все аспекты рефакторинга руководство не приемлет, но частично заинтересовано в его выполнении. Здесь получить одобрение проекта поможет любой из описанных методов, позволяющий подтолкнуть руководителя в правильном направлении.

Поиск убедительной аргументации

Теперь мы знаем, почему руководство не хочет проводить рефакторинг. Разберем несколько полезных стратегий, помогающих развеять связанные с рефакторингом страхи и убедить в его целесообразности. В этом разделе предполагается, что у вас уже была предварительная беседа о проекте с вашим руководителем. Если это не так, воспользуйтесь советами из следующей врезки. Предварительный разговор важен по двум причинам. Во-первых, он помогает понять, что больше всего влияет на решение вашего начальника. Во-вторых, дает представление о том, какие аргументы (эмоциональные или логические) могут его убедить. Фактически он показывает контекст для выбора наиболее эффективных стратегий убеждения.

ПЕРВЫЙ РАЗГОВОР

Не нужно обрушивать на начальника всю собранную информацию. Для начала поинтересуйтесь его мнением. Это может быть простой вопрос: «Я думал о том, как X влияет на способность делать Y, и хотел бы знать, что вы думаете по этому поводу». Предоставляя возможность откровенно высказаться, вы показываете, как важно для вас мнение руководства.

Если начальник уже в курсе проблемы, можно сразу задать вопрос о поддержке рефакторинга. В случае неуверенного и ли отрицательного ответа попытайтесь выяснить, что именно беспокоит начальника больше всего.

Руководителю, не знакомому с проблемой, нужно дать базовый беспристрастный обзор. Хороший начальник постарается понять причины вашей обеспокоенности и задаст вопросы, позволяющие ему верно охарактеризовать ситуацию.

Во время этих переговоров важно внимательно слушать оппонента. Вместо анализа сообщаемой информации многие думают о том, что будут говорить дальше. Фиксируйте важные моменты разговора. Это можно делать как в уме, так и письменно. Например, я очень быстро забываю подробности бесед, поэтому записываю важные детали.

Попробуйте вместо возражения задать вопрос. Например, вместо фразы: «Это не сработает из-за X», спросите: «Вы думали о X?» или «Каковы ваши планы относительно X?» Этим вы покажете начальнику, что вам важна его точка зрения и вы готовы к ней прислушаться, а не настроены на доказательство своей правоты.



Такие переговоры лучше проводить лично или по видеоконференцсвязи, а не по электронной почте или в чате. Выражение лица и общий тон руководителя во время беседы помогут лучше оценить его отношение к рефакторингу.

После первого разговора можно попробовать разные методы убеждения. Я расскажу про четыре простых, но их куда больше. Для каждого руководителя нужно выбирать свою стратегию в зависимости от того, что его больше мотивирует (например, перспектива карьерного роста) и как сильно он выступает против рефакторинга (например, в целом начальник согласен с наличием проблемы, но не уверен, что ее нужно решать немедленно). В конечном счете, эффективнее всего использовать сочетание приемов, выбирая наиболее удобные, на ваш взгляд, и те, что окажут наибольшее влияние. Уверенность в себе и хорошая подготовка к разговору позволят получить вам искомую поддержку руководства.

Как убедить коллегу

Некоторые мои коллеги могут прийти к несговорчивым инженерам и за полчаса убедить их в своей точке зрения. К сожалению, я так не умею. Если вы тоже, не волнуйтесь! Есть несколько простых (и честных) разговорных приемов, помогающих достичь большей убедительности.

Сделайте комплимент их ходу мыслей

Редкие люди невосприимчивы к лести. Скорее всего, ваш начальник не входит в их число. Если в какой-то момент беседы он с вами соглашается, подчеркните это комплиментом. Например, он подтверждает, что рефакторинг принесет пользу, но было бы недурно провести повторную оценку ситуации через полгода. Чтобы вернуть фокус на плюсы рефакторинга, можно сказать: «Вы замечательно подчеркнули потенциальные преимущества рефакторинга. Видно, что вы хорошо разбираетесь в проблемах, с которыми мы сталкиваемся». Так вы напомните начальнику о выявленных преимуществах, подтолкнув к тому, чтобы считать их существенное недостатков.

Представьте контраргумент

Нужно не просто быть готовыми к любым возражениям от руководства, но и заранее обеспечить себя контраргументами. Возможно, это выглядит странно, но психологические исследования показали, что аргументированные ответы на возражения убедительнее, чем обычное изложение аргументов «за». Прямое представление контраргументов дает несколько преимуществ:

- контраргументы на возражения показывают, что вы серьезно обдумали негативные последствия рефакторинга, а это подчеркивает вдумчивый и тщательный подход к задаче;
- подтверждая опасения начальника, вы косвенно делаете комплимент его способности разобраться в тонкостях рефакторинга. Обнаружив, что его рассуждения корректно поняты, руководитель будет с большей готовностью выслушивать следующие идеи.

Итак, обратить возражения себе на пользу можно путем их тщательного опровержения. Вернемся к примеру с приложением RadTunes. Допустим, руководитель группы планирует в следующем квартале потратить много времени на создание совместных списков воспроизведения композиций.

А вместо этого ему предлагаю сначала переписать представления песен в приложении, а только потом приступить к разработке нового функционала.

Вы можете сказать: «Понятно, что, если сейчас начать рефакторинг, работу над совместными списками воспроизведения придется отложить на несколько месяцев. Конечно, пользователи приложения, которые запрашивали этот функционал несколько лет, будут недовольны». И сразу же выдвинуть контраргумент: «Но я уверен, что, поменяв реализацию объектной модели песен сейчас, мы сократим процесс разработки совместных списков воспроизведения на несколько недель и поможем отвечающей за поиск команде реализовать сортировку результатов по жанрам».

Можно контраргументировать даже еще не высказанные руководством возражения. Такое поведение выглядит неэффективно, но оно повышает доверие к вам и укрепляет вашу позицию.



Эту книгу редко можно найти на полке программиста, но я настоятельно рекомендую приобрести томик Дейла Карнеги «Как завоевывать друзей и оказывать влияние на людей». Текст был написан более 80 лет назад, но большинство приведенных там советов актуальны и сегодня. Навыки, которым обучает Карнеги, будут полезны не только при поиске одобрения проектов, но и в других аспектах жизни!

Получение поддержки сверху и снизу

Конечно, не всех интересуют игры в офисную политику для отстаивания своих интересов, и это нормально. Таким читателям я советую сразу перейти к подразделу «Метрики» следующей главы. Тех же, кто хотел бы использовать организационный ландшафт для своей выгоды, я сейчас познакомлю с рычагами, помогающими получить от начальства добро на крупномасштабный рефакторинг. Для этого нужно заручиться поддержкой со стороны и товарищей по команде, и высшего руководства.

Важна именно поддержка обеих сторон. Если руководитель группы ощущает давление только от подчиненных, он без проблем может отказать, зная, что его действия не вызовут критики вышестоящих инстанций. Если же давление будет только сверху, а команда идею рефакторинга не поддержит (или, что еще хуже, выступает против), руководитель вряд ли будет продвигать проект, зная, что рискует подорвать моральный дух команды.

Имейте в виду, что такая стратегия влечет за собой неприятные последствия. Учитывая, что первый разговор на тему рефакторинга уже состоялся, ваш начальник знает о вашей заинтересованности. После обращения к нему на этот счет вышестоящего руководства есть вероятность, что начальник со-поставит эти два события и подумает, что вы хотели на него надавить. Это может вызвать негативную реакцию. Так что лучше начать с разговора со своим руководителем и узнать его взгляд на ситуацию. После этого не ждите, пока ваши союзники из высшего руководства обратятся к вашему начальнику напрямую, а постарайтесь организовать встречу всех трех сторон для дальнейшего обсуждения.

Объединяем товарищей по команде

Прежде чем искать поддержку у высшего руководства, заручитесь поддержкой товарищей по команде. Скорее всего, вы уже обсуждали рефакторинг с некоторыми из них на этапах сбора показателей для характеристики проблемы и составления плана выполнения, и их отношение уже известно. С остальными нужно поговорить, и желательно в неформальной обстановке. Например, можно пригласить их на кофе.

Вам нужно добиться их поддержки идеи рефакторинга в общем открытом обсуждении, где участвует и начальник (на собрании, в общем чате, в электронной переписке), или наедине с начальником. Возможно, надо будет обсудить последовательность разговоров, чтобы не получилось, что все члены рабочей группы подходят к начальнику с этим вопросом друг за другом. Хитрость в том, что эти разговоры не должны выглядеть как спланированная акция, нужна искренняя заинтересованность. Только тогда это будет полноценная поддержка снизу.

Прыжок через уровень

Если ваш руководитель не заинтересован в крупном рефакторинге, возможно, эта идея заинтересует его начальника. Руководство высшего уровня обычно более осведомлено о целях организации и о ее текущих и будущих проектах. Поэтому есть вероятность, что они воспримут идею большого рефакторинга более благосклонно, так как руководители выше уровнем могут лучше представить его преимущества.



Иногда иерархия в компаниях настолько строгая, что прямое обращение к вышестоящему руководству считается огромной бесактностью. Поэтому прежде, чем просить аудиенцию для обсуждения проекта, выясните, как воспримут этот маневр. И даже если такой прыжок через уровень допустим, будьте осторожны, и во время разговора не критикуйте своего начальника. Вместо этого сосредоточьтесь на попытках повысить интерес к проекту рефакторинга и добиться его согласования.

Если вы в хороших отношениях с руководителем более высокого уровня и есть основания полагать, что он поддержит ваш проект, назначьте с ним встречу. Первый разговор должен быть таким же, как и с вашим начальником. Важно понять, получите ли вы здесь поддержку предлагаемого рефакторинга. В случае отказа придется искать поддержку других влиятельных людей в компании. Если же вашу инициативу воспримут положительно, можно договариваться о второй встрече. На ней можно обсудить детали плана выполнения, согласовать требуемые ресурсы и выяснить, как этот начальник сможет вам помочь в получении одобрения вашего прямого руководителя.



Наличие прочных отношений с вышестоящим руководством может быть весьма полезным и в других ситуациях. Я настоятельно рекомендую по возможности проводить ежеквартальные (или даже ежемесячные) индивидуальные встречи с ним. Это ценный ресурс, позволяющий расширить вашу сферу деятельности. Если вы хотите для развития навыков поруководить эффективным проектом, вам смогут подобрать подходящий проект. Если вам нужен наставник, вас могут подключить к кому-нибудь из ведущих инженеров компании. Хорошие отношения с высшим руководством могут помочь справиться с трудностями при взаимодействии с руководителем группы, если они когда-то возникнут.

Другие отделы

В каждой компании есть несколько влиятельных отделов. Обычно именно их слово становится решающим, когда встает вопрос о способах разработки нового функционала, необходимости каких-то процессов или устраниении ошибок. Во многих отраслях (финансы, здравоохранение, человеческие ресурсы) это юридический отдел и отдел надзора за соблюдением нормативных требований. Люди, давно работающие в компании, скорее всего, знают, что это за отдел. Если вы не совсем уверены, спросите своих коллег. У них может быть парочка историй об участии отдела безопасности в инциденте или о вкладе отдела продаж в новую функцию.

Иногда (но не всегда) такие отделы могут быть заинтересованы в рефакторинге. Например, отдел надзора в уже знакомой вам компании Smart DNA, отвечающий прежде всего за безопасность секвенированных ДНК своих клиентов. Устаревшая версия Python для большинства систем компании может стать проблемой, ведь исправления безопасности уже невозможны. Если исследователи из Smart DNA выступают против масштабного обновления всех зависимостей до последней версии Python, группа разработчиков может связаться с отделом надзора за соблюдением нормативных требований и перечислить, какие уязвимости из-за этого возникают. После этого исследователи будут просто вынуждены отдать приоритет переходу на новую версию.

Привлечение на свою сторону авторитетных инженеров

В каждой компании есть авторитетные квалифицированные инженеры. Многие из них, если не большинство, продолжают работать с кодом. Если им знакома область, которую вы хотите улучшить, они не только сразу поймут, какие проблемы поможет решить рефакторинг, но и дадут ценные советы, которые можно будет внести в план. Их поддержка может стать решающей для получения разрешения прямого начальства. В некоторых компаниях нет более высокого одобрения, чем одобрение старшего инженера.

СТИМУЛИРОВАНИЕ РЕФАКТОРИНГА

Если у вас здоровые отношения с руководством среднего и высшего звена и вы хотите сделать рефакторинг и другие работы по обслуживанию программного обеспечения приоритетными в компании, попробуйте воспользоваться этими отношениями для создания системы, поощряющей эти вещи. Важно, чтобы никто из рядовых сотрудников не считал рефакторинг препятствием карьерному росту. Наоборот, соответствующим отделам нужно принимать меры, стимулирующие обслуживание кода. Ни один руководитель независимо от его положения не должен формировать культуру, где наличие технического долга считается нормой. В некоторых компаниях технический долг удалось сократить, когда сверху стали требовать включение технического обслуживания в ежеквартальные планы. Где-то даже зашли еще дальше: добавили качество кода в перечень факторов, влияющих на карьерный рост. В конце концов, управление — это не только обеспечение эффективной разработки нового функционала. Руководитель должен позаботиться и о проведении малопривлекательной работы по поддержке и улучшению существующего кода, чтобы он продолжал масштабироваться в соответствии с обстоятельствами.

Опора на доказательства

Если руководитель прислушивается к логическим аргументам, для укрепления своей позиции используйте доказательства, процесс сбора которых описан в главе 3. После первого разговора попросите о дополнительной встрече, сказав, что еще тщательнее все обдумали и подготовили количественные характеристики проблемы, чтобы лучше оценить ее масштаб (а возможно, и ее срочность).

Из множества собранных доказательств подготовьте к встрече два-три самых весомых. Некоторые вещи лучше воспринимаются визуально, поэтому для них имеет смысл составить график или диаграмму. Презентацию запишите на носитель, с которого она легко воспроизводится. В результате у вас появится исчерпывающий документ, который всегда можно предъявить другим заинтересованным лицам. Это очень пригодится, когда вы будете искать помощь других отделов или набирать людей в рабочую группу. И вам всегда будет на что сослаться во время обсуждений. Для тех, кто не обладает даром убеждения, четкий набор аргументов, которые можно предъявить оппоненту, часто имеет решающее значение.



Если вы не очень уверенный в себе человек и пока не обладаете полезными связями для воздействия на авторитетных коллег или своего начальника, советую в качестве основного аргумента использовать метрики. Это факты, которые легко подготовить, легко запомнить и с которыми очень трудно спорить.

Жесткие меры

Если вы точно уверены в важности рефакторинга для бизнеса, а ваш начальник продолжает вам препятствовать, подумайте о более серьезных вариантах воздействия. Учтите, что такие вещи ставят под угрозу ваши отношения с начальником и коллегами. Но если рефакторинг, которого вы добились таким путем, даст отличный результат, это поможет в продвижении по карьерной лестнице.

Конечно, не у всех есть ресурсы (карьерные или финансовые) на открытую борьбу с руководителем. И это нормально! Решить задачу так может только человек, обладающий авторитетом и неоднократно показывавший отличные результаты на нынешней должности.

Самое главное, что мне хотелось бы сказать, до того как мы перейдем к обсуждению деталей. Такие вещи работают, только когда вы готовы идти до конца. Если начальник продолжит стоять на своем и не увидит никакой реакции, это не только разрушит ваши отношения, но и не даст применить такой же подход, когда появится другой важный проект.

Прекращение технического обслуживания

Серьезная необходимость в крупномасштабном рефакторинге обычно указывает на активную закулисную работу по поддержке функционирования системы. Об этой работе начальство обычно не знает, а если и знает, то не осознает ее важности. Если вы регулярно находите способы смягчить проблему, которую призван решить рефакторинг, вы можете предупредить своего руководителя, что больше этого делать не будете. Идея в следующем: если перестать поддерживать систему на плаву, возможно, руководство увидит проблему, из-за которой нужен рефакторинг.

Для наглядности возьмем пример с миграцией на последнюю версию Python в фирме Smart DNA. При очередном обновлении безопасности команде приходилось тратить драгоценное время на его перенос на устаревшие системы. По сути, при обнаружении новой уязвимости команда приостанавливалась всю работу и бросала все силы на перенос исправления. Это трудоемкий и рискованный процесс, но без него было просто не обойтись. К сожалению, руководство не признавало производственные издержки, связанные с необходимостью поддерживать на плаву устаревшее программное обеспечение.

В этом случае на руководителя можно надавить, чтобы сделать обновление до последней версии Python приоритетной задачей, предложив группе не переносить все последующие обновления безопасности. Нужно сообщить ему, что, так как первоочередная задача команды — разработка нового функционала, вы можете прекратить поддержку устаревшего ПО, тем более что она не входит в должностные обязанности членов группы. Нужно подчеркнуть, что руководитель в ежеквартальных или годовых планах никак не учитывает работу, связанную с переносом новых обновлений безопасности на более старое ПО.

Да, это жесткая позиция. Возможно, вы даже будете себя винить за отказ обслуживания (которое, по мнению большинства разработчиков, является неотъемлемой частью их работы). И это совершенно нормально. Мне тоже приходилось испытывать по этому поводу чувство вины и страх, что такая

безответственность подводит компанию. Но потом я поняла, что, настаивая на своем, добиваюсь противоположного эффекта. Я показывала компании слабое место, которого никто не замечал, и его значение. Чтобы начальник пересмотрел приоритеты, я демонстрировала, сколько работы нужно для поддержания работоспособности системы. Работы, от которой может избавить рефакторинг.

Ультиматум

Если ничего не помогает, можно сказать начальнику, что, если он продолжит противодействовать рефакторингу, вы перейдете в другую команду или вообще уйдете из компании. Если вы хотели бы остаться в компании, подумайте о смене команды. Решите, куда вы хотите перейти, или попробуйте найти в компании руководителя, поддерживающего идею рефакторинга, и выяснить, как он смотрит на ваш переход под его крыло. Возможность легко перейти в другую команду придает уверенности при выдвижении ультиматума. Но перед таким обострением отношений лучше тщательно все обдумать и убедиться, что у вас есть для этого необходимая финансовая стабильность.

Этот разговор с начальником будет непростым. Начать его лучше с выражения обеспокоенности тем, что компания недостаточно серьезно относится к проблемам, о которых вы доложили. И если ваш начальник хочет сохранить вас в своей команде, ему следует обдумать ситуацию и одобрить рефакторинг.

Заинтересованность в рефакторинге

Несмотря на то что поиск поддержки происходит задолго до написания хоть одной строчки кода, этот этап рефакторинга может стать одним из сложнейших. Руководители часто не решаются соглашаться на такие длительные проекты, ведь у них есть свои ограничения и стимулы. Но любой из нас может изучить техники убеждения и попытаться показать, что ресурсы, затрачиваемые на рефакторинг, того стоят. Можно научиться эффективно опираться на товарищей по команде и на руководителей высшего звена, получая от них дополнительную поддержку.

Результатом всех этих усилий может стать разрешение на рефакторинг. Но возможен и отказ. Если ваш начальник все еще настроен скептически,

продолжайте собирать доказательства для рефакторинга и ждите подходящего момента. Например, после инцидента, вызванного проблемой, которую пытается решить рефакторинг, можно попробовать снова поговорить с начальником. Про рефакторинг можно напоминать и при составлении долгосрочных планов. Главное — хватайтесь за любую возможность показать рефакторинг в хорошем свете.

Если проект одобрен и вы заручились поддержкой руководства, используйте ее для сбора необходимых ресурсов. Определите, кто из инженеров нужен для успешного рефакторинга и когда потребуется их опыт. Все эти тонкости мы обсудим дальше.

ГЛАВА 6

Подбор команды

Триллер «Одннадцать друзей Оушена» — один из тех фильмов, которые нравятся всем. Он начинается с того, что Дэнни Оушен выходит из тюрьмы. Он встречается со своим бывшим подельником Расти Райаном, предлагая тому украсть 150 миллионов долларов из трех казино Лас-Вегаса: «Белладжио», «Мираж» и MGM Grand. Два вора понимают, что в одиночку им не справиться. Они начинают собирать команду из бывшего владельца казино, карманника, афериста, специалиста в сфере электроники и видеонаблюдения, эксперта-подрывника и акробата.

Команда делится на две группы. Первая изучает входы и выходы в казино «Белладжио», распорядок работы персонала и прочее. Вторая строит точную копию денежного хранилища, чтобы попрактиковаться в обходе сложной системы безопасности. Несколько дней группа вынашивает план. Дальше начинается веселье: все препятствия преодолеваются, и (внимание: спойлер!) команда в итоге сбегает с деньгами.

Оушен и Райан никогда не смогли бы ограбить «Белладжио» вдвоем. На финансовую подготовку к ограблению у них бы ушли месяцы. Да и вряд ли они смогли бы придумать план обхода систем безопасности казино. Собрав команду из людей с нужными навыками, преступники ускорили выполнение плана и увеличили свои шансы на успех.

Для крупного рефакторинга нам нужны свои одиннадцать друзей Оушена. За несколько месяцев в тюрьме Нью-Джерси Дэнни много раз представлял себе будущее ограбление. В процессе отшлифовки плана он составил список важных для успеха предприятия навыков и знаний и наметил возможных

кандидатов с этими способностями. В этой главе я расскажу вам, как собирать разные типы команд в зависимости от требуемых для эффективного рефакторинга знаний. Вы поймете, как технический лидер составляет список возможных партнеров по команде и убеждает их присоединиться. В конце мы поговорим о том, как извлечь максимальную пользу из неудачи, в случае которой проект придется реализовывать в одиночку.

Выбор экспертов

В главе 4 вы узнали, как составить эффективный план действий, схематично представив сложный процесс рефакторинга в виде нескольких этапов верхнего уровня с перечнем самых важных подзадач для каждого из них.

Многие из нас работают в команде. В этом случае план рефакторинга — совместная разработка, которая будет реализовываться силами всей команды. Но когда речь идет о большом рефакторинге, почти всегда нужна помочь других отделов. Хотя бывают случаи, когда подготовительная работа выполняется в одиночку либо с одним-двумя коллегами. Как бы то ни было, возьмем в руки план и попробуем наметить, какие специалисты нам нужны и когда.

Перечитаем план, на каждом этапе пытаясь представить код, с которым предстоит взаимодействовать. Легко ли это сделать? Можно уверенно сказать, какие изменения нужно внести и как они повлияют на систему? Осведомлены ли мы о возможных подводных камнях в этой области кодовой базы? Осознаем ли мы, какие последствия повлекут за собой наши изменения? Хорошо ли знакомы с технологиями, с которыми придется взаимодействовать? Если вы ответили «да» на все эти вопросы — отлично! Это значит, что у нас есть ресурс, позволяющий внести запланированные изменения самостоятельно. В противном случае будет нужна чья-то помощь. Сторонних помощников можно разделить на две категории: активные участники и профильные специалисты.

Активные участники с первого дня плотно задействованы в проекте и направне с другими пишут код. Их вклад принимается во внимание на ранней стадии реализации проекта и при каждой его редакции.

Профильные специалисты (Subject matter experts, SME) не принимают участия в активной работе над проектом. Они обсуждают детали решений,

отвечают на вопросы и могут провести обзор кода. Хотя их вклад может быть очень значимым, на проект они тратят минимум времени, так как в основном занимаются собственными проектами.

Например, представим, что команда мониторинга переходит с одного агрегатора метрик на другой (например, с StatsD на Prometheus). Они создали инфраструктуру, подготовили несколько узлов и готовы принимать трафик от некоего приложения. Для помощи с переходом команде нужен разработчик, который хорошо знает, как это приложение использует пакет StatsD. Для обеспечения взаимодействия с новым решением он должен написать внутреннюю библиотеку, в будущем способную заменить текущую. При этом нужно гарантировать, что функционал библиотеки для Prometheus ровно такой же, как и раньше, и что у этой библиотеки интуитивно понятный API. Разработчика просят дать практические рекомендации по работе с новой библиотекой и способствовать ее внедрению в организации.

Для профессионального выполнения своей задачи этому разработчику не нужно досконально разбираться в работе новой системы сбора показателей. Всегда можно обратиться за помощью и советом к команде мониторинга, а они в свою очередь могут обращаться с вопросами, если заметят что-то странное в процессе интеграции с приложением. Этот разработчик — активный участник, который сотрудничает с командой мониторинга.

В результате наблюдения за использованием библиотеки StatsD выявляется, что одна группа разработчиков продукта пользуется ею не так, как большинство других команд. Нужно понять причину и определить, стоит ли воспроизводить это поведение в новой системе. Если окажется, что оно необходимо, убедитесь, что пакет Prometheus дает возможность его приспособить. Для этого свяжитесь с людьми из той группы, чтобы задать им пару вопросов. После короткой беседы с одним из участников команды (допустим, Фрэнком), стало ясно, что новая библиотека Prometheus тоже поддерживает это поведение. Фрэнк соглашается проверять код по мере наращивания функциональности. В этом сценарии Фрэнк — профильный специалист. Для успешного выполнения рефакторинга нужны специалисты разного профиля.

В этом примере понадобились умение команды мониторинга работать с пакетами StatsD и Prometheus, осведомленность Фрэнка в разных сценариях использования продукта и собственные знания того, как кодовая база за-

действует библиотеки сбора метрик. Возможно, придется посоветоваться с кем-нибудь из группы безопасности для уверенности в том, что через новую систему не проходят никакие конфиденциальные данные клиентов (если это не так, у нас есть меры для быстрого решения этой проблемы).

При составлении списка необходимых специалистов нужно следить за его длиной. Крупный рефакторинг обычно затрагивает большую площадь, поэтому список может оказаться чрезмерным. Но чуть позже я расскажу, как его можно сократить.

Подбор специалистов

Итак, мы успешно составили список специалистов для выполнения рефакторинга. В рассмотренном случае нужен технический эксперт, эксперт по продукту и эксперт по информационной безопасности. Укажите в списке, на каком этапе выполнения понадобится каждый из них. Если кто-то необходим на нескольких этапах, отмечается момент его привлечения к проекту. Кроме того, помечается, какая роль (SME или активного участника) отводится каждому из сторонних специалистов. Но в процессе переговоров с кандидатами и обсуждения степени их участия в проекте эта пометка может измениться.

Затем для каждого пункта этого списка указывается одна или несколько подходящих кандидатур. Но если вы работаете в крупной компании и не знакомы с людьми из разных команд, это достаточно сложная задача. Поэтому для начала мы прикидываем, в каком отделе могут работать нужные специалисты, и при наличии доступа к актуальной структурной схеме организации пытаемся найти лучшую команду в этом отделе. Не бойтесь просить помощи у своего начальника. Это его обязанность — обеспечить своей команде все условия для эффективного выполнения проекта. Начальник, скорее всего, более осведомлен, какие команды в организации лучше подходят для оказания помощи.



Если доступа к актуальной структурной схеме организации нет, но практикуются дежурства инженеров и применяется платформа оповещений об инцидентах, как, например, PagerDuty, нужных экспертов можно поискать среди дежурных. Посмотрите, дежурным из какой команды уходят сообщения об инцидентах, связанных с интересующей вас функциональностью или компонентом инфраструктуры. Готово!

Пример списка, который мы разработали для перехода к другому пакету сбора метрик, дан в табл. 6.1.

Таблица 6.1. Список предметных областей и потенциальных экспертов

| Область знаний | Этап | Роль | Эксперт |
|--|------|-------------------|--|
| Знание, как код учета заказов использует StatsD (отличается от большей части другого функционала) | 1 | SME | Фрэнк, Макензи, группа обработки заказов |
| Автоматизированное сквозное тестирование от библиотеки к Prometheus | 2 | Активный участник | Джесс, группа автоматизированного тестирования |
| Слежение за трафиком от приложения к Prometheus, когда им начнут пользоваться | 3 | SME | Группа мониторинга |
| Как конвейер развертывания приложения повлияет на узлы Prometheus | 1 | SME | Джесс, команда выпуска и развертывания |
| Последствия для безопасности от сбора метрик, связанных с пользователями; клиенты, заботящиеся о безопасности, для которых метрики нужно собирать особенно внимательно | 1 | SME | Группа безопасности продукта |

Многопрофильные специалисты

Выберите из списка все имена, которые появляются больше одного раза. В таблице выше совпадений немного, но один кандидат подходит для двух этапов из пяти. В компании может быть несколько ведущих инженеров с широким спектром знаний. Консультации таких специалистов станут полезными во многих отношениях.

Во-первых, привлечение таких людей к проекту позволяет уменьшить общее число участников. Согласование действий команды крупного проекта — уже нелегкая задача, а теперь представьте, что нужно согласовывать действия разработчиков из разных команд. Каждый участник должен не только понять свою задачу, но и приспособиться к принятому в команде процессу разработки (например, к еженедельным или ежедневным встречам). На этот процесс может уйти много времени и усилий, прежде чем все начнут работать в нормальном темпе.

Во-вторых, у экспертов в разных аспектах проекта, скорее всего, есть четкое представление о том, как все это работает вместе. Такую ценную информацию почти невозможно получить от других инженеров компании. В табл. 6.1 к таким специалистам, скорее всего, относится Джесс. Как оказалось, он несколько месяцев тесно сотрудничал с командой выпуска и развертывания, помогая создавать процентную систему выпуска двух важных сервисов. Потом он перешел в другую команду, где работал над повышением доступности автоматизированных сред тестирования. Джесс — один из тех инженеров, кто успел поработать над обширным списком проектов и хорошо понимает принцип работы всех составляющих.

К сожалению, такие специалисты часто довольно загружены (возможно, потому что вносят вклад в ряд проектов в качестве SME и руководят своими проектами). Если они не могут регулярно помогать, но вы считаете, что их уникальные знания и опыт жизненно важны для рефакторинга, попросите их проанализировать план выполнения. Кроме того, они могут рекомендовать экспертов, способных принять в проекте активное участие.

Даже если ваш список получился очень длинным, а повторяющихся имен в нем почти нет, не беспокойтесь! Большой рефакторинг можно успешно выполнить с помощью небольшой группы энтузиастов.

Еще немного про активных участников

У меня есть хорошее эмпирическое правило: число активных участников лучше определять по размеру команды, с которой вам было удобно работать в прошлом. Если вы успешно работали в группе из шести человек, ограничьте команду шестью активными участниками. У всех нас разный опыт командной работы. Никто лучше вас не знает своих предпочтений. Поэтому воспроизведите наиболее эффективные для вас условия. Крупный рефакторинг достаточно сложен и с технологической, и с технической точек зрения. Так что не превращайте рабочую группу в еще один усложняющий фактор.

Если составленный список активных участников кажется вам слишком длинным, подумайте, есть ли области, помочь в которых могут предоставить SME. Это уменьшит усилия, затрачиваемые на координацию, так как эти специалисты просто оказывают разовые консультации. Стратегии эффективного взаимодействия с SME мы рассмотрим в следующей главе.

Необъективность при составлении списка

Если вы знакомы со специалистом, способным помочь с одним или несколькими пунктами списка, его можно напрямую попросить о помощи. Возможно, он будет рад вам помочь. Это самый простой вариант приглашения специалиста со стороны. Если у этого человека уже есть опыт сотрудничества с вами, он сможет довольно быстро влиться в общий ритм и рано начнет добиваться заметного прогресса.

Но у этого варианта есть и недостатки. Известно, что разработчики ПО плохо оценивают время и силы для выполнения задачи. Обычно это следствие свойственного им неугасающего оптимизма. Если просьба кажется необременительной, они соглашаются очень быстро, без должной оценки тяжести принимаемого на себя обязательства. Иногда только после начала проекта приходит осознание, что они согласились на слишком большую работу, а теперь изо всех сил пытаются со всем этим справиться. Я сама бывала в таких ситуациях, и уж поверьте моему опыту, сходу соглашаться на объемную работу так же нецелесообразно, как и сразу отвечать отказом.

У поиска помощи среди знакомых тоже есть минусы. Так, мы даже не рассматриваем других специалистов, которые могут лучше подходить для этой роли. Все мы страдаем от когнитивных искажений, с которыми желательно сознательно бороться. Одно из таких искажений — эффект недавнего. Это тенденция оценивать значение недавних событий выше, чем более ранних. Более вероятно, что мы укажем на коллегу как на хорошего возможного эксперта, если совсем недавно мы слышали его имя или разговаривали с ним. При составлении списка экспертов помните про этот эффект. Попытайтесь понять, действительно ли каждый человек из списка лучше всего подходит для выполнения задачи или он тут потому, что мы случайно увидели его имя в электронном письме пару дней назад. Для поиска квалифицированных кандидатов лучше всего связаться с руководителем рабочей группы. Он сможет познакомить с вашим предложением всех сразу и оценить вызванный интерес. Руководители знают, кто из команды сможет внести в рефакторинг значительный вклад и получит наибольшую выгоду в виде популярности и карьерного роста.

Еще при подборе кандидатур важно не путать опыт и стаж. Возможно, у Фрэнка не такой уж большой опыт работы в отрасли и даже в компании он работает недолго. Но за последние месяцы он активно участвовал в ряде проектов, и поэтому вы уверены, что он сможет ответить на ваши вопросы и сделает ценные выводы при проверке кода. Иногда самый опытный

специалист — не лучший выбор. Кроме того, такие разработчики часто заняты своими сложными проектами, и у них просто нет времени на помощь другим. Тогда как проект рефакторинга для кого-то может стать отличной возможностью продвижения и получения популярности за пределами своей команды. Несмотря на всю сложность рефакторинга, это не тот проект, в который не могут внести вклад инженеры с опытом работы всего год (или даже несколько месяцев).



Если вы решили, что какая-то команда может стать хорошим источником кандидатов, попробуйте напрямую поговорить с ее руководителем и попросить его порекомендовать подходящих специалистов. Это поможет минимизировать когнитивные искажения, мешающие процессу найма.

Типы команд, выполняющих рефакторинг

Мы уже достаточно обсудили процесс подбора специалистов в команду. Но ведь есть еще и команда, где работаете вы. Подходит ли она для рефакторинга? Чтобы добиться успеха в роли технического руководителя, нужно понимать, почему ваша команда лучше всего подходит для реализации проекта в организации, где вы все работаете. Обычно крупномасштабным рефакторингом занимаются команды трех типов.

Владельцы

Такие команды отвечают за определенную часть продукта и проводят рефакторинг кода, который они сами написали и поддерживают. Он взаимодействует с кодом других команд. И нужно решить, как быть с этими точками соприкосновения — редактировать самостоятельно или координировать свои действия с владельцами затрагиваемого кода.

Представим компанию с тремя основными отделами: один отвечает за продуктивность разработчиков, второй — за инфраструктуру и третий — за разработку продукта. Вы входите в группу, которая занимается тестированием библиотек и инструментов для приложения. Когда разработчики из разных отделов пишут больше модульных тестов, это здорово, но проблема в том, что постепенно время на выполнение тестов растет и начинает влиять на способность каждого быстро отправлять код. Ваш отдел начинает отслеживать время выполнения отдельных модульных тестов, чтобы понять, как быстро происходят определенные операции, такие как эмуляция окружения.

Результаты заставляют вашу команду начать рефакторинг для достижения более быстрой процедуры эмуляции. Проверка по эталонному тесту показывает резкое улучшение, но для использования новой логики нужно перенести существующие модульные тесты. Только так мы сможем получить выгоду от ускорения. Миграцию можно выполнить двумя основными способами.

Вариант 1: миграцию осуществляет одна команда

Можно перенести все тесты силами вашей команды. У этого подхода несколько явных преимуществ. Ваша команда лучше всех понимает, как перенести тест со старой процедуры эмуляции окружения на новую. Вы знаете, какие тесты подходят для легкой миграции и каких ловушек стоит избегать с помощью более сложных тестов. Вы представляете, как применять новую систему эмуляции окружения для получения максимального повышения производительности. Скорее всего, ваша команда сильнее других мотивирована на выполнение этой задачи. Как владельцы фреймворка тестирования вы можете отдать ей главный приоритет. Достаточно поставить в квартальный план цель сократить время выполнения полного набора тестов. Понимание того, что выполнение задачи будет оценено по достоинству, очень мотивирует (особенно ближе к концу квартала).

С другой стороны, нужно перенести тысячи тестов. Конечно, можно задействовать инструменты модификации кода для автоматического переноса некоторых простейших тестов, но это решит только малую часть задачи. Даже если все остальные тесты поровну распределить между членами команды, понадобятся недели ручной рутинной работы. Кроме того, команда плохо знакома с кодом, который проверяет каждый тест. Как бы ни хотелось предположить, что тесты рассматривают текущую эмуляцию окружения как черный ящик, мы не всегда можем предсказать, как сильно они могут быть связаны с поведением существующей реализации. Велика вероятность, что для корректной работы с эмуляцией окружения нам в итоге понадобится узнать контекст, в котором происходит каждый из тестов.

Вариант 2: каждый обновляет собственные тесты

Суть этого варианта в том, что команды разработчиков сами переносят тесты, относящиеся к написанному ими функционалу. В этом случае вашей команде уже не придется переносить тысячи тестов в одиночку. В случае распределения задачи по всей организации шанс быстрее ощутить положительное влияние миграции увеличивается. Это избавляет вашу команду

от необходимости самим разбираться в принципах работы более сложных тестов. Когда каждый обновляет собственные тесты, сохранять их поведение намного проще. Помимо этого, участвующие в этой работе команды получают прекрасную возможность критически взглянуть на текущее тестовое покрытие и, возможно, улучшить его.

У этого подхода есть и недостатки. На первый план выходит исходное качество (и своевременное обновление) документации, описывающей процедуру обновления тестов, которую должна подготовить ваша группа. Инженеры, которые переносят тесты, будут обращаться с вопросами и просьбами проанализировать код именно к вам. Даже подробный и доступный список ответов на часто задаваемые вопросы не избавит вас от необходимости много раз повторять одно и то же.

И как бы убедительно вы ни объясняли другим командам, что улучшение производительности из-за новой системы стоит затраченных на миграцию усилий, согласятся далеко не все. Будут и команды, которые, согласившись на миграцию, не смогут ее завершить, ведь в приоритете для них будет создание нового функционала. Призывая другие команды к участию в рефакторинге, помните, что даже если все они согласны с тем, что проект принесет ощутимую пользу, при появлении более приоритетных задач он будет первым отодвигаться в сторону. Конечно, если сторонние команды не добавят его по вашему примеру в ежеквартальный план.

Поиск баланса

Как видите, нет идеального варианта. И сделанный выбор влияет как на способность вашей команды достигать краткосрочных и долгосрочных целей, так и на отношения с другими командами. Поэтому я советую по возможности смешивать две стратегии для минимизации недостатков этих двух подходов и увеличения шансов на успешное завершение рефакторинга. В следующем разделе я напишу план для ситуации с переносом тестов.

Рекомендуемый подход

1. Выберите несколько простых тестов, которые могут получить наибольшую пользу от миграции. Спросите у других команд, какие, по их мнению, лучше подходят для этого.
2. Начните перенос тестов вручную силами своей команды, тщательно документируя процесс (если известны владельцы тестов, сообщите им

о переносе или привлеките их к работе для совместного завершения миграции).

3. Запустите эталонные тесты для перенесенных файлов, чтобы зафиксировать изменение производительности. Документируйте результаты тестирования.
4. Разработайте инструмент для автоматического переноса нескольких простых случаев. Запускайте его на небольших логических подмножествах набора тестов, пока не будут перенесены все доступные тесты.
5. Начните привлекать другие команды. Продвигайте новую эмуляцию окружения, подчеркивая ее преимущества и показывая примеры тестов после миграции. Выделите время для ответов на вопросы и личного участия в выявлении и устраниении неполадок. Подумайте над тем, чтобы организовать для разработчиков из других групп регулярную возможность присоединяться к вашей команде и совместно осуществлять миграцию нескольких тестов.
6. Побуждайте другие команды устанавливать ежеквартальные цели по повышению производительности тестов. Когда участие в проекте гарантированно будет оценено, шансы на выполнение переноса тестов повысятся.

Группы очистки

В некоторых крупных инженерных организациях есть специальные группы для повышения производительности труда разработчиков. Диапазон их задач довольно широк. Это предоставление сред разработки и управление ими, написание расширений для редактора и сценариев автоматизации повторяющихся задач, создание инструментов, помогающих лучше понять влияние предлагаемых изменений на производительность, поддержка и расширение базовых библиотек, необходимых всем разработчикам продукта, и прочее. Чаще всего члены этой команды берут на себя обязанности по очистке кода.

Команды по очистке берут на себя важную (но часто неблагодарную) работу по поиску и удалению мусора и антишаблонов и созданию вместо них более эффективных, более устойчивых шаблонов. Сюда обычно входят разработчики, заботящиеся о состоянии кода, которые хотят, чтобы их коллеги могли легко разрабатывать, тестировать и поставлять новый функционал. Им приятно видеть, как другие используют (и ценят) их библиотеки и инструменты.

За масштабный рефакторинг группы очистки берутся по двум причинам. Во-первых, им нет равных по уровню знаний кодовой базы. Как авторы основных

библиотек, они обычно имеют хоть какое-то отношение почти к каждому аспекту приложения. Это особенно верно для команд, работающих с монолитными базами кода. Во-вторых, такие команды ценят разработку эргономичных решений, доступных каждому, независимо от команды или стажа. У них есть полученное за счет опыта понимание того, какие интерфейсы обеспечивают правильный баланс между расширяемостью и практичностью. Для проектов, нацеленных на повышение производительности труда разработчиков (и сохранения ее уровня), это идеальная команда. Третья, неявная, причина в том, что, отдав рефакторинг на откуп группе очистки, команды разработки продукта могут спокойно сосредоточиться на разработке функционала.

К сожалению, группы очистки не вечны. При их продуктивной работе остальные команды обычно не считают себя обязанными выполнять обслуживание кода. Со временем накапливается объем работы, неподъемный для группы очистки, и люди начинают оттуда уходить. В итоге группа распадается или возникает текучка кадров. А команды, долго уклоняющиеся от технического обслуживания кода, постепенно теряют этот навык. И привлекать их к еще одному крупномасштабному рефакторингу просто бесполезно.

УДАРНАЯ ГРУППА

Ударная группа (*tiger team*) — это команда технических специалистов, отобранных для достижения определенной цели (как в фильме «Одннадцать друзей Оушена»!). Этот термин впервые появился в 1964 году в статье *Program Management in Design and Development*, где такие группы предлагались как эффективный метод повышения надежности авиационных и космических систем. Особую известность приобрела ударная группа, сформированная после аварии пилотируемого космического корабля Аполлон-13. Ее целью было возвращение астронавтов на Землю. Позже за проявленное мужество и исключительно высокопрофессиональную работу группа получила высшую гражданскую награду США — медаль Свободы.

В кризисной ситуации (например, при внезапном и продолжительном сбое, серьезных проблемах с производительностью, резком снижении надежности) руководство может попросить опытных специалистов отказаться от выполнения текущих задач и бросить все силы на решение проблемы. Обычно ударные группы работают в условиях нехватки времени. Например, им ставится условие «нужно все исправить до того, как произойдет X». Поэтому они недолговечны. Основное внимание здесь уделяется мозговому штурму и разработке минимально жизнеспособного решения. Поэтому крупномасштабным рефакторингом ударные группы не занимаются, но бывают и исключения. Если вы сможете убедительно доказать руководству, что рефакторинг позволит устранить проблему, мешающую бизнесу, и за время, оставшееся до обострения проблемы такие работы выполнить нереально, возможно, лучшим решением будет создание ударной группы.

Предложение

Итак, мы знаем, какие отношения сложились у нашей команды с проектом рефакторинга, определили необходимый нам опыт и обдумали список экспертов, которых хотим нанять. Теперь перед нами самая сложная задача: убедить их помочь нам. В отличие от героя фильма мы не можем предложить одиннадцатую часть от 150 миллионов долларов, украденных из казино. Но можно попытаться доказать, что участие в рефакторинге стоит их времени и сил. К каждому человеку нужен свой подход, поэтому я кратко опишу некоторые методы убеждения.

Не бойтесь применять к одному человеку несколько тактик. Самым занятым или самым скептически настроенным экспертам нужно больше одной причины, чтобы согласиться на сотрудничество, и в этом нет ничего удивительного! Ведь этот человек должен будет выделить часть своего драгоценного времени и энергии на ваш проект. Часто рефакторинг сопряжен со значительными рисками. То есть можно оказаться причастным к аварийной ситуации. Если рефакторинг затянется, придется упустить другие возможности. Словом, участие в проекте масштабного рефакторинга сопряжено с рисками. При разговоре не нужно пытаться их минимизировать, лучше попробуйте убедить эксперта, что преимущества участия в проекте гораздо значимее.

Наконец, не забывайте о настойчивости. Если разговор со всеми специалистами из списка не дал результата, вернитесь к его началу и попробуйте снова. Во-первых, у этих людей было время обдумать ваше предложение, во-вторых, возможно, в процессе собеседований у вас появились новые козыри и аргументы.

Обращение с просьбой к разработчику отличается от обращения к руководителю команды. Разработчики сами сталкиваются с болевыми точками, которые пытается решить ваш рефакторинг. И, по моему опыту, убеждать их в реальности проблемы приходится очень редко. При разговоре с разработчиками вы, скорее всего, сможете успешно использовать нижеприведенные техники подачи информации.

Руководители тоже ощущают проблемы кода, но косвенно. Они замечают, что при планировании спринтов увеличивается оценка времени выполнения задач из-за увеличения сложности кода. Кроме того, подчиненные могут лично выражать недовольство частыми инцидентами из-за нестабильного, плохо протестированного кода. Но у начальства обычно нет стимула ставить

рефакторинг на первое место. Это происходит из-за того, что продуктивность команды оценивается по частоте выпуска инновационных продуктов. Высшему руководству трудно объяснить необходимость потратить квартал-два на улучшение кода, за который отвечает команда, чтобы потом разработка проходила быстрее. Поэтому руководители команд не вступают в борьбу за рефакторинг до обострения необходимости очистки кода. При разговоре с ними я рекомендую больше опираться на метрики и предложения обмена.



Мотивировать руководителя, чтобы он начал уделять первоочередное внимание работоспособности и качеству производимого командой кода, можно через определение измеримых целей, связанных с рефакторингом. Убедить высшее руководство учитывать это при оценке производительности команд сложно, но, если получится, это сильно повлияет на процесс создания и поддержания ПО в организации.

Метрики

В главе 3 вы узнали про способы количественной оценки текущего состояния приложения. В следующей главе с помощью этих метрик оценивалась успешность следования плану выполнения. Еще они могут стать убедительным аргументом в разговоре со специалистами, помочь которых нам бы пригодилась.

Презентации на базе метрик лучше всего подходят для скептически настроенных экспертов и тех, кто в своей работе привык ориентироваться на данные. Это инженеры, которые всегда задают вопросы. Они активно отслеживают 95-й процентиль API, поддерживаемого их командой. Они первыми замечают рост среднего числа операций с определенным сегментом базы данных. Предоставьте им метрики, и, скорее всего, вы сможете получить специалиста в свою команду.

Обязательно сформулируйте, почему выбранные показатели стали индикаторами проблем. Опишите, какие проблемы призван решить рефакторинг и как будет проходить их количественная оценка. Поделитесь собранной вами исходной статистикой. Начать лучше с простых метрик, позднее дополнив их подтверждающими данными. Если есть иллюстративные материалы, укажите на них ссылку. Пояснительные графики и диаграммы оценит любой сотрудник.

Сопоставьте начальные показатели с желаемым результатом и проинформируйте о том, в каком темпе планируется вести работу, дав этим понять, что, несмотря на масштабность поставленной цели, она вполне достижима.

Великодушие

Есть странный психологический эффект – эффект Бенджамина Франклина: у вас больше шансов понравиться другому человеку, когда вы просите его об одолжении, а не наоборот. Это объясняется тем, что люди помогают только тем, кто им симпатичен. По крайней мере, так обосновывает ситуацию наш мозг, чтобы поддержать логическую согласованность между действиями и восприятием.

Инженер, плотно работающий с кодом, который вы хотите улучшить, с большей вероятностью поймет болевые точки этого кода. И, скорее всего, он знает несколько своих коллег, регулярно сталкивающихся с этими же проблемами. Если он следит за состоянием кодовой базы, вероятно, он проявит к вам большее сочувствие, и вы сможете удачно возвратить к его внутреннему альтруисту.

Спросите этого эксперта, на что жалуются его товарищи по команде. Запомните или запишите конкретные болевые точки, которые позволит убрать рефакторинг. После жалоб на сложности работы с кодом в его нынешнем состоянии по очереди расскажите про решение, которое предлагается для каждой проблемы из списка. Возможно, для каких-то еще нет четкого решения, и это нормально! Именно поэтому вы просите у эксперта помощи. Дайте ему понять, что именно его идеи помогут проекту. Не забудьте подчеркнуть, что его вклад может хотя бы немного облегчить жизнь его коллег и увеличить продуктивность их работы. Расскажите об ожидаемых выгодах от рефакторинга, кратко описав показатели успеха (многоплановая презентация даст вам больше шансов).

Возможности

Если эксперта интересует карьерный рост или он хочет поднять свою популярность в организации, то участие в крупном проекте добавит ему в резюме отличную строчку. Я уже упоминала, что руководители групп часто могут подсказать, кто из подчиненных достаточно квалифицирован для

участия в вашем проекте и хочет выделиться в других отделах. Обязательно поговорите со всеми, кого вам назовут, и узнайте, чего им не хватает для перехода на следующую ступеньку карьерной лестницы.

Иногда руководитель и его подчиненный имеют одинаковое представление о том, какие вещи нужно делать и в каких проектах участвовать для карьерного роста последнего. Но это не всегда так. Чтобы убедить инженера присоединиться к вашему проекту, добейтесь совпадения его ожиданий с ожиданиями его начальника. Подумайте, на каких этапах рефакторинга этот инженер может показать нужные для продвижения качества. И при личной встрече подчеркните, какой вклад он может внести в проект и как это поможет в его карьерных устремлениях. Ведите открытый диалог и будьте готовы к любым комментариям. Вы разные люди, а значит, его взгляд на то, что нужно для достижения успеха, может отличаться от вашего.

Обмен

Здесь нужно быть готовым к бартеру. Это отличный способ получить ресурсы для успешного завершения проекта, дав взамен некоторые обязательства. Обычно о таком обмене договариваются не исполнители, а руководство с обеих сторон. Знать заранее, что у вас попросят взамен, невозможно. Важно найти то, что ценно для второй стороны. Вот несколько примеров.

- Допустим, в вашей команде много людей, а команда, из которой вы хотите набрать экспертов, отчаянно нуждается в них. Если там, где вы работаете, это допустимо и ваша команда может себе позволить отказаться от некоторых сотрудников, можно предложить необходимое число людей в обмен на одного или двух инженеров, которые будут активно участвовать в рефакторинге.
- Если ваши команды вместе поддерживают определенный функционал, можно взять на себя поддержку компонентов, от которых вторая команда хотела бы отказаться. К сожалению, часто совместная поддержка заканчивается тем, что функционал остается без поддержки вообще. В обмен на помощь в рефакторинге ваша команда может согласиться полностью поддерживать функционал или его компоненты некоторое время (несколько кварталов или год).

- Если в вашей организации практикуются общие обязанности (отработка определенного количества часов на поддержке клиентов или участие в собеседованиях), можно предложить свою команду для выполнения в течение определенного времени их части (или всех) вместо привлеченных к рефакторингу экспертов. В идеале такой обмен начинается только по завершении рефакторинга, потому что отнятое у него время только затянет этот процесс в ущерб всем участникам.



Бартер между двумя инженерами обычно заключается в обмене экспертизами в предметной области. То есть эксперт, которого вы нанимаете как SME, хочет, чтобы вы также поучаствовали в текущем или будущем проекте с его стороны. Я видела, как предметом бартера становятся обзоры кода, дополнительные дежурства на телефоне или согласие документировать определенное количество postmortem-анализов.

Помните, что любая из сторон может не выполнить своего обещания, если за время рефакторинга приоритеты поменяются. Кроме того, любая реорганизация компании может сделать соглашения недействительными из-за смены руководства или ответственных за функционал. Уход из команды руководителя или инженеров тоже может повлиять на любые заранее согласованные договоренности. Чем дальше рефакторинг, тем больше шансов, что соглашение сорвется.

Повторные попытки

Если убедить эксперта, имя которого стоит в списке первым, не удалось, ничего страшного! Я не напрасно рекомендовала выбирать несколько кандидатур. Кроме того, у специалиста, который отказался от участия в рефакторинге, можно спросить, кого бы он порекомендовал вместо себя. Это позволит получить пару дополнительных кандидатур.

Если навыки отказавшего вам специалиста нужны не сразу, можно повторно попытаться его уговорить, когда проект дойдет до нужного этапа. Увидев достаточный прогресс и положительные сдвиги начальных показателей, эксперт может поменять свое мнение. Рефакторинг похож на снежный ком, катящийся вниз по склону. Чем дальше, тем больше поверхности он захватывает, собирая все больше ресурсов.

Возможные результаты

Если звезды сойдутся, вам удастся убедить всех выбранных кандидатов и собрать команду мечты. К сожалению, так бывает редко. Но даже с теми ресурсами, которые получилось собрать, можно найти способ провести качественный рефакторинг! Сейчас я расскажу вам, как выглядит реалистичный сценарий и как извлечь из него максимальную пользу. Еще мы кратко обсудим, что делать в худшем случае, когда приходится действовать в одиночку.

Реалистичный сценарий

Скорее всего, вы окажетесь с небольшой группой преданных делу экспертов и товарищ по команде. В небольших компаниях в процессе активного роста сотрудникам приходится совмещать разные обязанности, и работы у всех невпроворот. Поэтому вероятность найти экспертов для всех областей знаний из списка мала. В более крупных и стабильных компаниях получить от людей из других команд помошь бывает сложно из-за организационных границ и приоритетов. То, что человек разбирается в вещах, которые нужны для успешного рефакторинга, не означает, что это мероприятие станет главным приоритетом этого эксперта или его руководителей.

Неважно, сколько людей вы смогли привлечь, если удалось собрать команду из специалистов, необходимых хотя бы на ранних этапах проекта. В конце концов, вполне возможно начать рефакторинг с одной командой, а закончить с другой. Тем более что навыки и опыт, необходимые на первых этапах, в дальнейшем могут не понадобиться. Дополнительных специалистов можно привлечь в группу позже, когда уже будет заметен ощутимый прогресс и преимущества, которые дает рефакторинг.

Что делать в худшем случае

К сожалению, возможен и плохой сценарий — вам вообще не удается получить помошь. Здесь нужно понять, как извлечь максимальную пользу из сложившейся ситуации. Но прежде чем кидаться на выполнение рефакторинга в одиночку, подумайте, не разумнее ли вообще отказаться от него. Если ваше предложение не убедило ни руководство, ни опытных разработчиков, может,

стоит вернуться к началу и поискать более сильные аргументы в пользу рефакторинга. Или признать, что сейчас не самый подходящий момент для реализации этого проекта.

Если ваш начальник, товарищи по команде и некоторые коллеги разделяют мнение о важности рефакторинга, но на него просто нет ресурсов, подумайте о выполнении проекта в одиночку. Но имейте в виду, что это непростой путь. Прежде всего, вы будете чувствовать себя изолированно. Медленный темп создает впечатление отсутствия прогресса. Обсуждать проект с коллегами крайне сложно, так как каждый раз, когда хочется получить взгляд со стороны или совет, вам придется рассказывать контекст.

Но будут и положительные моменты. Например, вам ни с кем не придется согласовывать свои действия. Только вы знаете последовательность выполнения, и никто со стороны не внесет в них путаницу. Но придется тщательно за всем следить и публиковать информацию о состоянии проекта, чтобы коллеги, которые заинтересованы в нем, но пока не могут внести свой вклад, знали о происходящем.

Конечно, внесение значительных изменений в кодовую базу почти неизбежно будет сопровождаться какими-то инцидентами. Хотя postmortem-отчет по проекту, за который отвечал всего один человек, и не должен быть безупречным, у вас может возникнуть чувство, что бремя ответственности и необходимость исправить ситуацию лежит только на вас.



Я настоятельно рекомендую ознакомиться с тем, что пишет по поводу postmortem-процессов вице-президент по техническим операциям в Etsy Джон Аллспо (https://oreil.ly/DFSh_). Разработанный им подход к реагированию на инциденты способствует целенаправленному развитию внутри организации, заботясь о психологической безопасности отдельных инженеров.

Я советую вам подружиться с тем, кто тоже единолично отвечает за значительный проект. Вы будете отчитываться друг другу и поддерживать мотивацию. Можно установить регулярный график встреч для обсуждения достигнутого прогресса. Это даст возможность совместно искать решения сложных проблем и выполнять обзор кода. Наличие компаньона абсолютно необходимо, чтобы не сбиться с пути.

Создание сильных команд

Для создания эффективной команды нужен очень важный навык — умение общаться и передавать информацию. Именно он позволяет убедить нужных специалистов присоединиться к группе рефакторинга и четко сформулировать, чего от них ждут. Так активные участники и SME оказываются хорошо осведомленными о своей роли в проекте и осознают, смогут ли они оправдать ожидания.

Общаться и взаимодействовать важно на протяжении всего процесса, особенно на этапе внесения изменений в кодовую базу. В следующей главе я расскажу вам о методах установления и поддержания свободного потока информации между группой рефакторинга и теми, кого затрагивают вносимые изменения.

ЧАСТЬ III

Выполнение

ГЛАВА 7

Коммуникация

Моя подруга Элиза недавно начала строить дом. Несколько месяцев она участвовала во всех этапах строительства: координировала действия водопроводчиков, электриков, плотников, укладчиков плитки и бесчисленных бригад, которые приезжали к ней. Все эти профессионалы работали в сплоченных командах и вносили свою лепту в строительство будущего дома.

Иногда некоторые из друзей Элизы, в том числе и я, интересовались, как продвигается стройка. В ответ она рассказывала целые саги. Например, я до сих пор помню, как мне показывали множество фотографий образцов плитки для ванной и в подробностях описывали, сколько звонков пришлось сделать, когда оказалось, что большую часть доставили треснутой. Чуть позже Элиза вспомнила, что не познакомила меня с планами на вторую ванную комнату, и на меня свалился новый ворох историй.

Хотя мне нравилось слушать, как продвигается строительство, нелинейность повествования и избыток деталей сильно утомляли. В итоге Элизе посоветовали завести блог, где она могла отмечать происходящее с фотографиями и многочисленными деталями. Блог, который все интересующиеся могли посмотреть на досуге. И мы нашли для себя приемлемый способ следить за строительством.

В итоге, общаясь со строительными бригадами, Элиза применяла прямой подход, акцентируя внимание на деталях, а в блоге дописывала общую картину. По тому же принципу общаются в большом проекте рефакторинга. Коммуницировать можно двумя способами: один внутри своей команды (как у Элизы со строительной бригадой), второй с внешними заинтересованными сторонами (как у Элизы с друзьями). В этой главе вы узнаете, какие методы позволяют держать обе группы в курсе и поддерживать их

согласованность. Я расскажу, какие привычки нужно выработать у команды, и познакомлю с тактиками создания продуктивных групп. Отдельно мы поговорим, как держать в курсе людей, не занимающихся проектом. Обсудим мы и стратегии взаимодействия с чрезмерно и с недостаточно активными заинтересованными сторонами.

В этой главе я расскажу, как выработать у команды сильные коммуникативные навыки. Возможно, в вашей фирме уже практикуются хорошо налаженные методы координации, отслеживания и отчетности для крупных программных проектов. У вашего начальника, у ответственного за выпуск новых продуктов и у технического руководителя программы также могут быть свои представления о том, как лучше мотивировать команду. Прислушайтесь к их мнению, прочтите эту главу и выберите то, что с большей вероятностью подойдет всем. Надеюсь, к концу этой главы у вас будет новый набор инструментов для следующего большого рефакторинга.

Внутри команды

Хорошо, если общение в вашей команде происходит легко и часто. В этом случае в течение обычного рабочего дня коллеги часто обмениваются информацией. Члены команды занимаются парным программированием, анализируют код друг друга и вместе выполняют отладку. Ежедневно проводятся стендапы, а еженедельно — собрания для синхронизации. Многие не задумываются, как общаться в рамках этих взаимодействий. Все воспринимается как рутинная часть работы. Но если сделать некоторые из взаимодействий более осознанными, это поможет поддерживать долгосрочные, технически сложные проекты (например, рефакторинг).

Чтобы команда двигалась вперед без недопонимания, с самого начала введите несколько коммуникативных практик. Некоторые будут знакомы тем, кто хотя бы немного практиковал гибкую методологию разработки. Мы рассмотрим как частые практики (ежедневные или еженедельные), так и редкие (раз в месяц или в квартал), важные для объективного взгляда на достигнутые и запланированные результаты.

Возможно, в вашей команде общаются специфично. В течение рабочего дня члены команды, скорее всего, болтают, занимаются парным программированием, рецензируют чужой код и вместе выполняют отладку. Но бывают более структурированные варианты общения, гарантирующие

нужный рабочий ритм. Я опишу некоторые из них и расскажу, чем они могут быть полезны.

ВАЖНОСТЬ РЕГУЛЯРНОЙ КОММУНИКАЦИИ

Если коммуникативные навыки в вашей команде слабые, возможно, крупный рефакторинг стоит отложить. Такая ситуация указывает на наличие проблем. Быть может, между членами команды есть противоречия или они предпочитают говорить, а не слушать. В таких случаях успешно выполнить важный проект просто невозможно. Есть много ресурсов для улучшения коммуникации в команде, но я рекомендую книгу Маршалла Розенберга (*Marshall Rosenberg*) *Nonviolent Communication* («Ненасильственное общение»).

Если у вас молодая команда, потратьте время на формирование навыков коллективной работы. В таких проектах важно, чтобы участники чувствовали себя комфортно друг с другом. И на то есть причины. Во-первых, рано или поздно любой может ошибиться. И лучше, чтобы команда поддерживала и помогала быстро исправить оплошность, а не тратила время на критику. Во-вторых, нужно научиться решать сложные задачи сообща. Умение контролировать свое эго помогает быстрее прийти к хорошему решению. Ведь так каждый из членов команды думает о достижении хорошего результата, а не о продвижении своей идеи. В-третьих, большой рефакторинг может затянуться. И все это время коллегам нужно будет давать честную обратную связь. С людьми, которые раздражают, никто не захочет работать. Поверьте, я совершила такую ошибку.



По возможности откажитесь от ноутбуков и телефонов во время встреч. В идеале ноутбук может быть только у тех, кто активно участвует в обсуждении, делая заметки или делясь информацией. Если кто-то из группы на момент встречи дежурит на телефоне или активно участвует в устраниении инцидента, понятно, что без ноутбука ему не обойтись. Все остальные должны придерживаться этой политики. Я обнаружила, что гораздо лучше сохраняю концентрацию во время встреч, когда у меня нет компьютера. Это помогает внимательнее слушать, предлагать лучшие идеи и чаще покидать встречу с чувством, что она прошла продуктивно. Поэтому советую вам ввести такую политику хотя бы для пары встреч. Вы удивитесь, но они станут продуктивнее, а иногда и заканчиваются раньше!

Стендапы

Стендапы — отличное средство синхронизации всей команды. Они помогают обновлять статус задач в инструменте планирования проектов. А еще это прекрасная возможность вспомнить, что произошло за прошедшие сутки,

подумать, был ли достигнут достаточный прогресс или нужно обратиться за помощью к товарищам.

У каждой команды свой подход к стендалам. Некоторые выбирают личные встречи, где можно рассказать, какой прогресс достигнут за прошедший день. Требование каждый день в определенное время присутствовать на стендале дает точку привязки, вокруг которой члены команды могут планировать свою работу.

При работе над крупными программными проектами важно регулярно подводить итоги и отмечать прогресс, даже небольшой. Иногда общий масштаб работ кажется неподъемным, и сосредоточение на постепенных шагах вперед помогает ощутить достижимость результата. Кроме того, организованные ежедневные встречи позволяют контактировать лично. Какими бы скучными ни казались стендалы, если большинство участников в течение рабочего дня программируют самостоятельно, они могут стать одним из немногих видов личного общения.



Личное общение здесь — это любой вид очных взаимодействий. Они могут происходить как в одном офисе, так и по видеосвязи. Важно, чтобы каждый нашел время выслушать других, не отвлекаясь на посторонние дела.

Есть команды, предлагающие асинхронный способ взаимодействия. Они публикуют сводки предыдущего рабочего дня в мессенджерах (Slack или Discord). К сожалению, личная встреча предполагает, что все участники доступны ежедневно в одно время. Для сильно распределенных команд это очень неудобно, а иногда и почти невыполнимо (когда члены команды находятся в разных часовых поясах). Кроме того, не всегда с руки прерывать работу для участия в стендале. В результате такого разбиения может уменьшиться время, выделенное на решение текущей задачи.

Эффективный рефакторинг кода требует сильной концентрации. Нужно понять, что делает текущая реализация (читая код или запуская модульные тесты), затем выбрать подходящий способ ее улучшения и создать модернизированную реализацию, которая ведет себя как исходная. Большинству программистов нужно несколько часов сосредоточенной работы, чтобы добиться ощутимого прогресса в решении сложной задачи. Если начатую в девять утра работу в 10:30 прервут для участия в стендале, вряд ли можно будет говорить о значительном прогрессе.



Для лучшей имитации стендапа участники асинхронных встреч должны предоставлять актуальную информацию о выполненной работе к определенному времени. Например, по будням до 10:30. Ранние пташки могут сразу же отчитаться о проделанной работе и с головой погрузиться в следующую задачу. Если к 10:30 кто-то из команды не отправил отчет, он получает напоминание от руководства.

В случае большого рефакторинга стендапы можно проводить ежедневно, а можно со временем изменить их частоту. К примеру, если на каком-то этапе появляются параллельные рабочие потоки, ежедневные общие отчеты станут не нужны. Обновления, которые носят сугубо технический и детальный характер, большинство сотрудников не смогут оценить без знания контекста. В таком случае вместо ежедневных стендапов лучше предоставлять более подробный отчет о проделанной работе два раза в неделю или во время еженедельной синхронизации.

Еженедельная синхронизация

Цель стендапа — быстро показать прогресс каждого. Но это не универсальное средство регулярного общения с командой. Представьте получасовой стендап. Если члены группы тратят это время на обсуждение стоящих перед ними задач и возникающих в процессе их решения проблем, возможны два варианта. Во-первых, можно попросить продолжить обсуждение после стендапа. Если в затянувшемся разговоре не существует значительная часть команды, это должно сработать. Во-вторых, можно добавить еженедельную синхронизацию. Это позволит команде углубиться в самые важные для нее темы.

Если рефакторинг затрагивает очень большую поверхность, к нему обычно привлекают специалистов со всей организации. В кросс-функциональной команде не все участники посвящают 100 % своего времени рефакторингу, поэтому вместо стендапов лучше проводить еженедельные синхронизации. На таких коротких встречах члены команды могут обсудить только те обновления, которые относятся к рефакторингу.

Я бы рекомендовала выделить на это мероприятие около часа. Еженедельную синхронизацию можно структурировать как стендап, но с небольшими изменениями. В первой половине встречи все по очереди рассказывают о своих достижениях на прошлой неделе. Если прогресс меньше ожидаемого, нужно выяснить почему: из-за трудностей в работе или потому, что в центре вни-

мания была деятельность, не связанная с рефакторингом. Команде важно знать, что мешает выполнению проекта и над чем работает каждый. Если для продвижения нужно перераспределить рабочие обязанности, это легко обнаруживается и сразу исполняется. Важно отмечать и все темы, которые члены команды начинают подробно обсуждать.

Вторую половину встречи обычно посвящают важным темам, накопленным за неделю. В эту категорию попадает обнаружение во время тестирования нового пограничного случая. Вероятно, об этом будет упомянуто на ближайшем стендале. Но во время еженедельной синхронизации лучше обсудить ситуацию подробнее, дав команде возможность улучшить процедуру развертывания для гарантии корректной обработки таких пограничных случаев.

Обсуждаться могут и доклады о достигнутых успехах. Например, если кто-то создал прототип автоматизации часто повторяющихся фрагментов рефакторинга, остальным полезно узнать об этом и о том, как они могут его использовать. Но обязательно соблюдайте правила делового этикета и позволяйте всем поделиться своими мыслями.

Сильные команды создаются благодаря прочным связям, а прочные связи возникают на почве конструктивного личного общения. Еженедельная синхронизация — идеальная площадка для укрепления отношений с товарищами по команде. Почему так важно построить сильную команду? Поддержка членов команды особенно важна, когда дела идут плохо. Например, если внесенное кем-то изменение вызывает серьезную регрессию. Зная, что за твоей спиной команда и коллеги всегда готовы помочь устраниТЬ последствия, можно справиться с беспокойством и избежать выгорания. Возможность открытого общения и доверия друг другу очень важна, когда работа начинает затягиваться.

Крупный рефакторинг обычно включает в себя этапы, состоящие из утомительных и повторяющихся операций (даже в примерах из этой книги есть по паре длинных монотонных шагов). Ничего сложного или интересного во время таких этапов не происходит. Это скучная, но важная работа, в процессе которой обычно кажется, что темп выполнения проекта сильно замедлился. У многих может наблюдаться тенденция к выгоранию. Поэтому огромное значение имеют люди, на которых можно опереться, поделившись с ними чувством безысходности. Если кому-то в команде сложно найти энергию для движения вперед, ему просто обязан кто-то помочь.



Во время еженедельной синхронизации обязательно документируйте все, что обсуждалось (и любые выводы команды). В сочетании с задачами в своем ПО для управления проектами эти заметки пригодятся, когда нужно будет кратко описать все, что было сделано командой.

Еженедельную синхронизацию можно как совместить со стендапами, так и использовать вместо них. Мой опыт показывает, что даже при ежедневных встречах она невероятно полезна, так как позволяет каждому глубже погрузиться в обсуждение важнейших тем недели. Я особенно рекомендую еженедельную синхронизацию командам, предпочитающим асинхронные стендапы, — это обеспечит всем регулярное личное общение. Попробуйте разные варианты стендапов (асинхронные или личные, ежедневно или через день) вместе с еженедельной синхронизацией и посмотрите, какое сочетание лучше подходит для вашей команды.

Ретроспективы

Для команд, выполняющих крупномасштабный рефакторинг, ретроспективы так же полезны, как и для групп, практикующих гибкие методологии разработки. С их помощью можно обдумать последние итерации, выделить варианты их улучшения и определить любые дальнейшие действия. Время, выделенное для обсуждения того, что прошло хорошо, что можно было бы сделать лучше и что планируется изменить, способствует росту как команды, так и ее отдельных участников.

Большинство команд, практикующих гибкие методологии разработки, регулярно проводят ретроспективы. Команды, ориентированные на выпуск продукта, обычно проводят ретроспективу после запуска нового функционала или после определенного количества циклов разработки. Команды, работающие над долгосрочными проектами, могут проводить ее раз в месяц или квартал. Команды, проводящие крупный рефакторинг, обычно больше всего выигрывают от ретроспективы после основных этапов. Обычно они достаточно длинные, чтобы накопить нужное количество материала для рассмотрения, но не настолько длинные, чтобы команде было сложно запомнить все со времени последней ретроспективы. Иногда провести ретроспективу можно даже для подзадачи какого-либо этапа. Словом, универсального ответа на вопрос о времени ее проведения нет. Если вам кажется, что в какой-то точке нужна ретроспектива, просто обсудите это с командой. И если остал-

ные согласятся, запланируйте ее. В противном случае подождите завершения следующего большого этапа работ.

Если вы не знаете, как провести хорошую ретроспективу, поищите информацию на общедоступных ресурсах. Их сейчас великое множество. На сайте Atlassian (<https://oreil.ly/kgz9y>) немало статей и сообщений в блогах посвящено передовым практикам и оригинальным идеям, позволяющим оживить ретроспективы.

Вне команды

Как и в случае с любым крупномасштабным программным проектом, многим людям вне вашей команды будет интересен ваш прогресс. Это и высшее руководство, и разработчики из других команд, которых затронул рефакторинг, и старшие технические руководители. Высшему руководству нужно видеть, что процесс продвигается как надо и дает ожидаемые результаты. С его точки зрения, слишком большие затраты на рефакторинг приводят к неэффективной трате средств. Вместо разработки времени тратится на переписывание уже существующих функций. И при отклонении от запланированного графика временные и финансовые затраты только увеличиваются. В главе 5 упоминалась и такая вещь, как упущенная выгода в случае, когда руководитель выбирает рефакторинг вместо продолжения разработки функционала.

Высшему руководству нужны регулярные подтверждения правильности инвестиций в рефакторинг. Если в какой-то момент начинает казатьсяся, что это не так, скорее всего, поступит распоряжение приостановить либо полностью прекратить его. Чтобы гарантировать продолжение поддержки со стороны высшего руководства, нужно правильно подавать информацию о продвижении процесса.

Отслеживают все этапы проекта и руководители, и сотрудники затрагиваемых рефакторингом команд. Им важно видеть, насколько велик риск от вносимых изменений. Они хотят точно знать, когда планируется внедрение обновлений и сколько оно продлится. А вот старшие технические руководители внимательно следят за происходящим, чтобы в случае неудачи помочь с возвращением проекта в правильное русло. Обычно именно они отвечают за формирование технического видения компании и за обеспечение успеха сложных проектов, включая любой крупный рефакторинг.

В этом разделе мы обсудим, как правильно донести до внешних заинтересованных лиц информацию о последних достижениях в рефакторинге. Сначала я расскажу, что можно сделать заранее, чтобы сразу сформировать правильные навыки. Затем вы узнаете, как коммуницировать с внешним миром на протяжении всего проекта.

При запуске проекта

Перед началом рефакторинга определитесь с тем, как вы будете общаться с внешними заинтересованными сторонами. Выбор стратегии на раннем этапе поможет сэкономить драгоценное время при согласовании действий с коллегами из других команд и снижает вероятность недопонимания.

Единый источник информации

Даже в самых маленьких компаниях часто применяется несколько инструментов для решения одного набора задач. Например, можно пользоваться одновременно пакетами GSuite и Office 365. Причем у разных отделов могут быть разные предпочтения, и даже внутри одного отдела может сложиться ситуация, когда документация разбросана по GSuite, GitHub и внутренней электронной энциклопедии. Сведения об особенностях продукта или о текущем проекте очень тяжело искать на множестве платформ. Еще неприятнее, когда сведения в разных источниках не совпадают.

Поэтому первым делом выберите для хранения всей связанной с рефакторингом документации платформу, которая нравится команде. Вам придется регулярно создавать новые документы и обновлять существующие, поэтому выбираемое решение должно содержать все необходимые дополнительные функции. Если каждый раз, добавляя информацию, вы раздражаетесь из-за неудобства, есть шанс, что вы просто не станете это делать, и документация останется без обновления.

На выбранной платформе создайте каталог для хранения всех важных документов. Это будет единственный источник информации. Документация может содержать технические спецификации проекта, план выполнения из главы 4, заметки о встречах, postmortem-отчеты и прочее. После этого всем, кто ищет документацию по рефакторингу, можно давать ссылку на каталог или конкретный документ. Если вы предпочитаете использовать приложение Notion, но многие из ваших коллег привыкли к поиску технической

документации на GitHub, можно создать запись для своей документации в GitHub, связав ее с Notion. Это не только позволит легко искать нужные документы, но и гарантирует отсутствие устаревших копий.

На протяжении всего рефакторинга важно проверять, что новые документы сохраняются именно в каталог проекта, а внешние ссылки в других широко используемых источниках документов обновляются.

Договоренности о частоте информирования

Пока идет рефакторинг, заинтересованные лица будут регулярно связываться с вами для получения новой информации. К сожалению, когда их много, на ответы уходит много времени. Только представьте, каждый раз, когда у кого-то из высшего руководства возникает вопрос о продвижении рефакторинга, вы или ваш руководитель получаете электронное письмо, на которое нужно отреагировать. При этом есть и те, кого вы хотели бы держать в курсе, но они не присылают запросов. Словом, команда должна взять распространение информации на себя. Но постоянная необходимость брать на себя инициативу в таком деле раздражает, особенно когда принимающая сторона никак не дает понять, что ознакомилась со сведениями.

Поэтому нужно заранее выделить время и подумать, как вы будете информировать о выполнении работ, и заранее договориться с заинтересованными сторонами, где и как часто они будут получать сведения. При отступлениях от установленного регламента (например, вы получаете от кого-то сообщение о неполученной информации) достаточно отправить в ответ напоминание, где находится то, что им нужно.

Обязательно составьте примерный план коммуникации со следующими сведениями.

- *Где найти информацию о текущем этапе рефакторинга.* Есть несколько способов сделать эту информацию легкодоступной для всех желающих. Если команда использует Slack, можно создать канал для обсуждения связанных с рефакторингом вопросов и поместить в тему канала краткое описание текущего этапа проекта. В конце недели публикуется сводное сообщение с подробным описанием достижений за последние несколько дней (если вы проводите еженедельные синхронизации, черновик этого сообщения можно сразу же привязать к заметкам совещания). Если команда использует систему управления задачами JIRA, поместите

ссылку на доске проекта. Для тех, кто хочет получать регулярные ознакомительные обновления, добавьте поле summary.

- *Где найти общий график проекта.* График сроков и последовательности выполнения можно добавить в корневой каталог документации по проекту, в план коммуникации или в план выполнения. Главное, не забывать обновлять этот график в случае изменения дат.
- *Где найти техническую информацию о рефакторинге.* Дайте ссылку на каталог, где команда собирается сохранять связанную с рефакторингом документацию. Кратко опишите, что за документы там будут храниться.
- *Где можно задавать вопросы.* Случается, кто-то не может найти информацию в предоставленных ресурсах или хочет задать вопрос, а не заниматься самостоятельными поисками. В мессенджере Discord таких лиц можно направлять на канал проекта или настроить отдельный канал для вопросов. При общении по электронной почте можно настроить группу для всех членов команды и просить адресовать вопросы туда.
- *Как информируются команды, затронутые рефакторингом.* Общение с командами, на работу которых может повлиять рефакторинг, должно быть максимально прозрачным для гарантии того, что деятельность вашей группы не станет сюрпризом. Составьте список нормативов, которые вы будете соблюдать при взаимодействии с чужим кодом. Это может быть отметка одного или нескольких человек из команды как сигнал, что нужно проверить внесенные в код изменения, или по необходимости посещение стендалпов этой команды для предоставления информации об обновлениях.

Во время выполнения проекта

Теперь рассмотрим, как проинформировать всех заинтересованных лиц в компании о достигнутых успехах при минимальном количестве упреждающих внешних контактов. Мы обсудим и то, как лучше всего искать специалистов из других отделов, когда нужно мнение эксперта о происходящем.

Объявления об успехах

Оповещения о ходе работ нужны не только чтобы все узнали о завершении очередного этапа (и о получении ряда преимуществ). Они влияют на про-

дуктивность команды и на ее моральный дух. Большой рефакторинг порой оказывается настоящим испытанием даже для тех, кто привык к длительным проектам. И отмечание каждого завершенного этапа дает всем чувство удовлетворения.

Отчеты о ходе выполнения рефакторинга компания включает в рассылку новостей. Так ваша команда сможет получить признание за свой упорный труд и показать широкой аудитории, что рефакторинг — ценное вложение в инженерные разработки.

План выполнения

В главе 4 вы узнали, как составить эффективный план выполнения рефакторинга. Он может быть не только руководством к действию, но и местом документирования результатов вашей работы. Сделайте копию плана, так как в оригинал по необходимости вносятся только обновления оценок и основных показателей. Все другие пометки будут вноситься в копию. Они могут касаться чего угодно: странных ошибок, неожиданных пограничных случаев, отклонений от плана и т. п. Редактируемая версия изначального плана дает заинтересованным лицам детальное представление о ходе выполнения рефакторинга и помогает команде отслеживать, что сделано на сегодняшний день.

Для примера рассмотрим план из главы 4, составленный командой разработчиков в фирме Smart DNA для миграции всех сред на версию Python 2.7. Напомню, как выглядел первый этап этого плана:

- Создать единый файл `requirements.txt`.
 - **Метрика:** количество списков зависимостей; **начальное значение:** 3; **цель:** 1.
 - **Оценка:** 2–3 недели.
 - **Промежуточные задачи:**
 - сформировать список всех пакетов, используемых в каждом из репозиториев;
 - провести ревизию всех пакетов и оставить в списке только действительно необходимые с соответствующими версиями;
 - определить, с какой версии каждый пакет будет обновляться до Python 2.7.

В процессе работ этот план начнет заполняться подробностями выполнения каждой подзадачи. Например, вот так:

- Создать единый файл `requirements.txt`.
 - **Метрика:** количество списков зависимостей; **начальное значение:** 3; **цель:** 1.
 - **Оценка:** 2–3 недели.
 - **Промежуточные задачи:**
 - **Сформировать список всех пакетов, используемых в каждом из репозиториев.** При изучении пакетов в первом из трех репозиториев мы нашли шесть дополнительных зависимостей, не отраженных в файле `requirements.txt`. Сотрудники фирмы смогли предоставить нам обновленный список для первого репозитория и десяти зависимостей для двух других, которые тоже не были отражены в файле `requirements.txt`.
 - **Провести ревизию всех пакетов и оставить в списке только действительно важные с соответствующими версиями.** К счастью, у нас совпало 80 % пакетов, используемых в трех репозиториях. Из этого набора только у восьми пакетов были разные версии, которые нужно было обновить до последней.
 - **Определить, с какой версии каждый пакет будет обновляться до Python 2.7.** Для семи пакетов в финальном наборе это была непростая задача, потому что в 2.7-совместимых версиях не рекомендовалось использовать ряд API и функций, которые были активно задействованы в двух репозиториях из трех. Поэтому сначала пришлось уговорить сотрудников постепенно отказаться от этих устаревших функций, и только после этого можно было продолжить рефакторинг.

Внесение таких пометок по ходу выполнения позволяет всем заинтересованным лицам пользоваться этим планом для получения сведений о задачах команды на каждом этапе. Все SME, присоединившиеся к проекту позже, могут узнать, что на текущий момент сделала команда, просто прочитав план выполнения. Кроме того, я, например, иногда забываю, почему несколько месяцев назад было принято какое-то решение. И такие подробные отчеты позволяют легко вспомнить, что именно произошло и почему.

Подробный отчет пригодится и другим разработчикам, желающим понять, как кодовая база менялась со временем. Особо ценен этот отчет для специалистов, которые помогали вам в рефакторинге из карьерных соображений. Ведь в этой документации перечислены сложные технические проблемы, в решении которых они участвовали. Пригодится этот отчет и командам, которые сами хотят начать рефакторинг и ищут примеры его успешного выполнения.

Поиск обратной связи

При решении сложных задач все мы советуемся с более опытными коллегами. Конечно, в идеале хочется получить обратную связь от старших инженеров, но это непросто. Даже если такой специалист участвовал в рефакторинге как SME, вероятно, отвечать на запросы он будет очень медленно из-за сильной загруженности рабочими обязанностями. Но вероятность коммуникации можно повысить, корректно выразив свои ожидания.



Термином «старший инженер» я здесь обозначаю самого опытного сотрудника в команде, отделе или компании в целом. Он не имеет отношения к должности старшего инженера.

Запрашивая отзыв у старшего инженера, сначала определите нужный вам объем обратной связи. Для этого есть две основные причины. Во-первых, явное указание аспектов проблемы или решений, нуждающихся в оценке, избавит от получения отзывов об уже рассмотренных вещах. Во-вторых, старший инженер сразу сможет сосредоточиться только на том, что действительно важно.

Дальше нужно определить важность обратной связи от этого специалиста для дальнейшего прогресса. Если вы считаете, что прогресс возможен и без его мнения, скажите об этом сразу. Так он сможет корректно расставить приоритеты запросов, поступающих к нему со всей компании. Если же его вклад важен для продвижения вашей команды, попросите его по возможности пометить ваш запрос как срочный и оперативно дать ответ. Впрочем, независимо от срочности запроса сразу четко указывайте, в какой срок желательно дать ответ.



Сообщая старшему инженеру, что без его отзыва команда не может двигаться дальше, обязательно сразу обсудите, как быстро он сможет ответить. Если вопросов много, попросите выделить время для встречи с вами. Если же интересующие вас вопросы можно обсудить за несколько минут, попробуйте просто подойти и узнать, можно ли с ним поговорить. Разговор с человеком, который стоит перед вами, отложить сложнее, чем встречу, назначенную по переписке.

Помимо этого, нужно учитывать, насколько важным для динамики проекта считает свое мнение сам старший инженер. Если вы оба согласны с тем, что без его обратной связи можно обойтись, отлично! Но может получиться и так, что старший инженер будет удивлен и рассержен, если команда начнет продвигаться дальше, не поинтересовавшись его мнением. Этот момент очень важно выяснить заранее.

Приведем пример. Допустим, в рамках рефакторинга вы работаете над прототипом новой библиотеки. Он определяет базовые интерфейсы с временными неполными реализациями. Вы отправляете код на проверку с кратким описанием и ссылками на составленный командой проектный документ и отмечаете не только своих коллег, но и старшего инженера. К сожалению, вы забываете упомянуть, что вас интересуют отзывы об интерфейсах (а не о реализациях) и что за следующую неделю вы надеетесь слить эти изменения с основной веткой.

Следующие несколько дней вы получаете комментарии от товарищей по команде, но старший инженер молчит. Вы отправляете ему сообщение с вопросом, смог ли он ознакомиться с присланным на обзор кодом. В ответ получаете обещание выполнить обзор до конца недели. После обсуждения с товарищами по команде вы решаете объединить прототип с основной версией кода и продолжить его редактирование при последующих проверках.

Спустя день старший инженер, наконец, начинает комментировать детали реализации, но понимает, что объединение кода уже произошло. В итоге все раздражены. Вы — из-за того, что старший инженер слишком долго откладывал проверку и в итоге начал комментировать неактуальные вещи. Он — из-за того, что потратил время на комментирование временной версии кода, а вы объединили изменения, не дождавшись его ответа. Но этой ситуации можно было избежать, если бы с самого начала были четко оговорены сроки предоставления обратной связи.

РАБОТА В ОДИНОЧКУ

В грустной ситуации, когда вам предстоит выполнять рефакторинг самостоятельно, понадобится более частое и осознанное внешнее общение. Почему? Работая в одиночку, легко забыть, что ваши действия затрагивают других людей. Вы можете отказаться от программного обеспечения для управления проектами, считая, что разбросанных по столу стикеров достаточно. Даже если вы не прекратите участвовать в стендах или еженедельных синхронизациях, вы будете чувствовать обособленность от коллег, так как они мало знают о контексте вашей работы.

Даже если ваш начальник поддерживает вас, нужно найти способ дать доступ к вашей работе более широкому кругу: коллегам по команде, сотрудникам других отделов и высшему руководству. Фактически в такой ситуации внешними заинтересованными сторонами будут все, кроме вас. Поэтому подумайте об использовании всех техник из этой главы. Вот несколько идей.

- Если вы должны участвовать в стендапах, запишите для своих коллег информацию об обновлениях в вашем проекте и поместите ее в доступное всем место. Или можете организовать собственный асинхронный стендап, где будете рассказывать о вчерашних достижениях и о планах на день. Если найдете подходящий для вас инструмент управления проектами, используйте его для обновления своих задач.
- Если для еженедельных синхронизаций нужно присыпать повестку дня, постарайтесь добавлять хотя бы по одному пункту каждый раз. Это поможет держать команду в курсе того, чем вы занимаетесь, и может познакомить вас с разными точками зрения на проблемы, которые вы решаете. Если еженедельные синхронизации вашей командой не практикуются, постарайтесь выделить себе час для обзора сделанного и обновления документации (я считаю, что поддерживать актуальное состояние документов проще, если для этого специально выделяю время). Можно подумать о написании еженедельного обзора и размещении его в доступном всем месте.
- Выберите время, когда заинтересованные лица (включая инженеров из затронутых рефакторингом команд) могут прийти, чтобы задать вопросы или обсудить с вами проблему. Расписание таких встреч зависит от частоты вносимых вами изменений в чужой код и степени вовлеченности инженеров в рефакторинг. Можно начать с двух раз в месяц, уменьшая или увеличивая частоту при необходимости.

Экспериментируйте

Вот самое главное, что вы должны извлечь из этой главы: нет единственной верной стратегии коммуникации. При каждом рефакторинге она будет своей. Более того, она может меняться на протяжении проекта. Выбирайте порядок действий в зависимости от собранной для проведения рефакторинга команды, затрагиваемых изменениями отделов и уровня участия внешних заинтересованных сторон.

Если окажется, что выбранный алгоритм не дает нужного результата, сразу меняйте его! Правильно организованная коммуникация помогает команде работать эффективно и стабильно. Плохая же мешает движению проекта вперед. Если что-то не работает, лучше это поменять, чем держаться за тормозящую прогресс схему.

В следующей главе я продолжу тему создания шаблонов, помогающих продуктивной работе. Вы познакомитесь с концепциями (техническими и нетехническими), которые ваша команда, возможно, захочет опробовать во время рефакторинга.

ГЛАВА 8

Стратегии выполнения

Одна из старейших и наиболее популярных систем общественного транспорта в мире — открытое в 1904 году в Нью-Йорке метро — в будний день обслуживает в среднем чуть менее шести миллионов пассажиров. Те, кому часто приходится пользоваться этим видом транспорта, разработали десятки небольших вариантов оптимизации. Вечерами во вторник мы слушаем объявления об изменениях в движении поездов и знаем, под каким углом прикладывать к турникуту проездной. Мы можем дать много советов новичкам, чтобы упростить их первые опыты поездок на метро.

Эту главу можно сравнить с дружелюбным жителем Нью-Йорка, который дает совет, как лучше ориентироваться в системе городского метро. Я дам вам много советов для плавного выполнения рефакторинга. Сначала вы познакомитесь с передовыми методами построения команды. Есть несколько способов выйти за рамки обычного делового общения, обеспечив продуктивность и хороший настрой команды. Затем я расскажу, какие моменты нужно отслеживать, чтобы точно двигаться в верном направлении, и на что обращать внимание на последних этапах рефакторинга. Наконец, мы обсудим несколько стратегий кодирования, позволяющих надежно контролировать рефакторинг на стадии внедрения.

Формирование команды

В главе 6 вы узнали, почему в крупных программных проектах так важна сильная команда. В основном речь шла о преимуществах, которые надежные товарищи по команде дают в трудные времена (например, на этапе долгой рутинной работы или при столкновении с препятствием). Я не упомянула, что сплоченные команды более изобретательны, больше учатся друг у друга, лучше и быстрее решают проблемы. Поэтому так важно регулярно участвовать в мероприятиях для формирования командного духа. Сейчас я расскажу, как это может выглядеть. Конечно, возможны и другие способы.

Но то, о чем сейчас пойдет речь, я считаю одним из самых полезных для укрепления отношений с товарищами по команде.

Как только новые привычки станут вашей второй натурой, долгие проекты, в том числе и большой рефакторинг, начнут пролетать незаметно.

Парное программирование

Парное программирование — отличный инструмент для построения команды. Совместная работа дает прекрасную возможность узнать сильные и слабые стороны друг друга. Если у вашей команды еще не было опыта совместной работы, перед началом проекта подумайте, как их объединить. Важно сделать это именно на ранней стадии, ведь новый проект позволит с самого начала сформировать новые привычки.

С практической точки зрения с помощью парного программирования можно передавать знания от одного человека другому. Если только один человек обладает уникальными знаниями об одной или нескольких частях системы, то это не совсем нормально. Прежде всего, это чрезмерная психологическая нагрузка. Такие специалисты понимают, что не могут взять перерыв и на несколько дней полностью отключиться от работы. Ведь в чрезвычайной ситуации в той части системы, которую они знают, никто не сможет помочь. Во избежание этого организуйте парную работу для передачи опыта другим членам команды. Равномерное распределение знаний облегчает нагрузку на отдельных разработчиков, когда во время рефакторинга возникают какие-то проблемы.

Парная работа улучшает процесс отладки и упрощает поиск решения сложных или абстрактных проблем. Не зря говорят, что одна голова хорошо, а две лучше. Два человека, думающих над одной задачей, могут сгенерировать несколько идей, из которых будет выбрана лучшая. Активный диалог поможет найти истину и усовершенствовать решение. Ошибок тоже становится меньше. Исследования Университета Юты (<https://oreil.ly/yA75W>) показали, что написанный в паре код содержит примерно на 15 % меньше ошибок. Наконец, работающие в паре меньше отвлекаются. При совместном поиске решения не будет времени на электронную почту или социальные сети.

Во время рефакторинга парная работа особенно эффективна: пока один печатает, второй свободно обдумывает происходящее с более широкой перспективы. При рефакторинге легко увязнуть в деталях, пытаясь распутать сбывающий с толку унаследованный код. Напарник может помочь снова сосредоточиться на цели и, подумав на несколько шагов вперед, указать на трудности, которые вас поджидает.

У парного программирования есть и недостатки. Когда доходит до оценки проблемы или изучения чего-то нового (нового фреймворка, инструмента или языка программирования), некоторые, в том числе и я, предпочитают действовать самостоятельно. Я обнаружила, что лучше запоминаю важные концепции, когда разбираюсь в них сама.

Не особенно продуктивна парная работа в случае четко определенных и несложных задач. Конечно, остается вероятность, что задача будет решена быстрее и с меньшим числом ошибок, но выделять для такой работы сразу двух специалистов нецелесообразно.

Да и сама по себе работа в паре может утомлять. Постоянное озвучивание мыслительного процесса требует больше энергии, чем обдумывание проблемы в одиночку. К концу сеанса парного программирования вполне может понадобиться перезарядка. Особенно сложно такие вещи даются тем, кто не любит и не умеет общаться. Для них любая работа в паре станет кошмаром. Поэтому выступая за создание пар, важно помнить обо всех особенностях и предпочтениях членов команды.

Подытожим. Разбивая команду на пары, нужно придерживаться следующих правил.

- *Парное программирование нужно поощрять, но не делать обязательным.* Возможно, далеко не все члены вашей команды большие сторонники парного программирования. Подчеркивая преимущества такого подхода и оказывая ему поддержку, можно убедить попробовать работать в паре тех, кто не определился со своим отношением к ней (или никогда ее не пробовал). Возможно, потом они захотят повторить этот опыт. Но принуждать к объединению тех, кто испытывает от этого дискомфорт, неправильно. Это может привести к недовольству командой и проектом или даже к уходу из команды.
- *Объединяйте инженеров с одинаковым уровнем опыта.* Если парное программирование используется не для передачи опыта, лучше объединять специалистов примерно одного уровня. Только тогда работа над сложными задачами будет эффективной, ведь будет возможность на равных обмениваться идеями.
- *Ограничивайте продолжительность сеансов.* Парное программирование — утомительный процесс, поэтому важно заранее четко определить продолжительность сеансов (с перерывами при необходимости). Начните с часа совместной работы. Если после этого у вас останется энергия, можно продлить еще на час. Позвольте друг другу останавливать процесс — слишком большая продолжительность парной работы снижает ее производительность.

Сохранение мотивации

В главе 4 мы обсуждали процесс построения сбалансированного плана выполнения, обладающего достаточной гибкостью для сохранения продуктивности сотрудников. Но важно продумать и механизмы сохранения мотивации во время длительного рефакторинга, позволяющие членам команды лучше узнать друг друга и отмечать совместные достижения. Для налаживания прочных связей внутри команды и определения вклада каждого в общий результат не нужен большой бюджет или престижные выездные мероприятия. Поддерживать моральный дух можно просто, но эффективно.

Мотивация членов команды

Один из наиболее убедительных способов повышения мотивации членов команды — предоставление возможности максимально использовать уникальные навыки и способности человека для внесения вклада в рефакторинг. Члены команды станут намного счастливее (и продуктивнее), если начнут работать над наиболее полезными с их точки зрения аспектами рефакторинга. Если человеку хочется развить новые навыки или контролировать важный участок проекта, позвольте ему заняться этим. В главе 6 я уже рекомендовала это как способ привлечь к участию в проекте сторонних специалистов.

По возможности установите свободный график. Не всем комфортно ежедневно работать с 9 до 17 с полусырым перерывом на обед. Кому-то удобней начать рабочий день на рассвете и после обеда уже отправиться по своим делам. Кто-то предпочтет начать попозже и задержаться после официального завершения рабочего дня. Если удастся приспособиться к свободному расписанию участников при поддержании достаточного уровня взаимодействия (см. в предыдущей главе), продуктивность работы возрастет.

Признание особого вклада отдельных членов команды — отличный способ поддерживать мотивацию. Показывая людям, как ценится их упорный труд, руководство подтверждает, что они поступают правильно, поощряет их продолжать в том же духе и укрепляет чувство принадлежности к команде. Признание может выглядеть как угодно — от формализованной программы, принятой в отделе или компании, до таких простых вещей, как написание заметки. Помните, что каждый в команде хочет быть узнаваемым. Но некоторым может нравиться видеть всеобщее признание, а другие будут склоняться от публичной похвалы. Неправильно выбранная форма признания

в лучшем случае сделает его не очень эффективным, а в худшем окончится провалом. Иногда хватает благодарственного письма или хвалебного упоминания в отчете о достижениях.



Можно установить традицию выбирать лучшего сотрудника недели. Для начала выберите переходящий приз (любой выделяющийся предмет, который можно поставить на рабочий стол), который вручается человеку, отличившемуся на прошлой неделе. Отмечать можно любые достижения — от вмешательства, которое помогло справиться со сложной проблемой, до отличного документирования последних исправлений. Через неделю победитель сам выбирает, кому передать приз. Традиция сохраняется до завершения проекта или пока команда не решит от нее отказаться.

Мотивирование команды

Теперь поговорим о способах мотивации команд. Заинтересовать всех занятых в крупном проекте гарантированно можно путем превращения работы в игру. Геймифицируя самые рутинные и скучные фрагменты рефакторинга, можно ускорить продвижение по этапам. Возьмем для примера игру «Бинго». Выбирается небольшая, но важная задача, которую нужно выполнить на текущем этапе рефакторинга (например, десять проверок кода). Для нее создается таблица, как для Бинго. Печатаются копии этой таблицы и раздаются членам команды. За победу предлагается небольшой приз.

Но при таком подходе нужно следить за тем, чтобы не разгоралась конкуренция. Это отличный источник мотивации, но когда процесс выходит из-под контроля, возникает риск конфликта и падения морального духа. Кроме того, большой рефакторинг оставляет мало места для небрежности, так что этот метод нужно применять с осторожностью. Ведь важно стимулировать качество работы, а не скорость ее завершения. Если сделать упор на достижении финиша, может начаться экономия на качестве в стремлении решить поставленные задачи быстрее.



При составлении плана выполнения в числе прочего примерно оценивается время на решение каждой подзадачи. Эту процедуру можно превратить в аналог шоу «Цена удачи» (Win of the Week) (то есть нужно попытаться угадать, сколько времени займет выполнение подзадачи, не превысив реальный показатель). После завершения подзадачи на следующем собрании объявляется имя члена команды, оценка которого оказалась ближе всего к истине. Все получат удовольствие от игры, и постепенно улучшится способность оценивать время выполнения разных задач!

По возможности отмечайте достижения команды одним-двумя сображенными на протяжении проекта. Особенно после завершения важных этапов. Моменты праздника дают ощущение постоянной вовлеченности и поддерживают хороший моральный дух. Если команда не может взять перерыв и отметить достигнутые результаты, рефакторинг превратится в бесконечные крысиные бега. Важно выделять время на общее собрание, выбрав для него оптимальную форму, будь то совместный обед или тост во время полдника. Всем будет приятно, что нашлась минутка отметить совместные достижения.

Учет результатов

Во время рефакторинга важно часто проверять прогресс и вести постоянный учет достигнутых результатов. Так вы удостоверитесь в правильности выбранного направления и с меньшей вероятностью что-нибудь упустите на заключительных этапах. Обязательно следует вносить информацию о ходе выполнения в копию плана, как в подразделе «План выполнения» из предыдущей главы.

Промежуточные измерения

В главе 3 вы познакомились со способами количественной характеристики проблем, которые призван решить рефакторинг. Позже эти показатели были добавлены в план выполнения, где проект разбивался на отдельные этапы с собственным набором показателей. На них мы и будем ориентироваться в процессе рефакторинга. В крупных программных проектах на каждом шагу возникает опасность неконтролируемого расширения масштабов.

Раз в неделю-две измеряя прогресс команды на пути к очередной промежуточной вехе, мы помогаем себе придерживаться верного курса. Частые проверки удерживают команду от искушения решать побочные задачи, расширяющие масштаб проекта. Кроме того, периодические проверки позволяют оценить скорость работы. Если все заняты делом, но показатели уже несколько недель почти не меняются, что-то явно не так. Прогрессу могут мешать сложные ошибки, с которыми сталкивается команда, или выбранные показатели неточны. Но вне зависимости от причины вы поймете, что успешно справились с трудностями, когда снова увидите положительную динамику.

Обнаруженные ошибки

Даже если ваш рефакторинг проводится не для выявления и исправления системных ошибок, в процессе его выполнения они все равно появятся. Будете вы исправлять ошибку или нет, нужно обязательно документировать, когда вы заметили, при каких условиях она возникает (для облегчения воспроизведения) и что было предпринято. В контексте рефакторинга у вас есть два варианта действий: можно исправить ошибку, а можно воспроизвести ее в исправленной версии кода.

Если у вас получается полностью исправить ошибку простыми средствами, на это можно ссылаться как на демонстрацию эффективности рефакторинга при обсуждениях с заинтересованными сторонами или коллегами. Иногда всего пары опасных, хорошо документированных ошибок может убедить любого сомневающегося в том, что рефакторинг стоило проводить. Если же команда решает оставить ошибку в новой версии кода, нужно точно описать, где она и как ее воспроизвести, и передать эту информацию тем, кто будет заниматься ее исправлением.

ИСПРАВЛЯТЬ ИЛИ НЕТ

Решая, что делать с обнаруженной при рефакторинге ошибкой, важно учитывать ряд факторов. Во-первых, гарантировать точное воспроизведение исходного поведения проще, если оно копируется целиком, включая ошибки и все остальное. Исправляя ошибку при рефакторинге части кода, мы отклоняемся от эталонного поведения. Теперь нужно не только учитывать исправленную ошибку при всестороннем регрессионном тестировании любого рода, но и готовиться к новым ошибкам или неожиданному поведению.

Крупный рефакторинг всегда связан с глубоким погружением в чужой код. То, что нам может показаться ошибкой, вполне может ею не быть. Даже если ответственная за этот функционал команда подтвердит ошибку, у нас может не быть контекста для ее исправления. Но если вы решите ее исправить сейчас, то это удовлетворит пользователей и сделает реорганизованное решениеальным. В определенном смысле устранять проблемы выгоднее и удобнее сразу же после их обнаружения.

Советую написать модульные тесты, подчеркивающие обнаруженную ошибку, и передать их команде, ответственной за затронутый функционал. Опишите условия, при которых воспроизводится ошибка. Если команда считает такое поведение ожидаемым, просто продолжите рефакторинг. Если вам скажут, что дефект допустим, попросите определить приоритет его исправления. В случае высокого приоритета попросите команду *сначала исправить ошибку и подтвердить это с помощью написанных вами модульных тестов*. После этого исправление можно добавить к остальному отредактированному коду.

Удаление артефактов

В разделе «Очистка кода» главы 4 я рассказала о важности добавления в план отдельного пункта для удаления оставшихся от рефакторинга артефактов. При рефакторинге в приоритете должно быть создание упорядоченной кодовой базы, которой удобно будет пользоваться другим разработчикам. Улучшение эргономики кодовой базы — это уже серьезная мотивация для большого рефакторинга. Представить, какие артефакты появятся при работе над кодом, сложно. Можно быть уверенными только в одном: они сами будут возникать по ходу выполнения рефакторинга.

Следите за всем, что нужно привести в порядок, независимо от того, когда вы планируете это делать: в конце текущего этапа или только на последних этапах проекта. Список артефактов важно обновлять, как только часть кода будет объявлена устаревшей. Только так на этапе очистки удастся удалить все. Разработчики, которым код попадет в руки после рефакторинга, будут благодарны вам за такой подход.



Как повара рекомендуют мыть кастрюли и сковородки сразу после приготовления еды, так и я советую проводить очистку по мере выполнения рефакторинга. Гораздо проще (и безопаснее) сразу удалять ненужные фрагменты кода. В этом случае память еще хранит сведения о взаимодействии между устаревшими фрагментами и остальным кодом. Это позволяет совершить в процессе удаления меньше ошибок.

Элементы, выходящие за рамки проекта

Почти каждый разработчик во время рефакторинга сталкивается с возможностями расширить его масштаб. Понятно, что если не поддаваться такому искушению, шансы уложиться в сроки повышаются. Но все-таки полностью игнорировать эти возможности не стоит. Составьте список. Он подтверждает универсальность рефакторинга, показывая возможность продолжить улучшение кодовой базы. Благодаря ему будет больше шансов, что заинтересованные стороны (и коллеги) поверят в ценность рефакторинга. Если ваша или любая другая команда в компании захочет, опираясь на заложенный рефакторингом фундамент, продолжить постепенное улучшение кодовой базы, можно выбрать из списка несколько проектов и запустить их.

Продуктивное программирование

Есть несколько полезных стратегий, помогающих сделать длительный рефакторинг понятнее и для себя, и для членов вашей команды. Разрабатывать крупные программные проекты не всегда сложно. При написании нового функционала сложные маневры в основном нужны только при его встраивании в существующую кодовую базу. А вот для рефакторинга важно написать значительный объем кода, большая часть которого должна воспроизводить существующее поведение. Такой код нужно тщательно проектировать и аккуратно интегрировать с исходной реализацией. В этом случае вероятность сбоев возрастает. Я надеюсь, что описанные в этом разделе методы помогут вам успешно ориентироваться в процессе разработки рефакторинга.

Прототипирование

Разрабатывая план рефакторинга в главе 4, мы стремились добиться нужного уровня детализации. Предполагалось, что план будет доходчивым и конкретным, чтобы его могли читать как заинтересованные лица из руководства, не очень хорошо знакомые с техническими деталями, так и члены команды, которой предстояло заняться рефакторингом. Там, где план намеренно оставался расплывчатым, открывается прекрасная возможность для прототипирования.

Создание прототипов на раннем этапе часто помогает работать быстрее, но нужно придерживаться двух важных принципов.

- *Помнить, что решение не будет идеальным.* Важно получить рабочее решение, не тратя слишком много времени на шлифовку деталей. Затраты на идеальное решение здесь не окупятся, потому что изменение требований в будущем может легко сделать его устаревшим (примеры этого вы видели в главе 2).
- *Быть готовыми выбросить код.* Если написанное решение не работает, возьмите из него рабочие части, остальное отбросьте и начните заново. Прототипирование — это всегда попытка, из которой извлекается урок, и делается следующая попытка.

Представим, что во время рефакторинга было решено разделить раздущий класс на несколько компонентов. Согласно предварительному плану,

основной функционал помещается в три новых класса. Остается ряд второстепенных, но тоже важных функций, которые предстоит добавить в один из них. Не разрабатывайте решение сразу. Можно создать несколько прототипов и проверить, как себя ведут новые классы в нескольких разделах кодовой базы. Так вы узнаете, какой из вариантов лучше работает, и сможете выработать решение, хорошо интегрирующееся с остальной частью кодовой базы.

Движение вперед маленькими шагами

Когда радикальные изменения затрагивают большую площадь, легко увлечься. Представим, что функцию `pre_refactor_impl` нужно заменить на `post_refactor_impl`. Кодовая база содержит примерно 300 экземпляров функции `pre_refactor_impl`, разбросанных более чем по 80 файлам. Задачу можно решить через поиск и замену, после чего полученный коммит отправляется на рассмотрение товарищу по команде. Если миграция выполняется просто, как в примере, кажется, что удобнее всего создать один набор изменений. Но у этого подхода есть несколько серьезных недостатков.

Во-первых, получить хороший код проще всего можно, фиксируя небольшие постепенные изменения. Создание небольших коммитов позволяет быстро и часто получать адекватную обратную связь от инструментов (например, интеграционных тестов, выполняемых на сервере путем непрерывной интеграции). Редкое, но объемное редактирование увеличивает вероятность множества ошибок тестирования, с которыми придется бороться. Чем больше модификаций на один коммит, тем выше шанс каскадных сбоев тестирования, когда исправление одной ошибки приводит к обнаружению следующей. Небольшие, но частые коммиты позволяют лучше понять происходящее и быстрее разобраться с любыми сбоями тестов. То же касается и проверки изменений вручную.

Во-вторых, отменить маленький коммит намного проще, чем большой. Если что-то пойдет не так во время разработки или даже спустя время после развертки кода, отмена небольшого коммита путем возврата позволяет аккуратно избавиться от нежелательного изменения.

В-третьих, краткие коммиты всегда конкретнее. Это позволяет добавить к ним точное описание, с помощью которого можно быстрее найти нужные наборы изменений. Кроме того, позже, при сканировании истории версий, другие разработчики смогут лучше понять содержание каждого коммита.

Ну и нельзя забывать о том, что крошечные коммиты обычно проверяются и утверждаются намного быстрее!

Наконец, в одиночку почти невозможно адекватно просмотреть весь отредактированный код. Полагаться на его рецензирование как на способ обнаружения ошибок тоже не стоит (для этого проводится тщательное и серьезное тестирование). При недостаточном покрытии кода поиском возможных ошибок придется заняться проверяющему. Сначала может показаться, что проверять изменения легко. Но позже, если специально не сосредоточиваться на коде, способность подмечать несоответствия ослабевает. Поэтому рецензировать большие наборы изменений гораздо легче, если они разбиты на логически структурированные и содержательные коммиты.



При рефакторинге желательно максимально сохранить исходную историю версий. Страйтесь перемещать файлы командой `git mv`, а не удалять их и добавлять обратно. В описаниях коммитов указывайте, что изменение — часть более крупного рефакторинга, чтобы инженеры знали, что для поиска владельца кода нужно глубже копать в истории коммитов. Страйтесь сделать описания как можно более исчерпывающими, явно указывая, на какие изменения должен обращать внимание рецензент кода, и добавляя необходимый контекст.

Тестирование

Рефакторинг предполагает воспроизведение существующего поведения, и важно постоянно убеждаться в том, что поведение не изменилось. Прoverить отсутствие изменений гораздо труднее, чем их наличие. Поэтому особую важность приобретает постепенное и многократное тестирование на протяжении всего рефакторинга. Часто запуская модульные и интеграционные тесты или проводя ручное тестирование, можно либо подтвердить, что все осталось неизменным, либо указать точный момент изменения поведения.



Перед началом редактирования фрагмента кода убедитесь в наличии для него хороших модульных тестов. Возможно, уже есть тесты для подтверждения поведения, но важно определить, не упускают ли они какие-либо дополнительные задачи. В случае неточности тестов (например, поток выполнения проверяется только для функций верхнего уровня, не затрагивая отдельных вспомогательных функций) разбейте их. Детализированные тесты, как и детализированные коммиты, помогут сузить круг проблем на ранней стадии.

«Глупые» вопросы

Наверное, в такую ситуацию попадал каждый: на собрании старшие инженеры обсуждают технологию или какой-то функционал, а вы ничего не понимаете. Все сидят тихо, слушают и кивают. Вы же сбиты с толку, но боитесь задавать уточняющие вопросы, чтобы не показаться некомпетентным. Обычно сценарий таких встреч развивается в двух направлениях. Первый: вы молчите и пытаетесь все-таки сообразить, о чем речь, не имея возможности поучаствовать в беседе. Второй: кто-то из присутствующих вдруг вежливо задает интересующий вас вопрос, и вы слышите интересующие вас пояснения.

Советую не сильно рассчитывать на второй вариант. Не стоит ни молчать слушать, ни надеяться на других. Нужно просто встать и задать «глупый» вопрос. Отдавая предпочтение ясности, а не пытаясь сохранить вид компетентного специалиста, вы моделируете важный шаблон поведения в команде. Вы показываете, что глупых вопросов не существует и что в любом обсуждении важно убедиться, что все понимают, о чем речь. Только так в команде будет больше продуктивных обсуждений и меньше недопонимания. Это позволит быстрее приступить к решению актуальных проблем.

Поскольку большой рефакторинг — это почти всегда обширная площадь изменений, вы почти гарантированно будете сталкиваться с незнакомыми частями кодовой базы. Не бойтесь искать тех, кто хорошо разбирается в этом коде, и просить у них совета. Нужно ли вам краткое объяснение или подробное пошаговое руководство, главное — как следует понять код, который вы собираетесь редактировать. Этим вы не только сэкономите время на разработку и внесете меньше ошибок, но и сможете оптимально выполнить рефакторинг.

Заключение

Итак, вы завершили последние несколько коммитов и все привели в порядок. Осталось выполнить последнюю важную задачу — найти способ закрепить результат ваших усилий. В следующей главе вы узнаете, какие шаги нужно предпринять, чтобы предотвратить возвращение кодовой базы в исходное состояние.

ГЛАВА 9

Закрепление результатов рефакторинга

Чуть больше года назад мой друг Тим решил перестать употреблять сахар, чтобы похудеть и лучше себя чувствовать. Первая неделя была тяжелой: Тим был вялым и очень хотел чего-нибудь сладкого. Но к концу третьей он привык обходиться без сахара и снова чувствовал себя бодрым. Постепенно начали проявляться преимущества нового режима питания: Тим стал меньше уставать и сбросил несколько килограммов.

Позже соблюдение этой диеты стало проблемой. Тим неоднократно наблюдал у своих друзей срывы и понимал, как важно установить для себя реалистичные ожидания. Чтобы избежать соблазнов, он убрал из квартиры всю сладкую еду. Вел дневник питания, считая калорийность своей еды. Но иногда позволял себе небольшие удовольствия при встречах с друзьями. Через два месяца его девушка тоже решила отказаться от сахара, что позволило им поддерживать и воодушевлять друг друга. Сейчас здоровье Тима значительно улучшилось, а по уровню энергии с ним может соперничать, наверное, только его щенок.

Рефакторинг можно сравнить с переходом на новую диету. Может показаться, что самая большая проблема — выяснить, какие изменения нужно внести, и реализовать их. Но такие же усилия нужны и для обеспечения долговременности этих изменений. Поэтому сейчас мы поговорим об инструментах и методах, позволяющих гарантировать долговечность внесенных во время рефакторинга изменений. Я расскажу, как на уровне организации заставить принять установленные рефакторингом шаблоны и как использовать непрерывную интеграцию для дальнейшего содействия их внедрению. Вы узнаете о важности презентации последствий рефакторинга.

ринга коллегам. Наконец, мы коснемся способов внедрения постепенных улучшений в культуру разработки, чтобы в дальнейшем не возникала потребность в рефакторинге.

Содействие принятию рефакторинга

Часто рефакторинг затрагивает множество инженеров. И по двум причинам важно, чтобы они поддержали как рефакторинг, так и установленные им шаблоны.

Во-первых, нужно обеспечить сохранение внесенных изменений в долгосрочной перспективе. Большой рефакторинг часто приводит к появлению двух лагерей: горячих сторонников и противников выбранного проекта. Если противники отказываются писать код по новым шаблонам, на границе между изменениями, внесенными в процессе рефакторинга, и их собственным кодом появляется новый «мусор», который начинает накапливаться и постепенно уничтожает все выгоды от рефакторинга.

От качества выполнения рефакторинга это не зависит. Просто не все согласны с новым видением. Новички в команде могут не до конца понимать, какие проблемы пытался решить рефакторинг. Когда у коллег-разработчиков нет контекста для должной оценки результатов рефакторинга, им сложно начать работать в новой среде. Они могут неправильно реализовывать новые шаблоны или вообще игнорировать их, когда те могут принести большую пользу для кода.

Во-вторых, поддержка инженеров-разработчиков нужна, чтобы обеспечить проникновение новых шаблонов во всю кодовую базу. Нам важно, чтобы внесенные во время рефакторинга изменения не просто остались, а влияли на будущие решения инженеров, работающих с кодовой базой. Рефакторинг можно сравнить с прополкой заросшего огорода, вскапыванием грядок и посадкой луковиц. Нам нужно, во-первых, сохранить посаженный лук, а во-вторых, побудить остальных членов семьи сажать на подготовленных грядках другие овощи.

Например, после нескольких случаев утечки личных данных был проведен рефакторинг основной библиотеки журналирования для отказа от произвольных строк. После него новые поля и типы журналов регистрации нужно было сначала добавить в библиотеку. Команда решила не менять каждую

точку входа системы журналирования, а просто изменить логику существующей библиотеки.

Но некоторых сотрудников компаний не устраивала потеря гибкости, которую давали произвольные сроки. Новички из других компаний, где к ведению журнала подходили более свободно, тоже не понимали сути ограничений. Зачастую без должного информирования о мотивах и целях рефакторинга начинается поиск способов обхода встроенных в новую библиотеку мер безопасности. А это только увеличивает риск утечки личных данных.

Даже если инженеры согласятся с изменениями после рефакторинга, они могут не поддержать активное преобразование существующих точек входа под новую библиотеку. Они могут и игнорировать необходимость добавления в новую библиотеку полей и типов журналов, предпочитая использовать существующие для более широкого диапазона записей. Упростив процесс расширения библиотеки журналирования и показав всем, как это делается, вы упростите переход к новой библиотеке и, возможно, увеличите число ее применений в кодовой базе.

Есть много способов стимулировать принятие результатов рефакторинга в организации. По моему опыту, лучше создать эргономичные интерфейсы для взаимодействия с отредактированным кодом. Их нужно планировать на ранних этапах проекта и постепенно дорабатывать. Узнайте мнение товарищ по команде и опытных коллег, как сделать более эргономичной границу между кодом после рефакторинга и остальной частью кодовой базы. Если к завершению рефакторинга интерфейсы недостаточно протестированы на пользователях, проведите семинар с коллегами из разных отделов и с их помощью постепенно доработайте интерфейсы.

Сейчас мы поговорим о таких эффективных методах закрепления результатов, как обучение сотрудников работе с новой версией кода, предоставление документации и стимулирование использования новых шаблонов.

Образование

Есть два основных метода информирования. Активный — планирование и проведение семинаров или аналогичных вариантов обучения, и пассивный — создание пошаговых инструкций или коротких онлайн-курсов на учебной платформе компании.

Активное образование

Если рефакторинг затрагивает критическую часть кодовой базы, часто используемую разработчиками из других отделов, на первый план выходит активный образовательный компонент. Инженеров, привыкших к существующему набору шаблонов, нужно ознакомить с совершенно новым способом работы.

Семинары

Убедиться в эффективном взаимодействии инженеров с реорганизованным кодом проще всего на обучающем семинаре, где требуется интерактивная работа с примерами кода и можно задавать вопросы. Важное преимущество семинаров в том, что их участники специально выделяют время, чтобы разобраться в новом материале. Иначе некоторые разработчики из-за слишком большой загрузки не стали бы знакомиться с результатами рефакторинга.

Лучшее время для такого семинара — сразу по завершении рефакторинга. До этого изучать новые код и шаблоны бессмысленно, ведь есть риск, что все изменится. Да и в принципе не до конца очищенный код лучше не давать в пользование людям, не очень хорошо знакомым с деталями рефакторинга. Поэтому прежде, чем приступить к планированию первого семинара, убедитесь, что все в порядке. А еще лучше проведите пробный семинар со своей командой для шлифовки всех нюансов.

Занятия не должны длиться вечно. В идеале за несколько месяцев большинство инженеров, на работу которых рефакторинг повлиял сильнее всего, должны быть хорошо знакомы с новым кодом. Этот код должен стать новой нормой, а потребность в пояснениях исчезнуть. Лучше всего подготовить два-три семинара, следя за уровнем интереса и посещаемостью. Как бы ни были увлекательны такие тренировки, они отнимают очень много времени. Поэтому их должно быть немного. Если по завершении семинаров остаются вопросы, подумайте о написании более исчерпывающей документации.

Посмотрим, как мог бы выглядеть сеанс обучения на примере с новой библиотекой журнализирования.

1. Кратко расскажите о причине рефакторинга, приведя несколько убедительных примеров. Объясните, чем была обусловлена утечка данных в старой версии библиотеки журнализирования и как пользоваться новой версией этой библиотеки, созданной для предотвращения утечки информации.

2. Разделите участников на пары и попросите их перенести те же конфигурационные настройки для использования новой библиотеки. Отвечайте на все возникающие вопросы. Если какие-то пары дадут несколько вариантов решения, попросите объяснить.
3. Пусть пары выберут более сложный вариант, в идеале требующий расширения библиотеки (через добавление нового типа журнала или поля). Проверьте результаты каждой пары и ответьте на их вопросы.

График приема

График приема тоже можно отнести к средствам активного обучения. В приемные часы любой может зайти и задать проводившим рефакторинг специалистам вопросы по работе с новой версией кода. Не у всех, кого затронет рефакторинг, будет время (или интерес) посетить семинар. Если они смогут получить подробную консультацию, вероятно, они позитивно примут внесенные изменения. Приемные часы будут полезны и участникам семинаров, чтобы обратиться за дополнительными рекомендациями или советами.

График приема удобен тем, что позволяет ограничить время ответа на вопросы. Сразу после завершения рефакторинга шквал вопросов резко обрушивается ото всей компании. Они способны полностью украсть ваше время и внимание (и работать, постоянно отвлекаясь, очень сложно). Предложение подходить со всеми несрочными вопросами в приемные часы снимает нагрузку с ответственной за рефакторинг команды.

Отслеживайте запросы, с которыми приходится сталкиваться во время приема, и используйте их для написания FAQ. Этот документ избавит команду от необходимости отвечать на одни и те же вопросы.

Обмен опытом

Многие инженерные группы проводят открытые собрания, где любой может рассказать о своей работе. Крупный рефакторинг часто сопровождается множеством немаловажных событий: непредсказуемые ошибки, код, который в последний раз редактировали 15 лет назад, пошедшее не по плану внедрение. Многие любят слушать чужие рассказы о работе с кодом, а самые яркие надолго остаются в памяти.

Расскажите на таком собрании об интересных моментах рефакторинга. Скорее всего, многим после этого захочется больше узнать о причине рефак-

торинга и о том, как извлечь из него пользу. Иногда небольшого, но яркого рассказа о проекте хватает, чтобы заручиться поддержкой ваших коллег.

Пассивное образование

В главе 7 вы узнали о том, что нужно не только тщательно документировать весь процесс рефакторинга, но и выбрать канал передачи информации и организационную схему, подходящие для вашей команды. На последних этапах проекта создание документации должно быть в приоритете. Она должна описывать цели рефакторинга и то, как коллеги, работающие с подвергшейся редактированию кодовой базой, могут извлечь выгоду из происходящего. Еще раз напомню: все нужно сохранять в один специально созданный для этой цели каталог.

Информацию можно предоставлять в разных формах: FAQ, файл README с кратким изложением целей проекта или учебное пособие. Это отлично экономит время команды после завершения рефакторинга, когда у сотрудников других отделов начинают появляться вопросы. Вместо того чтобы каждый раз говорить одно и то же, достаточно дать ссылку на каталог с документацией.

Писать руководства, помогающие ориентироваться в кодовой базе после рефакторинга, я советую с исторической точки зрения. Начните с того, почему нужен рефакторинг, и закончите результатами, которых вы достигли. Такой подход позволит сохранить актуальность материала. По возможности добавляйте даты, чтобы дать читателю ориентиры. Вот как это выглядит в примере с библиотекой журнализирования.

1. Начните с объяснения причин деградации кода. В случае с библиотекой журнализирования это будет обзор первоначального проекта. Расскажите, что, по замыслу авторов, библиотека должна была быть легкой и простой в использовании, позволяющей любому добавлять туда что угодно в удобной форме.
2. Объясните, как усложнение продукта и появление в команде новых людей повлияло на риск утечки конфиденциальных данных. Перечислите серьезные случаи утечки, сделав акцент на том, что в последние месяцы их частота стала расти.
3. Опишите, как ваше решение препятствует утечке этих данных. Сравните несколько конфигурационных настроек для старой и новой версий

библиотеки. Подчеркните, что речь идет о нынешнем состоянии кода, так как есть вероятность, что он продолжит развиваться. То есть вместо «сегодня» лучше говорить «по состоянию на сентябрь 2020 года».

Закрепление

Положительное подкрепление — мощный инструмент. Всем разработчикам без исключений нужно напоминать о новых шаблонах (возможно, не один раз). И это можно делать двумя способами. Во-первых, применять тактики из подраздела «Мотивация членов команды» предыдущей главы. Публичное поощрение сотрудников, принявших и использующих новые шаблоны, стимулирует остальных последовать их примеру.

Во-вторых, можно автоматизировать закрепление нового процесса разработки с помощью непрерывной интеграции. Она позволяет запустить ряд процессов при отправлении нового коммита, установке пометки о готовности кода к рецензированию или перед объединением изменений с основной веткой. Обычно эти процессы сводятся к тестированию внесенных изменений, проводимых линтером и инструментами анализа кода. Сейчас мы рассмотрим, как это происходит, и поговорим о способах настройки этих инструментов, избавляющих команду рефакторинга от необходимости активно стимулировать и контролировать его внедрение.

Прогрессивный линтинг

Прогрессивный линтинг позволяет постепенно улучшать кодовую базу с помощью анализа недавно написанного или измененного кода. Благодаря ему разработчики могут решать проблемы по мере их появления, а не просить инженеров исправлять каждый случай. Если один шаблон заменен другим, написание нового правила для линтера — простой способ подтолкнуть разработчиков к использованию новым шаблоном.

Например, в рамках рефакторинга библиотеки журналирования нужно убрать ссылки на метод `logEvent`, позволяющий принимать произвольные строки, и заменить его методом `logEventType`, который регистрирует только определенные фрагменты данных. В этом случае для линтера можно написать новое правило, согласно которому при попытке использовать метод `logEvent` появляется сообщение, что метод устарел и вместо него нужно пользоваться методом `logEvent`.

Обязательно сообщите о цели введения нового правила, чтобы оно не стало неожиданностью. В сообщении об ошибке максимально точно опишите контекст происходящего, чтобы столкнувшиеся с этой ошибкой разработчики не искали дополнительную документацию.



Не во всех языках линтер позволяет разработчикам писать свои правила. Для прогрессивного линтинга еще меньше встроенных возможностей. Некоторые инженерные группы сами пишут такие инструменты для внутренних нужд компаний (и иногда выкладывают решения в открытый доступ). Но если у вас есть линтер, допускающий написание своих правил, быстрее всего ввести прогрессивный линтинг можно, запустив линтер только для измененных файлов в коммите или только для внесенных в код изменений.

Инструменты анализа кода

Динамику многих метрик из главы 3 можно отслеживать с помощью инструментов анализа кода, запускаемых во время интеграции. Есть широкий спектр бесплатных и платных решений с открытым исходным кодом, которые автоматически рассчитывают сложность кода в разных масштабах (отдельные функции, классы, файлы и т. п.) и генерируют статистику тестового покрытия. Многие из этих решений расширяемы. Это позволяет добавлять свои варианты метрик и новые правила.

Допустим, нужно, чтобы ни одна функция в базе кода не превышала 500 строк. Выбранный инструмент анализа кода настраивается выдавать предупреждение или ошибку, когда внесенное изменение приводит к превышению этого порога. Если кто-то добавляет к существующей функции несколько строк, увеличивая их количество с 490 до 512, его заставляют разделить функцию на более мелкие подфункции. И только после этого можно выполнить слияние изменений.

Ворота и ограждения

Каждый шаг проверки в потоке интеграции можно сравнить с воротами, замедляющими движение, или с ограждением, на котором висит предупреждение для автора кода.

Когда ворот слишком много, это замедляет разработку и сбивает с толку (особенно если они возникают неожиданно). Представим десять наборов

блокирующих тестов, параллельно запускающихся при появлении подлежащего проверке кода. Выполнение примерно половины этих наборов занимает больше десяти минут, а некоторые из них часто дают нестабильные результаты. Приходится тратить драгоценное время, ожидая, пока код пройдет через все десять ворот.

Теперь допустим, что вместо ворот установили ограждения. Если раньше каждый из этих наборов тестов блокировал прогресс, теперь команда решает, какие из них критически важны для бизнеса, а остальные помечает как необязательные. В итоге разработчики сами ответственны за выбор тестов и могут игнорировать те, что дают нестабильные результаты. Конечно, такой сценарий сопряжен с риском и может привести к появлению в производственной среде множества ошибок. Но в целом я считаю, что разработчикам нужно доверять больше.

АВТОРСТВО КОДА

Многих разработчиков в крупных компаниях (в том числе и меня) часто спрашивают о коде, который они редактировали последними. Как человек, который участвовал в нескольких крупномасштабных рефакторингах и отредактировал большой объем кода, я стараюсь сохранять историю версий. Но команда `git blame` часто выдает мое имя с функциями, о которых я мало что знаю. В результате меня вовлекали в разбор инцидентов, отмечали в тикетах в системе JIRA и назначали для проверки кода с почти незнакомым контекстом, тем самым разочаровывая людей, которым была нужна моя помощь. К счастью, обычно мы можем верно определить команду, отвечающую за конкретный код. Но бывают и случаи, когда код никому не принадлежит.

Такой бесхозный код — острые проблема, с которой сталкивается почти каждая компания с большой командой разработчиков. Что же делать, если на вас или на вашу команду укажет команда `git blame`? Возможно, что вы будете рады помочь, но лучше не создавать таких прецедентов и не браться за разгребание проблем в чужом коде. Ведь потом может оказаться, что за вами закрепили его авторство. Кроме того, к вам будут подходить люди, которым тоже нужна помощь с кодом без владельца.

Поэтому на все обращения сразу отвечайте, что ваша команда не ответственна за этот код. Предложите вместе поискать лучшего кандидата (или попросите о помощи у руководства). Если запрос небольшой (например, простое исправление ошибки) и срочный, возможно, вам повезет найти того, у кого есть время и достаточное знакомство с кодом, чтобы сразу заняться решением этой задачи. Попробуйте поискать в истории версий, кто редактировал этот код до вас, или определить разработчика (и команду) с последними коммитами в том же файле.

Как только найдется человек, согласный обработать запрос немедленно, попросите своего начальника найти для этого кода новых владельцев среди своих коллег. Скорее всего, код будут кидать друг другу, как горячую картошку, но надеюсь, что к этому моменту вас это касаться уже не будет.

Интеграция улучшений в культуру

Потребность в рефакторинге будет всегда — невозможно предсказать, как на системы повлияют изменения в технологиях или требованиях. Но я считаю, что иногда рефакторинга можно избежать и что важно прилагать все усилия, чтобы предотвратить такую необходимость. В заключение этой главы я хочу поделиться мыслями о том, как создать культуру постоянного совершенствования. Если всегда искать возможности для ощутимого улучшения кода и использовать их, можно прожить какое-то время без проведения большого рефакторинга.

Лучший способ поддерживать здоровье кодовой базы — по возможности проводить рефакторинг небольших, хорошо изолированных фрагментов кода. Конечно, не нужно проводить рефакторинг всего подряд (я писала об этом в подразделе «Вы случайно проходили мимо» главы 2). Сосредоточьтесь на постепенном улучшении областей кодовой базы, которые поддерживает наша команда.

Навести порядок в своей области можно всегда. Если же мы видим, что улучшить код может другая команда, с ними лучше поговорить и попытаться понять причину проблем, а не сразу предлагать решение. Такое сотрудничество способствует более чистой реализации.

Старайтесь часто обсуждать проектные решения в команде, делая упор на обратную связь от других. Рецензирование кода — не только возможность с чужой помощью найти ошибки, но и база для открытого обсуждения и совместного поиска лучших решений. Авторам отправляемого на рецензирование кода лучше добавить к нему аннотации с конкретными вопросами. К рецензированию чужого кода нужно подходить так же вдумчиво, как и к написанию своего.

Рецензирование проектных решений должно проводиться на ранних этапах разработки функционала. Для оценки и обсуждения проектов нужно приглашать разработчиков любого уровня. Чем разнообразнее будет опыт и стаж рецензентов, тем больше точек зрения удастся собрать и тем выше будут шансы обнаружить фатальные недостатки на раннем этапе, устраниТЬ их и получить лучшее решение.

Каждый раз, приступая к работе, критически обдумывайте, может или не может то, что вы делаете сегодня, в будущем стать причиной большого рефакторинга. Иногда достаточно небольшого напоминания о возможных долгосрочных последствиях принятых решений, чтобы вернуться в правильное русло.

ЧАСТЬ IV

Разборы примеров

Прежде чем показать вам разбор первого примера, я хотела бы рассказать, что такое Slack. Для лучшего понимания материала желательно знакомство с историей продукта и компании.

Мессенджер Slack был разработан для внутреннего использования небольшой компанией Tiny Speck из Ванкувера. Команда, состоящая из инженеров, дизайнеров и специалистов по продуктам из Flickr, хотела создать фантастическую многопользовательскую онлайн-игру Glitch, ориентированную на создание сообщества.

Сотрудники Tiny Speck были разбросаны по Северной Америке, поэтому сначала для общения применялась система IRC (Internet Relay Chat, ре-транслируемый интернет-чат). Вскоре стало ясно, что нужно что-то мощнее: инструмент, позволяющий поддерживать связь в асинхронном режиме, выполнять поиск в истории сообщений и отправлять файлы. Его было решено создать своими силами.

В 2012-м игра закрылась из-за недостаточной популярности, и компания уволила большинство сотрудников. Но у них оставался один последний козырь. Те, кто остался, решили ввести в коммерческое обращение инструмент внутренней коммуникации. Они отшлифовали интерфейс и дали мессенджеру название *Slack: searchable log of all conversation and knowledge* (журнал всех разговоров и знаний с возможностью поиска).

Протестировать новый инструмент команда Tiny Speck попросила друзей и бывших коллег. Собирая отзывы каждой новой группы пользователей, они исправляли ошибки и создавали новый функционал. К маю 2013 года была готова пробная версия продукта, доступная немногим избранным, которые об этом попросили. Девять месяцев спустя мессенджер Slack представили широкой публике.

Всего за год количество его активных пользователей увеличилось с 15 000 до 500 000. Когда продукту исполнилось два года, ежедневно им пользовалось более 2,3 миллиона человек. Спустя почти шесть лет после запуска их число превысило 12 миллионов. При этом еженедельно отправлялось более миллиарда сообщений.

Многие ранние технологические и проектные решения Slack были основаны на опыте создания фотохостинга Flickr и игры Glitch. Поэтому разработчики мессенджера использовали привычные им PHP и MySQL. Фактически большинство основных серверных функций Slack уходит корнями в Framework —

PHP-фреймворк для веб-приложений, послуживший основой для процессов и фирменного стиля Flickr. Этот фреймворк можно найти на сайте GitHub. Большая часть инфраструктуры обмена онлайн-сообщениями была взята из IRC-подобного внутреннего инструмента компании Tiny Speck.

В начале 2016 года проект Slack начал искать альтернативы PHP-движку Zend Engine II. На эту роль было два основных претендента. Можно было перейти на PHP 7 и использовать движок Zend Engine III или попробовать виртуальную машину HipHop (HHVM) от Facebook. После некоторых раздумий руководство решило развернуть на своих серверах среду выполнения HHVM. Развёртывание прошло успешно, и разработчики перешли на PHP-подобный язык программирования Hack для виртуальной машины HHVM. На момент публикации этой книги часть кодовой базы Slack, которая когда-то была написана на PHP, была переписана на Hack.

Примеры в этом разделе посвящены рефакторингу части кодовой базы, написанной на PHP, а затем на Hack. Чтобы как можно нагляднее отразить суть проблемы в каждом случае, все фрагменты кода даны на языке Hack. Ничего страшного, если вы с ним не знакомы! Приведенный для примера код — не основной предмет повествования. В первую очередь я хочу показать вам сам процесс рефакторинга и задействованных в нем людей.

Очень надеюсь, что материал этой и следующей глав покажет именно этот аспект. Если вы беспокоитесь, что не сможете понять и проанализировать примеры кода, могу заверить, что приведенная в книге версия кода Hack во многом напоминает код на PHP. Для тех, кто не знаком ни с одним из этих языков, я подробно рассмотрю каждый фрагмент. Надеюсь, это поможет сориентироваться в происходящем.

Еще хотелось бы упомянуть, что на момент публикации Slack был доступен широкой публике всего шесть лет. Это молодой продукт, и его код требовалось быстро масштабировать, чтобы справиться с растущим числом клиентов. При этом разработчиков в компании тоже прибавилось. Большой рефакторинг, который пришлось провести компании, во многом был ответом на быстрый рост проекта, обусловленный внешними и внутренними причинами.

ГЛАВА 10

Избыточные схемы базы данных

В этой главе я расскажу про рефакторинг, в котором я участвовала на первом году работы в проекте Slack. Перед нами стояла задача объединить две избыточные схемы базы данных. Обе были тесно связаны с нашей растущей кодовой базой. При этом у нас было очень мало надежных модульных тестов. Словом, это отличный пример крупномасштабного рефакторинга в молодой развивающейся компании с небольшим числом сотрудников и с разрастающейся кодовой базой.

Успех этого проекта был обусловлен в основном нашей сосредоточенностью на конечной цели — объединения избыточных таблиц базы данных. Мы составили простой и эффективный план выполнения (по методике из главы 4), тщательно взвесив риски и оценив реальную скорость реализации. Мы выбрали упрощенный подход к сбору показателей (о них я говорила в главе 3), сосредоточившись только на основных данных. После завершения каждого этапа информация о внесенных изменениях доносилась до всех членов инженерной группы (методы информирования см. в главе 7). Мы всячески содействовали сохранению изменений (методами из главы 9). Наконец, мы успешно показали значение рефакторинга, предоставив новый функционал всего через несколько недель после внедрения объединенной схемы. Это обеспечило поддержку нашего следующего проекта рефакторинга (см. главу 5).

Рефакторинг дал желаемое улучшение производительности, но мы допустили несколько ошибок. Сильное давление со стороны самых важных клиентов заставило нас слишком быстро двигаться вперед. Мы не узнали, почему появились несовпадающие схемы, не снабдили другие команды письменной копией плана выполнения (см. главу 4). Мы самостоятельно выполнили большую часть работы, не прибегая к помощи специалистов (как в главе 5).

Но несмотря на все попытки сохранить запланированный темп, в последние недели рефакторинг затянулся (глава 8).

Перед изложением подробностей рефакторинга я расскажу, зачем нужен Slack и как он работает. Тем, кто пока не сталкивался с этим продуктом, рекомендую внимательно прочитать этот раздел. Постоянные пользователи Slack могут смело переходить к разделу «Архитектура Slack 101».

Slack 101

Прежде всего Slack — это инструмент для совместной работы. Он подходит компаниям любого размера и отрасли. Обычно фирма настраивает рабочее пространство Slack и создает учетные записи для всех сотрудников. После загрузки приложения (на ПК или мобильный телефон) можно сразу начать общение с коллегами.

Темы и беседы распределены по каналам. Допустим, разрабатывается новый функционал, ускоряющий загрузку файлов в приложение. Назовем этот проект Faster Uploads («Более быстрая загрузка»). Создаем канал `#feature-faster-uploads` для согласования разработки между членами команды, руководством и менеджером по продукту. Любой, кому интересно узнать о ходе этой разработки, может перейти по тегу `#feature-faster-uploads` и прочитать недавнюю историю или присоединиться к беседе и напрямую задать вопросы членам команды.

На рис. 10.1 показан интерфейс Slack начала 2017 года, примерно во время описываемых в этой главе событий.

Здесь показан мессенджер сотрудника Acme Sites Мэтта Кампа (Matt Kump). В верхнем левом углу мы видим название рабочего пространства и сразу под ним имя Мэтт.

На левой боковой панели перечислены все каналы Мэтта. Мы пока проигнорируем раздел `starred` и сосредоточимся на разделе `Channels`. Из списка этого раздела видно, что Мэтт участвует в обсуждениях бухгалтерских издержек (`#Accounting-Cost`), мозгового штурма (`#brainstorming`), работы компании (`#business-ops`) и некоторых других. Все эти каналы может найти любой, у кого есть учетная запись Acme Sites, посмотреть их содержимое и присоединиться к ним.

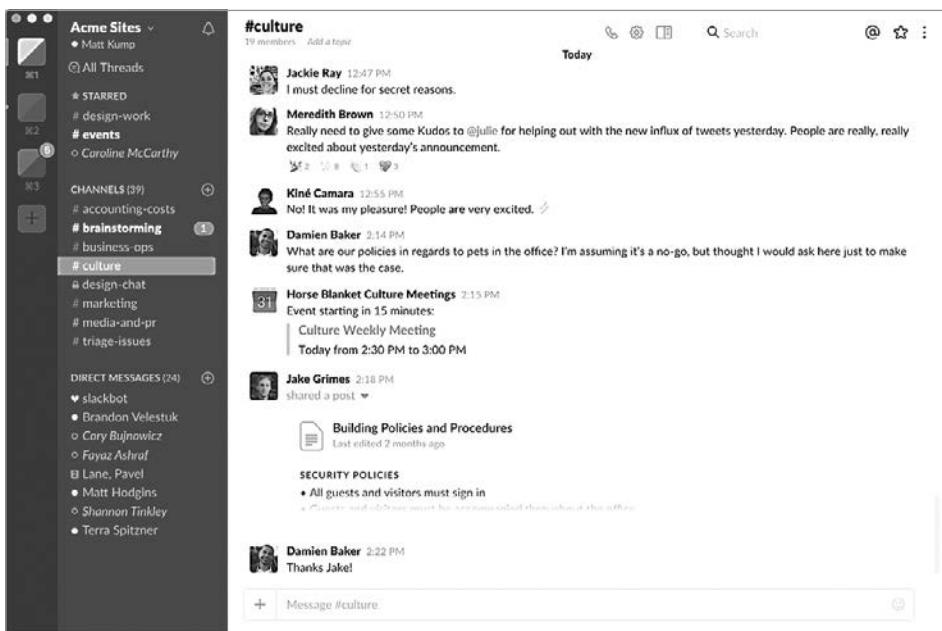


Рис. 10.1. Интерфейс Slack в январе 2017 года

Обратите внимание, что у канала `#design-chat` вместо символа `#` замок. Это частный канал, содержимое которого доступно только его участникам. Для присоединения к нему нужно приглашение от кого-то из участников.

Нижний список на этой панели — личные сообщения (Direct Messages). Мы видим, что он переписывается с коллегами: Брэндоном, Кори и Фаязом. Еще он принимает участие в групповом чате с Лейном и Павлом. Чат работает по тому же принципу, что и личные сообщения, но в нем может участвовать несколько человек.



Понимание различий общедоступного и частного каналов понадобится, когда мы перейдем к обсуждению основных проблем, которые пытался решить рефакторинг.

Названия некоторых каналов выделены ярко-белым цветом. Это означает наличие непрочитанных сообщений. При выборе на панели строчки `#brainstorming` открывается доступ к ним, а название канала тускнеет.

Конечно, у Slack еще много возможностей, но этого вполне достаточно для понимания материала следующих разделов.

Архитектура Slack 101

Теперь рассмотрим компоненты архитектуры Slack, лежащие в основе нашего исследования. Важно отметить, что некоторые из них сильно изменились, и не только из-за рефакторинга, так что современная архитектура Slack немного отличается от описанной в этой главе.

На рис. 10.2 показан результат простого запроса на получение истории сообщений одного из моих любимых каналов `#core-infra-sourdough`, где сотрудники компании обсуждают выпечку хлеба.

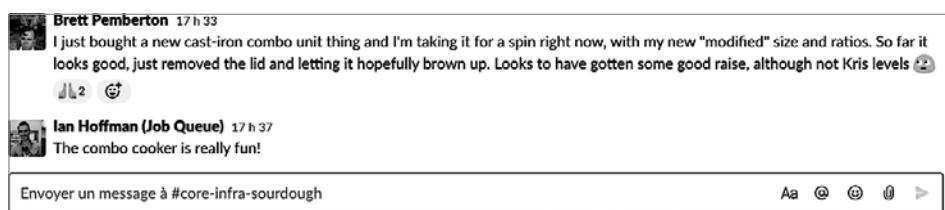


Рис. 10.2. Последние советы по выпечке хлеба на канале `#core-infra-sourdough`

Если посмотреть сетевой трафик, обнаружится запрос GET к Slack API для метода `channels.history` с идентификатором канала `#core-infra-sourdough`. Сначала этот запрос попадет в подсистему балансировки нагрузки, а потом на доступный сервер, который проверит несколько вещей: действительность предоставленного токена и мое право на доступ к указанному каналу. Если проверка пройдена, сервер извлечет последние сообщения из соответствующей базы данных, отформатирует их и вернет клиенту. За несколько миллисекунд я получу последние сообщения на выбранном канале.

Как сервер узнал, к какой базе данных обращаться? Внутри мессенджера все относится к единому рабочему пространству. Все сообщения содержатся в каналах, а каналы — в рабочем пространстве. Всего один логический элемент дает нам удобный способ шардинга — горизонтального распределения данных.

Каждое рабочее пространство было закреплено за одним шардом базы данных, где хранилась вся необходимая информация. Если участник рабочего пространства хочет получить список всех общедоступных каналов, серверы делают запрос, чтобы узнать, какой шард содержит данные указанного рабочего пространства, а затем запрашивают соответствующий шард для каналов.

Если крупный клиент, занимающий шард вместе с другими компаниями, сильно вырос, компании перераспределяются по другим шардам, давая растущему клиенту больше места для маневра. Если растет единственный «обитатель» шарда, мы обновляем аппаратное обеспечение, чтобы приспособиться к этому росту. Пример структуры нашей базы данных показан на рис. 10.3.



Рис. 10.3. Рабочие пространства, распределенные по шардам

Посмотрим на способы хранения основных фрагментов информации в каждом шарде рабочего пространства. Рассмотрим *каналы* и *членство* в них. В начале 2017 года в Slack за хранение информации о каналах отвечало несколько таблиц. Таблица `team_channels` хранила информацию для общедоступных каналов, а `groups` — информацию для частных каналов и чаты (сообщения, отправляемые нескольким пользователям). Каждая из таблиц содержала сведения о названии канала, дате его создания и о его создателе. Примеры таблиц показаны на рис. 10.4.

Информация об участниках каналов хранилась в таблицах `team_channels_members` и `groups_members`. Каждому участнику соответствует строка, однозначно распознаваемая по комбинации идентификатора рабочего пространства, идентификатора канала и идентификатора пользователя. Сохранены и дополнительные сведения о членстве пользователя: дата присоединения к каналу и unix-время его последнего появления на канале. Из рис. 10.5 видно, что эти таблицы почти идентичны.

| teams_channels | | | | |
|----------------|----------------|-----------------------|-----|--|
| id | team_id | Название | ... | Назначение |
| 10001 | 20001 | sourdough-baking | | Обсуждение успехов и неудач в хлебопечении |
| 10002 | 20001 | company-announcements | | Важные сообщения компании |

| groups | | | | |
|-----------|----------------|--------------------|-----|---|
| id | team_id | Название | ... | Назначение |
| 10007 | 20001 | secret-acquisition | | Обновления, связанные с потенциальными приобретениями |
| 10008 | 20001 | eng-promotions | | Комитет по продвижению инженеров |

Рис. 10.4. Упрощенная схема таблиц teams_channels и groups

| teams_channels_members | | | | |
|------------------------|----------------|----------------|-----|------------------|
| channel_id | team_id | user_id | ... | last_read |
| 10001 | 20001 | 30001 | | 1553656778 |
| 10002 | 20001 | 30002 | | 1553658978 |
| 10003 | 20001 | 30003 | | 1553688978 |
| 10004 | 20001 | 30004 | | 1553658978 |

| group_members | | | | |
|-------------------|----------------|----------------|-----|------------------|
| channel_id | team_id | user_id | ... | last_read |
| 10007 | 20001 | 30001 | | 1553656778 |
| 10008 | 20001 | 30002 | | 1553658978 |
| 10008 | 20001 | 30003 | | 1553688978 |
| 10009 | 20001 | 30004 | | 1553658978 |

Рис. 10.5. Упрощенная схема таблиц teams_channels_members и groups_members

Наконец, для хранения личных сообщений применяется таблица `team_ims` (см. рис. 10.6), содержащая информацию о самом канале и о членстве в нем.

| teams_ims | | | | | |
|-------------------|----------------|----------------|----------------|-----|------------------|
| channel_id | team_id | user_id | user_id | ... | last_read |
| 10007 | 20001 | 30001 | 30003 | | 1553656778 |
| 10008 | 20001 | 30002 | 30001 | | 1553658978 |
| 10008 | 20001 | 30003 | 30003 | | 1553688978 |
| 10009 | 20001 | 30004 | 30001 | | 1553658978 |

Рис. 10.6. Упрощенная схема таблицы teams_ims

Итак, три таблицы использовались для хранения информации о каналах, а еще три таблицы — для хранения информации о членстве в каналах. На рис. 10.7 показана роль каждой таблицы в зависимости от типа канала, с которым она работает.

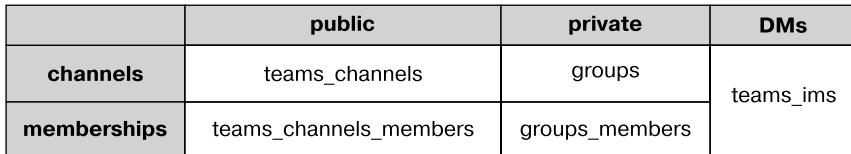


Рис. 10.7. Диаграмма, показывающая, какие таблицы отвечают за хранение информации о канале и членстве в нем, в зависимости от типа канала (общедоступный, частный, групповая переписка или личная переписка)

Проблемы масштабирования

Теперь вы знаете о базовой архитектуре Slack и о способах представления каналов и членства в них. Можно поговорить о проблемах в процессе работы. Я опишу три проблемы, с которыми столкнулся наш крупнейший на тот момент клиент. Назовем его очень большим бизнесом (Very Large Business), далее — VLB.

Этот клиент очень хотел, чтобы все 350 000 его сотрудников использовали Slack. Он начал активно наращивать использование нашего продукта в течение первых месяцев 2017 года. К апрелю число пользователей перевалило за 50 000. Это почти вдвое превысило показатели нашего второго по величине клиента. Постепенно они начали упираться во все ограничения. Я тогда входила в команду, отвечавшую за работу с крупными клиентами. Несколько недель мы устраивали дежурства: в 6:30 утра двое должны были сидеть в штаб-квартире в Сан-Франциско и быть готовы реагировать на любые неотложные запросы. Как раз в это время в VLB, которая находилась на Восточном побережье, нашим продуктом пользовались интенсивнее всего. Мы оперативно реагировали на возникающие проблемы и постепенно начали замечать, что каждая из них усугублялась из-за избыточных таблиц базы данных.

Загрузка клиента Slack

Каждый будний день с девяти утра по восточному времени сотрудники VLB начинали входить в Slack. Постепенно росла нагрузка на шард базы данных, выделенный под VLB. Наши инструменты показывали, что виновником, вероятно, был метод `rtm.start` — один из важнейших API, вызываемых при запуске мессенджера. Именно он возвращает информацию для заполнения боковой панели пользователя. Он извлекает все общедоступные и частные каналы пользователя, все его чаты и личные сообщения и определяет наличие непрочитанных. Клиент анализирует эту информацию и заполняет свой интерфейс аккуратным списком.

Для сервера это очень ресурсоемкий процесс. Чтобы определить, на какие каналы подписан пользователь, требовалось сделать запрос к трем таблицам: `team_channels_members`, `groups_members` и `team_im`s. Из каждого ответа мы извлекали `channel_id` и выбирали соответствующую строку `team_channels` или `groups` для отображения имени канала. Мы делали запрос к таблице `messages` для получения временной метки самого последнего сообщения и сравнения ее с временной меткой пользователя `last_read`. Это позволяло определить наличие у него непрочитанных сообщений. Большинство этих запросов выполнялось по отдельности — на каждый мы расходовали ресурсы сервера.

Видимость файла

В течение дня мы иногда замечали всплески ресурсоемких запросов к базе данных. Мы определили несколько возможных мест происхождения таких запросов. Например, функцию определения видимости файлов, лежавшую в основе большинства связанных с файлами API. Внутри нее мы обнаружили набор сложных запросов.

Когда пользователь загружает файл в Slack, серверы записывают в таблицу `files` новую строку с именем файла, его местоположением на удаленном файловом сервере и другой важной информацией. При каждой передаче файла в канал в таблице `files_share` появляется новая запись с идентификаторами файла и канала, которому он был предоставлен. Файл, переданный на общий канал, становится видимым для любого пользователя в рабочем пространстве и помечается как общедоступный — на пересечении столбца `is_public` и строки `files` появляется значение `true`. Это простейший случай, когда информация о видимости файла получается быстро и пользователю дается доступ к нему.

В случае с закрытыми файлами логика усложняется. Нужно сопоставить все каналы, в которых участвует пользователь, со всеми каналами, на которых доступен файл. Здесь, как и в случае с методом `rtm.start`, снова нужно делать запрос к трем таблицам. Результаты объединяются с ответом на запрос к таблице `files_shares` для целевого файла. В случае совпадения пользователь получает доступ к файлу, в противном случае видит сообщение об ошибке.

Упоминания

Чаще всего нагрузку на шард VLB в течение рабочего дня вызывал запрос, определяющий, есть ли в канале не прочитанные пользователем упоминания его самого (или тем, на которые он подписан). *Упоминанием* (mention) в Slack может стать многое: имя пользователя, имя пользователя с префиксом @ или даже слово, указанное им в предпочтениях. Ответ на этот запрос клиент применяет для заполнения сведений о числе непрочитанных упоминаний справа от имени соответствующего канала на боковой панели. Вариант такого сложного запроса отражен в примере 10.1.

Этот запрос опять требует обращения к трем таблицам для получения информации о членстве пользователя. Сложность в том, что нужно исключить

любое членство в уже удаленных или архивированных каналах. Для этого полученные результаты объединялись с данными о членстве из строки соответствующего канала таблицы `groups` или `teams_channels`.

Пример 10.1. Запрос, определяющий, нужно ли сообщать пользователю об упоминании. После знака % подставляется соответствующее значение

```
SELECT
    tcm.channel_id as channel_id,
    'C' as type,
    tcm.last_read
FROM
    teams_channels tc
    INNER JOIN teams_channels_members tcm ON (
        tc.team_id = tcm.team_id
        AND tc.id = tcm.channel_id
    )
WHERE
    tc.team_id = %TEAM_ID
    AND tc.date_delete = 0
    AND tc.date_archived = 0
    AND tcm.user_id = %USER_ID
UNION ALL
SELECT
    gm.group_id as channel_id,
    'G' as type,
    gm.last_read
FROM
    groups g
    INNER JOIN groups_members gm ON (
        g.team_id = gm.team_id
        AND g.id = gm.group_id
    )
WHERE
    g.team_id = %TEAM_ID
    AND g.date_delete = 0
    AND g.date_archived = 0
    AND gm.user_id = %USER_ID
UNION ALL
SELECT
    channel_id as channel_id,
    'D' as type,
    last_read
FROM
    teams_ims
WHERE
    team_id = %TEAM_ID
    AND user_id = %USER_ID
```

Консолидация таблиц

Вы узнали, с какой проблемой мы столкнулись. Теперь пришло время поговорить о рефакторинге. Мне очень хотелось бы рассказать, что проект консолидации таблиц `team_channels_members` и `groups_members` был хорошо спланированным и грамотно выполненным. Но, к сожалению, это не так. Более того, самые хаотичные фрагменты рефакторинга стали источником идей и вдохновения для написания этой книги. Мы приступили к работе с чувством, что ее нужно сделать как можно быстрее. Мы не отслеживали свой прогресс. И хотя мы знали, что снизили нагрузку на большую часть базы данных, у нас была всего одна метрика для количественной демонстрации этого. В итоге проект привели к успеху умные, преданные делу люди. И хотя самую большую выгоду от рефакторинга получил крупный клиент, все остальные тоже почувствовали улучшения.

Мы начали проект без письменного плана. Основной целью была консолидация таблиц, которая позволила бы изменить запрос, больше всего загружавший шарды, — запрос упоминаний.

Конечно, объединение таблиц выгодно и для множества других запросов, но их миграция была вторичной. В главе 1 я настоятельно не рекомендовала вам начинать крупномасштабный рефакторинг, если вы не уверены, что сможете его завершить. Здесь же мы были уверены в том, что завершим консолидацию таблиц, но не знали, не появятся ли более неотложные проблемы с производительностью, которые выйдут на первый план. Учитывая срочность проблемы, было решено пойти на риск. Мы прекрасно осознавали, что будет, если нам не удастся завершить миграцию.

Сначала мы создали новую таблицу `channels_members`, объединили схемы таблиц с информацией о членстве, сохранив все индексы и добавив столбец для указания того, откуда взялась строка — из таблицы `team_channels_members` или `groups_members`. Это должно было упростить миграцию и обеспечить соблюдение любых зависимостей бизнес-логики, связанных с исходными таблицами. Целевая таблица показана на рис. 10.8, а начальное состояние — на рис. 10.7.

| | public | private | DMs |
|--------------------|------------------|----------------|------------------------|
| channels | teams_channels | groups | <code>teams_ims</code> |
| memberships | channels_members | | |

Рис. 10.8. Целевое состояние

Сбор разрозненных запросов

Переписать запросы так, чтобы они обращались к одной новой таблице, было непросто. Кодовая база Slack была написана в очень императивном стиле. Все от коротких до длинных функций было распределено по сотням файлов с вольными именами. Первоначальные авторы придерживались хорошо знакомых им вещей и избегали объектно-ориентированных шаблонов из-за проблем с производительностью PHP. Они предпочли встраивать индивидуальные запросы в код, не полагаясь на библиотеку объектно-реляционного отображения, с риском быстро раздуть кодовую базу.

Единичные запросы к таблицам `team_channels_members` и `groups_members` были разбросаны по 126 файлам. Многие из них появились еще до запуска продукта. В довершение всего большая часть кода, содержащего эти запросы, не имела достаточного покрытия модульными тестами. Я специально нашла старый код, чтобы вы смогли понять, как все это выглядело (пример 10.2).

Пример 10.2. Встроенный SQL-запрос к таблице `teams_channels_members`

```
function chat_channels_members_get_display_counts(
    $team,
    $user,
    $channel
){
    // Какая-то бизнес-логика

    $sql = "SELECT
        COUNT(*) as display_counts,
        SUM(CASE
            WHEN (is_restricted != 0 OR is_ultra_restricted != 0)
                THEN 1
            ELSE 0
        END) as guest_counts
    FROM
        teams_channels_members AS tcm
        INNER JOIN users AS u ON u.id = tcm.user_id
    WHERE
        tcm.team_id = % team_id
        AND tcm.channel_id = % channel_id
        AND u.deleted = 0";
    $ret = db_fetch_team($team, $sql, array(
        'team_id' => $team['id'],
        'channel_id' => $channel['id']));
}

// Еще немного бизнес-логики

return $counts;
}
```

Код бизнес-логики был тесно связан со схемами базы данных. При каждом создании нового столбца приходилось обновлять соответствующий фрагмент кода. Вспомним столбец `is_public` из таблицы `files`, где указывалось, общедоступен ли файл. Представим, что мы добавили новую логику, и определение прав доступа к файлу стало требовать проверки дополнительного свойства. После этого нужно было обновлять любой код, проводивший простую проверку `if ($file['is_public'])`, чтобы добавить туда новое условие.

Для консолидации таблиц `teams_channels_members` и `groups_members` в таблицу `channels_members` нужно было определить все запросы к обеим таблицам, разбросанные по базе кода. Утилита `grep` дала нам список всех мест, где выполнялся запрос к таблицам `groups_members` и `team_channels_members`. Полученный список файлов и номера строк были вставлены в общий файл Google Sheets (рис. 10.9).

The screenshot shows a Google Sheets document with the title "Channel Membership". The spreadsheet has four columns: "File", "Call", "Assignee", and "Done". The "File" column lists various PHP files from a codebase. The "Call" column contains snippets of SQL code. The "Assignee" column lists names: Bradley, Dhruv, Maude, Eric, and WHEI. The "Done" column contains status markers: Y (Yes) or N (No). The rows correspond to the numbered items in the text below.

| File | Call | Assignee | Done |
|--|---|----------|------|
| 2 include/lib_chat_channels_shared_ext.php:693 | db_insert_dupe_team(\$team, 'teams_channels_members', array(| Bradley | Y |
| 3 x_export_team_channel_metrics_walmart_tech | \$sql = <<<EOQT | | N |
| 4 include/lib_chat_channels_shared.php:583 | \$sql = "SELECT members.team_id, members.channel_id, members.user_id Dhruv | Dhruv | Y |
| 5 include/lib_enterprises_channels_shared.php:2759 | \$sql = "SELECT user_id FROM teams_channels_members WHERE Maude | Maude | Y |
| 6 include/lib_enterprises_channels_shared.php:756 | \$ret = db_fetch_team(\$enterprise, "SELECT channel_id from teams_chann Bradley | Bradley | Y |
| 7 include/lib_enterprises_move_channels.php:1585 | Schannels_ret = db_fetch_team(\$team, "SELECT user_id, channel_id FRO Eric | Eric | Y |
| 8 include/lib_enterprises_users_merge.php:504 | \$ret = db_write_team(\$enterprise, "UPDATE teams_channels_members SE Bradley | Bradley | Y |
| 9 include/lib_team_consistency.php:841 | \$ret = db_fetch_team(\$team, "SELECT members.* FROM teams_channels Bradley | Bradley | Y |
| 10 include/lib_search.php:2532 | \$ret = db_fetch_team(\$team->cur, "SELECT channel_id FROM teams_cha Dhruv | Dhruv | Y |
| 11 include/lib_consistency_fix.php:626 | \$ret = db_fetch_team(\$team, "SELECT members.* FROM teams_channels Dhruv | Dhruv | Y |
| 12 include/lib_consistency_fix.php:877 | \$ret2 = db_fetch_team(\$team, "SELECT COUNT(*) FROM teams_channels Dhruv | Dhruv | Y |
| 13 include/lib_migrate_channels.php:1169 | \$ret = _migrate_channels_create_destination_channel_members('teams_c Eric | Eric | Y |
| 14 include/lib_enterprises_move_channels.php:1993 | \$sql = "UPDATE teams_channels_members SET user_id = %user_id WHEI Eric | Eric | Y |
| 15 tests/unit/test_migrate_channels.php:2257 | is_error_response_>_migrate_channels_create_destination_channel_membe Eric | Eric | Y |
| 16 tests/unit/test_migrate_channels.php:2277 | \$ret = _migrate_channels_create_destination_channel_members('teams_c Eric | Eric | Y |
| 17 include/lib_enterprises_move_channels.php:441 | \$ret = db_fetch_team(\$destination_team, "SELECT channel_id, user_id, 0 i Eric | Eric | Y |

Рис. 10.9. Файл Google Sheets для отслеживания запросов к таблицам `teams_channels_members` и `groups_members`

Мы решили создать единый файл для всех связанных с членством в каналах запросов. Примерно в то же время инженеры задумались о централизации запросов. Мы были растущей командой, которая пыталась работать быстро. Так что необходимость обновлять запросы в случайных местах кодовой базы после каждого редактирования таблицы очень утомляла. В итоге было рассмотрено несколько предложений, но предпочтение отдали хранению всех запросов к этой таблице в одном файле.

Конечно, были и сторонники подхода, позволявшего генерировать запросы с заданным набором параметров. Для этого нужно было создать более сложный уровень доступа к данным. Были и те, кто хотел сохранить возможность встраивания запросов в код. Мы решили, что для ограничения количества отдельных функций в новом файле создадим прототип минимального запроса. Новый шаблон было решено назвать *unidata*, или сокращенно *ud*, а целевой файл — *ud_channel_membership.php*.

Разработка стратегии миграции

С таблицей и набором подлежащих переносу запросов мы могли приступить к работе. В списке, сформированном утилитой *grep*, нужно было идентифицировать все запросы, которые вставляли/удаляли строки или обновляли значения. Для каждого запроса была создана соответствующая функция в библиотеке *unidata*. Каждая принимала параметр, указывающий, нужно ли выполнить запрос для таблицы *members_channels_members* или таблицы *groups_members*. Кроме того, присутствовала логика условного выполнения того же запроса для новой таблицы *channels_members*. Общая идея показана в примере 10.3.

Пример 10.3.

```
function ud_channel_membership_delete(
    $team,
    $channel_id,
    $user_id,
    $channel_type
){
    if ($channel_type == 'groups'){
        $sql = 'DELETE FROM groups_members WHERE team_id=%team_id AND
                group_id=%channel_id AND user_id=%user_id';
    }else{
        $sql = 'DELETE FROM teams_channels_members WHERE team_id=%team_id AND
                channel_id=%channel_id AND user_id=%user_id';
    }
    $bind = array(
        'team_id' => $team['id'],
        'channel_id' => $channel_id,
        'user_id' => $user_id,
    );
    $ret = db_write_team($team, $sql, $bind);

    if (feature_enabled('channel_members_table')){
        $sql = 'DELETE FROM channels_members WHERE team_id=%team_id AND
```

```
channel_id=%channel_id AND user_id=%user_id';
$double_write_ret = db_write_team($team, $sql, $bind);

if (not_ok($double_write_ret)){
    log_error("UD_DOUBLE_WRITE_ERR: Failed to delete row for
              channels_members for {$team['id']}->{$channel_id}-{$user_id}");
}
}

return $ret;
}
```

После успешного перемещения всех операций записи мы написали сценарий обратного заполнения для копирования данных обеих таблиц в нашу новую. Заметьте, что миграция операций записи была выполнена *перед* запуском обратного заполнения для обеспечения точности данных в новой таблице. Далее мы выполнили обратное заполнение всех данных о членстве для нашего рабочего пространства, а после в нерабочее время оперативно проделали то же для VLB. Мы трижды проверили, что за пределами новой библиотеки не осталось ошибочных записей ни в одной из таблиц. Но так как заданная инженерами проекта скорость была высока, нельзя исключать, что мы пропустили один-два запроса. У нас еще не было придумано механизмов, препятствующих члену другой команды без предупреждения добавить новый запрос. Поэтому для гарантии согласованности результатов заполнения с реальными данными мы письменно предупредили инженеров о будущем процессе (рис. 10.10). Еще был написан сценарий, который запускался вручную и выявлял любые несоответствия.



На некоторых снимках экрана в этой главе есть аббревиатура TS. Это сокращение от Tiny Speck — предыдущего названия компании Slack. Поэтому такие формулировки, как «доступно для TS», означают всего лишь разрешение на редактирование своего рабочего пространства.

После включения двойной записи для VLB мы внимательно следили за состоянием базы данных этого клиента. Строки `team_channels_members` и `groups_members` обновлялись очень часто. Каждый раз, когда пользователь читал новое сообщение, клиент отправлял запрос на обновление временной метки `last_read` пользователя в строке, указывающей его членство. А после добавления таблицы `channels_members` количество записей удвоилось. Мы целый день следили за трафиком, чтобы убедиться, что пропускной способности этого рабочего пространства хватит для обработки дополнительной нагрузки.

 Maude 2:43 PM
 Important Perf Update 

 In case you're one of those people that just casually goes through and reads all PHP webap commits, you may have noticed that we've been double writing updates to the `team_channels_members` and `groups_members` tables to a brand new  `channels_members` tab for TS and VLB

How are we doing this?

All writes to either of these tables are now done through the unidata channel membership libra Within this library, we check if a feature flag is set for your team and conditionally write all cha to the `channels_members` table as well. (You may have noticed fewer lines of raw SQL in `lib_chat_groups` and `lib_chat_channels`.)

Why do I need to care?

It is now *imperative* that any writes to either of these tables are done by calling the appropriate function within the unidata channel membership library in order to ensure correct state in all tables. We are currently powering mentions off of `channels_members` for both IBM and TS.

TL;DR If you're going to write a new SQL query to write (update, delete, whatever) to `teams_channels_members` or `groups_members`, use the appropriate `ud_channel_membership` function instead. 

Рис. 10.10. Объявление о начале двойной записи в новую таблицу

После синхронизации таблиц и двойной записи обновлений появилась возможность перейти к важнейшей задаче — миграции запросов, связанных с упоминаниями. Все новшества мы сначала внедряли для нашей команды и только потом для клиентов. Это была (и есть) типичная стратегия тестирования результатов нашей работы перед запуском в производство, будь то новый функционал, новый фрагмент инфраструктуры или, как в этом случае, повышение производительности. При таком подходе логично сначала перейти на бесплатные рабочие пространства и медленно продвигаться вверх по уровням оплаты, оставляя самых крупных клиентов, для которых производительность наиболее важна, напоследок. Но здесь перед нами стояла задача в первую очередь облегчить нагрузку на клиентов высшего уровня. Поэтому мы перевернули стратегию наоборот.

Сначала оптимизированные упоминания были включены для нашей команды. У нас было немного средств для автоматизированного тестирования, а фреймворк модульного тестирования не мог как следует проверять запросы. Поэтому оставалось полагаться на товарищей по команде, которые должны были выявить любые регрессии, прежде чем разрешить предостав-

ление новой версии запросов другим клиентам. Мы тщательно отслеживали каналы, посвященные сообщениям об ошибках. Через некоторое время новое поведение было включено для VLB.

Количественная оценка нашего прогресса

Мы знали о перегруженности наших баз данных. Их работоспособность оценивалась по уровню бездействия CPU. Обычно этот показатель был примерно 25 %, но регулярно опускался до 10 % и ниже. Это вызывало беспокойство, потому что резерва, чтобы справиться с внезапным увеличением нагрузки, было мало. Мы проверяли наш продукт с помощью загрузки, которую обеспечивал VLB, но не могли предугадать, что в следующий раз приведет к неожиданному росту использования баз данных.

Когда мы приступили к консолидации, у нас уже были параллельные проекты для оптимизации нагрузки. При этом множество рабочих потоков, дополнительная нагрузка из-за двойной записи, разные флюктуации и продолжающаяся разработка нового функционала не позволяли подтвердить эффективность рефакторинга с помощью коэффициента загрузки базы данных. Кроме того, данные мониторинга хранились всего около недели. Поэтому, чтобы использовать их после рефакторинга, нужно было делать снимки экрана и записывать показатели вручную, а на это у нас не было времени.

Было решено смотреть сначала на время запроса. Мы добавили к каждому запросу временные метрики, что позволило посмотреть рост их эффективности после рефакторинга. Много полезной информации дает утилита EXPLAIN PLAN, но ничто не сравнится с метрикой, показывающей длительность выполнения запроса с точки зрения сервера. Мы добавили в рабочее пространство переключатель функциональности, а потом в случайном порядке распределили трафик в пропорции 50/50. Это позволило вводить изменения более плавно и подтвердило, что для такого большого клиента можно было поднять производительность с помощью нового варианта запросов.

Через несколько часов мы снова проверили данные, чтобы убедиться, что новый запрос выполняется быстрее при любых условиях, как при средней, так и при пиковой загрузке базы данных. К счастью, метрики показали ускорение на 20 % (рис. 10.11)! Запрос к таблицам `teams_channels_members` и `groups_members` выполнялся примерно за 4,4 секунды. Запрос к таблице

`channels_members` в среднем выполняется примерно за 3,5 секунды. Так что с помощью объединения таблиц нам удалось сэкономить почти секунду (оба запроса слишком длинные, поэтому на диаграмме видны только первые несколько строк).

| 5 row(s), 7 column(s) | | | | | | |
|--|---------|-----|-----|-----|---------|----------------|
| sql | total | p50 | p95 | p99 | maximum | avg_time |
| <pre>SELECT mn.*,tcm.last_read,tcm.last_read_abs,mnpp.state as is_pushed FROM mentions_normalized mn JOIN teams_channels_members tcm ON (mn.team_id=%team_id AND mn.channel_type=%channel_type_c AND mn.user_id=%mentioned_user_id AND tcm.user_id=%mentioned_user_id AND mn.team_id=tcm.team_id AND mn.channel_id=tcm.channel_id AND mn.timestamp>tcm.date_joined*999999 AND mn.timestamp>tcm.last_read_abs AND mn.timestamp >= % field_binding_0) LEFT JOIN subteams_users su ON (mn.team_id=su.team_id AND mn.item_id=id) SELECT mn.*,cm.last_read,cm.last_read_abs,mnpp.state as is_pushed FROM mentions_normalized mn JOIN channels_members cm ON (mn.team_id=%team_id AND mn.user_id!=%mentioned_user_id AND cm.user_id=%mentioned_user_id AND mn.team_id=cm.team_id AND mn.channel_id=cm.channel_id AND mn.timestamp>cm.date_joined*999999 AND mn.timestamp>cm.last_read_abs AND mn.timestamp >= % field_binding_2) LEFT JOIN subteams_users su ON (mn.team_id=su.team_id AND mn.item_id=su.subteam_id AND su.user_id=%mentioned_user_id AND SELECT</pre> | 4043474 | 2 | 11 | 37 | 3092 | 4.404268458261 |
| | 3967120 | 1 | 9 | 31 | 3657 | 3.511519943939 |

Рис. 10.11. Временные данные запроса упоминаний для VLB

Рефакторинг помог решить самую большую проблему. Подтверждение этого позволило нам обосновать необходимость дальнейшей консолидации. Мы вернулись к нашему перечню Google Sheet и начали рассылать инженерам оповещения о рефакторинге запросов на чтение.

Попытка получить помощь

К сожалению, получить помошь для завершения миграции было сложно. Все члены команды были загружены множеством неотложных дел, и никто не хотел выделять несколько часов в день на тщательное извлечение запросов. Ко всему прочему большая часть кода, связанного с этими запросами,

была плохо протестирована. Это делало несложную по своей сути миграцию опасной. Коллеги не хотели тратить на это время.

Я подумывала привлечь к работе несколько ориентированных на производительность разработчиков из других отделов, но в итоге решила продолжать самостоятельно, изредка прибегая к помощи ближайших коллег. Поскольку работа предстояла рискованная и не особенно интеллектуальная, я сомневалась, что кого-то можно будет уговорить внести свой вклад. Думая об этом сейчас, я понимаю, что если бы я постаралась быть убедительнее и распределила работу более равномерно, то мы сэкономили бы несколько недель.

Когда всего несколько недель спустя прогресс замедлился до практически полной остановки, я попыталась подкупить команду печеньем (рис. 10.12). Хотя есть более традиционные варианты мотивации коллег (см. главу 8), иногда лучшим стимулом становится еда.

June 26th, 2017

Maude 3:31 PM
Status of unidata channel membership!

We're SO CLOSE.

| User | Instances | Files |
|----------|--------------|---------|
| @bradley | 14 instances | 6 files |
| @ericv | 12 instances | 3 files |
| @johan | 11 instances | 5 files |
| @Pedro | 14 instances | 4 files |

I've assigned some more to myself and a few to @Dhruv Luthra to take a look at once he's wrapped up his current work 😊

If you each did 1 file per day, we will be done before Wednesday of next week! How awesome would that be!? (edited)

Maude 3:32 PM
I'm out for a few days next week and working from my parents' place next Thursday/Friday but if you all finish by next Friday, I will make a large batch of cookies 😊

Maude 3:37 PM
Also @kevin feel free to add your name to a few 😊

Maude 3:45 PM
Vote for your preferred baked good here:
 for brownies
 for cookies (chocolate chip oatmeal)
 for raspberry cookies
 for chocolate chip rice krispies

10 replies Last reply 3 years ago

Рис. 10.12. Попытка подкупа

Сообщения об успехах

Члены нашей команды работали над множеством проектов, но мы нуждались в поддержке друг друга. Мы помогали друг другу при проверке кода, при устранении серьезных ошибок и при проверке состояния дел. Поэтому проблемы с производительностью регулярно обсуждались в общедоступных каналах (часто в канале нашей команды). Еще мы проводили еженедельные встречи для обсуждения достижений и препятствий. Мне это помогало выяснить, какой процент запросов оставался в кодовой базе, и обсудить все обнаруженные ошибки и несоответствия.

О завершении очередного этапа (включение двойной записи в нашем рабочем пространстве или добавление нового варианта запросов, связанных с упоминаниями в VLB) объявлялось как в канале нашей команды, так и в нескольких технических каналах. Чем больше инженеров узнавало о внесенных изменениях, тем лучше! Ведь это уменьшало вероятность того, что инженер из чужой команды сделает запрос к любой из таблиц, над депрекацией которых мы активно работаем, не обращаясь к новой библиотеке. А поскольку мы сортируем поступающие от клиентов ошибки, это также означало, что инженеры могли эффективнее выделить и решить связанную с этими ошибками проблему.

БОРЬБА С РАЗДРАЖАЮЩЕЙ ОШИБКОЙ

Ни один крупный рефакторинг не обходится без пары раздражающих ошибок. Наш случай не исключение. Примерно через месяц после переноса в новую библиотеку запросов на чтение мы начали замечать в нашей таблице несколько устаревших строк по поводу членства в каналах. Через день стало понятно, что иногда успешная запись в таблицу `team_channels_members` или `groups_members` сопровождается параллельной записью в таблицу `channels_members`. Возврат из функции записи происходил при успешной записи в старую таблицу. То есть проверка успешности второй записи не проводилась, так как вызывающие функции считали, что все в порядке.

Отредактированный код стал возвращать сообщение об ошибке при проблемах записи, но мы не были уверены, что этим можно ограничиться. Для проверки мы решили заново заполнить таблицу `channels_members` для нашего рабочего пространства. Мы инициировали еще одно обратное заполнение, очистив ее от содержимого, а затем скопировали туда данные из таблиц `team_channels_members` и `groups_members`.

Все было бы здорово, но я забыла, что вся информация о членстве тогда считывалась только из новой таблицы. Буквально через несколько секунд после запуска сценария исчезли все мои каналы и каналы всех сотрудников. Мне потребовалось несколько минут, чтобы воспользоваться переключателем функциональности и получить возможность еще раз извлечь информацию о членстве из исходных таблиц. К счастью, все это происходило, когда большинство сотрудников уже разошлись по домам. Но тех, кто в этот момент пользовался клиентом, ситуация сильно напугала.

Заключительные шаги

Когда в нашем планировщике не осталось записей, мы начали постепенно разрешать чтение из новой таблицы всем остальным командам (изначально оно было доступно только нам и VLB). Две недели мы наблюдали за происходящим, пока не решили, что двойную запись данных можно прекращать. Мы хотели удостовериться в нормальной реакции на новую таблицу на уровне нашей базы данных, в согласованности данных в этой таблице и в отсутствии новых связанных с рефакторингом ошибок. Если бы двойная запись не была такой дорогой и не давала бы такую нагрузку, ее можно было бы оставить на время, но мы старались уменьшить издержки.

Прекращение двойной записи тоже было постепенным. Сначала для нашей команды, затем для VLB и, наконец, для остальных клиентов. Этот важный этап был широко освещен (рис. 10.13). После этого мы быстро привели в порядок новую библиотеку, удалив все ссылки на таблицы `team_channels_members` и `groups_members`. Были добавлены новые правила для линтера, запрещавшие запросы к устаревшим таблицам и заставляющие корректно размещать в новой централизованной библиотеке запросы к таблице `channels_members`. Мы старались дать ориентиры для работы в новых условиях. Не все сотрудники читают объявления в каналах, особенно если они в отпуске. Поэтому только на объявления лучше не полагаться, нужны инструкции, как поступать с измененным во время рефакторинга кодом.

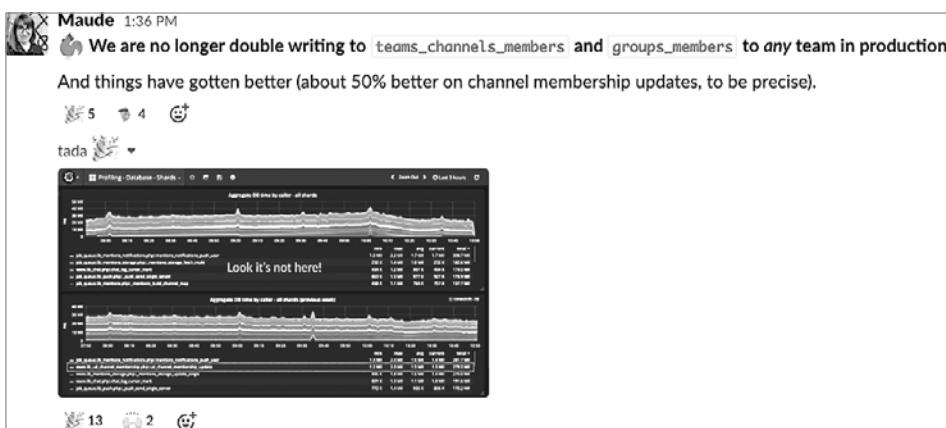
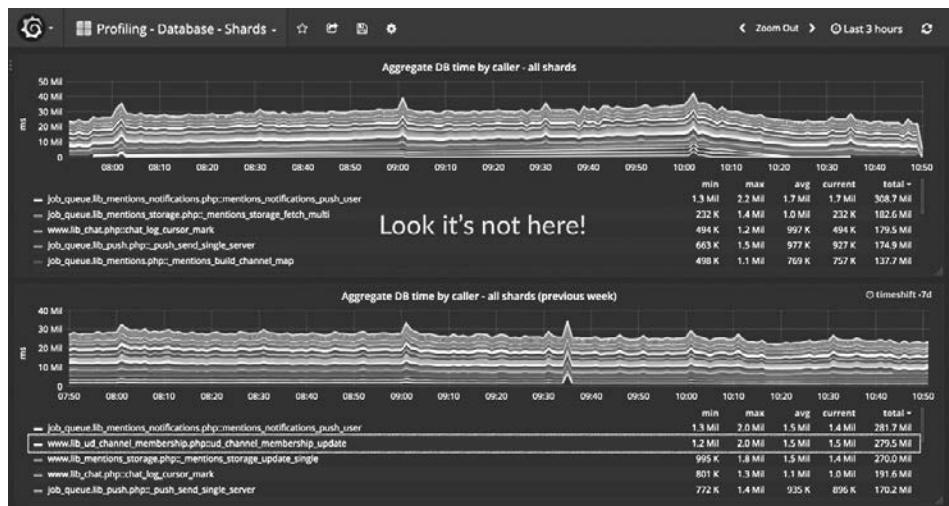


Рис. 10.13. Объявление о прекращении двойной записи в нашем рабочем пространстве

Вот крупный план графика из сообщения в Slack, показанного на рис. 10.13:



Конечно, мы не забыли про самый важный финальный шаг — отметить победу! По принятой у большинства инженеров в Сан-Франциско традиции мы заказали торт, где красовалось название новой таблицы (рис. 10.14).



Рис. 10.14. Торт Funfetti в честь завершения рефакторинга!

График выполнения проекта показан на рис. 10.15. На нем можно увидеть число запросов к каждой таблице ежедневно с мая по сентябрь 2017 года.

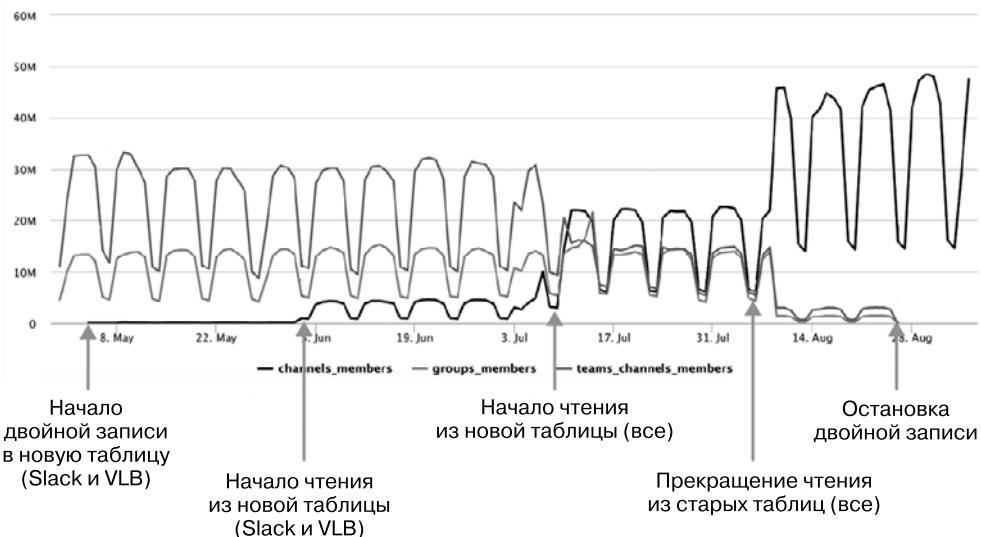


Рис. 10.15. Объем запросов к таблицам `teams_channels_members`, `groups_members` и `channels_members` за время рефакторинга

Извлеченные уроки

Посмотрим, какие уроки можно извлечь из примера выше. Начну я с того, что мешало выполнению проекта: отсутствие плана выполнения, отказ от попыток понять причины деградации кода, экономия на количестве тестов и неспособность мотивировать товарищей по команде. А дальше мы поговорим о вещах, способствовавших успешному выполнению проекта: динамические этапы и четко определенный набор метрик.

Разработка четкого плана выполнения

Поскольку проект требовал оперативных действий, мы не стали тратить время на составление плана. Процесс переноса данных из одной таблицы в другую был нам знаком. Мы знали, что в первую очередь надо заниматься запросами упоминаний и что мы завершим только связанный с ними фрагмент миграции, а остальное обдумаем позже. Единственный раз, когда в письменном виде появилось нечто похожее на план, был связан с публикацией обновлений на канале нашей команды (а не на канале по проекту).

Но даже тогда это были только фрагменты плана. Отсутствие перед глазами перечня этапов выполнения и подзадач увеличивало вероятность того, что в процессе мы что-то упустим.

Хуже всего было то, что из-за отсутствия письменного плана мы не могли разослать его по другим отделам. Поэтому они не могли проверить, насколько их затронет рефакторинг, и высказать свои соображения по этому поводу. Мы просто пошли вперед, ведь считалось, что самым важным, что можно было сделать для улучшения отношений с крупнейшим клиентом (а значит, и для нашей компании), это улучшение производительности. Мы думали, что сможем провести рефакторинг, по минимуму затронув работу других команд.

Конечно же, мы заблуждались. Во-первых, при неизбежном столкновении с ошибками мы не смогли как следует организовать коллективную работу по их устранению, что неприятно удивило инженерную группу. Во-вторых, мы упустили из виду команду, на которую вносимые изменения влияли особенно сильно. Примерно за месяц до завершения миграции связанных с членством запросов мне напомнили, что отключение возможности записи в старые таблицы нарушало большинство рабочих процессов одной из команд. В том числе процессы, отвечающие за расчет коэффициентов загрузки. Нам повезло, что эта команда смогла быстро адаптироваться к изменениям, предотвратив серьезный кризис.

Такие инциденты показывают, как важно разработать и тщательно проверить план выполнения. К счастью, мы смогли оперативно выйти из положения. Но зачем оставлять на волю случая вещи, которые можно осознанно решить на ранних этапах? Как подчеркивалось в главах 4 и 7, четкий план выполнения позволяет сразу увидеть проблемные места и найти взаимопонимание с другими командами.

Анализ истории кода

Я еще раз настоятельно рекомендую разработчикам начинать рефакторинг с изучения истории кода, потому что дополнительный контекст может изменить форму и направление проекта. Из-за срочности нашей работы мы пренебрегли этим процессом и сразу приступили к выполнению. И только после начала переноса запросов у меня возник вопрос, почему вообще появились таблицы `teams_channels_members` и `groups_members`.

Шли недели, подлежащих миграции запросов оставалось еще много, а меня уже сильно утомили избыточные таблицы, по которым они были разбросаны. Чем больше я расстраивалась, тем дольше, казалось, длился проект (и тем больше был соблазн пойти по пути наименьшего сопротивления, чтобы быстрее добраться до финиша).

После рефакторинга от сотрудников, работающих в компании дольше всех, я узнала, что хранение информации о каналах в отдельных таблицах было мерой безопасности. Это изолировало все сведения друг от друга. Кроме того, на заре существования Slack концепции общедоступных и частных каналов были разными. Но постепенно они сближались, как и схемы таблиц.

Эта информация пригодилась в следующем рефакторинге, так как прояснила способ консолидации таблиц `teams_channels` и `groups`. У меня сформировалось более позитивное отношение к рефакторингу как к возможности улучшить код, исправно служивший нам долгое время, а не как к переделке «плохого» кода. Именно поэтому в главе 2 я советую найти время, чтобы понять, как появился код, который нужно улучшить, и что могло привести его к деградации. Это понимание позволяет сохранить непредвзятое отношение и снисходительность на протяжении всего рефакторинга.

Обеспечение адекватного тестового покрытия

В главе 1 я рассказывала о важности адекватного тестового покрытия. Именно благодаря ему поведение приложения во время рефакторинга будет неизменным на каждом этапе. В нашем случае подавляющая часть редактируемого кода появилась на ранних этапах разработки Slack. Поскольку продукт стремились быстро вывести на рынок, тестов для большей его части просто не было. Таблицы мы тоже объединяли в срочном режиме, так как вопросы производительности очень беспокоили клиента. Поэтому нужные изменения вносились максимально осторожно, а тесты мы решили писать только для самых важных непроверенных путей выполнения кода.

В итоге в рефакторинге было сделано несколько ошибок, которые можно было бы предотвратить с помощью необходимых тестов. Скорее всего, на восстановление после регрессий мы потратили больше времени, чем ушло бы на написание тестов. Адекватное тестовое покрытие позволяет выполнять рефакторинг плавно, предотвращая столкновение клиентов с ошибками и избавляя команду от траты времени на их устранение.

Сохранение мотивации

Спустя некоторое время прогресс замедлился. И чтобы не продолжать работать в одиночку, мне нужно было придумать, как привлечь к проекту коллег. На миграцию последних 10 % запросов ушло примерно столько же времени, что и на первые 50 %. После успешного совершенствования связанного с упоминаниями запроса для VLB чувство срочности, которое мы испытывали в начале проекта, ушло. Каждая новая ошибка или несоответствие в данных гасили наш энтузиазм. Когда проект был почти завершен, он больше напоминал попытку закатить на гору огромный камень.

Но мы даже не думали обратиться за помощью к другим командам. А ведь можно было попросить специалистов из других групп разработки продукта перенести запросы в рамках их функционала. Им можно было объяснить, как окупятся потраченные усилия, показав итоговый прирост производительности. Распределение обязанностей могло бы вдвое сократить время рефакторинга.

Если темп работ начинает замедляться, сразу ищите способы ускорить его, пока не застопорились еще больше. Медленный рефакторинг часто теряет приоритетность, оставив после себя много кода, работа над которым не закончена. А это становится причиной новых проблем. В главе 8 я описала несколько способов для поддержания мотивации команды. Не стесняйтесь обращаться за поддержкой, когда она вам нужна!

Концентрация на стратегических этапах

У нас были предварительные данные от утилиты EXPLAIN PLAN, подтверждающие, что объединение двух таблиц ускорит выполнение запросов. На ранних этапах рефакторинга нужно было подтвердить эту гипотезу, чтобы, если консолидации окажется недостаточно, можно было переориентироваться. Мы сосредоточились только на изменениях для миграции запросов упоминаний для VLB. За несколько недель они позволили нам получить необходимое подтверждение, так как нагрузка на шард базы данных VLB снизилась. Это позволило нам довести до конца остальную часть рефакторинга.

Вовремя доказанная эффективность рефакторинга — гарантия того, что команда не теряет времени зря, работая над длительным проектом, который может не принести желаемых результатов. Если сосредоточиться

на стратегических этапах, клиенты, которые должны получить выгоду от рефакторинга, могут получить ее раньше. В дальнейшем это обеспечит поддержку проекта с их стороны. Способы определения стратегических этапов обсуждались в главе 4.

Выбор метрик

У нас был набор метрик, позволивший убедительно продемонстрировать успешность проекта как во время промежуточных этапов, так и после внедрения нововведений для всех клиентов. Данные утилиты EXPLAIN PLAN, которые мы собирали до и после консолидации, позволили документировать прогресс по мере переноса более сложных, связанных с членством запросов. Добавив к запросам, связанным с упоминаниями, метрики времени, мы смогли сразу отслеживать его производительность и видеть положительное влияние.

Внимательное отслеживание метрики позволит убедиться, что рефакторинг все время проходит в правильном направлении. Если показатели перестанут улучшаться (или начнут снижаться), можно немедленно заняться решением возникшей проблемы, не откладывая это до завершения проекта. Способы количественной оценки рефакторинга мы обсуждали в главе 3.

Ключевые моменты

Вот перечень основных положений нашего рефакторинга по консолидации таблиц членства в каналах Slack.

- Составьте подробный письменный план и поделитесь им с коллегами.
- Разберитесь в истории кода, чтобы воспринимать его рефакторинг позитивнее.
- Убедитесь в достаточном тестовом покрытии кода, который вы собираетесь улучшать. Если тестов не хватает, напишите их.
- Поддерживайте мотивацию своей команды. Если темп работы снижается, ищите способы его восстановить.
- Сосредоточьтесь на стратегических этапах, чтобы чаще доказывать эффективность вашего рефакторинга.
- Определите значимые метрики и ориентируйтесь на них для движения в верном направлении.

ГЛАВА 11

Переход к новой базе данных

*Написано в соавторстве с Мэгги Чжоу,
штатным инженером по инфраструктуре
компании Slack Technologies, Inc.*

Здесь мы рассмотрим рефакторинг, выполненный инженерами из группы разработки продукта и группы инфраструктуры в Slack. Основой для него послужила консолидация таблиц членства в каналах компании из прошлой главы. Если вы ее не прочти, советую сделать это сейчас. Там вы найдете важный контекст, необходимый для лучшего понимания этой главы.

Если в предыдущем случае рефакторинг проводился для увеличения производительности, теперь вопрос стоял об обеспечении большей гибкости продукта. Привязка членства в каналах к разным шардам рабочих пространств затрудняла создание более сложного функционала, выходящего за рамки одного рабочего пространства. Мы хотели позволить смешанным организационным структурам с несколькими рабочими пространствами беспрепятственно сотрудничать при одном наборе каналов и облегчить общение между отдельными клиентами Slack, позволяя компаниям договариваться с поставщиками в приложении. Для этого требовалось, чтобы шардирование данных о членстве в каналах выполнялось по пользователям и каналам, а не по рабочим пространствам. Крупномасштабная миграция баз данных, растянувшаяся на несколько кварталов и выполнявшаяся разными отделами, сопровождалась множеством проблем.

Тем не менее рефакторинг прошел успешно. Все четко понимали поставленную задачу и то, что продукт просто перерос нынешнее архитектурное решение в процессе развития (глава 2). Мы тщательно планировали проект и ввели в рассмотрение еще несколько переменных, зная, что это увеличит пользу от рефакторинга (глава 4). Была спланирована осторожная стратегия

развертывания и разработаны инструменты для ее максимально надежной реализации (глава 8).

Хотя рефакторинг в конечном итоге позволил нам расширять наш продукт новыми и интересными способами, он занял почти вдвое больше времени, чем мы изначально планировали. Мы оказались слишком оптимистичными в оценках (глава 4). Потребовалось больше года для завершения проекта, изначально рассчитанного на шесть месяцев. Мы недооценили влияние рефакторинга на продукт и привлекли к сотрудничеству инженеров по продукту только после того, как несколько месяцев почти не двигались вперед (глава 6).

Как и в прошлой главе, я начну рассказ с краткого обзора ситуации. Он позволит лучше понять, почему способ распределения данных стал узким местом и почему мы решили перейти к новой системе шардирования Vitess. После этого мы опишем решение и пройдемся по всем этапам его реализации.

Распределение по рабочим пространствам

Чтобы понять суть проблем, которые мы хотели решить рефакторингом, нужно описать способ распределения данных по базам MySQL. Изначально большинство данных было распределено по рабочим пространствам. Каждое из них было выделено для одного клиента компании Slack. Я немного рассказывала об этом ранее в разделе «Архитектура Slack 101» главы 10. Распределение данных клиентов по разным шардам было показано на рис. 10.3.

Несколько лет это работало нормально, но постепенно такая схема становилась более неудобной. Это происходило по двум причинам.

Во-первых, усложнялась поддержка операций для самых больших шардов. Шарды, выделенные крупнейшим и наиболее быстрорастущим клиентам, часто страдали от проблем с доступом. Данные таких клиентов, уже занимавшие изолированные шарды, быстро разрастались настолько, что требовалось обновление аппаратного обеспечения, которое мы сделать не могли. Простых механизмов разделения данных по горизонтали у нас не было, и мы не знали, что делать.

Во-вторых, в мессенджер вносились важные изменения для разрушения барьеров между рабочими пространствами, которые долго поддерживались способами написания кода и структурированием данных. Но в какой-то момент был создан функционал, позволивший крупнейшим клиентам

объединить несколько рабочих пространств, а двум клиентам напрямую общаться в общем канале.

Несоответствие между тем, как компания видела будущее продукта и способом построения систем вело к усложнению приложения. Это пример деградации кода из-за изменения требований к продукту (глава 2). Проблему наглядно иллюстрирует тот факт, что за год до рефакторинга для успешного определения местоположения канала и его участников приходилось делать запрос к трем шардам. Разработчикам приходилось запоминать правильный набор шагов для извлечения связанных с каналом данных и управления ими.

Чтобы разобраться с операционными проблемами с MySQL и сложностями масштабирования, мы начали оценивать другие варианты хранения. Выбор пал на Vitess (<https://vitess.io/>) — систему кластеризации баз данных, созданную YouTube, которая обеспечивает горизонтальное масштабирование MySQL. Миграция на Vitess позволит нам делить данные на шарды не только по рабочим пространствам. Так мы сможем освободить место в самых загруженных шардах и распределить данные так, чтобы упростить запросы!

Миграция таблицы channels_members на Vitess

Учитывая эти обстоятельства, мы решили перенести таблицу `channels_members` в систему Vitess. Это была одна из таблиц с наибольшим трафиком. Изменение способа ее шардирования освобождало много места и уменьшало нагрузку на самые загруженные шарды рабочего пространства. Миграция сильно упрощала логическую схему получения членства в каналах за пределами рабочей области.

Проект был инициирован командой разработчиков инфраструктуры Vitess с помощью инженеров по продукту, хорошо разбиравшихся в шаблонах запросов к таблице `channels_members`. Это было выигрышное сочетание.

Разработчики инфраструктуры досконально знали систему баз данных. Это позволяло предотвратить ошибки миграции и эффективно решать проблемы с базой данных по мере их возникновения. Тогда они обладали самым большим опытом в области миграции таблиц и лучше всего подходили для руководства проектом. Возглавляла эту группу Мэгги. Инженеры по продукту, включая меня, могли предоставить информацию о новой схеме и схеме шардирования. Нашей задачей была помочь с переписыванием логики приложения под новые варианты запросов.

Мы создали канал `#feat-vitess-channels` для обмена идеями и координации рабочих потоков. Разослали приглашения присоединиться к нему и сразу приступили к решению первой задачи.

Схема шардирования

Прежде чем переносить в Vitess данные о членстве в каналах, нужно было решить, как они будут распределяться (какие ключи использовать для изменения шардирования таблицы). У нас было два варианта:

- по каналу (`channel_id`), чтобы запрос к одному шарду давал всю информацию об участниках канала;
- по пользователю (`user_id`), чтобы запрос к одному шарду давал всю информацию о членстве пользователя.

Поскольку консолидация таблиц завершилась недавно, я помнила, что большинство запросов касались получения членства для данного канала, а не для данного пользователя. Многие из них имели решающее значение для приложения, обеспечивая поиск и возможность упоминания всех участников канала (@`channel` или @`here`).

Тогда (и по сей день) мы регистрировали образцы всех запросов к базе данных в хранилище данных, чтобы отслеживать использование MySQL при обращениях к нашим производственным системам. Для подтверждения своего предположения, что большая часть обращений к таблице `channels_members` связана с ключом `channel_id`, я посмотрела выборку связанных с членством запросов за месяц и передала результат (рис. 11.1) команде.

| # | Фильтр по channel ID | count |
|---|----------------------|-----------|
| 1 | true | 846150562 |
| 2 | false | 770456108 |

Рис. 11.1. Количество запросов к таблице `channels_members`, отфильтрованных по ключу `channel_id`

Но один из сотрудничавших с нами инженеров по продукту, имеющий больший опыт работы с Vitess, указал, что будет лучше провести шардирование по пользователям. В том же самом журнале он показал нам десять самых частых запросов к таблице, отфильтрованных по `channel_id` (рис. 11.2). Для обеспечения эффективной работы приложения нужно было предусмотреть это поведение.

| # | sql | Фильтр по channel ID | count |
|----|--|----------------------|------------|
| 1 | SELECT * FROM channels.members FORCE INDEX (PRIMARY) WHERE team_id=%team_id... | true | 1129091681 |
| 2 | SELECT * FROM channels.members WHERE team_id=%team_id AND... | true | 268354996 |
| 3 | SELECT cm.channel_id As channel_id, cm.last_read, c.is_shared, c.is_m... | true | 218944219 |
| 4 | SELECT 'team_id','channel_id','user_id','date_joined','date_deleted','1... | true | 134941810 |
| 5 | UPDATE channels_members SET 'last_read'=%last_read,'last_read_ab...' | true | 128874281 |
| 6 | UPDATE channels_members SET 'last_read'=%last_read,'last_read_ab...' | true | 86645894 |
| 7 | SELECT 1 as count FROM messages m JOIN channels_members c ON... | false | 69190411 |
| 8 | SELECT channel_id, is_general, c.channel.type, name, date_archived, la... | false | 65318561 |
| 9 | SELECT c.* FROM %field:table c JOIN channels_members cm ON cm.c... | false | 62554095 |
| 10 | SELECT team_id, channel_id,user_id FROM channels_members WHERE... | false | 57343805 |

Рис. 11.2. Десять самых частых запросов к таблице channels_members и данные о том, проводилась ли фильтрация по ключу channel_id

Мы подсчитали примерный объем запросов к базе данных, необходимый для поддержки любого из этих вариантов. Было решено пойти на компромисс, прибегнув к денормализации данных о членстве. В итоге появилось две таблицы: одна шардированная по пользователям, другая — по каналам плюс двойная запись для обоих вариантов. Так точечные запросы получатся дешевыми в обоих случаях.

Разработка новой схемы

Дальше нужно было внимательно взглянуть на существующую схему таблицы, шардированной по рабочим пространствам, и определить, как мы будем ее преобразовывать для получения шардирования по пользователям и каналам. Конечно, можно было бы просто воспользоваться существующей схемой. Но рефакторинг дал нам уникальную возможность пересмотреть некоторые решения. Мы подробнее рассмотрим обе новые схемы, начав с шарда для пользователей. В примере 11.1 мы видим схему шардов по рабочим пространствам, которая применялась до миграции.

Пример 11.1. Оператор CREATE TABLE показывает таблицу channels_members, шардированную по рабочим пространствам

```
CREATE TABLE `channels_members` (
  `user_id` bigint(20) unsigned NOT NULL,
  `channel_id` bigint(20) unsigned NOT NULL,
  `team_id` bigint(20) unsigned NOT NULL,
  `date_joined` int(10) unsigned NOT NULL,
  `date_deleted` int(10) unsigned NOT NULL,
  `last_read` bigint(20) unsigned NOT NULL,
  ...
  `channel_type` tinyint(3) unsigned NOT NULL,
  `channel_privacy_type` tinyint(4) unsigned NOT NULL,
  ...
  `user_team_id` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`user_id`,`channel_id`)
)
```

Таблица членства, шардированная по пользователям

В случае шардирования по пользователям мы решили сохранить большую часть исходной схемы. Серьезные изменения были внесены только в способ хранения идентификаторов пользователей. Чтобы вам стали понятны мотивы такого решения, расскажу о двух типах идентификаторов пользователей и о том, откуда они взялись.

В начале главы я кратко упомянула, что мы пытались упростить использование мессенджера Slack для крупных клиентов, у которых разным отделам выделялись разные рабочие пространства. Это затрудняло взаимодействие между сотрудниками из разных отделов и управление отдельными рабочими пространствами. Поэтому мы позволили самим крупным клиентам объединить свои многочисленные рабочие пространства.

Но для корректной группировки требовался способ синхронизации пользователей. Давайте посмотрим, как это работало, на простом примере.

В корпорации Acme Corp. для каждого отдела выделено свое рабочее пространство. Одно из них — для команды инженеров и отдела обслуживания клиентов. У каждого сотрудника Acme Corp. есть учетная запись пользователя от организации в целом. Инженеры имеют членство в рабочем пространстве Engineering для общения с товарищами по команде и в Customer Experience (служба поддержки).

Но с точки зрения внутренней структуры то, что выглядит как одна учетная запись Acme Corp., на самом деле — несколько учетных записей. На уровне организации каждый пользователь имеет *канонический идентификатор*. Одновременно ему сопоставлены несколько локальных для всех рабочих пространств, где он подписан на каналы. Например, у члена рабочих пространств Engineering и Customer Experience три уникальных идентификатора пользователя. Их количество рассчитывается по формуле $n + 1$, где n — число доступных пользователю рабочих пространств.

Несложно догадаться, что такая система быстро стала очень сложной и благоприятствующей появлению ошибок. Поэтому через год появился план замены всех локальных идентификаторов пользователей каноническими.

Поскольку большинство хранящихся в системах Slack данных связано с определенным идентификатором пользователя (создание сообщения, загрузка файла и прочее), корректно (и по возможности незаметно) перезаписать эти идентификаторы было непросто.

При шардировании по рабочему пространству в таблице `channels_members` локальные идентификаторы пользователей хранились в столбце `user_id`. Проект по замене всех локальных идентификаторов каноническими уже стартовал. Мы решили работать совместно с ними и проследить за сохранением во все столбцы идентификаторов канонических user ID.

Схема шардированной по каналам таблицы

Нас волновали не только идентификаторы пользователей, но и пропускная способность записи во вторую таблицу членства, шардированную по каналам. Мы изучили, какие запросы планировались к этим шардам, для определения способов снижения трафика. В процессе обнаружилось, что большинство столбцов исходной таблицы вообще не использовались. Причем к ним относились и столбцы, обновлявшиеся чаще всего, например дата последнего чтения пользователем материалов канала. Если запросить всех пользователей с доступом к какому-то каналу, логическая схема приложения для выполнения этого запроса обычно обращается только к столбцам `user_id` и `user_team_id`. Опуская эти неиспользуемые столбцы в новой схеме, мы значительно снижали частоту записи, предоставляя больше места шардам по каналам. Схема шардированной по каналам таблицы членства показана в примере 11.2.

Пример 11.2. Оператор CREATE TABLE для второй из новых таблиц `channels_members`, шардированной по каналам

```
CREATE TABLE `channels_members_bychan`
  `user_id` bigint(20) unsigned NOT NULL,
  `channel_id` bigint(20) unsigned NOT NULL,
  `user_team_id` bigint(20) unsigned NOT NULL,
  `channel_team_id` bigint(20) unsigned NOT NULL, ❶
  `date_joined` int(10) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`channel_id`, `user_id`)
)
```

❶ `team_id` переименовано в `channel_team_id`.

Разбиение запросов с оператором JOIN

Затем пришло время обновить логику приложения с учетом изменений в схемах таблиц и появления кластера Vitess. К счастью, большинство изменений были простыми, и мы быстро обновили большую часть логических схем.

Миграция усложнилась запросами, включающими операции соединения с другими таблицами в нашем кластере MySQL. После перемещения таблицы в новый кластер продолжать поддержку этих запросов было невозможно. Поэтому их пришлось разбить на более мелкие точечные, к которым оператор `JOIN` применялся прямо в коде приложения.

С самого начала мы знали о стоящей перед нами задаче по разделению запросов с операторами `JOIN`. Но мы не ожидали, что большинство из них обеспечивает работу основного функционала Slack, и долгое время тщательно настраивались вручную для повышения производительности. Разделение таких запросов могло закончиться чем угодно, от замедления уведомлений до утечки данных и полного прекращения работы Slack. Мы очень нервничали, но выбора у нас не было.

Наша команда приостановила обычные миграции и составила список запросов, вызывающих наибольшие опасения. Их получилось 20. Мы беспокоились об отсутствии у нашей команды опыта работы с продуктом, который позволил бы адекватно разделить эти запросы. По нашим подсчетам получалось, что на решение этой задачи уйдут месяцы. К счастью, на наш призыв откликнулись инженеры по продукту, которые помогли нам разработать простой процесс безопасного разделения.

Для наглядности рассмотрим процедуру разделения запроса (пример 11.3). Этот запрос показывал, есть ли у пользователя разрешение на просмотр определенного файла.

Пример 11.3. Запрос с оператором `JOIN`, который нужно было разбить на части; после знака `%` подставляется соответствующее значение

```
SELECT COUNT(*)
FROM files_shares s
LEFT JOIN channels_members g
  ON g.team_id = s.team_id
  AND g.channel_id = s.channel_id
  AND g.user_id = %USER_ID
  AND g.date_deleted = 0
WHERE
  s.team_id = %TEAM_ID
  AND s.file_id = %FILE_ID
LIMIT 1
```

Сначала нужно было определить наименьшее подмножество данных, которое можно было извлечь раньше всего. Это позволяло минимизировать пересечения с данными, с которыми нужно было начать работу как можно раньше.

Типичные шаблоны запросов видимости файла показали, что мест с доступом к файлу обычно было намного меньше, чем каналов, на которые подписан пользователь (этую гипотезу можно было проверить, посмотрев на

число элементов во множестве запроса). Поэтому вместо запроса членства пользователя в каналах и последующего поиска каналов с доступом к файлу мы извлекали каналы, где был общий доступ к файлу, а затем определяли, был ли на каком-то из них пользователь. Подробности разделения запроса на два компонента видно на примере 11.4.

Пример 11.4. Разбиение запроса с оператором JOIN к таблице files_shares на две части

```
SELECT DISTINCT channel_id
FROM files_shares
WHERE team_id=%TEAM_ID AND file_id=%FILE_ID

...
SELECT COUNT(*)
FROM channels_members
WHERE
    team_id=%TEAM_ID
    AND user_id=%USER_ID
    AND channel_id IN (%list:CHANNEL_IDS)
LIMIT 1
```

Затем мы убедились в достаточности тестового покрытия. Если бы оно оказалось недостаточным, мы написали бы несколько дополнительных тестов для проверки результатов исходного запроса. Но все было в порядке, и мы провели эксперимент с новой логической схемой, позаботившись о возможности быстро откатиться назад в экстренной ситуации. Мы тестировали обе реализации, исправляли все обнаруживаемые ошибки и повторяли процесс, пока не убедились в работоспособности новой логической схемы. После чего к обоим вызовам были добавлены метрики времени для сравнения времени выполнения исходного запроса с оператором JOIN и новой версии, полученной после разбиения. Пример 11.5 показывает, как была реализована проверка видимости файла и какие инструменты при этом использовались.



Более рискованные разделения запросов (включая запрос видимости файлов) мы выполняли вместе с командой контроля качества. Так мы могли вручную проверить внесенные изменения как в наших средах разработки, так и в производственной среде. Большинство запросов с оператором JOIN, которые мы пытались разбить, были связаны с критически важным функционалом Slack. Поэтому приходилось быть особенно осторожными, чтобы наши изменения идеально воспроизвели нужное поведение.

Пример 11.5. Функция, определяющая, видит ли пользователь указанный файл

```
function file_can_see($team, $user, $file): bool {  
    if (experiment_get_user('detangle_files_shares_query')) {  
        $start = microtime_float();  
  
        # Сначала нужно найти все каналы, на которых представлен файл.  
        # На одном канале файл может фигурировать несколько раз, поэтому  
        # можно обнаружить несколько строк таблицы files_shares  
        # с одинаковым идентификатором канала, но с разными временными  
        # метками, указывающими момент предоставления файла.  
        $channel_ids =  
            ud_files_shares_get_distinct_channel_ids(  
                $team,  
                $file['id'])  
        );  
  
        # Нужно найти пересечение каналов, на которых представлен файл  
        # ($channel_ids), и каналов, на которые подписан пользователь.  
        $membership_counts =  
            ud_channels_members_get_counts(  
                $team,  
                $user['id'],  
                $channel_ids  
            );  
  
        $end = microtime_float() - $start;  
  
        # Если найдется хоть одна строка membership, значит,  
        # пользователь может видеть файл.  
        _files_can_see_unjoined_histogram()->observe($end);  
        return ($membership_counts['count'] > 0);  
    }  
  
    $start = microtime_float();  
    $sql .= "SELECT 1 FROM files_shares s  
        LEFT JOIN channels_members g  
        ON g.team_id = s.team_id  
        AND g.channel_id = s.channel_id  
        AND g.user_id = %USER_ID  
        AND g.date_deleted=0  
    WHERE s.team_id = %TEAM_ID  
        AND s.file_id = %FILE_ID  
        AND (g.user_id > 0) LIMIT 1";  
  
    $bind = [  
        'file_id' => $file['id'],  
        'user_id' => $user['id'],  
        'team_id' => $team['id']  
    ];
```

```
$ret = db_fetch_team($team, $sql, $bind);
$end = microtime_float() - $start;
_files_can_see_join_histogram()->observe($end);

return (bool)db_single($ret);
}
```

Перед развертыванием мы включили новую реализацию для нашего внутреннего экземпляра Slack. Это были важный шаг для подтверждения правильности обработки метрик времени и дополнительная гарантия того, что мы случайно не внесли ошибку.

Рабочее пространство Slack имеет множество особенностей. Поэтому наши шаблоны использования не всегда совпадают с шаблонами клиентов. Часто рабочее пространство может стать хорошей лакмусовой бумажкой для раннего выявления ошибок. Но оно не подходило для определения того, насколько приемлема задержка в выполнении разбитых на части запросов. Производительность некоторых запросов с оператором `JOIN` в нашем рабочем пространстве особенно упала. Но по мере выполнения развертывания сначала для бесплатных, а потом и для платных клиентов метрики стабилизировались.

Этот процесс повторялся почти для каждого оператора `JOIN`. Мы осторожно разбивали запросы на части и снабжали их метриками. Исключением стали два ужасных запроса упоминаний, которые мы не трогали несколько месяцев. К сожалению, эти запросы создавали ряд уникальных проблем, в том числе `JOIN` для таблиц, которые подвергались собственной миграции Vitess. Мы решили отложить преобразование этих запросов до момента, когда все их составные части встанут на свои места. В целом для завершения миграции запросов с оператором `JOIN` группе из пяти человек потребовалось около шести недель. Причем мы занимались не только рефакторингом, но и своими прямыми рабочими обязанностями.



Часто рефакторинг проходит не совсем по плану. Возникают сложности, требующие перестановки приоритетов или вообще временной остановки задач текущего этапа. Такие вещи не только раздражают, но иногда и влияют на возможность вовремя завершить проект.

Если бы мы начали дожидаться завершения миграции компонентов, от которых зависели запросы на упоминание, рефакторинг продлился бы на несколько месяцев дольше. Но мы решили, что у нас уже успешно переписано большинство запросов к таблице `channels_members`, так что можно продолжить движение вперед, решая проблемы по мере их возникновения. К моменту, когда мы смогли вернуться к работе над запросами упоминаний, ситуация стала гораздо стабильнее.

Сложности развертывания

Когда мы приступили к миграции таблицы `channel_members`, примерно 15 % от всех запросов в секунду (queries per second, QPS) обеспечивалось системой Vitess. Мы уже перенесли и обновили элементы, создававшие основную нагрузку (таблицы, связанные с уведомлениями, и таблицу `teams`, отвечающую за перечень всех экземпляров клиента Slack). Мы создали надежные методы и инструменты для облегчения почти 20 миграций, укомплектованные фреймворками для контроля и эффективного сравнения наборов данных в старом и новом кластерах.

Но миграция таблицы `channel_members` была уникальна тем, что она обеспечивала почти 20 % от общего числа запросов. Это почти вдвое превышало количество запросов в секунду, которым мы тогда научились управлять в Vitess. Мы боялись, что из-за масштаба во время миграции могут возникнуть неожиданные проблемы. Но у нас была сильная мотивация убрать нагрузку с MySQL, создающую трудности для наших крупнейших клиентов.

Мы решили положиться на инструментарий, созданный во время предыдущих миграций на Vitess, и надеялись, что для нашей таблицы он тоже даст достаточно стабильный результат.

Процесс развертывания выполнялся в четырех режимах.

1. *Backfill*. На этом этапе велась запись запросов к старому и к новому кластеру (с новой схемой шардирования), и произошло заполнение нового кластера данными из старого.
2. *Dark*. Чтение осуществлялось из обоих кластеров, а результаты сравнивались. Любые несоответствия в данных, полученных из нового кластера Vitess, регистрировались. Пользователям предоставлялись результаты, полученные из старого кластера.
3. *Light*. В этом случае чтение тоже осуществлялось из обоих кластеров, результаты сравнивались и любые несоответствия регистрировались. Но в приложение возвращались уже результаты из кластера Vitess.
4. *Sunset*. Запись велась в оба кластера, но запросы на чтение шли строго в Vitess. Ресурсоемкое чтение из двух источников данных прекратилось, но любые нижестоящие потребители могли полагаться на данные из старых кластеров, пока они не будут перенесены в Vitess (включая такие системы, как наше хранилище данных). На этом этапе в случае проблем

единственным вариантом было продвижение их вперед. Простого и безопасного способа вернуться к использованию данных из устаревшего источника просто не было.

Возможность быстрого развертывания конфигураций позволяла легко переключаться между режимами и регулировать объем трафика. Система представляла детализированные элементы управления, благодаря которым мы могли быстро переключать клиентов и пользователей в конкретные режимы. Возможность настройки этих параметров позволяла быстро регулировать трафик при возникновении проблем.

Режим Backfill

Каждая миграция начиналась с режима *Backfill*. Во-первых, в рамках подготовки к миграции запросов на чтение нужно было заложить фундамент для заполнения данными из старого кластера. Для большинства прошлых миграций это был достаточно простой этап. Например, запросы на запись для нового кластера были почти идентичными таким же запросам на запись в старый кластер. Так как мы активно меняли модель данных, многие запросы SQL для нашего приложения пришлось переписать, чтобы привести в соответствие с новой схемой (включая правильное распространение таблицы `share_type` и переход от локальных идентификаторов пользователей к каноническим). К счастью, благодаря консолидации таблиц (глава 10) мы могли легко определить все требующие перезаписи запросы.

Во-вторых, нам нужно было понять, не вызвала ли запись в новый кластер проблем с производительностью. Во время прошлых миграций считалось, что режимы *Backfill* и *Dark* почти не влияют на производительность приложения в рабочей среде. Этим мы были обязаны двум пунктам.

- Мы использовали асинхронный кооперативный многозадачный режим языка Hacklang для одновременной отправки запросов в оба кластера. Для запросов к новому кластеру в Vitess был установлен короткий тайм-аут в одну секунду (1 с). Так что в худшем случае потеря производительности для этих запросов описывалась формулой 1 (минус время выполнения запроса из старого кластера).
- В приложение еще не возвращались результаты из кластеров в Vitess! Этот процесс начался только в режиме *Light*.

Но при этой миграции все пошло иначе. Кластер базы данных в Vitess, шардированный по пользователям, куда мы перемещали таблицу `channels_members`, уже был заполнен часто используемыми производственными данными (сюда входили сохраненные сообщения и уведомления). И после перехода в режим *Backfill* насыщение ресурсов базы данных в Vitess привело к тому, что запросы к уже находящимся в кластере таблицам стали возвращать тайм-ауты и ошибки.

Покопавшись, мы нашли несколько запросов на обновление и удаление без ключа (`user_id`). В результате этого они оказались разбросаны по разным шардам кластера. Мы изменили конфигурацию и снова аккуратно начали увеличивать трафик в режиме *Backfill*. По достижении 100 % мы перешли в режим *Dark*!

Режим *Dark*

Вход в режим *Dark* произошел после тщательного переписывания большинства запросов к таблице `channels_members` (включая множество проблемных запросов с оператором `JOIN`) и успешного завершения обратного заполнения. На все это у нас ушло три месяца. Наша система миграции позволяла выбирать подмножества запросов на разных этапах (например, один мог быть в режиме *Dark*, а другой — в режиме *Light*). Поэтому для ускорения процесса мы начали наращивать трафик в режиме *Dark* еще до того, как все запросы были переписаны для чтения из кластеров в Vitess.

Как и *Backfill*, режим *Dark* преследовал две основные цели, одна из которых — выявление потенциальных проблем с производительностью, связанных с количеством запросов на чтение, отправляемых в новый кластер.

Производительность

Когда одновременно с запросами на чтение к устаревшей системе мы начали увеличивать количество запросов на чтение в Vitess, оказалось, что некоторые запросы с высоким показателем QPS возвращают слишком много строк. Из-за такого сочетания количество возвращаемых в секунду строк стало самым большим в нашем кластере. Рисунок 11.3 показывает, что на пике из таблицы `channel_members` одного шарда возвращалось около 9000 строк в секунду. Эти запросы были настолько частыми и требовательными к памяти,

что вызывали ошибки из-за ее нехватки (out-of-memory error, ООМ). Это привело к прекращению работы хоста базы данных! После наращивания мощности несколько дней мы наблюдали нехватку памяти на многих хостах.

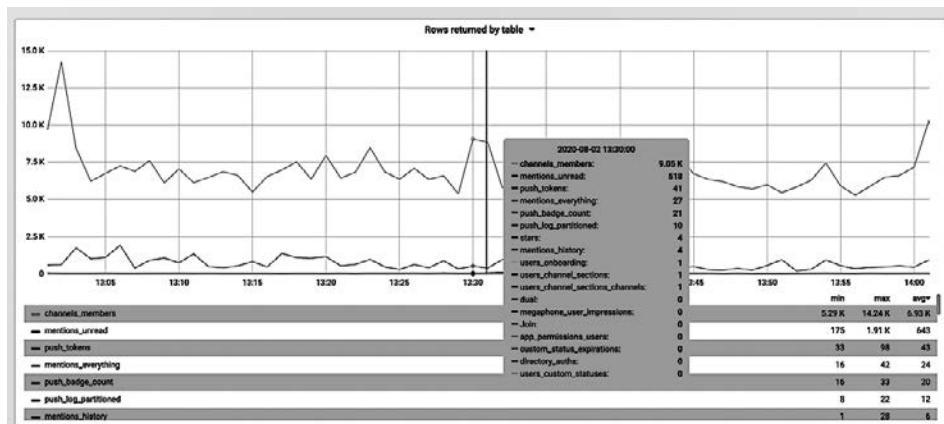


Рис. 11.3. Строки, возвращенные из таблицы +channels_members+ на одном шарде

Сначала мы думали, что виноват наш облачный провайдер. Возможно, что-то было не так со способом подготовки нашего самого большого кластера базы данных. Но потом стало ясно, что причина не в неудачной конфигурации, и мы быстро перешли к поиску источника ООМ.

Рисунок 11.4 показывает объявление про ООМ в еженедельном отчете о новостях проекта.

Maggie Zhou (she/her) 1 year ago
Status
 100% DARK reads for all teams minus a few callsites*, and two remaining JOINs.
Big change this week
 We ramped up more DARK reads and hit 5 OOMs on master tablets in 12 hours.
 We're still working on uncovering the various causes of this problem and how to move forward so that high query load doesn't cause OOMs. In the meantime, we've actually re-ramped up most of the DARK reads.

Рис. 11.4. Еженедельный отчет о статусе проекта с информацией про ООМ

Рефакторинг был приоритетным проектом как внутри инфраструктуры, так и среди инженерных организаций. С точки зрения надежности базы данных перемещение таблицы `channels_members` в систему Vitess было очень важным шагом. Поэтому, когда стало понятно, что сами мы не можем обнаружить причины ООМ, на помощь пришла команда, отвечавшая в Slack за работу с базами данных. Отладка шла со всех сторон при постоянном общении в канале `#feat-vitess-channels`. Мы попытались изменить размер выделяемой памяти для процессов MySQL, подробно изучая фрагментацию и распределение памяти как на уровне базы данных, так и на уровне ОС. Мы обновили второстепенные версии MySQL для получения доступа к новой настройке, позволяющей указать политику чередования неравномерного доступа к памяти (NUMA) для пула буферов! Тем временем мы продолжали разделять запросы с оператором `JOIN` и увеличивали их количество в режиме *Dark*. При этом все наши попытки найти причины ООМ ни к чему не приводили. Мы продолжали сталкиваться с этой ошибкой по мере увеличения нагрузки.

К этому моменту прошло полгода с момента начала проекта. То есть вышел предполагаемый срок окончания рефакторинга. У команды было ощущение, что мы каждый раз делаем два шага вперед и один назад. Через несколько недель проб и ошибок мы обнаружили, что в других системах хранения в Slack (включая наш кластер мониторинга и кластер поиска) возникли проблемы с очень маленьким значением параметра `min_free_kbytes`. Это низкоуровневая настройка ядра, отвечающая за минимальный размер свободной памяти, который нужно поддерживать. Чем больше его значение, тем большую передышку дает себе ядро, удаляя из оперативной памяти больше данных. Большое количество запросов, возвращающих множество строк с высоким QPS, требует быстрого выделения большого количества RAM. Но ядро не могло освободить RAM достаточно быстро, что и приводило к ООМ. Повышение значения `min_free_kbytes` решило эту проблему.

В режиме *Dark* мы работали целых восемь месяцев. Эта фаза не только заняла у нас больше времени, чем предполагалось потратить на весь проект. От всего объема потраченных на проект усилий она забрала почти две трети. Почему так получилось?

Несоответствия в данных

После изменений конфигурации мы смогли пустить на кластер в Vitess 100 % трафика без риска снижения производительности. К этому моменту мы разбили почти все запросы с оператором `JOIN` и обновили все точечные

запросы на чтение. Оставалось выявить любые несоответствия в наборах данных, возвращаемых по новым запросам. Сделать это было легко, так как запросы посыпались одновременно к новому и старому кластерам. Расхождения в ответах на них накапливались несколькими способами, так как нам нужно было получить общее представление о масштабах проблемы. В частности, мы регистрировали первичные ключи запросов, для которых старый и новый кластер возвращали разные результаты.

На это мы потратили несколько недель тщательного изучения различий. Шардированная по пользователям схема включала больше информации, чем исходная шардированная по рабочему пространству таблица `channels_members`. Поэтому во время перезаписи приходилось использовать намного больше переменных. Чтобы улучшить условия разработки для инженеров, работающих с общими каналами и приложением Enterprise Grid, при переносе каждого запроса нужно было учитывать неоднозначную логическую схему продукта. Это означало, что вероятность ошибок была намного выше, чем при миграции в Vitess других таблиц.

Оказалось, что большинство различий в наборах данных было связано с единичными проблемами. Часто хватало внести исправления в один экземпляр, и регистрируемых различий становилось меньше. Например, если в старой системе и в Vitess выбрать разные наборы столбцов, результаты каждого запроса будут разными, и это попадет в число зарегистрированных случаев. Как показано на рис. 11.5, обнаружение и устранение случаев с несовпадающими столбцами уменьшило количество различий с 10 % всего до 0,01 % от числа всех запросов.

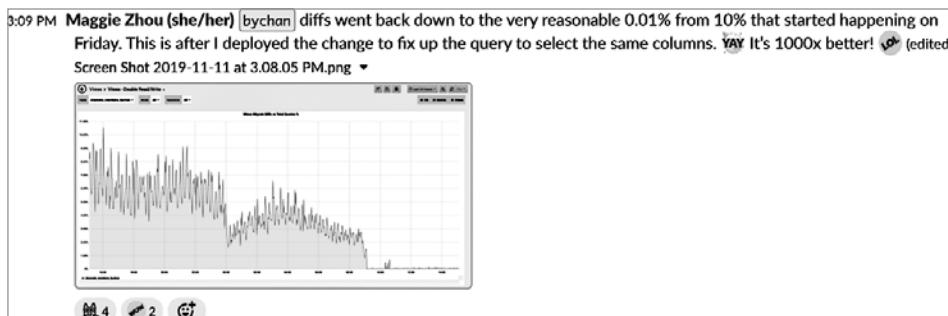
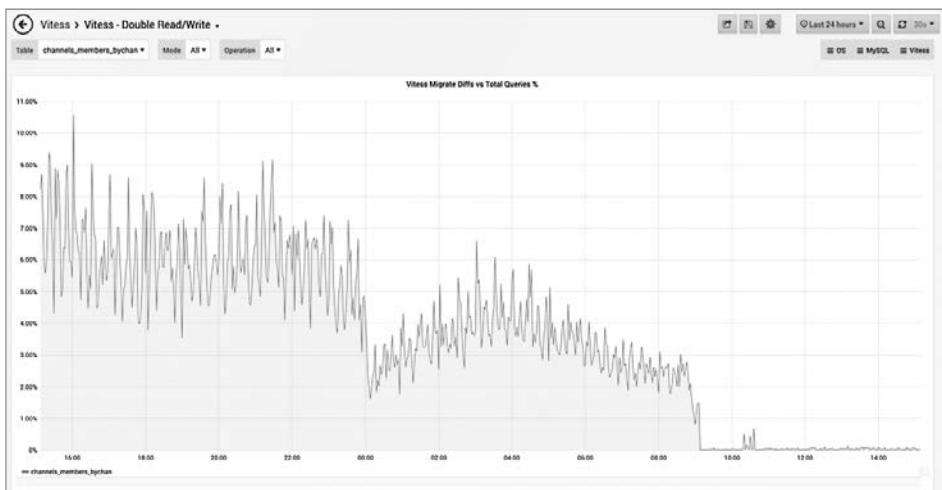


Рис. 11.5. Уменьшение различий для таблицы `channels_members`, шардированной по каналам

Вот увеличенное изображение графика с рис. 11.5:



Увы, не от всех различий было так же легко избавиться. При анализе их причин мы обнаружили несколько мест с не совсем правильной логической схемой для общих каналов и несколько ошибок, допущенных при обратном заполнении таблицы. Это была утомительная работа. И чтобы она не повлияла плохо на продукт, нужно было глубоко разобраться в принципе работы приложения. Конечно, мы активно использовали переключатели функционала. Но вносимые изменения могли сильно затронуть производственные системы, поэтому приходилось действовать очень осторожно. Из-за всего этого проект продвигался медленно, и мы запросили дополнительные ресурсы у разработчиков продукта.

Привлечение к проекту новых людей заново вдохнуло в него жизнь. Те из нас, кто варился в этом кotle уже много месяцев, хотели получить взгляд со стороны на проблемы, с которыми мы столкнулись. Чтобы быстрее ввести в курс дела сотрудников, привлеченных для устранения оставшихся несоответствий в данных, мы работали с ними в паре. Контекст идеально подходил для демонстрации инструментов миграции в систему Vitess и поэтапного процесса развертывания, а также для обсуждения новых схем. Работа была утомительной. Но появление дополнительных специалистов резко ускорило темп и помогло устраниТЬ последние несоответствия.

Мы не убрали различия полностью, но сочли устранение 99,999 % достаточным. Так как было понятно, что строки таблицы `channel_members` быстро меняются, когда пользователь читает сообщения в Slack (в частности, меняется состояние `last_read`), мы согласились на небольшие несоответствия, которые могли быть обусловлены быстрым переходом от записи к чтению. Изучив оставшиеся 0,001 % различий прямо в базе данных сразу после их появления, мы заметили, что строки сходятся к одному состоянию.

Завершение работы в режиме *Dark* стало весомым достижением. Знание того, что все запросы к таблице `channels_members` эффективно выполняются в системе Vitess и возвращают правильные результаты, много значило для успеха всего рефакторинга. Хотя до завершения было далеко, окончание режима *Dark* стало для всех большим облегчением. Наконец, все было готово к переходу в режим *Light* для небольшого числа бета-пользователей внутри компании.

Режим *Light*

В режиме *Light* мы собирались протестировать данные выполнения запросов в кластере Vitess, чтобы удостовериться в том, что переключение трафика на новые таблицы не приведет к регрессиям для пользователей. Мы были уверены в небольшом количестве ошибок, в основном из-за проделанной на прошлых этапах работы по устраниению расхождений в данных. Но так как членство в каналах лежит в основе работы Slack, ошибки, если были, рисковали оказаться довольно серьезными.

Поэтому наращивать мощность режима *Light* мы начали с небольшой группы добровольцев из Slack, рассчитывая постепенно включить этот режим для всех клиентов. Большинство функционала работало нормально. Но внезапно оказалось, что после присоединения к каналу некоторые пользователи не могли отправлять сообщения. Мы немедленно свернули эксперимент и углубились в журналы запросов, которые хранили на всех хостах баз данных в течение двух часов. Там можно было отследить любые изменения, связанные с членством пользователя в канале, и ответственные за них функции.

Мы быстро определили виновника. Это был фоновый процесс, запускаемый после того, как любой пользователь приложения Grid на уровне рабочего пространства присоединялся к каналу, членом которого он раньше

был. Этот процесс ищет строки членства с каноническим идентификатором пользователя и заменяет его локальным. Но новая схема базы данных в системе Vitess намеренно использовала канонические идентификаторы пользователей. А значит, строку о членстве пользователя с переписанным идентификатором было невозможно найти, и это не давало отправлять сообщения в канал.

Мы не понимали, почему этот процесс вообще есть. Важно было определить, стоит ли сохранять такое странное поведение или это ошибка, которую нужно исправить. Изучение истории бесед в Slack и истории `git` за несколько лет показало, что код был написан, чтобы решить специфичную для функционала приложения Enterprise Grid проблему. Тогда периодически создавались строки-заполнители для членства с каноническими идентификаторами пользователей, которые обновлялись, когда пользователи снова присоединялись к каналам.

Эта проблема не проявлялась ни при борьбе с несоответствиями, которую мы вели в режиме *Dark*, ни во время нескольких раундов ручного контроля качества (QA), ни в написанных нами модульных тестах. Она появилась только при конкретных необычных обстоятельствах. К счастью, мы решили, что этот процесс больше не нужен, и удалили его. Задача была решена!

На внедрение режима *Light* для всех клиентов ушел месяц. После того как мы с небольшой группой добровольцев убедились в корректности данных в кластере Vitess, наращивание объемов продолжилось. Мы начали с нашего экземпляра Slack, потом перешли к бесплатным пользователям, чуть позднее к платным клиентам и, наконец, к самым крупным корпоративным клиентам. Во время наращивания трафика мы заметили, что клиент с самым большим числом общих каналов сталкивался с тайм-аутами у API, который вызывался при просмотре одного из них (`conversations.view`). Мы быстро отследили запрос к таблице `channels_members`, который выполнялся во время вызова API и сопровождался тайм-аутом. К сожалению, из-за редкости этого запроса мы не узнали о проблеме в режиме *Dark*. Мы немедленно отключили для этого клиента режим *Light*, внесли корректизы в запрос и сразу же вернулись обратно.

Режим *Sunset*

Всего через три дня после успешного переключения всех клиентов в режим *Light* мы начали последний этап — режим *Sunset*. Хотя двойная запись в оба

источника данных продолжалась, запросы на чтение направлялись только в новые кластеры в системе Vitess. Включив для пользователей режим *Sunset*, мы на 22 % снизили количество запросов к нашим перегруженным старым системам, предоставив им необходимую передышку. На рис. 11.6 показано падение объема запросов, которое мы наблюдали в шардах по рабочим пространствам.

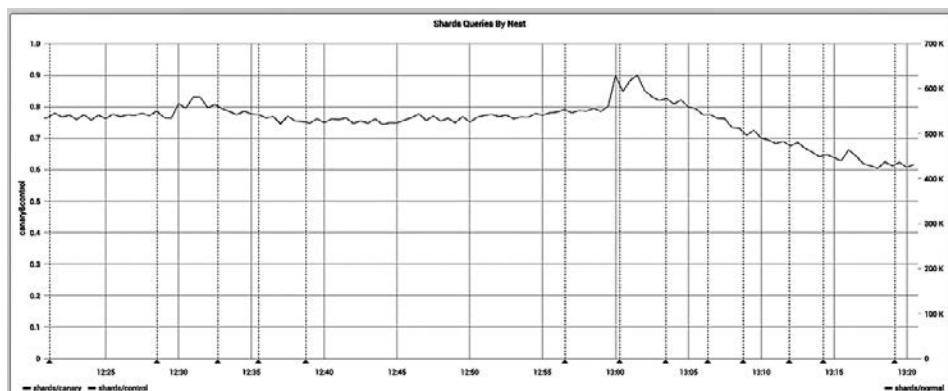


Рис. 11.6. Удаление запросов на чтение из устаревших кластеров, шардированных по рабочим пространствам

Заключительные шаги

По завершении режима *Sunset* осталось несколько важных задач. После того как мы перенесли зависимости хранилища данных и начали брать данные о членстве в каналах из системы Vitess, нужно было отказаться от старых таблиц `channels_members`, шардированных по рабочим пространствам. Примерно за месяц мы это сделали. Следующие недели были потрачены на приведение в порядок библиотеки *Unidata*, тщательное удаление всех переключателей функционала и логических схем, отвечающих за двойную запись.

Огромным шагом вперед стал отказ от записи в старые шарды. Удалив 50 % операций записи, мы полностью устранили задержку репликации в корпоративном сегменте для нашего крупнейшего клиента (VLB, см. главу 10). До этого запросы на запись шли непрерывным потоком, и задержка репликации

в шарде порой превышала 20 минут. На рис. 11.7 можно увидеть резкое падение числа запросов на запись в корпоративный шард VLB.

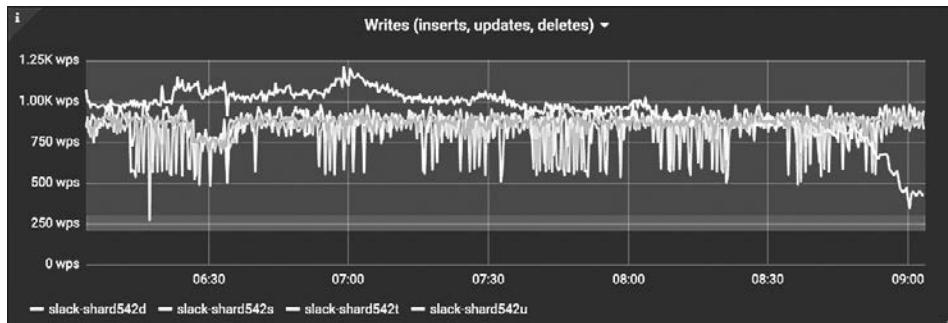


Рис. 11.7. Удаление запросов на запись в шард VLB

На рис. 11.8 отчетливо видно, как после уменьшения количества запросов на запись прекратились внезапные задержки репликации.

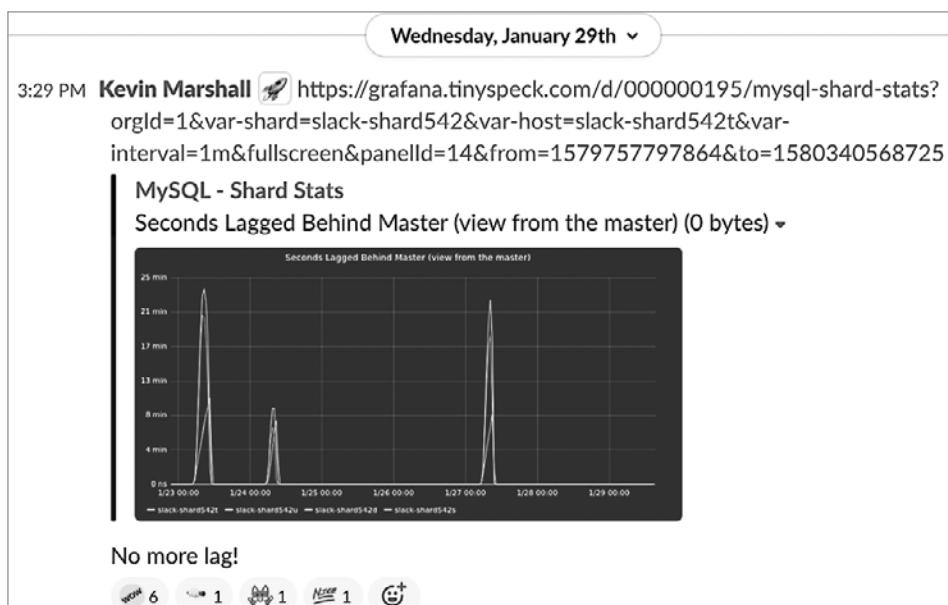
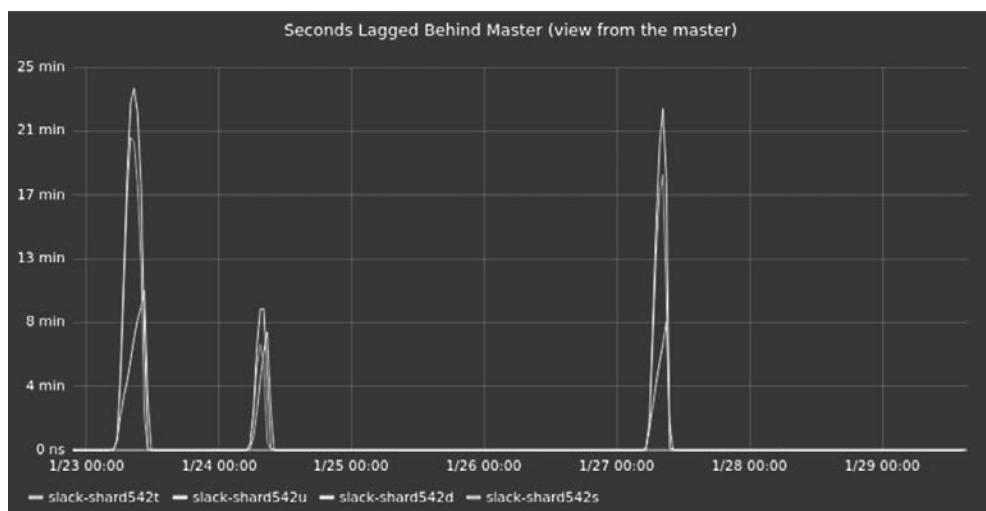


Рис. 11.8. Задержек репликации больше нет!

Вот крупный план графика с рис. 11.8:



К сожалению, когда мы почти закончили работу, началась пандемия коронавируса. Это привело к закрытию наших офисов по всему миру и переводу персонала на удаленную работу. Спрос на Slack резко вырос. Продукт с головокружительной скоростью привлекал новых клиентов, а уже имеющиеся отправляли больше сообщений, чем когда-либо прежде. Команда разработчиков инфраструктуры, в том числе те, кто занимался миграцией таблиц `channels_members`, срочно занялись масштабированием наших систем.

Мы были счастливы завершить рефакторинг, но отпраздновать это достижение не смогли.

После завершения этого проекта инженеры Slack начали придумывать, как использовать преимущества переноса таблиц в новые шарды. Вскоре стали появляться прототипы нового функционала, причем процесс стартовал, когда мы еще находились в режиме *Sunset*. Команды новых проектов были быстро укомплектованы. Перед ними стояла цель воспользоваться преимуществами новой модели данных и упростить другие запросы в приложении Grid и в общих каналах.

Извлеченные уроки

Как и в предыдущем случае, миграция таблицы `channel_members` в Vitess преподнесла нам ряд важных уроков. Посмотрим, как можно было бы улучшить проект, как мы могли бы обеспечить более реалистичные оценки и быстрее найти нужных людей в команду. Затем я расскажу, как мы добились успеха, подробно описав принятые в самом начале решение об увеличении объема проекта и достоинства нашей простой коммуникационной стратегии.

Реалистичные оценки

К моменту переноса таблицы `channels_members` ряд миграций в систему Vitess уже был выполнен. Мы создали инструменты для этой процедуры и постоянно дорабатывали их, с каждой итерацией делая процесс проще и безопаснее. Изначально наши оценки были основаны на опыте последних миграций, которые были выполнены гораздо быстрее первых. Поэтому мы оптимистично предполагали, что эта пройдет так же гладко.

Но мы не учли некоторые особенности таблицы `channels_members`. Во-первых, запросов оказалось намного больше, чем при любой из наших прошлых миграций. Во-вторых, мы шардировали данные по двум ключам (по пользователям и по каналам), а не по одному, как раньше. Наконец, так как мы решили применять канонические идентификаторы пользователей, в схему таблицы были внесены значительные изменения для повышения производительности разработчиков. Но из-за этого проект стал еще сложнее. И при оценках следовало учитывать эти важные решения и их последствия.

Когда мы не уложились в запланированный срок, руководство стало пристальнее следить за проектом. К счастью, мы смогли получить дополнительные ресурсы и продвинуть рефакторинг вперед. Но наша предварительная оценка не оправдала возложенных на нее ожиданий.

Нереалистичные оценки могут иметь последствия и посерьезнее. Например, рефакторинг перестанет быть приоритетной задачей, а руководство разочаруется в ваших способностях управлять крупными программными проектами. Это плохо скажется на ваших карьерных перспективах. Если бы

мы смогли выделить время на мозговой штурм и, опираясь на стратегии из главы 4, подумать обо всех возможных ловушках, сроки выполнения проекта были бы гораздо реалистичнее.

Коллеги, которые вам нужны

В начале проекта мы думали, что с большей частью работы лучше всего справляться инженеры по инфраструктуре. При необходимости можно было обратиться к инженерам по продукту, чтобы задать вопросы или попросить о разовой проверке кода. Лишь после столкновения с трудностями при разбиении запросов с оператором `JOIN` мы попросили более значительной поддержки от группы проектировщиков продукта. Именно тогда мы поняли, что темп можно ускорить, привлекая к сотрудничеству коллег, хорошо знакомых с подлежащими миграции запросами. Их вмешательство помогло устраниТЬ несоответствия в данных, которые привели к странному поведению мессенджера. Если бы они участвовали в проекте с самого начала, переносить запросы (в том числе с оператором `JOIN`) можно было бы быстрее и правильнее, сэкономив немало времени.

Как я уже писала в главе 5, иногда товарищи по команде не совсем подходят для рефакторинга. У крупных работ по внедрению улучшений далеки от последствия. Поэтому к ним часто привлекают специалистов разной направленности из разных команд. Команда, с которой вы начинаете проект, очень редко доходит до конца в том же составе. Если вы считаете, что текущий состав не справляется с задачами, выясните, каких людей вам не хватает, и найдите их. Если окажется, что нужно больше ресурсов, чем вы ожидали, попросите.

Планирование объема работ

Важным решением, принятым на ранней стадии рефакторинга, было использование в схемах таблицы `channels_members` для системы Vitess канонических идентификаторов пользователей для всех связанных с идентификаторами пользователей столбцов. Мы знали, что Slack стремился полностью перейти на канонические идентификаторы, но этот переход вряд ли мог завершиться раньше переноса нашей таблицы.

Решив работать с каноническими идентификаторами пользователей, мы намеренно увеличили объем рефакторинга. Мы могли сначала перейти

к этим идентификаторам в наших устаревших кластерах, шардированных по рабочим пространствам, и только после их полного обновления мигрировать на Vitess. Или можно было перенести таблицу без перехода к каноническим идентификаторам и запустить процесс их миграции только потом. Но мы постарались решить обе задачи одновременно для экономии времени и сил (хотя мы и не могли измерить это, но были уверены, что это так!).

В главе 4 мы узнали, что лучше придерживаться умеренного объема работ. Так рефакторинг можно завершить в разумные сроки, и он не затронет слишком большую площадь. Но бывают обстоятельства, когда разумнее взять в работу дополнительную область, так как это сделает проект успешнее. О такой возможности важно помнить на этапе планирования проекта. А принимать решение о добавлении чего-то в проект лучше ближе к его началу. В этом случае на этапе знакомства с планом выполнения заинтересованные стороны смогут высказать свое мнение о расширении проекта, и вы сможете согласовать все ожидания.

Выбор места коммуникации

На протяжении рефакторинга все рабочие обсуждения, координация и публикации обновлений происходили в канале нашего проекта `#feat-vitess-channels`. Это было основное место коммуникации, и новые сообщения видели все. Там можно было задать вопросы или прислать код для рецензирования и очень быстро получить ответ. Во время рефакторинга в режиме *Light* добровольцы, помогавшие с новыми запросами, сообщали в канале `#feat-vitess-channels` об ошибках и странном поведении, с которыми они столкнулись. Словом, на этом канале можно было найти всю информацию, связанную с миграцией таблицы `channels_members` в систему Vitess.

Главное, что на канале `#feat-vitess-channels` мы поддерживали друг друга. Рефакторинг затягивался, люди подключались к проекту и выходили из него, а мы продолжали бороться с проблемами в режиме *Dark*. Но сохранять оптимизм становилось все труднее. Иногда заглядывали коллеги из других отделов и писали «вы справитесь!» в ответ на еженедельное обновление статуса. Небольшие продуманные жесты поддержки могут сильно повысить моральный дух команды. Наличие удобного канала коммуникации с группой рефакторинга сделало эти жесты поддержки обычным явлением.

Когда все связанное с проектом обсуждается в одном месте, всем заинтересованным лицам легко оставаться на одной волне. Товарищи по команде могут

делиться успехами и жаловаться на трудности. Внешние заинтересованные стороны легко узнают о последних достижениях, не беспокоя команду вопросами. Но главное, что это место поддержки и ободрения (подробнее об этом см. в главе 7).

План развертывания

Перенос таблицы членства в каналах на Vitess имел четкую стратегию развертывания, разделенную на четыре этапа. На каждом новый функционал включался для разных групп пользователей (сначала пользователи из нашей компании, затем бесплатные, постоянные платные и самые крупные клиенты). Мы применяли высоконадежные инструменты, разработанные специально для облегчения миграции на Vitess. Именно они позволили нам быстро наращивать (и снижать) в каждом режиме трафик запросов для отдельных сегментов пользователей, выбирая нужный темп.

Все эти факторы помогали быстро продвигаться. Но самым эффективным фактором стала наша способность немедленно откатиться на исходные позиции сразу, как только что-то идет не так. Именно благодаря этому мы не боялись энергично идти вперед. Особенно это пригодилось в режиме *Light*, когда для чтения данных из кластера в Vitess были привлечены волонтеры.

Даже самый тщательно спланированный и выполняемый рефакторинг не избавлен от ошибок. Часто выявить их все до начала развертывания попросту невозможно. Если при вводе каждого критически важного новшества следить за теми, кого затрагивают изменения, и при необходимости быстро откатываться назад, прогресс придет гораздо быстрее, а разрушительные регрессии получится выявить до того, как они приведут к серьезным последствиям.

Основные моменты

Вот наиболее важные уроки проекта по переносу таблицы `channels_members` из кластеров, разбитых на шарды по рабочим пространствам, в шардированные по пользователям и каналам кластеры в системе Vitess.

- Важно реалистично оценивать свои силы. Оптимизм — это прекрасно, но у сорванных сроков могут быть серьезные последствия.

-
- Подбирайте в команду нужных специалистов. Ваши товарищи по команде могут быть не самыми подходящими кандидатурами для рефакторинга. Не бойтесь просить дополнительные ресурсы, когда они нужны.
 - Тщательно планируйте объем проекта. Любые дополнительные операции важно учитывать на этапе планирования, чтобы верно оценить сроки выполнения.
 - Заранее выберите одно место для обсуждения всего, что связано с проектом, и не меняйте его.
 - Продумайте план развертывания и заранее создайте нужный инструментарий, чтобы максимально упростить расширение (и сужение) границ проекта.

06 авторе

Мод Лемер (Maude Lemaire) — инженер компании Slack Technologies, Inc. Отвечает за масштабирование продукта и поддержку крупных клиентов компании. На работе в основном занимается поиском нужных специалистов, переговорами, рефакторингом громоздких фрагментов кода, консолидацией избыточных схем баз данных и созданием инструментов для других разработчиков. Мод обеспокоена вопросом удобства процесса разработки, поэтому активно ищет более простые и эффективные способы структурирования кода на всех уровнях.

С отличием окончила Университет Макгилла. Имеет степень бакалавра по специальности «программная инженерия».

Об обложке

На обложке книги изображены моржи (*Odobenus rosmarus*) — крупные морские млекопитающие, обитающие в Арктике и субарктических регионах. Моржи славятся длинными острыми клыками, которые помогают им ломать лед, вылезать из воды, бороться за первенство в стаде и защищаться от хищников. Их толстую кожу покрывают короткие желто-бурые волосы. Цвет окраски этих животных варьируется от серого до желто-коричневого. Толстый слой подкожного жира сохраняет тепло, позволяя моржам выживать в суровых условиях.

Эти медленно передвигающиеся животные любят жить рядом с мелководьем, чтобы иметь легкий доступ к пище. В поисках льда оптимальнойтолщины они совершают сезонные миграции. Короткие передние ласты и более крупные задние плавники помогают этим существам весом около тонны перемещаться в воде, а для навигации и поиска пищи они используют не глаза, а усы. Питаются моржи в основном моллюсками и ракообразными. Но известны случаи поедания более крупных животных, таких как морские птицы и даже тюлени.

Глобальное изменение климата и человеческая халатность привели к тому, что вид стал относиться к группе риска. На обложках книг издательства O'Reilly часто появляются животные, находящиеся под угрозой исчезновения. Все они важны для мира.

Цветная иллюстрация сделана Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из книги *Natural History of Animals* Карла Фогта и Фридриха Шпехта.

Мод Лемер

**Масштабируемый рефакторинг.
Возвращаем контроль над кодом**

Перевела с английского И. Рузмайкина

Руководитель дивизиона
Ведущий редактор
Литературный редактор
Художественный редактор
Корректоры
Верстка

*Ю. Сергиенко
Н. Гринчик
Т. Сажина
В. Мостипан
С. Беляева, Г. Шкатова
Г. Блинов*

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: 05.2022. 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 30.03.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1000. Заказ 0000.

Роберт Мартин

ЧИСТЫЙ AGILE. ОСНОВЫ ГИБКОСТИ



Прошло почти двадцать лет с тех пор, как появился Манифест Agile. Легендарный Роберт Мартин (Дядя Боб) понял, что пора стряхнуть пыль с принципов Agile и заново рассказать о гибком подходе не только новому поколению программистов, но и специалистам из других отраслей. Автор полюбившихся айтишникам книг «Чистый код», «Идеальный программист», «Чистая архитектура» стоял у истоков Agile. «Чистый Agile» устраниет недопонимание и путаницу, которые за годы существования Agile усложнили его применение по сравнению с изначальным замыслом.

По сути Agile — это всего лишь небольшая подборка методов и инструментов, помогающая небольшим командам программистов управлять небольшими проектами, но приводящая к большим результатам, потому что каждый крупный проект состоит из огромного количества кирпичиков. Пять десятков лет работы с проектами всех мыслимых видов и размеров позволяют Дяде Бобу показать, как на самом деле должен работать Agile.

Если вы хотите понять преимущества Agile, не ищите легких путей — нужно правильно применять Agile. «Чистый Agile» расскажет, как это делать, разработчикам, тестировщикам, руководителям, менеджерам проектов и их клиентам.

КУПИТЬ

Роберт Мартин

ЧИСТЫЙ КОД: СОЗДАНИЕ, АНАЛИЗ И РЕФАКТОРИНГ. БИБЛИОТЕКА ПРОГРАММИСТА



Плохой код может работать, но он будет мешать развитию проекта и компании-разработчика, требуя дополнительные ресурсы на поддержку и «укрощение».

Каким же должен быть код? Эта книга полна реальных примеров, позволяющих взглянуть на код с различных направлений: сверху вниз, снизу вверх и даже изнутри. Вы узнаете много нового о коде. Более того, научитесь отличать хороший код от плохого, узнаете, как писать хороший код и как преобразовать плохой код в хороший.

Книга состоит из трех частей. Сначала вы познакомитесь с принципами, паттернами и приемами написания чистого кода. Затем приступите к практическим сценариям с нарастающей сложностью — упражнениям по чистке кода или преобразованию проблемного кода в менее проблемный. И только после этого перейдете к самому важному — концентрированному выражению сути этой книги — набору эвристических правил и «запахов кода». Именно эта база знаний описывает путь мышления в процессе чтения, написания и чистки кода.

[КУПИТЬ](#)