Вы инженер-программист. Что теперь?

READ ME

СУРОВЫЕ РЕАЛИИ РАЗРАБОТЧИКОВ

КРИС РИККОМИНИ
ДМИТРИЙ РЯБОЙ

THE MISSING README

A GUIDE FOR THE NEW SOFTWARE ENGINEER

by CHRIS RICCOMINI
and DMITRIY RYABOY

no starch
press

San Francisco

КРИС РИККОМИНИ
ДМИТРИЙ РЯБОЙ

READ ME

СУРОВЫЕ РЕАЛИИ РАЗРАБОТЧИКОВ

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018-02
УДК 004.415
Р50

Риккомини Крис, Рябой Дмитрий

Р50 README. Суровые реалии разработчиков. — СПб.: Питер, 2023. — 304 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1972-1

Начинающим программистам требуется нечто большее, чем навыки программирования. Столкнувшись с реальной работой, вы моментально понимаете, что самым нужным вещам, имеющим критическое значение для карьеры, не обучают ни в университетах, ни на курсах. Книга «README. Суровые реалии разработчиков» призвана восполнить этот пробел.

Познакомьтесь с важнейшими практиками инжиниринга, которым обучают разработчиков в ведущих компаниях.

Вы узнаете о том, что вас ждет при устройстве на работу, затем познакомитесь с особенностями кода промышленного уровня, эффективным тестированием, ревью кода, непрерывной интеграцией и развертыванием, созданием проектной документации и лучшими практиками архитектуры ПО. В последних главах описываются навыки гибкого планирования и даются советы по построению карьеры.

Ключевые концепции и лучшие практики для начинающих разработчиков — то, чему вас не учили в университете!

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

ББК 32.973.2-018-02
УДК 004.415

ISBN 978-1718501836 англ.

ISBN 978-5-4461-1972-1

Краткое содержание

Оглавление

Моей семье. Спасибо за вашу любовь и поддержку!
Крис Риккомини

Гите.
Дмитрий Рябой

Об авторах

Крис Риккомини уже более десяти лет занимается разработкой программного обеспечения. За это время он успел поработать специалистом по анализу данных и инженером-программистом в таких крупных технологических компаниях, как PayPal, LinkedIn, WePay и JPMorgan Chase. Крис также активно участвует в создании программ с открытым исходным кодом, занимается стартап-инвестированием и консалтингом.

Дмитрий Рябой работает инженером-программистом с начала 2000-х годов. Сотрудничал со стартапами по разработке программных продуктов для организаций (Cloudera), интернет-компаниями (Ask.com, Twitter) и исследовательскими институтами (Национальная лаборатория им. Лоуренса в Беркли). Участвовал в создании нескольких проектов с открытым исходным кодом, в том числе Apache Parquet. В настоящее время является вице-президентом по разработке программного обеспечения в компании Zymergen.

Благодарности

Мы искренне благодарны нашему редактору Атабаске Витчи — без нее эта книга не была бы такой, как она есть. Благодарим Ким Вимпсетт за редактирование, Джейми Лауэр — за корректуру, а также Билла Полло-ка, Барбару Йен, Катрину Тейлор и других сотрудников издательства No Starch за помощь двум новичкам в процессе написания книги.

Спасибо нашей команде рецензентов: Джой Гао, Алехандро Кросе, Джейсону Картеру, Чжэнляну (Зэйну) Чжу и Рэйчел Гите Шифф — ваши отзывы были бесценны. Спасибо Тодду Палино за отзыв о главах, посвященных операционной деятельности, Мэтью Клауэру — за чест-ный и исчерпывающий отзыв о главе 6, Тому Хэнли, Джонни Киндеру и Киту Вуду — за отзывы о главах по теме управления, а также Мартину Клеппману и Питу Скоморочу — за наставничество и помощь в работе с издателем.

Мы не смогли бы написать эту книгу без поддержки наших работо-дателей и руководителей. Спасибо Крису Конраду, Биллу Клерико, Аарону Кимбаллу и Дуэйну Валцу за то, что они способствовали нам в реализации этого проекта.

Предисловие

Вы устраиваетесь на новую работу с готовностью решать сложные проблемы, писать элегантный код и совершенствовать свое мастерство. Отлично! Поздравляем! Мы надеемся, что вам доведется столкнуться с интересными задачами и, создавая нечто полезное, вы будете работать с умными и увлеченными коллегами.

Однако вскоре вы обнаружите, а может, и уже обнаружили, что умение программировать, то есть использовать компьютеры для решения задач, — только половина дела. Несомненно, это важная часть вашего набора умений и знаний, но, чтобы стать по-настоящему эффективным инженером-программистом, требуются и другие навыки, которые не освоить в университетах. Восполнить подобный недостаток и призвана книга «README. Суровые реалии разработчиков».

В издании описаны современные методы создания, тестирования и эксплуатации программного обеспечения, а также нормы поведения и подходы, повышающие эффективность командной работы. Мы дадим вам практические советы о том, как получать помощь, писать проектную документацию, работать со старым кодом, поддерживать связь во время дежурств, планировать свою нагрузку и взаимодействовать с руководством и остальной командой.

Книга не является исчерпывающей и не содержит все-все, что вам следует знать, — такая задача была бы невыполнимой. Мы сосредоточились на самой важной информации, которая обычно не входит в учебные программы. Темы достаточно сложные, поэтому в конце каждой главы вы найдете раздел «Повышение уровня» с рекомендуемыми источниками для получения дополнительной информации.

В первых нескольких главах мы поговорим о том, чего следует ожидать при устройстве на работу в новую компанию. Затем мы обсудим такие технические темы, как написание кода промышленного уровня,

эффективное тестирование, ревью кода, непрерывная интеграция и развертывание, написание проектной документации и лучшие практики создания архитектуры ПО. Последние три главы посвящены навыкам гибкого планирования и работы с руководством, а также содержат советы по построению карьеры.

При написании книги мы опирались на собственный опыт работы в быстрорастущих компаниях Кремниевой долины, находящихся на стадии pre-IPO, финансируемых венчурным капиталом. Вы можете быть в другой ситуации, но при всех различиях компаний основные принципы универсальны.

«README. Суровые реалии разработчиков» — это книга, которую мы сами хотели бы иметь в начале своего пути и которую мы планируем вручать новым инженерам, приходящим в нашу команду. В процессе ее прочтения вы освоите все навыки, необходимые настоящему инженеру-программисту. Приступаем!

От издательства

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

Мы будем рады узнать ваше мнение!

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

1 Предстоящее путешествие

На протяжении карьеры каждому инженеру-программисту приходится выступать в роли новичка, инженера, техлида и иногда даже руководителя. Большинство новоиспеченных инженеров обладают хорошей технической подготовкой, но не имеют практического опыта. Следующие главы помогут вам достичь первой вехи в своей карьере, то есть научиться безопасно вносить изменения в код и эффективно взаимодействовать с командой.

Достичь первой вехи не так уж и просто, поскольку нужные сведения разбросаны по Всемирной паутине или, что еще хуже, хранятся в чьей-то голове. В этой книге собрана вся ключевая информация, необходимая для достижения успеха. Но что именно подразумевается под понятием «успешный инженер-программист» и как им стать?

Пункт назначения

Для продвижения вперед по карьерной лестнице потребуется развивать свои навыки в нескольких основных направлениях.

- **Техническая подготовка.** Вы знаете основы computer science. Вы умеете использовать интегрированные среды разработки (IDE), системы сборки, отладчики и фреймворки для автоматизации тестирования. Знаете, что такое непрерывная интеграция, метрики и мониторинг,

конфигурации и системы управления пакетами. Вы активно пишете и улучшаете тестовый код. Вы учитываете особенности операций при принятии архитектурных решений.

- **Мастерство.** Вы создаете ценный продукт, решая задачи с помощью кода, и понимаете связь между вашей работой и бизнесом компании. Вам уже доводилось создавать и внедрять функции малого и среднего масштаба. Вы умеете писать, тестировать и рецензировать код. Вы готовы участвовать в дежурствах и решать операционные вопросы. Вы активны и надежны. Вы участвуете в технических совещаниях, групповых обсуждениях и собеседованиях.
- **Коммуникация.** Вы умеете четко доносить свои мысли как в письменной, так и в устной форме. Вы можете эффективно давать и получать обратную связь. В неоднозначных ситуациях способны обратиться за помощью и разъяснениями. Вы в конструктивной манере поднимаете вопросы и выявляете проблемы. Вы оказываете помощь, когда это возможно, и к вашим советам прислушиваются коллеги. Вы документируете свою работу. Вы составляете понятные проектные документы и запрашиваете обратную связь. Вы проявляете терпение и уважение в общении с другими.
- **Лидерские качества.** Вы способны самостоятельно реализовывать хорошо спланированные проекты. Вы быстро учитесь на своих ошибках. Вы хорошо адаптируетесь к нововведениям и справляетесь с нестандартными ситуациями. Вы принимаете активное участие в проектном и квартальном планировании. Вы помогаете новым членам команды осваиваться на рабочем месте. Вы предоставляете содержательную обратную связь своему руководителю.

Карта путешествия

Чтобы добраться до пункта назначения, вам потребуется карта. Материал этой главы поможет вам сориентироваться в книге и определиться с действиями в начале карьеры. Стартуете вы с пика Новичка, как и все начинающие специалисты. Оттуда вы отправитесь вниз по реке Интенсивной работы, когда приступите к написанию кода и изучению соглашений

и практик, принятых в вашей компании. Со временем вы достигнете мыса Активного участника, где внедрите ряд важных функций. По ходу вам придется преодолевать штормы в Операционном океане. В конце концов вы окажетесь в тихих водах бухты Компетентности.

Многие абзацы мы снабдили ссылками на главы. Вы можете читать все подряд или переходить к тем главам, которые вас интересуют больше всего. Некоторые ссылки на главы появляются более одного раза, и это сделано намеренно. Главы сгруппированы по темам, которые охватывают весь ваш карьерный путь. Возвращаясь к уже пройденному материалу, вы каждый раз будете открывать для себя что-то новое.

Пик Новичка

Вы начинаете свой путь в статусе новичка и знакомитесь с компанией, командой и методами работы. Посетите вводные встречи. Настройте среду разработки и доступ к системе, а также выясните особенности командной работы. Просмотрите документацию и обсудите ее с коллегами. Внесите свой вклад, заполнив обнаруженные вами пробелы в документах.

В компании может действовать программа адаптации для новых сотрудников, которая поможет вам освоиться на новом рабочем месте. В рамках такой программы вам расскажут о принятых в компании практиках, проведут экскурсию и представят руководству, а также познакомят с новыми сотрудниками из других отделов — вашими будущими коллегами. Если в компании нет такой программы, попросите своего руководителя рассказать об организационной структуре (кто за что отвечает и кто кому подчиняется), о различных отделах и их взаимосвязи. Делайте заметки.

В некоторых компаниях предусмотрены дополнительные процедуры адаптации новых инженеров-программистов, которые помогают им получить доступ к системам и настроить среду разработки, а также ознакомиться с кодом и поэкспериментировать с ним. Если в вашей компании такой процедуры не предусмотрено, вы можете ее создать! Для этого

запишите все, что вам пришлось сделать в процессе подготовки к работе. (См. главу 2 «Достижение осознанной компетентности».)

ЗАКОНЫ КАННИНГЕМА И ПАРКИНСОНА

Мы советуем вам делать заметки обо всех соглашениях, практиках онбординга и традициях команды. Будьте готовы к тому, что ваши заметки будут комментировать и исправлять. Не принимайте это близко к сердцу: ваша цель состоит не в том, чтобы написать идеальный документ, а в том, чтобы спровоцировать обсуждение, позволяющее конкретизировать детали. Как гласит *закон Каннингема*, «лучший способ получить правильный ответ в интернете — это не задать вопрос, а опубликовать неверный ответ».

Будьте готовы к тому, что дискуссия по тривиальным вопросам затянется надолго. В англоязычной литературе появился даже специальный термин bikeshed effect (дословно «эффект велосипедного сарая»), ставший метафорой закона тривиальности — его сформулировал Сирил Норткот Паркинсон. В качестве примера Паркинсон привел заседание комитета, которому поручено рассмотреть проект электростанции. Поскольку проект слишком сложен для обсуждения, комитет утверждает его за считаные минуты, а затем тратит еще около часа на выбор материалов для постройки велосипедного сарая рядом с электростанцией. Этот эффект весьма часто проявляется в технической работе.

Вам должны дать небольшое задание, чтобы вы познакомились со способами изменения кода и его внедрения в эксплуатацию. Если вы не получили такого задания, попросите дать вам возможность внести какое-нибудь небольшое, но полезное изменение. Это может быть даже обновление комментария: ваша цель — понять процедуру, а не произвести впечатление. (См. главу 2 «Достижение осознанной компетентности» и главу 8 «Доставка программного обеспечения».)

Настройте редактор кода или IDE. Используйте ту IDE, с которой работает ваша команда. Если вы не знакомы с данной средой, найдите информацию в интернете — в дальнейшем освоение IDE сэкономит вам

много времени. Настройте IDE с учетом стандартов форматирования кода, а для этого выясните, каковы они и как их применять. (См. главу 3 «Работа с кодом».)

Убедитесь в том, что руководитель включил вас в число участников всех встреч и совещаний. Напомните своему руководителю о назначении личной встречи, если это практикуется в вашей компании. (См. главу 12 «Гибкое планирование» и главу 13 «Взаимодействие с менеджментом».)

Река Интенсивной работы

Выполнив задание для новичков, вы приступите к работе над настоящим проектом вместе с командой. Скорее всего, вы будете работать с уже существующей кодовой базой. Если что-то покажется непонятным, не стесняйтесь задавать вопросы. Просите, чтобы коллеги чаще проверяли вашу работу. (См. главу 3 «Работа с кодом» и главу 7 «Ревью кода».)

На этом этапе решающее значение имеет обучение. Изучите процесс создания, тестирования и развертывания кода. Ознакомьтесь с пул-реквестами (запросами на включение изменений в коде) и ревью кода. Смело запрашивайте дополнительную информацию. Участвуйте в технических семинарах, неформальных встречах, групповых обсуждениях, программах наставничества и т. п. (См. главу 2 «Достижение осознанной компетентности», главу 5 «Управление зависимостями», главу 6 «Тестирование» и главу 8 «Доставка программного обеспечения».)

Самое время наладить отношения с непосредственным руководителем. Ознакомьтесь с его стилем работы, выясните его ожидания и поговорите о своих целях. Если ваш руководитель проводит индивидуальные встречи, будьте готовы к ним. Как правило, начальники стремятся отслеживать прогресс, поэтому спросите руководителя о предпочтительной форме отчетности о проделанной работе. (См. главу 13 «Взаимодействие с менеджментом».)

Вероятно, в этот период вы побываете на своем первом совещании по вопросам планирования, и, скорее всего, это будет собрание по

планированию спринта. Вы также можете присоединиться к ретроспективным или общим собраниям. Попросите ознакомить вас с календарным планом проекта и процессом планирования развития. (См. главу 12 «Гибкое планирование».)

Мыс Активного участника

Мыса Активного участника вы достигнете, как только начнете работать над более крупными задачами и функциями. К тому времени команда уже будет доверять вам самостоятельную работу, и вы научитесь писать удобный для оператора код промышленного уровня, позволяющий правильно управлять зависимостями и успешно проходящий тесты. (См. главу 3 «Работа с кодом», главу 4 «Написание работоспособного кода», главу 5 «Управление зависимостями» и главу 6 «Тестирование».)

На этом этапе вы уже должны помогать товарищам по команде. Участвуйте в ревью кода и будьте готовы делиться идеями и предоставлять обратную связь. Ваши коллеги могут забыть, что вы присоединились совсем недавно, поэтому смело задавайте вопросы, если чего-то не понимаете. (См. главу 2 «Достижение осознанной компетентности», главу 7 «Ревью кода» и главу 10 «Процесс технического проектирования».)

В большинстве компаний предусмотрены квартальные циклы планирования и постановки целей. Участвуйте в процессе командного планирования и определяйте цели и ключевые результаты вместе со своим руководителем. (См. главу 12 «Гибкое планирование» и главу 13 «Взаимодействие с менеджментом».)

Операционный океан

Работая над более крупными задачами, вы узнаете, каким образом код доставляется пользователям. Процесс включает в себя такие этапы, как тестирование, сборка, релиз, развертывание и внедрение. Налаживание процесса требует определенных навыков. (См. главу 8 «Доставка программного обеспечения».)

После внедрения изменений вы будете отвечать за эксплуатацию программного обеспечения вашей команды, что потребует от вас большого напряжения и выдержки: нестабильная работа ПО может вызвать недовольство клиентов. Вам предстоит отлаживать программное обеспечение в режиме реального времени, используя метрики, журналы и инструменты трассировки. На этом этапе вас также могут попросить принять участие в дежурстве. Занимаясь операционной деятельностью, вы увидите, как код отзывается на действия пользователей, и научитесь обеспечивать работу своих программ. (См. главу 4 «Написание работоспособного кода» и главу 9 «Дежурство».)

Бухта Компетентности

Теперь ваша команда уже готова поручить вам реализацию небольшого проекта. От вас потребуется написание проектной документации и помощь с планированием. В процессе разработки программного обеспечения вам откроется новый уровень сложности. Не останавливайтесь на первом варианте проекта: обдумайте возможные компромиссы и учтите развитие системы во времени. (См. главу 10 «Процесс технического проектирования», главу 11 «Создание эволюционной архитектуры» и главу 12 «Гибкое планирование».)

К этому моменту изначальный блеск вашей работы потускнеет, и вы начнете замечать недостатки в архитектуре, среде тестирования, системе сборки и развертывании. На данном этапе вы научитесь сочетать регулярную работу с обслуживанием и рефакторингом кода. Не пытайтесь все переписать. (См. главу 3 «Работа с кодом».)

У вас также будут появляться мысли относительно рабочих процессов вашей команды. Запишите свои наблюдения по поводу того, что работает, а что нет, и обсудите свои идеи в ходе личной встречи с руководителем. (См. главу 13 «Взаимодействие с менеджментом».)

Сейчас вам стоит уделить время постановке долгосрочных целей и анализу эффективности. Поработайте над этим с руководителем и запросите обратную связь от коллег. Обсудите с руководителем

свои карьерные устремления, видение дальнейшей работы, проекты и идеи. (См. главу 13 «Взаимодействие с менеджментом» и главу 14 «Навигация по карьерной лестнице».)

В путь!

Теперь у вас есть карта и вы знаете, куда направляетесь. Достигнув бухты Компетентности, вы станете полноценным инженером-программистом, способным работать вместе с командой над созданием важных функций. А наша книга поможет вам лучше ориентироваться в пути. Итак, путешествие начинается.

2 Достижение осознанной компетентности

В своей статье *Teaching for Learning* Мартин М. Бродвелл выделил четыре уровня компетентности: неосознанная некомпетентность, осознанная некомпетентность, осознанная компетентность и неосознанная компетентность. На уровне *неосознанной некомпетентности* вы не способны правильно решить задачу и не отдаете себе в этом отчета. *Осознанная некомпетентность* означает, что вы не можете правильно решить задачу, но знаете об этом. На уровне *осознанной компетентности* вы способны решить задачу с некоторым усилием. Наконец, на уровне *неосознанной компетентности* вы можете решить задачу без особого труда.

Все инженеры начинают с уровня осознанной или неосознанной некомпетентности. Даже если вы знаете все о разработке ПО (что в принципе невозможно), вам придется изучить рабочие процессы и правила конкретной компании. Вам также нужно будет освоить определенные практические навыки. Ваша цель — как можно быстрее достичь уровня осознанной компетентности.

Большая часть этой главы посвящена тому, как учиться самостоятельно и получать необходимую помощь. Учеба вне учебного заведения — особый навык, и далее вы найдете несколько советов по развитию привычки к самостоятельному обучению. Мы также подскажем, как найти баланс между слишком частым и недостаточно частым обращением за

помощью. В конце главы мы обсудим синдром самозванца и эффект Даннинга — Крюгера, которые могут вызвать у новых инженеров или неуверенность, или, наоборот, чрезмерную самоуверенность, сдерживая тем самым их рост. Мы объясним, как избегать обеих крайностей и выявлять их признаки. Самостоятельное обучение в сочетании с постановкой правильных вопросов и избеганием ловушек неуверенности и самоуверенности позволит вам в кратчайшие сроки достичь осознанной компетентности.

Научитесь учиться

Обучение поможет вам стать компетентным инженером и положит начало вашему дальнейшему процветанию. Сфера разработки программного обеспечения постоянно развивается, поэтому, кем бы вы ни были — новоиспеченным выпускником или опытным программистом, — если вы не учитесь, вы отстаете.

В этом разделе кратко описаны различные подходы к обучению. Не пытайтесь одновременно использовать их все! Это приведет лишь к выгоранию. Цените свое личное время: непрерывный рост очень важен, но не следует работать каждую свободную минуту. Выбирайте подходы, исходя из своих условий и природных склонностей.

Пройдите предварительное обучение

Первые несколько месяцев на новом месте работы потратьте на изучение рабочих процессов: это позволит вам участвовать в обсуждении вопросов разработки и решении операционных проблем, а также в дежурстве и ревью кода. На данном этапе у вас может появиться ощущение дискомфорта, поскольку вместо разработки программного обеспечения вам придется тратить время на чтение документации и знакомство с инструментами. Не волнуйтесь, для всех будет вполне ожидаемым, что вы выделите некоторое время на то, чтобы набрать обороты.

Предварительное обучение представляет собой настолько ценное вложение, что многие компании специально разрабатывают учебные программы для новых сотрудников. Например, компания Facebook проводит шестинедельный интенсивный курс обучения новых инженеров.

Учитесь на практике

Предварительное обучение не сводится к чтению документации. В ходе практической работы есть возможность узнать гораздо больше, чем в процессе чтения. Вы должны написать код и ввести его в эксплуатацию. Если вы боитесь что-нибудь сломать, не переживайте: как правило, руководители не ставят новичков в такие условия, когда те могут нанести серьезный ущерб (хотя в редких случаях новые сотрудники все же выполняют высокорискованные операции).

Постарайтесь понять, какое влияние окажет ваша работа, и действуйте с должным уровнем осторожности. Например, при написании модульного теста можно проявлять меньшую осторожность и, следовательно, действовать быстрее, чем при изменении индексов в базе данных с высоким трафиком.

ДЕНЬ, КОГДА КРИС УДАЛИЛ ВЕСЬ КОД

Во время одной из своих первых стажировок Крис работал над проектом вместе с сеньор-разработчиком. После внесения в код некоторых изменений Крису нужно было их внедрить. Сеньор-разработчик показал, как добавить код в систему управления версиями CVS. Слепо следуя инструкциям, Крис выполнил все необходимые действия, включая ветвление, тегирование и слияние. Потом вернулся к своим делам и в конце рабочего дня отправился домой.

На следующее утро Крис пришел на работу в хорошем настроении и весело поприветствовал коллег. Они попытались ответить тем же, но вид у них был удрученный. Когда Крис спросил, в чем дело, ему сообщили, что он повредил репозиторий CVS, в результате чего весь код компании был уничтожен. Его коллеги не спали ночь, отчаянно пытаясь восстановить хоть что-нибудь. В конце концов им удалось вернуть большую часть кода (за исключением коммитов Криса и некоторых других сотрудников). Крис был потрясен. Руководитель отвел его в сторону и посоветовал не переживать. Сказал, что Крис поступил правильно, работая под руководством сеньор-разработчика, однако ошибки случаются.

Подобную историю может рассказать любой программист. Старайтесь понимать, что вы делаете, но знайте, что и такие ситуации бывают.

Ошибки неизбежны. Быть инженером-программистом сложно, и все понимают, что никто не застрахован от неудач. Задача вашего руководителя и команды — создать систему защиты, способную предотвратить фатальные последствия ошибок. Допустив промах, не мучайте себя: извлеките уроки и двигайтесь дальше.

Экспериментируйте с кодом

Поэкспериментируйте, чтобы выяснить, как именно работает код: ведь и документация устаревает, и коллеги могут о чем-то забыть. Эксперименты, проведенные вне промышленной среды, совершенно безопасны и поэтому допускают применение более инвазивных техник. Например, вы знаете о вызове некоего метода, но не можете понять, как именно он осуществляется. Проведите эксперимент: сгенерируйте исключение и отобразите трассировку стека или подключите отладчик для определения пути вызова.

Отладчики — ваши лучшие друзья при проведении экспериментов с кодом. Вы можете использовать их для приостановки выполнения кода, а также для просмотра запущенных потоков и для трассировки стека и значений переменных. Подключите отладчик, инициируйте событие и осуществите пошаговое выполнение кода, чтобы увидеть, как именно код обрабатывает данное событие.

Несмотря на то что отладчики являются мощными инструментами, иногда понять поведение программы проще всего с помощью нескольких строк журнала или операторов `print()`. Вы наверняка знакомы с этим методом, однако имейте в виду, что в сложных ситуациях, особенно в случае с многопоточными приложениями, отладка с помощью операторов `print()` может приводить к путанице: операционные системы будут буферизовать записи в стандартный вывод, задерживая отображение данных в консоли, а сообщения, выводимые несколькими потоками, делающими запись в стандартный вывод, будут чередоваться.

Один простой, но на удивление эффективный метод заключается в добавлении оператора `print()` в начале выполнения программы. Такое действие позволит легко отличить модифицированную версию программы от исходной, и вам не придется тратить много часов на понимание загадочного поведения программы, если вместо ее измененной версии вы запустите исходную.

Читайте

Выделите часть рабочего времени на чтение. Может быть немало различных источников: документы, составленные вашей командой, проектная документация, код, бэклоги, книги, статьи и технические сайты. Не пытайтесь прочитать все сразу. Начните с документов команды и проектной документации, чтобы получить общее представление о том, как устроен проект. Уделите особое внимание обсуждениям компромиссов и контекста. После чего можете сосредоточиться на подсистемах, имеющих отношение к вашим первым полученным заданиям.

Как говорит Рон Джеффрис, «код никогда не лжет; порой обманывают комментарии» (https://ronjeffries.com/articles/020-invaders-70ff/i-77/). Читайте код, поскольку он не всегда точно соответствует проектной документации! Не ограничивайтесь кодовой базой своей компании — обращайтесь к высококачественному открытому исходному коду, в частности коду библиотек, с которыми работаете. Не читайте код от начала до конца, как роман: используйте IDE для навигации по кодовой базе. Создайте граф потока управления и состояний для ключевых операций. Изучите структуры данных и алгоритмы. Обратите внимание на обработку пограничных случаев. Следите за идиомами и стилем, выучите местный сленг.

Незавершенные задачи или проблемы отслеживаются с помощью *тикетов*. Ознакомьтесь с тикетами своей команды, чтобы понять, над чем работают ваши коллеги и что вас ожидает в ближайшее время. Хорошим местом для поиска тикетов является бэклог. Старые задачи делятся на три категории: более не актуальные; полезные, но второстепенные; важные, но слишком сложные, чтобы их можно было решить в данный момент. Определите, к какой категории относятся тикеты вашей команды.

Печатные издания и онлайн-ресурсы дополняют друг друга. Книги и статьи отлично подходят для более глубокого изучения предмета, однако следует иметь в виду, что содержащиеся в них сведения, несмотря на свою надежность, могут быть устаревшими. Напротив, такие онлайн-ресурсы, как сообщения в Twitter, блоги и информационные рассылки, содержат менее надежные сведения, но более подходящие для отслеживания текущих тенденций. Только не спешите реализовывать последние идеи, почерпнутые на сайте Hacker News: иногда быть скучным совсем неплохо (подробнее об этом мы поговорим в главе 3).

Присоединитесь к группе для совместного чтения, чтобы быть в курсе последних исследований. Выясните, организует ли ваша компания подобные группы. Если нет, подумайте о том, чтобы организовать ее самостоятельно. Вы также можете присоединиться к местному клубу сообщества Papers We Love, участники которого регулярно читают и обсуждают научные статьи по компьютерной тематике.

УЧИТЕСЬ ЧИТАТЬ КОД

В начале карьеры Дмитрию поручили разобраться в унаследованном Java-приложении. Руководитель хотел внести в него некоторые изменения, а Дмитрий был единственным человеком в команде, кто чувствовал себя достаточно комфортно при работе с языком Java. Исходный код был полон, скажем так, особенностей. В качестве имен переменных использовались буквы *a*, *b*, *c* и т. д. Что еще хуже, переменная *a* в одной функции превращалась в переменную *d* в другой. Не было ни истории изменений кода, ни тестов, а разработчик приложения давно покинул компанию. В общем, настоящее минное поле.

Всякий раз, когда нужно было что-то изменить в кодовой базе, Дмитрий полностью погружался в код и внимательно его читал, переименовывал переменные, отслеживал логику, набрасывал схемы на бумаге, экспериментировал. Работа шла очень медленно. Однако со временем он оценил кодовую базу, которая, как выяснилось, выполняла весьма сложные операции. Дмитрия восхищала способность разработчика программы удерживать все это в голове без нормальных имен переменных. Однажды за обедом он выразил свое восхищение, и коллега посмотрел на него так, будто у Дмитрия выросла вторая голова: «Но у нас нет оригинального исходного кода. Вы работаете с выводом декомпилятора. Никто в здравом уме не стал бы писать код подобным образом!»

Мы не рекомендуем учиться читать код именно так, однако стоит отметить, что данный опыт научил Дмитрия не спешить, вникать в прочитанное и никогда не полагаться на имена переменных.

Смотрите презентации

Хорошая презентация может многому научить. Начните с просмотра видеопрезентаций, записанных в прошлом: это могут быть как записи мероприятий вашей компании, так и внекорпоративные видеоролики на YouTube. Смотрите обучающие программы, лекции, посвященные техническим вопросам, и презентации, проводимые в рамках конференций. Попросите коллег порекомендовать вам хороший контент. Для экономии времени можно увеличивать скорость воспроизведения видео в 1,5 или даже 2 раза. Однако не смотрите пассивно: делайте заметки, чтобы не забыть важные моменты, и уточняйте любые незнакомые понятия или термины.

Участвуйте в неформальных встречах типа *brown bag* (короткая презентация или семинар) и технических обсуждениях, если ваша компания их организует: они проводятся прямо на рабочем месте, поэтому вам не придется тратить время на дорогу. На подобных мероприятиях, как правило, обсуждаются рабочие вопросы вашей компании, что позволяет получить по-настоящему актуальную информацию.

Посещайте митапы и конференции (иногда)

Конференции и митапы хороши для нетворкинга и поиска новых идей. Их стоит посещать время от времени, однако не переусердствуйте. Отношение «сигнал/шум», то есть отношение релевантного контента ко всему контенту в целом, часто бывает низким, а кроме того, многие презентации впоследствии публикуются во Всемирной паутине.

Существуют три типа конференций: научные конференции, собрания по интересам и выставки-презентации, устраиваемые поставщиками. Научные конференции отличаются замечательным контентом, однако их посещение лучше заменить чтением научных статей и участием пусть в менее крупных, зато целенаправленных обсуждениях. Встречи по интересам отлично подходят для получения практических советов и общения с более опытными специалистами: поучаствуйте хотя бы в нескольких таких встречах. Выставки-презентации, организуемые

поставщиками, — самые масштабные и значимые. Они представляют собой маркетинговый инструмент для технологических компаний и не подходят для обучения. На них можно хорошо провести время с коллегами, однако стоит ограничиться одним подобным мероприятием в год. Опять же, попросите коллег порекомендовать вам самые лучшие выставки. Имейте в виду, что некоторые работодатели оплачивают и билеты на такие мероприятия, и проезд с проживанием.

ПОДДЕРЖИВАЙТЕ СВЯЗИ С УНИВЕРСИТЕТАМИ

Несколько лет назад Дмитрий и его коллега Питер Альваро изо всех сил пытались заставить работать свое хранилище данных и думали, что правильным решением было бы распределение задач агрегации по узлам кластера серверов. В процессе работы Питер обнаружил документ с описанием модели MapReduce, выпущенный чуть ранее компанией Google. Оказалось, что Google уже делала именно то, что предлагали Питер и Дмитрий! Затем коллеги нашли другие интересные статьи и решили поискать людей, с которыми можно было бы их обсудить. Дмитрий выяснил, что в Калифорнийском университете в Беркли регулярно проводятся открытые встречи для всех желающих, посвященные обсуждению вопросов, связанных с базами данных. Питер и Дмитрий стали их завсегдатаями (но к бесплатной пицце не притрагивались до тех пор, пока свою долю не получали студенты). Иногда они даже принимали участие в разговоре!

Процесс их обучения перешел на новый уровень. В конце концов Питер остался в университете Беркли и поступил в докторантуру: теперь он профессор Калифорнийского университета в Санта-Крузе. Дмитрий тоже поступил в аспирантуру. Спустя два года после их ухода дорогостоящее хранилище данных заменили распределенной системой агрегации с открытым исходным кодом Hadoop.

Если вы почувствуете, что перестали учиться, сходите в местный университет. Как правило, там есть масса программ, открытых для публики. Расширьте свой круг общения. Поступать в аспирантуру при этом необязательно.

Попробуйте шедоуинг и парное программирование

Шедоуинг предполагает наблюдение за тем, как более опытный сотрудник выполняет то или иное задание. Наблюдатель в данном случае является активным участником: он делает записи и задает вопросы. Шедоуинг — отличный способ освоить новый навык. Чтобы получить максимальную пользу, выделите время до и после сессии для планирования и подведения итогов.

Когда будете готовы, поменяйтесь ролями. Пусть теперь сеньор-разработчик наблюдает за вами. При этом он тоже должен предоставить обратную связь. Кроме того, он сможет подстраховать вас, если что-то пойдет не так. Это хороший способ подготовиться к таким пугающим мероприятиям, как собеседование.

Парное программирование также отличный метод обучения: два инженера пишут код вместе, вводя его по очереди. К подобному нужно привыкнуть, но это один из самых быстрых способов обмена знаниями. Сторонники парного программирования утверждают, что таким образом улучшается качество кода. Если ваши коллеги не возражают, мы настоятельно рекомендуем попробовать данный метод. Парное программирование приносит пользу не только младшим инженерам — извлечь из него выгоду могут сотрудники всех уровней.

Некоторые компании поощряют наблюдение за работой и других специалистов. Например, наблюдение за работой сотрудников отдела поддержки клиентов и отдела продаж — отличный способ больше узнать о клиентах компании. Запишите свои наблюдения и поделитесь ими с коллегами. Поработайте с руководителем и сеньор-разработчиками над приоритизацией идей, вдохновленных этим опытом.

Экспериментируйте со сторонними проектами

Работа над сторонними проектами позволит познакомиться с новыми технологиями и идеями. Вы можете пропустить все, что связано с разработкой программного обеспечения, то есть тестирование, операционную работу, проверку кода и т. д.: игнорируя эти процессы, можно быстро

освоить новые технологии. Только не забывайте об этих важных этапах, трудясь над собственным проектом.

Вы также можете участвовать в разработке проектов с открытым исходным кодом. В большинстве из них вклад участников очень приветствуется. Это отличный способ научиться чему-то новому и наладить профессиональные связи. С помощью таких сообществ вы даже можете найти работу. Имейте в виду, что разработкой проектов с открытым исходным кодом часто руководят добровольцы. Поэтому не стоит ожидать, что задачи будут решаться с такой же скоростью, как на работе: иногда люди бывают заняты и могут на некоторое время пропадать.

При выборе проекта не руководствуйтесь тем, что, по вашему мнению, вам следует изучить. Думайте об интересующих вас проблемах и пробуйте решать их, используя инструменты, которые вы хотите изучить. При наличии мотивации можно дольше оставаться вовлеченным в процесс, а значит, и получить больше знаний.

Обычно в компаниях приняты определенные правила в отношении работы над сторонними проектами. Ознакомьтесь с политикой своей компании. Не используйте ресурсы компании (например, ноутбук) для работы над сторонними проектами. Не работайте над сторонними проектами в рабочее время. Избегайте проектов, которые конкурируют с продуктами вашей компании. Уточните, можно ли заниматься разработкой проекта с открытым исходным кодом на работе. Одни компании могут разрешить вам использовать специальную рабочую учетную запись, но другие будут настаивать на использовании только личной учетной записи. Выясните, сохранится ли за вами право собственности на разработанный вами сторонний проект. Спросите у руководства, требуются ли какие-либо согласования. Уточнение всех этих нюансов избавит вас от проблем в будущем.

Научитесь задавать вопросы

Все инженеры должны научиться правильно задавать вопросы. Новички обычно боятся лишний раз побеспокоить коллег и пытаются разобраться во всем самостоятельно, что негативно сказывается на скорости

и эффективности работы. Правильные вопросы помогут вам учиться быстро, не раздражая окружающих. Для этого нужно провести исследование, четко сформулировать вопрос и выбрать подходящее время, чтобы его задать.

Проведите исследование

Сначала попытайтесь найти ответ самостоятельно. Даже если ваши коллеги знают ответ, приложите усилия сами — так вы узнаете больше. Если вам не удастся найти ответ, результат вашего исследования станет отправной точкой для формулирования вопроса.

Проводите поиск не только в интернете. Нужную информацию можно найти в документации, базах знаний wiki, README-файлах, исходном коде и системах отслеживания ошибок. Если ваш вопрос касается кода, попробуйте преобразовать его в модульный тест, демонстрирующий конкретную проблему. Возможно, ваш вопрос уже задавали ранее, так что проверьте архивы рассылки или чат-групп. В процессе сбора информации у вас появятся идеи, которые вы сможете проверить. Если не знаете, с чего начать, экспериментируйте, отслеживая все свои действия, их причины и результаты.

Установите временные рамки

Ограничьте время, выделяемое на изучение проблемы. Установите дедлайн, чтобы дисциплинировать себя и предотвратить снижение продуктивности. Подумайте, когда вам потребуется ответ, а затем оставьте достаточно времени для того, чтобы задать вопрос, получить на него ответ и принять соответствующие меры.

Когда выделенное время закончится, обратитесь за помощью. Выходить за установленные рамки стоит только в том случае, если вы замечаете прогресс: тогда, пропустив исходный срок, установите другой. Если по его истечении вы все еще не уверены в ответе, попросите о помощи. Умение останавливаться требует практики и дисциплины — развивайте ее у себя.

Продемонстрируйте результаты проделанной работы

Задавая вопрос, сообщите о том, что вам уже удалось выяснить. Не ограничивайтесь заметками. Кратко опишите те способы, которые вы попробовали, и полученные результаты. Так вы покажете, что потратили время, пытаясь разобраться в проблеме самостоятельно. Кроме того, это может послужить другим людям отправной точкой для ответа.

Вот плохой способ задать вопрос.

> *Привет, Элис!*
>
> *Есть идеи, почему testKeyValues вызывают сбой в TestKVStore? Повторные запуски сильно замедляют процесс сборки.*
>
> *Спасибо!*
>
> *Панкадж*

Подобное мало о чем скажет Элис. Кроме того, вопрос звучит так, будто Панкадж обвиняет ее, хотя у него, вероятно, вовсе не было такого намерения. В данном случае составитель вопроса явно поленился. Сравните это сообщение со следующим вариантом.

> *Привет, Элис!*
>
> *Я никак не могу понять, почему testKeyValues вызывают сбой в TestKVStore (в репозитории DistKV). Шон порекомендовал мне обратиться к тебе. Надеюсь, ты сможешь помочь.*
>
> *Примерно каждое третье выполнение теста заканчивается сбоем, но никакой закономерности при этом не прослеживается. Я пробовал запускать его изолированно, но сбои по-прежнему имеют место, поэтому не думаю, что они вызваны взаимодействием между тестами. Шон провел тест в цикле на своем компьютере, но не смог*

его воспроизвести. В исходном коде я не вижу ничего, что могло бы объяснить происходящее. Похоже на состояние гонки. У тебя есть какие-нибудь предположения?

Особой срочности нет, поскольку, по моим сведениям, это вряд ли повлияет на производство. Тем не менее каждый такой сбой обходится нам в 20–30 минут рабочего времени, поэтому мне очень хотелось бы исправить проблему. На всякий случай я прикрепил журналы со сведениями обо всех сбоях и текущих настройках среды.

Спасибо!

Панкадж

Во втором примере Панкадж предоставляет некоторый контекст, описывает проблему, говорит Элис о том, что он уже попробовал, и просит ее о помощи. Он также отмечает влияние проблемы на рабочий процесс и уровень срочности. Сообщение достаточно краткое, но к нему прилагается подробная информация, поэтому Элис не придется ничего искать. Элис поможет Панкаджу и, кроме того, отметит его основательность. Подобные запросы помогут Панкаджу заслужить доверие коллег.

Написание второго сообщения требует больших усилий. Но оно определенно того стоит.

Не отрывайте коллег от работы

Так же, как и вы, ваши коллеги занимаются делом и пытаются сосредоточиться. Не отрывайте их от работы, даже если испытываете сложности и уверены в том, что они знают ответ на ваш вопрос. Не отвлекайте их без крайней необходимости.

В разных компаниях сигнал «Не беспокоить!» подается разными способами. Универсальным является использование наушников или берушей. С лаундж-зонами может возникнуть путаница. Видя инженеров, работающих в общих помещениях, одни люди понимают, что те

не хотят, чтобы их отвлекали, а вот другие, наоборот, полагают, что они открыты для общения. Убедитесь в том, что знаете негласные правила своей компании!

Подходя и обращаясь к человеку, вы вынуждаете его вам ответить. Даже если он просто скажет, что занят, вы тем не менее отвлечете его от работы. Если нужный вам человек занят, используйте асинхронный способ общения с ним.

Выбирайте многоадресную асинхронную коммуникацию

Многоадресная рассылка предполагает отправку сообщения не отдельному адресату, а группе людей. При использовании *асинхронной коммуникации* полученное сообщение не требует немедленного ответа. Эти концепции применимы и к человеческому общению.

В многоадресной рассылке задавайте свои вопросы так, чтобы люди имели возможность отвечать на них в своем темпе, то есть асинхронно. Позаботьтесь о том, чтобы всем было видно, когда вам помогли, и чтобы в будущем ответ могли прочитать и другие интересующиеся.

Как правило, речь идет о групповой рассылке или групповом чате (в компании Дмитрия, например, для этого есть канал #sw-helping-sw). Используйте общие форумы, даже если вам нужен ответ от конкретного человека — его имя можно просто упомянуть в сообщении.

Группируйте синхронные запросы

Чат и электронная почта отлично подходят для решения простых вопросов, однако обсуждение сложных проблем в асинхронном режиме не работает. Личное общение характеризуется «высокой пропускной способностью» и «низкой задержкой», что позволяет быстро обсудить множество вещей, но это не самый лучший вариант: если коллеги отвлекаются от работы, снижается их продуктивность. Избегайте этого, выделив время на обсуждение несрочных вопросов со своим техлидом

или менеджером. Запланируйте встречу или используйте приемные часы, если в компании они предусмотрены. Запишите интересующие вас вопросы, чтобы задать их на встрече, а пока проведите дополнительное исследование. По мере возникновения новых вопросов ваш список будет расти, и это хорошо — включите его в повестку дня. Не полагайтесь на свою память и приходите подготовленным.

При отсутствии вопросов отмените встречу. Если заметите, что постоянно отменяете подобные встречи, задайтесь вопросом, по-прежнему ли они для вас актуальны. Если нет, отмените их совсем.

Преодолевайте препятствия на пути роста

Уметь учиться и задавать вопросы — важно, но недостаточно: вы также должны избегать ловушек, замедляющих ваш профессиональный рост. Многие инженеры сталкиваются с такими явлениями, как синдром самозванца и эффект Даннинга — Крюгера. Вы будете прогрессировать быстрее, если поймете, что это за явления и как их избежать.

Синдром самозванца

Большинство новых инженеров начинают с уровня осознанной некомпетентности. Они понимают, что им нужно многому научиться, и им кажется, что все остальные ушли далеко вперед. Новички даже могут чувствовать себя не в своей тарелке или считать получение работы большой удачей — мы знаем, что такое относиться к себе слишком строго. Вне зависимости от того, как часто мы говорим инженерам о том, что они проделали отличную работу, некоторые не верят этому, даже когда их повышают! При этом они испытывают дискомфорт, говорят, что им просто повезло и они не заслуживают признания, или ссылаются на слишком «мягкие» критерии продвижения по службе. Так проявляется *синдром самозванца*. Впервые он был описан в 1978 году в исследовании доктора Паулины Роуз Клэнс и доктора Сюзанны Эймент Аймс под названием *The Impostor Phenomenon in High Achieving Women: Dynamics*

and Therapeutic Intervention («Феномен самозванца у успешных женщин: динамика и терапевтическое вмешательство»):

> *«Несмотря на выдающиеся научные и профессиональные достижения, женщины, подверженные синдрому самозванца, упорно считают, что на самом деле они неумны, и полагают, что просто обманывают тех, кто думает иначе. Многочисленные достижения, которые, казалось бы, предъявляют достаточное количество объективных доказательств превосходных интеллектуальных способностей, по-видимому, никак не влияют на убежденность в собственном самозванстве».*

Если это описание находит у вас отклик, знайте, что неуверенность в себе — явление весьма распространенное. Справиться с ней помогут осознанность, рефрейминг и общение с коллегами.

Синдром самозванца сам себя подпитывает. Каждая ошибка рассматривается как доказательство некомпетентности, а каждый успех — как признак хорошего «обманщика». Когда человек попадает в подобный замкнутый круг, ему бывает очень трудно из него выбраться. Здесь поможет осознанность: если отслеживать у себя этот паттерн, его можно сознательно преодолеть. Вы чего-то добиваетесь потому, что действительно хорошо работаете, а не потому, что вам просто везет.

Не игнорируйте комплименты. Записывайте все, даже мелочи. Ваши коллеги — профессионалы, и если они положительно отзываются о вашей работе, значит, для этого есть веские основания. Попрактикуйтесь в рефрейминге, то есть в превращении негативных мыслей в позитивные. Вместо: «Мне пришлось оторвать Дарью от работы, чтобы она помогла мне решить проблему с состоянием гонки», — лучше скажите: «Я обратился к Дарье и теперь знаю, как решить проблему с состоянием гонки!» Подумайте, чего вы хотите достичь, а когда достигнете цели, отметьте это. Так вы укрепите уверенность в себе.

Помочь может и обратная связь. Попросите какого-нибудь уважаемого вами человека сказать, что он думает о вашей работе. Это может

быть ваш менеджер, наставник или просто инженер, на которого вы равняетесь. Важно, чтобы вы доверяли этому человеку и чувствовали себя в безопасности, обсуждая с ним проблему неуверенности в себе.

Кроме того, вам может помочь терапия. Благодаря ей вы поймете свои сильные стороны и справитесь с краткосрочными проблемами. Синдром самозванца и сопровождающие его тревога и депрессия — тема весьма сложная. Если вы испытываете подобные трудности, попробуйте поговорить с несколькими терапевтами, чтобы найти того, чей метод подходит именно вам.

Эффект Даннинга — Крюгера

Противоположностью синдрома самозванца является *эффект Даннинга — Крюгера*: когнитивное искажение, при котором люди считают себя более способными, чем они есть на самом деле. Инженеры, находящиеся на уровне неосознанной некомпетентности, не понимают, насколько они невежественны, поэтому не могут точно оценить свою (или чужую) работу. Они слишком уверены в себе, когда, например, агрессивно критикуют технологический стек компании, жалуются на качество кода и принижают дизайн. Такие люди уверены в правильности своих идей. Обычно они негативно реагируют на обратную связь или вообще ее игнорируют. Отклонение всех предложений — весьма тревожный сигнал. Полная уверенность в чем-то — явный признак недостаточной компетентности в соответствующей области.

К счастью, эффект Даннинга — Крюгера среди начинающих инженеров распространен меньше. Кроме того, существует множество способов борьбы с ним. Советуем вам начать с развития любознательности. Будьте готовы ошибаться. Запросите обратную связь у уважаемого вами инженера по поводу своей работы и внимательно его выслушайте. Обсуждайте дизайнерские решения, особенно те, с которыми не согласны. Спросите, почему были приняты те или иные решения. Учитесь мыслить в терминах компромиссов, вместо того чтобы делить вещи на правильные и неправильные.

Что следует и чего не следует делать

Следует	Не следует
Играть и экспериментировать с кодом	Бездумно создавать большие объемы кода
Читать проектную документацию и код, созданный другими людьми	Бояться риска и ошибок
Принимать участие в митапах, присоединяться к онлайн-сообществам, группам по интересам и программам наставничества	Слишком часто посещать конференции
Читать статьи и блоги	Бояться задавать вопросы
Предпочитать многоадресную и асинхронную коммуникацию	
Присутствовать на собеседованиях с другими кандидатами в качестве «тени» и участвовать в дежурстве	

Повышение уровня

Книга Дэйва Гувера и Адевале Ошиная (Dave Hoover, Adewale Oshineye) *Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman* (O'Reilly Media, 2009) содержит подборку отличных советов, которые помогут начать работу в новой среде: как обращаться за помощью, как осваивать навыки и справляться с распространенными проблемами.

Если хотите научиться задавать вопросы, рекомендуем книгу Уэйна Бейкера (Wayne Baker) *All You Have to Do Is Ask: How to Master the Most Important Skill for Success* (Currency, 2020). Она состоит из двух частей: в первой говорится о ценности умения задавать вопросы и сложности овладения подобным умением, вторая часть представляет собой набор инструментов, позволяющих эффективно задавать вопросы.

Более подробную информацию о парном программировании можно найти в работе Кента Бека и Синтии Андрес (Kent Beck, Cynthia Andres) *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 2004). В этой книге речь идет не только о парном программировании. Если хотите сэкономить время, можете ознакомиться с отличной статьей

Биргитты Бокелер и Нины Сиссеггер (Birgitta Böckeler, Nina Siessegger) *On Pair Programming* по адресу https://www.martinfowler.com/articles/on-pair-programming.html.

Если рассказ о синдроме самозванца и эффекте Даннинга — Крюгера нашел у вас отклик, обратитесь к книге Эми Кадди (Amy Cuddy) *Presence: Bringing Your Boldest Self to Your Biggest Challenges* (Little, Brown & Company, 2016). В ней вы найдете описание распространенных причин как беспокойства, так и излишней самоуверенности, проявляющихся в работе.

3 Работа с кодом

Во французском городе Арль есть древнеримский амфитеатр. Когда-то там устраивали зрелища — гонки на колесницах и гладиаторские бои — для 20 000 человек. После падения Римской империи прямо на арене начали строить городок. Это имело смысл, учитывая то, что арена имела стены и была оснащена дренажной системой. Позднее жители, вероятно, сочли постройку странной и неудобной: возможно, архитекторов амфитеатра осудили за те решения, из-за которых его было трудно превратить в город.

Кодовые базы чем-то напоминают амфитеатр в Арле. Код пишется слоями, а слои впоследствии подвергаются изменениям. Над кодом трудится множество людей. Тесты или отсутствуют, или способствуют укоренению устаревших идей. Изменяющиеся требования влияют на процесс использования кода. Работать с ним сложно, но это именно то, что вам придется делать в самом начале.

В этой главе мы поговорим о том, как работать с существующим кодом. Мы познакомим вас с такими концепциями, как энтропия программного обеспечения и технический долг. Затем дадим практические рекомендации по безопасному изменению кода. В конце главы вы найдете советы, позволяющие предотвратить случайное возникновение беспорядка в коде.

Энтропия программного обеспечения

В процессе изучения кода вы начнете замечать его недостатки. Беспорядок в коде — это естественный побочный эффект изменений: не вините в его возникновении разработчиков. Явление нарастания беспорядка в коде называется *энтропией ПО*.

Увеличению энтропии ПО способствует множество факторов. Разработчики могут неправильно понимать код друг друга или использовать разные стили при его написании. Развитие технологических стеков и изменение требований к продукту порождают хаос (см. главу 11). Исправление ошибок и оптимизация производительности еще больше усложняют ситуацию.

К счастью, на степень энтропии ПО можно влиять. Порядок в коде помогают поддерживать инструменты для проверки стиля кода и обнаружения ошибок (глава 6). Благодаря ревью кода можно делиться знаниями и сокращать степень несогласованности (глава 7). Непрерывный рефакторинг снижает энтропию (см. раздел «Изменение кода» далее в этой главе).

Технический долг

Технический долг является основной причиной увеличения энтропии программного обеспечения. Технический долг — это работа, которую необходимо выполнить для исправления недостатков в существующем коде. Как и финансовый долг, технический состоит из основной суммы долга и процентов. Исходный недостаток, который необходимо исправить, можно назвать основной суммой долга. Выплата процентов осуществляется по мере доработки кода без устранения основного недостатка и выражается в реализации все более сложных обходных путей. Воспроизведение и закрепление этих обходных путей аналогично начислению сложного процента. Усложнение программного обеспечения

приводит к возникновению дефектов. Невыплаченный технический долг — обычное дело, и в случае унаследованного кода его размер бывает очень большим.

Технические решения, с которыми вы не согласны, не являются техническим долгом, как и код, который вам не нравится. Чтобы превратиться в технический долг, проблема должна требовать от команды «уплаты процентов» для предотвращения возникновения критической ошибки, требующей срочной выплаты долга. Не злоупотребляйте этим выражением. Слишком частое упоминание «технического долга» сделает соответствующие заявления слишком легковесными, усложнив решение по-настоящему важных проблем.

Мы знаем, как напрягают долги, однако не всякий долг плох. Мартин Фаулер выделяет четыре типа технического долга (табл. 3.1).

Таблица 3.1. Квадрант технического долга

	Безрассудный	Благоразумный
Умышленный	«У нас нет времени на разработку проекта»	«Давайте выпустим релиз, а с последствиями разберемся потом»
Неумышленный	«Что такое многослойная архитектура?»	«Теперь мы знаем, как это надо было делать»

Источник: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html.

Благоразумный умышленный долг представляет собой классическую форму технического долга и является прагматичным компромиссом между известным недостатком кода и скоростью доставки. Это хороший долг — при условии, что команда дисциплинированно подходит к его дальнейшей выплате.

Безрассудный умышленный долг возникает в условиях недостатка времени. При обсуждении такого долга часто используется слово «просто»: «Позднее мы *просто* добавим структурное логирование» или «Давайте *просто* увеличим время ожидания».

Безрассудный неумышленный долг возникает из-за *неизвестных неизвестных*. Вы можете свести его к минимуму, заранее продумывая планы

реализации и своевременно получая по ним обратную связь, а также проводя ревью кода и используя непрерывное обучение.

Благоразумный неумышленный долг — это естественное следствие растущего опыта. Некоторые уроки можно извлечь только задним числом: «Нам следовало создавать учетные записи пользователей даже для тех людей, которые не прошли процесс регистрации до конца. Мы должны фиксировать неудачные попытки регистрации в маркетинговых целях. Теперь нам придется добавить дополнительный код, чего можно было бы избежать, если бы мы предусмотрели это в модели данных». В отличие от случая с благоразумным умышленным долгом, здесь команда не знает о возникновении долга. В отличие от неумышленного безрассудного долга, долг такого типа является скорее естественным следствием более глубокого изучения предмета или профессионального роста архитектора ПО, а не результатом плохого выполнения домашней работы. Хорошей практикой является проведение ретроспективы проектов, что позволяет команде обнаруживать такой неумышленный долг и решать, когда его погашать и стоит ли это делать.

Важный вывод заключается в том, что возникновение некоторого долга неизбежно, поскольку неумышленные ошибки невозможно предотвратить. Кроме того, технический долг может быть признаком успеха и свидетельствовать о том, что проект просуществовал достаточно долго для возникновения в нем беспорядка.

Выплата технического долга

Не ждите конца света, чтобы исправить проблемы, возникшие в течение месяца. Вместо этого старайтесь приводить код в порядок прямо по ходу работы, выполняя мелкий рефакторинг и внося небольшие изменения с помощью независимых коммитов и пул-реквестов.

Может оказаться, что постепенного рефакторинга недостаточно и необходимы более значительные изменения. Масштабный рефакторинг — это серьезное обязательство. В краткосрочной перспективе погашение технического долга замедляет внедрение новых функций, в то время как увеличение размера долга ускоряет этот процесс. В долгосрочной перспективе верно обратное: выплата долга ускоряет внедрение новых

функций, а увеличение размера долга его замедляет. Продакт-менеджеры заинтересованы в разработке большего количества функций (что ведет к увеличению размера технического долга). Нахождение правильного баланса сильно зависит от контекста. Если у вас есть предложения, касающиеся масштабного рефакторинга или переписывания кода, обсудите их со своей командой, используя такую последовательность действий.

1. Изложите ситуацию, опираясь на факты.
2. Оцените степень риска и стоимость долга.
3. Предложите решение.
4. Обсудите альтернативы (включая бездействие).
5. Взвесьте компромиссы.

Изложите предложение в письменной форме. Не основывайте свою позицию на оценочном суждении («этот код является устаревшим и плохо написанным»). Сосредоточьтесь на стоимости долга и выгоде, связанной с его выплатой. Будьте конкретны и не удивляйтесь, если вас попросят продемонстрировать обещанные преимущества после завершения работы.

Привет всем!

Я думаю, пора разделить сервис входа в систему на два сервиса: один должен быть для аутентификации, другой — для авторизации.

Нестабильность работы этого сервиса является причиной более чем 30 % проблем, решаемых дежурными специалистами. Похоже, что такая нестабильность обусловлена смешиванием логики аутентификации и авторизации. Текущий проект сильно затрудняет тестирование всех заявленных функций безопасности. Мы гарантируем защиту данных наших клиентов, но процесс входа в систему в его нынешнем виде усложняет выполнение этого обещания. Я не говорила со специалистами комплаенс-службы, но думаю, что они поднимут данную проблему во время следующей проверки.

Мне кажется, логика управления доступом была встроена в сервис входа в основном из соображений целесообразности, с учетом

различных временных и ресурсных ограничений, которые существовали на тот момент. Это решение не было обусловлено каким-либо основополагающим архитектурным принципом. Тем не менее в настоящее время решение проблемы предполагает реализацию крупного проекта, связанного с рефакторингом сервиса входа в систему и удалением из него кода авторизации. И все же, я думаю, нам стоит исправить проблемы, связанные с нестабильностью и некорректностью работы этого сервиса.

Для сокращения объема работы можно было бы использовать сервис авторизации бэкенд-команды вместо создания собственного. Однако я не думаю, что это правильный подход, потому что бэкенд-команда решает другой набор задач. Мы занимаемся авторизацией пользователей, а они — межсистемной авторизацией. Но, может быть, есть хороший способ справиться и с тем, и с другим. Что вы об этом думаете?

Спасибо за внимание!

Джоанна

Изменение кода

Изменение кода не похоже на написание кода в новом репозитории. При внесении изменений нельзя нарушать существующее поведение программы. Вы должны понимать ход мыслей других разработчиков, придерживаться существующих стилей и шаблонов и улучшать кодовую базу постепенно.

Методы изменения кода примерно одинаковы, идет ли речь о добавлении новых функций, рефакторинге, удалении кода или исправлении ошибки. Как правило, имеет место сочетание разных типов изменений. Рефакторинг, то есть улучшение внутренней структуры кода без изменения его функциональности, производится при добавлении функции: таким образом упрощается ее внедрение. Удаление кода имеет место во время исправления ошибки.

Изменение больших кодовых баз — это навык, оттачиваемый годами, а то и десятилетиями. Следующие советы помогут вам приступить к работе в данном направлении.

Используйте алгоритм изменения унаследованного кода

В своей книге *Working Effectively with Legacy Code* (Pearson, 2004)[1] Майкл К. Физерс предлагает следующие этапы безопасного изменения существующих кодовых баз.

1. Определение точек изменения.
2. Нахождение тестовых точек.
3. Разрывание зависимостей.
4. Написание тестов.
5. Внесение изменений и рефакторинг кода.

Думайте о первых четырех этапах как о расчистке земли и установке забора вокруг поля перед посадкой семян, которая будет выполнена на пятом этапе. Если вы не установите забор, на поле могут забрести дикие животные и выкопать высаженные вами растения. Найдите код, требующий изменения, и подумайте, как его можно протестировать. При необходимости выполните рефакторинг, чтобы сделать тестирование возможным. Добавьте тесты, подтверждающие существующее поведение. После установки забора, защищающего территорию вокруг выбранных точек, можете приступать к внесению соответствующих изменений.

Сначала найдите код, который требуется изменить *(точки изменения)*, используя стратегии, описанные в главе 2. Прочтите код, поэкспериментируйте и задайте вопросы. В нашей метафоре с полем точки изменения — это места посева семян.

После нахождения кода, подлежащего изменению, необходимо выбрать *тестовые точки* — точки входа в код, который вы хотите изменить, и места для написания тестов. Тестовые точки показывают поведение кода до внесения изменений и используются для их тестирования.

Если повезет, то тестовые точки окажутся легко достижимыми, в противном случае, чтобы до них добраться, вам придется разорвать зависимости.

---

[1] *Физерс М. К.* Эффективная работа с унаследованным кодом.

В данном контексте под зависимостями понимаются не зависимости библиотек или сервисов, а объекты или методы, необходимые для тестирования вашего кода. *Разрывание зависимостей* означает изменение структуры кода с целью облегчения процесса его тестирования. То есть вам придется изменить код, чтобы подключить свои тесты и предоставить синтетические входные данные. Эти изменения *не* должны приводить к изменению поведения.

Рефакторинг, выполняемый с целью разрывания зависимостей, — самая рискованная часть работы. Он может даже предполагать изменение уже существующих тестов, что затруднит выявление изменений в поведении. Действуйте осторожно и на данном этапе не внедряйте никаких новых функций. Убедитесь в том, что можете проводить тесты быстро: это позволит вам выполнять их достаточно часто.

Существует множество методов разрывания зависимостей, в том числе такие:

- разделение большого сложного метода на несколько более мелких с целью независимого тестирования отдельных фрагментов функциональности;
- использование интерфейса (или другого посредника), позволяющего тестам предоставлять простую реализацию сложного объекта — неполную, но достаточную для тестирования;
- внедрение явных контрольных точек, позволяющих имитировать трудно контролируемые аспекты выполнения, например течение времени.

Не меняйте модификаторы доступа для упрощения тестов. Если вы сделаете общедоступными приватные методы и переменные, это, конечно, позволит тестам получить доступ к коду, но нарушит инкапсуляцию, так что такой обходной путь не назовешь удачным. Нарушение инкапсуляции увеличивает количество вариантов поведения, которые вы должны гарантировать в течение всего жизненного цикла проекта. Более подробно мы обсудим это в главе 11.

В процессе рефакторинга и разрывания зависимостей следует добавлять новые тесты для проверки старого поведения. Старайтесь как можно чаще запускать набор тестов, включающий как новые, так и старые тесты.

Рассмотрите возможность применения инструментов для автоматизированного тестирования с целью создания тестов, покрывающих существующее поведение. Подробнее о написании тестов поговорим в главе 6.

После разрыва зависимостей и создания качественных тестов можно приступать к внесению «настоящих» изменений. Добавьте тесты, подтверждающие изменения, а затем выполните рефакторинг кода для улучшения его структуры. Зная о том, что периметр кода надежно защищен, можно вносить более радикальные изменения.

Оставляйте код более чистым, чем он был до вас

Интернет-ресурсы, посвященные программированию, часто цитируют принцип бойскаутов: «Всегда оставляйте лагерь более чистым, чем он был до вас». Как и палаточный лагерь, кодовая база является общей, а с чистой базой работать гораздо приятнее. Если вы начнете применять этот подход к коду, то есть будете оставлять его более чистым, чем он был до вас, то со временем качество кода улучшится и вы сможете заменить масштабный рефакторинг постепенным внесением множества небольших изменений.

По мере исправления ошибок и добавления функций очищайте соседний код. Не ищите грязный код специально — действуйте по ситуации. Старайтесь делать так, чтобы коммиты, связанные с очисткой кода, были отделены от коммитов, изменяющих поведение. Такое разделение коммитов упрощает отмену изменений кода без потери результатов, полученных при его очистке. Менее масштабные коммиты также упрощают процесс анализа изменений.

Рефакторинг не единственный способ очистки кода. Иногда код буквально плохо пахнет: обращайте внимание на такой дурно пахнущий код. *Код с запашком* — это термин, обозначающий код, который не обязательно содержит ошибки, но в котором используются шаблоны, как правило, вызывающие проблемы. Говорят, что такой код дурно пахнет. Рассмотрим следующий фрагмент кода Java:

```
if (a < b)
  a += 1;
```

Этот фрагмент абсолютно корректен. В языке Java один оператор может следовать за условным выражением, при этом его не обязательно заключать в фигурные скобки. Однако это код с запашком, потому что в нем можно легко допустить следующую ошибку:

```
if (a < b)
  a += 1;
  a = a * 2;
```

В отличие от языка Python, в Java игнорируются отступы и используются фигурные скобки для группировки операторов. Таким образом, значение `a` будет удвоено вне зависимости от условия `if`. Такую ошибку было бы намного сложнее допустить, если бы при написании исходного кода фрагмент `a += 1;` был заключен в фигурные скобки, отсутствие которых в данном случае делает код «дурно пахнущим».

Многие линтеры и инструменты для проверки качества кода обнаруживают эту и другие проблемы, связанные, например, со слишком длинными методами или классами, с дублированием кода, чрезмерным ветвлением, зацикливанием или слишком большим количеством параметров. Для выявления и исправления менее явных антипаттернов требуются дополнительные инструменты и опыт.

Вносите изменения постепенно

Как правило, рефакторинг принимает одну из двух форм. Первая — это масштабная ревизия кода, предполагающая одновременное изменение десятков файлов. Вторая — это запрос на внесение изменений, сочетающий рефакторинг с внедрением новых функций. И то и другое сложно анализировать. В случае комбинированных коммитов бывает трудно отменить функциональные изменения, не затронув результаты рефакторинга, которые требуется сохранить. В процессе рефакторинга лучше вносить небольшие изменения. Создавайте отдельные пул-реквесты, соответствующие каждому из шагов алгоритма изменения унаследованного кода (описанного ранее в этой главе). Используйте более мелкие коммиты, если изменения сложно отслеживать. Наконец, заручитесь поддержкой своей команды, прежде чем приступать к рефакторингу. В конце концов, вы меняете код, созданный вашими коллегами, поэтому их мнение тоже должно учитываться.

Относитесь к рефакторингу прагматично

Проведение рефакторинга не всегда целесообразно: есть дедлайны и другие приоритеты, а рефакторинг требует времени. Ваша команда может отказаться от рефакторинга в пользу выпуска новых функций. Такие решения увеличивают технический долг команды, но подобный подход бывает оправданным. Кроме того, стоимость рефакторинга может превышать его ценность. Устаревшему и постепенно заменяемому коду не нужен рефакторинг, равно как и коду, который характеризуется низким уровнем риска или редко используется. Старайтесь относиться к рефакторингу прагматично.

Используйте IDE

Интегрированные среды разработки (IDE) стигматизируются l33t-программистами, которые считают получение «помощи» от редактора слабостью, но фетишизируют Vim или Emacs — «элегантное оружие более цивилизованной эпохи». Это нонсенс. Пользуйтесь преимуществами доступных вам инструментов. Если ваш язык программирования предусматривает хорошую IDE, используйте ее.

Особенно полезны IDE бывают в процессе рефакторинга. Они содержат инструменты для переименования и перемещения кода, извлечения методов и полей, обновления сигнатур методов и выполнения других общих операций. При работе с большими кодовыми базами простые операции с кодом часто бывают утомительными и сопряженными с ошибками, а IDE автоматически анализируют код и обновляют его в соответствии с внесенными изменениями. (Мы умеем делать это с помощью Vim и Emacs, так что не пишите нам гневные письма.)

Однако не стоит увлекаться: IDE настолько упрощают процесс рефакторинга, что несколько простых изменений могут перерасти в масштабную ревизию кода. В конце концов, анализировать ваши автоматизированные изменения придется человеку. Кроме того, у автоматического рефакторинга есть и ограничения: ссылка на переименованный метод может не обновиться, если он вызывается посредством отражения или метапрограммирования.

Используйте передовые методы управления версиями

Изменения необходимо фиксировать в *системе управления версиями* (VCS, version control system), например в Git. VCS отслеживает историю изменений кодовой базы, в том числе и сведения о том, кто внес изменение *(коммит)* и когда оно было сделано. Каждый коммит сопровождается *коммит-сообщением*.

Во время разработки фиксировать изменения следует как можно чаще. Частые коммиты отражают процесс изменения кода во времени, позволяют отменять изменения и выступают в качестве удаленной резервной копии. Однако частые коммиты порой сопровождаются бессмысленными сообщениями вроде «ой» или «исправьте сломанный тест». В кратких коммит-сообщениях нет ничего плохого при написании большого объема кода, но другим людям они ни о чем не говорят. Перебазируйте свою ветку, объедините коммиты и напишите четкое коммит-сообщение, прежде чем отправлять изменение на рассмотрение.

Итоговое коммит-сообщение должно быть оформлено в соответствии с правилами, принятыми вашей командой. Обычно к таким сообщениям добавляется префикс с идентификатором проблемы: «[MYPROJ123] Попытка заставить серверную часть работать с Postgres». Привязка коммита к конкретной проблеме позволяет разработчикам лучше понимать контекст изменения, а также применять сценарии и инструменты. При отсутствии конкретных правил следуйте советам Криса Бимса (https://chris.beams.io/posts/git-commit).

- Отделите тему от тела сообщения пустой строкой.
- Ограничьте длину строки темы 50 символами.
- Напишите каждое слово темы с заглавной буквы.
- Не ставьте точку в конце строки темы.
- В формулировке темы сообщения используйте повелительное наклонение.
- Ограничьте длину строки в теле сообщения 72 символами.
- В теле сообщения объясните *что и почему*, а не *как*.

Статью Криса стоит прочитать, поскольку в ней описывается хороший подход.

Избегайте ловушек

Существующий код сопровождается багажом в виде библиотек, фреймворков и шаблонов. Некоторые из уже использующихся стандартов могут вам не понравиться. Желание работать с чистым кодом и современным технологическим стеком вполне понятно, однако от соблазна переписать код или проигнорировать существующие стандарты стоит удержаться. Неправильно выполненное переписывание кода может дестабилизировать кодовую базу, и кроме того — переписывание негативно сказывается на новых функциях. Стандарты кодирования обеспечивают читаемость кода, и их игнорирование может существенно усложнить жизнь разработчикам.

В своей книге *The Hard Thing About Hard Things* (Harper Business, 2014)[1] Бен Хоровиц пишет следующее:

> *«Главное, что должен сделать любой технологический стартап, — создать продукт, который справляется с какой-либо задачей как минимум в десять раз лучше по сравнению с существующим решением. Если новый продукт будет превосходить существующий лишь в два или три раза, этого не хватит для того, чтобы заставить людей переключиться на него достаточно быстро или в достаточно значительном объеме, — а смысл именно в этом».*

Бен говорит о стартапах, однако та же идея применима и к существующему коду. Переписывать код или отклоняться от стандартов следует только тогда, когда ваше изменение может на порядок повысить его качество. Небольшого улучшения недостаточно — цена слишком высока. Большинство инженеров склонны недооценивать важность соглашений и переоценивать значимость их игнорирования.

Проявляйте осторожность при переписывании кода, нарушении соглашений и добавлении в стек новых технологий. Перезаписывайте код

---

[1] *Хоровиц Б.* Сложные решения.

только при большой ценности вносимых изменений. По возможности используйте так называемые скучные технологии. Не игнорируйте правила и конвенции, даже если вы с ними не согласны, и избегайте разветвления кода.

Используйте скучные технологии

Сфера разработки ПО развивается очень быстро: постоянно появляются новые инструменты, языки и фреймворки. По сравнению с тем, что вы видите в интернете, существующий код может показаться устаревшим. Однако, несмотря на использование старых библиотек и шаблонов, код успешных компаний остается надежным. И для этого есть причина: достижение успеха требует времени, а перебор технологий сопряжен с тратой времени.

Проблема новых технологий заключается в их незрелости. В своей презентации *Choose Boring Technology* Дэн Маккинли отмечает: «Виды отказов скучных технологий хорошо изучены» (http://boringtechnology.club/). Все технологии дают сбой, но в отличие от новых технологий старые делают это предсказуемым образом. Незрелость технологии означает малочисленность ее сообщества, меньшую стабильность, менее подробную документацию и худшую совместимость. На сайте Stack Overflow содержится гораздо меньше ответов на вопросы, связанные с новыми технологиями.

Иногда новые технологии позволяют решить проблемы вашей компании, а иногда — нет. Чтобы понять, когда использование новой технологии является оправданным, требуются дисциплина и опыт. Потенциальная выгода должна превышать стоимость. За каждое решение об использовании новой технологии приходится платить «токеном инноваций». С помощью этой концепции Дэн стремится показать, что усилия, затрачиваемые на внедрение новых технологий, могут быть потрачены на разработку инновационных функций. Количество таких токенов у компаний ограничено.

Чтобы сбалансировать затраты и выгоду, потратьте свои токены на технологии, которые обслуживают самые важные направления деятельности (ключевые компетенции) вашей компании, имеют широкий спектр

применения и могут использоваться несколькими командами. Если ваша компания специализируется на прогнозной аналитике ипотечных кредитов и в ней работает целая команда специалистов-аналитиков, то внедрение передовых алгоритмов машинного обучения имеет смысл. Если же в компании десять инженеров, которые создают игры для iOS, используйте готовые решения. Новые технологии приносят больше пользы, когда делают вашу компанию более конкурентоспособной. Широкий спектр применения означает, что они принесут пользу большему количеству команд и вашей компании в целом придется поддерживать меньший объем кода.

Выбор нового языка программирования для реализации проекта имеет особенно далекоидущие последствия. Использование нового языка предполагает включение в экосистему вашей компании целого технологического стека, а это требует обеспечения поддержки новых систем сборки, библиотек, сред разработки и тестирования. Выбранный вами язык может обладать такими серьезными преимуществами, как особая парадигма программирования, простота проведения экспериментов или устранения некоторых видов ошибок, однако преимущества должны быть сбалансированы с недостатками. Если стоимость использования нового фреймворка или базы данных составляет один токен инновации, то использование нового языка стоит три токена.

Зрелость экосистемы нового языка имеет огромную важность. Хорошо ли продумана система сборки и управления пакетами? Как поддерживается IDE? Есть ли среди мейнтейнеров важных библиотек опытные разработчики? Доступны ли тестовые фреймворки? Есть ли возможность заплатить за поддержку, если она понадобится? Можно ли нанять инженеров, обладающих соответствующими навыками? Насколько легко освоить данный язык? Какова его эффективность? Можно ли интегрировать языковую экосистему и существующие в компании инструменты? Ответы на эти вопросы столь же важны, как и особенности самого языка. Многие крупнейшие компании пришли к успеху, применяя скучные технологии. На языках C, Java, PHP, Ruby и .NET было написано отличное программное обеспечение. Если язык не умирает, то его возраст и отсутствие ажиотажа вряд ли можно считать достаточными аргументами против него.

SBT И SCALA

В 2009 году разработчики LinkedIn открыли для себя Scala. Писать код на этом языке было гораздо удобнее, чем на языке Java, который в то время был популярен в LinkedIn. Scala отличался мощной и выразительной системой шрифтов, меньшей многословностью и позволял использовать методы функционального программирования. Более того, поскольку Scala работал на виртуальной машине Java (JVM), операционная группа могла запускать Scala-код, используя привычные JVM-инструменты. Это также означало, что Scala-код мог взаимодействовать с существующими библиотеками Java. Язык Scala использовался в нескольких крупных проектах, в том числе в распределенном графе LinkedIn и новой системе логирования Kafka.

Крис создал для проекта Kafka систему потоковой обработки данных под названием Samza. Он быстро понял, что простая в теории интеграция с JVM не работает на практике. Многие библиотеки Java было неудобно совмещать с библиотеками коллекций Scala. Среда сборки Scala тоже отличалась неудобством. Компания LinkedIn использовала в качестве системы сборки молодую технологию Gradle, а она не поддерживала Scala: сообщество Scala использовало Scala Build Tool (SBT).

SBT — это встроенная в сам язык Scala система сборки, определяющая предметно-ориентированный язык (DSL, domain-specific language) для создания файлов сборки. Крис выяснил, что для разных версий SBT предусматривались два совершенно разных DSL, и в большинстве примеров, найденных во Всемирной паутине, использовался старый синтаксис, а документация по новому синтаксису была ему совершенно непонятна.

На протяжении следующих лет язык Scala продолжал оставаться для Криса источником таких проблем, как двоичная несовместимость между версиями, ошибки сегментации JVM, незрелость библиотек и отсутствие интеграции с внутренними инструментами LinkedIn. Команда начала скрывать Scala-код, удаляя его из клиентских библиотек. Для Криса, которого больше интересовала потоковая обработка, а не особенности языка, использование Scala в 2011 году казалось неудачным выбором, ведь ему приходилось тратить слишком много времени на проблемы, связанные с языком и инструментами. Выбранная технология была достаточно скучной.

Не самовольничайте

Не игнорируйте стандарты вашей компании (или отрасли) только потому, что они вам не нравятся. Нестандартный код не впишется в среду компании. Система непрерывной интеграции, плагины IDE, модульные тесты, линтеры, инструменты агрегирования журналов, панели показателей и конвейеры данных уже интегрированы. Применение вашего индивидуального подхода обойдется слишком дорого.

Ваши методы действительно могут быть лучше, однако самовольничать не надо. В краткосрочной перспективе следует делать то же, что и остальные. Попытайтесь понять причины, лежащие в основе стандартного подхода. Вполне возможно, он решает какую-то неочевидную для вас проблему. Если вы сами не можете найти причину, спросите коллег. Если и они вам не помогут, поговорите со своим руководителем и с командой, владеющей технологией.

При изменении стандартов необходимо учитывать множество аспектов: приоритет, право собственности, стоимость и нюансы реализации. Убедить команду отказаться от того, что ей принадлежит, будет непросто. Вы услышите множество мнений. Придерживайтесь прагматичного подхода.

Как и в случае с переписыванием кода, изменение того, что уже широко распространено, происходит очень медленно. Это не значит, что ничего делать не надо. Вы можете добиться своей цели, если воспользуетесь правильными каналами и при этом познакомитесь с другими отделами организации, что поспособствует налаживанию новых связей и вашему карьерному продвижению. Кроме того, вы одним из первых начнете пользоваться новым решением. Сделав свой вклад, вы получите то, что хотите. Однако старайтесь не слишком сильно отвлекаться от основной работы и предупредите руководителя, что тратите время на новый проект.

Не делайте форк, если не собираетесь обновлять исходный репозиторий

Форк — это полная независимая копия репозитория исходного кода. Форк имеет собственный ствол, ветки и теги. На платформе для совместного использования кода, например GitHub, форки создаются перед отправкой в исходный или вышестоящий репозиторий запроса

на внесение изменений. Форки позволяют людям, не имеющим права производить запись в основной репозиторий, вносить свой вклад в проект. И это распространенная практика.

Плохая практика — создавать форк без намерения вносить изменения в исходный репозиторий. Такое бывает при возникновении разногласий по поводу направления развития проекта, при его заморозке или наличии сложностей с внесением изменений в основную кодовую базу.

К особо пагубным последствиям приводит развитие внутреннего форка компании. При этом разработчики обычно обещают друг другу внести изменения «потом», но этого, как правило, так и не происходит. Незначительных изменений, не вносимых в вышестоящий репозиторий, со временем становится все больше, и в конце концов форк превращается в совершенно независимое программное обеспечение. Становится все труднее вносить в исходный репозиторий функции и исправления. В итоге команде приходится поддерживать еще один полноценный проект. Некоторые компании даже создают форки собственных проектов с открытым исходным кодом, потому что не вносят внутренних изменений.

Сопротивляйтесь искушению переписать код

Рефакторинг часто перерастает в полномасштабное переписывание кода. Рефакторинг существующего кода — задача непростая, и часто появляется соблазн просто отказаться от старой системы и переписать все с нуля. Считайте переписывание крайней мерой. Этот совет основан на многолетнем опыте.

В каких-то случаях переписывание кода оправданно, в каких-то — нет. Сообщите команде о своем желании переписать код, однако имейте в виду, что ваша антипатия к языку или фреймворку не является достаточно уважительной причиной для этого. Переписывать код следует лишь в том случае, если потенциальная выгода превышает затраты. Кроме того, следует помнить, что переписывание кода сопряжено с риском, а его стоимость довольно высока. Помимо всего прочего, инженеры склонны недооценивать связанные с этим временные затраты: особенно ужасной является миграция кода, которая предполагает перемещение

данных и обновление нижестоящих и вышестоящих систем — на это могут уйти годы или даже десятилетия.

Переписывание кода не всегда будет наилучшим решением. В своей известной книге *The Mythical Man-Month* (Addison-Wesley Professional, 1995)[1] Фредерик Брукс ввел понятие «синдром второй системы», которое описывает тенденцию к замене простых систем более сложными. Первая система имеет ограниченный масштаб из-за недостаточного понимания разработчиками предметной области: система справляется со своими задачами, но ее возможности ограниченны. Приобретя некоторый опыт, разработчики начинают осознавать допущенные ошибки и приступают к созданию второй системы, в которой намереваются реализовать свои новые идеи. Новые системы обычно призваны обеспечить гибкость, но, как правило, получаются раздутыми. Если вы решили переписать систему, старайтесь не допустить ее чрезмерного расширения.

МИГРАЦИЯ DUCK DUCK GOOSE

Инструмент A/B-тестирования компании Twitter называется Duck Duck Goose (DDG). Первая версия DDG была создана на самых ранних этапах развития компании, но после нескольких лет стремительного роста начала устаревать. Некоторые из самых опытных инженеров задумались об обновлении DDG: они хотели улучшить архитектуру системы, сделав ее более надежной и удобной, хотели сменить язык с Apache Pig на Scala, а заодно и устранить другие ограничения. Дело было спустя несколько лет после истории Криса с Samza, так что компания Twitter на тот момент уже потратила годы на внедрение Scala. Дмитрию было поручено управлять командой, сформированной для реализации проекта, — она состояла из разработчиков прежней версии, инженеров, обладающих новыми идеями, и еще нескольких человек. Предполагалось, что три месяца уйдет на создание и выпуск новой системы и еще три месяца — на списание старой.

[1] *Брукс Ф.* Мифический человеко-месяц.

Однако ушел год на то, чтобы сделать работу DDGv2 достаточно стабильной, и еще полгода — чтобы отказаться от старой версии DDG. В ходе работы инженеры оценили старую кодовую базу, и уровни сложности в коде обрели для них смысл. В конце концов новый инструмент во многих отношениях превзошел своего предшественника и прошел проверку временем: теперь он уже старше, чем была первая версия, когда ее заменили. Реализация проекта, которая, по оценкам опытных инженеров и руководителей, должна была занять лишь шесть месяцев, в итоге заняла полтора года и потребовала дополнительных затрат на разработку в размере более миллиона долларов. Не обошлось и без непростых разговоров с вице-президентами: «Нет, правда, босс, так будет лучше. Но на это уйдет еще три месяца». Из трех разработчиков, создававших прототип переписанной системы, двое покинули компанию, а третий перешел в другую команду еще до того, как работа над проектом была завершена.

Не беритесь за переписывание кода, думая, что это пустяк. Готовьтесь к тому, что процесс будет долгим и утомительным.

Что следует и чего не следует делать

Следует	Не следует
Выполнять рефакторинг постепенно	Злоупотреблять выражением «технический долг»
Отделять коммиты, связанные с рефакторингом, от коммитов, связанных с добавлением новых функций	Делать методы или переменные общедоступными в целях тестирования
Вносить небольшие изменения	Проявлять языковой снобизм
Оставлять код более чистым, чем он был до вас	Игнорировать стандарты и инструменты компании
Использовать скучные технологии	Делать форк без намерения обновить исходный репозиторий

Повышение уровня

Мы сами довольно часто обращаемся к книге Майкла К. Физерса (Michael C. Feathers) *Working Effectively with Legacy Code* (Pearson, 2004)[1] и рекомендуем ее вам, если вы сталкиваетесь с объемными и запутанными кодовыми базами. Кроме того, полезной будет и книга Джонатана Боккара (Jonathan Boccara) *The Legacy Code Programmer's Toolbox* (LeanPub, 2019).

Мартин Фаулер (Martin Fowler) очень много написал о рефакторинге, и когда у вас будет свободное время, почитайте его блог. В качестве канонической книги по рефакторингу мы рекомендуем работу Фаулера *Refactoring: Improving the Design of Existing Code* (Addison-Wesley Professional, 1999)[2].

Наконец, следует еще раз упомянуть книгу Фредерика Брукса (Fred Brooks) *The Mythical Man-Month* (Addison-Wesley Professional, 1995). Это классический труд, с которым нужно ознакомиться каждому инженеру-программисту. В нем рассказывается о практических аспектах проектирования программного обеспечения. Вы удивитесь, насколько эта книга будет актуальна в вашей повседневной деятельности.

---

1 *Физерс М. К.* Эффективная работа с унаследованным кодом.

2 *Фаулер М.* Рефакторинг: улучшение существующего кода.

4 Написание работоспособного кода

Код ведет себя странно, когда выполняется в «реальном мире». Поведение пользователей невозможно предсказать. Работа сетей оказывается ненадежной. Дела идут не так, как задумано. При этом программное обеспечение должно продолжать работать. Написание работоспособного кода позволяет справиться с непредвиденными обстоятельствами. Работоспособный код имеет встроенные средства защиты, диагностики и управления. Защитите свою систему, применив при ее создании методы безопасного и отказоустойчивого программирования. Безопасный код предотвращает множество сбоев, а отказоустойчивый код помогает быстро восстановить работу системы, если сбой все-таки случится. Чтобы иметь возможность проводить диагностику сбоев, следите за тем, что происходит: для упрощения этого процесса обеспечьте доступ к информации, содержащейся в журналах, а также к метрикам и результатам трассировки вызовов. Наконец, системами нужно управлять, не переписывая код. Работоспособная система предусматривает наличие параметров конфигурации и системных инструментов.

В этой главе описываются некоторые передовые практики, упрощающие процесс промышленной эксплуатации кода. Чтобы затронуть как можно больше тем, мы постарались изложить материал максимально сжато, но представить все основные концепции и инструменты, необходимые для обеспечения работоспособности вашего программного обеспечения. Комментарии, касающиеся работоспособности кода,

часто оказываются актуальными при его рецензировании, поэтому предложенная информация поможет вам давать и получать более качественную обратную связь.

Защитное программирование

Хорошая защита кода — это проявление сострадания ко всем, кто его использует, включая вас самих! Защищенный код реже дает сбой, а если подобное все-таки происходит, работоспособность кода легче поддается восстановлению. Сделайте свой код безопасным и отказоустойчивым.

Безопасный код предполагает проверку на этапе компиляции, что снижает вероятность возникновения сбоев во время его выполнения. С целью предотвращения ошибок используйте неизменяемые переменные, модификаторы доступа для ограничения области видимости и средства статической проверки типов. Во время выполнения проверяйте входные данные, чтобы избежать неожиданностей. В *отказоустойчивом коде* применяются передовые методы обработки исключений и корректно обрабатываются сбои.

Избегайте значений null

Во многих языках программирования переменным без значения по умолчанию присваивается значение `null` (`nil`, `None` или какой-либо другой его вариант). Обычным явлением при этом оказываются исключения нулевого указателя. Результаты трассировки стека вызывают недоумение и заставляют задаться вопросом наподобие такого: «Как вышло, что для этой переменной не было задано значение?» Чтобы избежать исключений нулевого указателя, удостоверьтесь, что переменные не имеют значение `null`: используйте шаблон проектирования Null Object и опциональные типы.

Выполняйте проверки на наличие значений `null` в начале методов. По возможности используйте аннотации `NotNull` и аналогичные языковые функции. Предварительная проверка того, что значение переменных не равно `null`, даст последующему коду основание полагать, что он имеет дело с реальными значениями: это сделает код более чистым и удобочитаемым.

Шаблон Null Object использует объекты вместо значений `null`. Примером является метод поиска, который возвращает пустой список вместо значения `null` в том случае, если объекты не найдены. Возврат пустого списка позволяет вызывающим объектам перебирать результаты без необходимости использовать специальный код для обработки их пустых наборов.

Некоторые языки предусматривают встроенные *опциональные типы* — `Optional` или `Maybe`, — которые заставляют разработчиков задуматься о способе обработки пустых ответов. Воспользуйтесь преимуществами опциональных типов, если у вас есть такая возможность.

Делайте переменные неизменяемыми

Неизменяемыми называются переменные, которые нельзя изменить после установки. Если ваш язык предусматривает способ явного объявления неизменяемых переменных (`final` в Java, `val` вместо `var` в Scala, `let` вместо `let mut` в Rust), по возможности пользуйтесь им. Неизменяемые переменные позволяют предотвратить неожиданные изменения.

Неизменяемыми можно сделать гораздо больше переменных, чем кажется на первый взгляд. Как дополнительный бонус: использование неизменяемых переменных упрощает параллельное программирование и повышает эффективность работы компилятора или среды выполнения благодаря их уверенности в том, что переменная не будет меняться.

Используйте подсказки типа и средства статической проверки типа

Задайте ограничения для значений переменных. Например, переменные, для которых допустимыми являются лишь несколько строковых значений, должны иметь тип `Enum`, а не `String`. Ограничение значений переменных гарантирует, что неожиданные значения не просто вызовут появление ошибок, а сразу же приведут к сбою (или не позволят компилятору завершить работу). При определении переменных используйте наиболее конкретный тип.

Динамические языки, такие как Python (начиная с версии Python 3.5), Ruby (благодаря инструменту Sorbet, который планируется включить

в Ruby 3) и JavaScript (благодаря TypeScript), теперь предусматривают гораздо более надежные *подсказки типа* и *средства статической проверки типа*. Подсказка типа позволяет указать тип переменной при написании кода на языке, который обычно является динамически типизированным. Например, следующий метод Python 3.5 использует подсказку типа для получения и возврата строки:

```
def say(something: str) -> str:
  return "You said: " + something
```

Больше всего радует, что подсказки типов можно постепенно добавлять в существующие кодовые базы. Сочетая их со средством статической проверки типа, использующим подсказки типа для нахождения ошибок перед выполнением кода, можно предотвратить сбои во время его выполнения.

Проверяйте входные данные

Никогда не доверяйте входным данным, которые получает ваш код. Из-за ошибок разработчиков, неисправности оборудования и человеческого фактора они могут оказаться поврежденными. Чтобы защитить свой код, удостоверьтесь в корректности входных данных. Используйте предусловия и контрольную сумму, проводите валидацию данных, применяйте передовые методы обеспечения безопасности и инструменты для нахождения распространенных ошибок. Старайтесь как можно раньше отклонять некорректные входные данные.

Уточняйте входные переменные метода с помощью предусловий и постусловий. Обращайтесь к библиотекам и фреймворкам, которые проверяют предусловия, когда используемый вами тип охватывает не весь диапазон допустимых значений переменных. Большинство языков предусматривают библиотеки с такими методами, как `checkNotNull`, или с аннотациями вроде `@Size(min=0, max=100)`. Старайтесь максимально конкретизировать ограничение. Убедитесь, что входные строки имеют ожидаемый формат, и не забудьте о начальных или конечных пробелах. Удостоверьтесь в том, что численные значения принадлежат соответствующим диапазонам: если значение параметра должно быть больше нуля, проверьте, чтобы так оно и было; если значением параметра является IP-адрес, подтвердите его действительность.

ДОКУМЕНТЫ WORD НЕ ПРИНИМАЮТСЯ

Во время учебы в колледже Дмитрий подрабатывал в лаборатории сравнительной геномики. Его команда создала веб-сервис для ученых, который позволял загружать последовательности ДНК и анализировать их с помощью инструментов лаборатории. Одна из наиболее распространенных причин возникновения ошибок заключалась в том, что биологи помещали текст последовательности ДНК — длинную строку, состоящую из букв A, C, T и G, — в документ Word, а не в простой текстовый файл. Разумеется, это приводило к сбою в работе парсеров, и никаких результатов система не выдавала. При этом пользователь получал сообщение о том, что соответствующая последовательность не найдена. И подобное было обычным явлением. Пользователи отправляли отчеты об ошибках, в которых говорилось о неисправности поисковой системы, неспособной обнаружить последовательности ДНК, точно находящиеся в базе данных.

Так продолжалось довольно долго. Команда винила пользователей, потому что в инструкциях было четко сказано о необходимости работать с «текстовым файлом». В конце концов Дмитрию надоело отвечать на электронные письма, рассылая инструкции по сохранению текстового файла, и он обновил сайт. Думаете, он добавил парсер документов Word? Не угадали. Полагаете, внедрил средства проверки формата файла и обработки ошибок с целью предупреждать пользователя о том, что сайт не может обработать отправляемые им данные? И снова нет. Он добавил большой значок Microsoft Word, перечеркнутый красной линией, и ссылку на инструкции. Количество обращений в службу поддержки резко сократилось! Успех!

Старый сайт все еще работает, хотя и был обновлен. Перечеркнутый значок Microsoft Word исчез, осталось лишь предупреждение: «Только текстовые файлы. Документы Word не принимаются». Спустя 15 лет после ухода с этой работы Дмитрий попытался загрузить документ Word, содержащий последовательность хорошо изученного гена. При этом он не получил ни нужного результата, ни сообщения об ошибке. То есть больше десяти лет система выдавала вводящие в заблуждение результаты только потому, что Дмитрию было лень позаботиться об обработке входных данных.

Не будьте такими, каким был 20-летний Дмитрий, — вполне возможно, что своей ленью он саботировал поиски лекарства от рака.

Компьютерное оборудование не всегда заслуживает доверия. Сети и диски могут повредить данные. Если вам нужны гарантии надежности, используйте контрольные суммы для проверки того, что данные не были изменены неожиданным образом.

Не забывайте и о безопасности. Входные данные, получаемые извне, могут представлять опасность. Злоумышленники могут попытаться внедрить в них вредоносный SQL-код или выйти за границы буфера при чтении, чтобы получить контроль над вашим приложением. Используйте проверенные библиотеки и фреймворки для защиты от межсайтового скриптинга. Всегда экранируйте входные данные для предотвращения SQL-инъекций. Явно задавайте параметры размера при манипулировании памятью с помощью таких команд, как `strcpy` (в частности, используйте `strncpy`), чтобы предотвратить переполнение буфера. Используйте широко распространенные библиотеки или протоколы безопасности и криптографии, а не пишите собственные. Ознакомьтесь с отчетом, который публикует Open Web Application Security Project (OWASP) (https://owasp.org/www-project-top-ten/), и узнайте об актуальных угрозах безопасности.

Используйте исключения

Не используйте специальные возвращаемые значения (`null`, `0`, `-1` и т. д.), чтобы сообщить об ошибке. Все современные языки программирования поддерживают исключения или предусматривают стандартный шаблон их обработки (например, тип `error` в Go). Специальные значения не всегда прописываются в сигнатуре метода. Разработчики могут не знать о возникновении ошибок, требующих обработки. Также трудно бывает запомнить, какое возвращаемое значение соответствует тому или иному состоянию отказа. Исключения сообщают гораздо больше информации, чем значения `null` или `-1`; они предусматривают конкретное название, данные трассировки стека, номера строк и сообщения.

Например, в языке Python исключение `ZeroDivisionError` предоставляет намного больше информации, чем возвращаемое значение `None`:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Во многих языках проверяемые исключения прописываются в сигнатуре методов:

```
// в языке Go метод Open явно возвращает значение error
func Open(name string) (file *File, err error)

// в языке Java метод open() явно генерирует исключение IOException
public void open (File file) throws IOException
```

Объявление ошибки в языке Go и генерация исключения в Java ясно указывают на то, что эти методы `open` могут вызывать ошибки, требующие обработки.

Конкретизируйте исключения

Конкретные исключения упрощают процесс использования кода. По возможности применяйте встроенные исключения и избегайте создания обобщенных исключений. Используйте исключения для обработки сбоев, а не для управления логикой приложения.

Большинство языков предусматривают встроенные типы исключений (`FileNotFoundException`, `AssertionError`, `NullPointerException` и т. д.). Не создавайте пользовательское исключение, если встроенный тип позволяет описать проблему. Разработчики имеют опыт работы с существующими типами исключений и уже знают, что они означают.

Создавая собственные исключения, не делайте их слишком обобщенными. Обобщенные исключения трудно обрабатывать, потому что из них разработчики не смогут понять, о какой именно проблеме идет речь. Не получив точной информации о произошедшей ошибке, они будут вынуждены остановить выполнение программы. Старайтесь использовать максимально конкретные типы исключений, чтобы разработчики могли адекватно реагировать на сбои.

Не стоит также использовать исключения для управления логикой приложения: ваш код должен быть предсказуемым, а не умным. Использование исключений для выхода из метода может сбить с толку и затруднить отладку кода.

В следующем фрагменте кода на языке Python используется исключение `FoundNodeException` вместо непосредственного возврата найденного узла:

```
def find_node(start_node, search_name):
  for node in start_node.neighbors:
    if search_name in node.name:
      raise FoundNodeException(node)
    find_node(node, search_name)
```

Не делайте так. Просто возвратите узел.

Генерируйте исключения как можно раньше, а перехватывайте как можно позже

Следуйте принципу «выбрасывай рано, лови поздно». *Ранний выброс* означает генерацию исключений в непосредственной близости от места возникновения ошибки, что позволяет разработчикам быстро найти соответствующий код. Затягивание с генерацией исключения затрудняет поиск места, в котором произошел сбой. Когда перед генерацией исключения выполняется другой код, существует риск возникновения еще одной ошибки. Если для второй ошибки будет сгенерировано исключение, вы не узнаете о возникновении первой. Выявление ошибок такого рода способно свести с ума. Исправив одну ошибку, вы можете обнаружить, что настоящая проблема возникла где-то раньше.

Поздний перехват означает распространение исключений вверх по стеку вызовов вплоть до достижения уровня программы, способного обработать это исключение. Рассмотрим приложение, которое пытается произвести запись на заполненный диск. В данном случае возможны разные варианты последующих действий: блокировка и повторная попытка, асинхронная повторная попытка, запись на другой диск, отображение предупреждения для пользователя или даже аварийное завершение программы. Адекватная реакция зависит от специфики приложения. Информация в журнале предзаписи базы данных должна быть зафиксирована сразу, в то время как фоновое сохранение в текстовом редакторе может быть отложено. Фрагмент кода, способный принять это решение, вероятнее всего, будет отделен от низкоуровневой библиотеки, столкнувшейся с проблемой заполненного диска, несколькими уровнями. Исключение

начнет распространяться вверх через все эти промежуточные уровни, на которых не должно предприниматься попыток его преждевременного исправления. Худшим вариантом здесь являются «проглатывание» исключения и невозможность его обработки, что обычно выражается в игнорировании исключения в блоке `catch`:

```
try {
  // ...
} catch (Exception e) {
  // игнорирую, поскольку ничего не могу с этим поделать
}
```

Исключение не будет ни зарегистрировано, ни сгенерировано повторно: оно просто проигнорируется, и никаких мер по его поводу принято не будет. О существующей неисправности никто не узнает, а это может привести к катастрофическим последствиям. При вызове кода, способного сгенерировать исключения, следует либо полностью их обрабатывать, либо обеспечить их распространение вверх по стеку.

Выполняйте повторные попытки с умом

Зачастую самой правильной реакцией на ошибку является повторная попытка. В отдельных ситуациях имеет смысл запланировать некоторое количество повторных попыток при обращении к удаленным системам. Повторная попытка не сводится к простому перехвату исключения и повтору операции. На практике принятие решения относительно того, когда и как часто необходимо повторять попытки, требует определенного ноу-хау.

Самый наивный подход предполагает перехват исключения и немедленное повторение операции. Но что если очередная попытка тоже окажется неудачной? Ситуация, когда на диске не хватает места, не изменится, скорее всего, ни через 10, ни через 20 миллисекунд. В этом случае повторяющиеся снова и снова попытки приведут лишь к замедлению работы системы и помешают ее восстановлению.

Разумным подходом является использование так называемой *выдержки* (backoff). Такая стратегия предполагает нелинейное увеличение интервала между попытками (обычно используется экспоненциальная выдержка, например `(retry number)^2`). При этом не забудьте задать для

выдержки предельно допустимое значение, чтобы она не стала слишком большой. Однако если на сетевом сервере возник сбой и все клиенты столкнулись с ним одновременно и затем все они, применяя один и тот же алгоритм, использовали выдержку, то так же одновременно они повторят и отправку своих запросов. Это называется проблемой *несметной орды (thundering herd)*: повторная отправка множества запросов может помешать восстановлению работы сервиса. Чтобы справиться с проблемой, используйте *джиттер*, добавляющий к интервалу выдержки случайный промежуток времени. Такой элемент случайности позволит рассредоточить запросы и уменьшить вероятность перегрузки сервера.

Не следует слепо повторять все неудачные вызовы, особенно связанные с записью данных или выполнением какого-либо бизнес-процесса. Лучше позволить приложению аварийно завершить работу при обнаружении ошибки, для обработки которой оно не предназначено: то есть лучше *потерпеть неудачу быстро*. Такой подход предотвращает дальнейшие повреждения и позволяет человеку определиться с правильным планом действий. Позаботьтесь о том, чтобы терпеть неудачу не только быстро, но и громко: обеспечьте отображение соответствующей информации для облегчения процесса отладки.

Создавайте идемпотентные системы

То, в каком состоянии осталась система после сбоя, не всегда бывает очевидно. Если во время отправки удаленного запроса на запись происходит сбой в работе сети, вам неизвестно, был ли запрос обработан до сбоя. Это ставит вас в затруднительное положение, поскольку вы не знаете, стоит ли повторить попытку с риском выполнить запрос дважды или же лучше отказаться от повторной попытки, рискуя потерять данные. В случае биллинговой системы повторная попытка может привести к удвоению суммы счета, выставленного клиенту, в то время как отказ от повторной попытки может означать, что счет вообще не будет выставлен. Иногда можно осуществить проверку, обратившись к удаленной системе, — но не всегда. Мутации локального состояния способны порождать аналогичные проблемы. Мутации нетранзакционной резидентной структуры данных могут оставить вашу систему в несогласованном состоянии.

Лучший способ решения проблем, связанных с повторными попытками, состоит в создании идемпотентных систем. *Идемпотентной* называется операция, которую можно выполнять многократно, получая при этом один и тот же результат. Примером такой операции является добавление значения в набор. Вне зависимости от того, сколько раз вы добавляете значение, в наборе оно присутствует в единственном экземпляре. Удаленные API можно сделать идемпотентными, позволив клиентам предоставлять уникальный идентификатор для каждого запроса. Повторяя попытку, клиент предоставит тот же уникальный идентификатор, что и во время неудачной попытки, и сервер сможет исключить дополнительный запрос, если тот уже был обработан. Сделав все операции идемпотентными, вы можете значительно упростить процессы взаимодействия с системой и предотвратить возникновение целого класса ошибок.

Очищайте ресурсы

Предусмотрите очистку ресурсов на случай сбоя. Освобождайте память, структуры данных, сетевые сокеты и дескрипторы файлов, которые вам больше не нужны. Операционные системы выделяют фиксированный объем пространства для дескрипторов файлов и сетевых сокетов: после превышения этого значения новые дескрипторы и сокеты не будут открываться. Утечки, возникающие тогда, когда сетевые сокеты не закрываются после использования, будут поддерживать бесполезные соединения, способствуя переполнению их пулов. Следующий фрагмент кода представляет опасность:

```
f = open('foo.txt', 'w')
#
f.close()
```

Любые сбои, произошедшие до `f.close()`, предотвратят закрытие указателя на файл. Если ваш язык не поддерживает автоматическое закрытие, заключите код в блок `try/finally`, чтобы обеспечить закрытие дескрипторов файлов даже в случае возникновения исключения.

Многие современные языки программирования предусматривают функции автоматического закрытия ресурсов. Rust автоматически закрывает

ресурсы, вызывая деструктор, когда объекты покидают область видимости. Оператор `with` в Python автоматически закрывает дескрипторы, когда путь вызова покидает блок:

```
with open('foo.txt') as f:
  # ...
```

Логирование

Впервые написав фразу «Hello, world!» в окне терминала, вы внесли свою первую запись в журнал. Вывод на экран записей журнала упрощает понимание кода и помогает отладить небольшую программу. Для более сложных приложений языки программирования предусматривают библиотеки логирования, которые дают операторам больше контроля над тем, что и когда именно регистрируется в журналах. Операторы могут регулировать объем журналов с помощью уровней их ведения, а также управлять их форматами. Помимо этого, фреймворки позволяют включить контекстную информацию, например имена потоков, имена хостов и идентификаторы, для использования ее при отладке. Фреймворки для логирования хорошо работают с системами управления журналами, которые агрегируют журнальные записи, позволяя оператору производить их фильтрацию и поиск.

Используйте фреймворк для логирования с целью упростить свой код и облегчить его отладку. Задайте уровни ведения журналов, чтобы операторы могли контролировать объем журналов вашего приложения. Сделайте журналы атомарными, быстрыми и безопасными.

Используйте уровни логирования

Фреймворки для логирования предусматривают *уровни логирования* или *ведения журнала*, которые позволяют операторам фильтровать сообщения по степени важности. После задания уровня логирования будут регистрироваться только сообщения, относящиеся к указанному и более высокому уровням. Обычно эти уровни контролируются как глобальными настройками, так и переопределениями на уровне пакета или класса. Уровни логирования позволяют операторам регулировать объем журналов в соответствии с конкретной ситуацией и создавать

журналы необходимой степени детализации, начиная от чрезвычайно подробных журналов отладки и заканчивая журналами, содержащими лишь общие сведения об обычных операциях.

В качестве примера рассмотрим фрагмент Java-кода из файла `log4j.properties`, который задает уровень детализации ERROR для корневого логгера (регистратора) и специфичный для пакета уровень детализации INFO для логов, поступающих из пространства пакета `com.foo.bar`:

```
# задать уровень ERROR для корневого логгера,
# связанного с файловым аппендером fout
log4j.rootLogger=ERROR,fout

# задать уровень INFO для com.foo.bar
log4j.logger.com.foo.bar=INFO
```

Для того чтобы извлечь пользу из уровней логирования, необходимо определиться с нужной степенью критичности выводимого сообщения. Следующие уровни логирования не являются стандартом, но используются довольно часто.

- **TRACE.** Чрезвычайно высокий уровень детализации, который включается, как правило, в процессе разработки только для определенных пакетов или классов. Если вам нужны журналы, в которых регистрируется все подряд, или дампы структур данных, выбирайте именно этот уровень. В случае слишком частого применения уровня TRACE подумайте об использовании отладчика для пошагового выполнения кода.
- **DEBUG.** Сообщения этого уровня бывают полезны для решения проблем в процессе производства, а не обычной работы. Однако не следует использовать данный уровень логирования для вывода чрезмерно детализированных сообщений, которые затрудняют процесс отладки: для обеспечения такой детализации лучше применять уровень TRACE.
- **INFO.** Сообщения содержат полезную информацию о состоянии приложения, но не указывают на какие-либо проблемы. Примерами являются такие сообщения, как «Сервис запущен» и «Прослушивание порта 5050». Уровень логирования INFO используется по умолчанию. Для создания журналов, в которых регистрируется информация «на всякий случай», следует использовать уровень TRACE или DEBUG.

Сообщения уровня INFO должны содержать сведения, которые могут пригодиться во время нормальной работы.

- **WARN.** Сообщения содержат информацию о потенциальных проблемных ситуациях. Примером такой ситуации является перспектива исчерпания того или иного ресурса. Каждое регистрируемое предупреждение должно сопровождаться описанием конкретного действия, которое должен предпринять читающий его человек. Если предупреждение не требует выполнения каких-либо действий, используйте для его регистрации уровень INFO.
- **ERROR.** Сообщения свидетельствуют о возникновении ошибки, требующей внимания, например о недоступности базы данных для записи. Сообщения уровня ERROR должны быть достаточно подробными для диагностики проблем. Регистрируйте подробные сведения, включая данные трассировки стека и результирующие действия, предпринимаемые программным обеспечением.
- **FATAL.** Это сообщения об ошибках, являющихся фатальными для работы системы. Если программа сталкивается с настолько серьезной проблемой, что вынуждена немедленно завершить свою работу, то для регистрации сообщения о причине этой проблемы можно использовать уровень FATAL. Включите в него контекстную информацию о состоянии программы, а также сведения о местах восстановления или диагностические данные.

Вот пример журнала уровня INFO, созданного в Rust:

```
info!("Failed request: {}, retrying", e);
```

Эта строка содержит сведения об ошибке, из-за которой не выполняется запрос. Уровень INFO используется потому, что приложение автоматически повторяет попытку. Никаких действий со стороны оператора не требуется.

Обеспечьте атомарность логов

Если та или иная информация оказывается полезной лишь в сочетании с другими данными, записывайте ее всю в одном сообщении. *Атомарные* логи, содержащие всю необходимую информацию в одной строке, лучше работают с агрегаторами журналов. Не стоит рассчитывать

на то, что логи будут отображаться в определенном порядке: многие операционные инструменты изменяют порядок сообщений или даже отбрасывают их. Не стоит полагаться и на упорядочение с помощью временных меток, поскольку показания системных часов могут быть сброшены или отличаться у различных хостов. Не используйте в журналах символы перевода строки, поскольку многие агрегаторы журналов воспринимают новые строки в качестве отдельных сообщений. Удостоверьтесь в том, что данные трассировки стека сохраняются в одном сообщении, поскольку они часто содержат символы перевода строки при выводе на экран.

Вот пример неатомарных сообщений журнала:

```
2022-03-19 12:18:32,320 - appLog — WARNING — Request failed with:
2022-03-19 12:18:32,348 — appLog — INFO — User login: 986
Unable to read from pipe.
2022-03-19 12:18:32,485 — appLog — INFO — User logout: 986
```

Сообщение WARNING содержит символ перевода строки, что затрудняет его чтение. Последующие строки этого сообщения не содержат временной метки и перемежаются сообщениями INFO из другого потока. Данное сообщение WARNING должно быть атомарным, то есть записанным в одну строку.

При невозможности атомарного вывода сообщений журнала снабдите их уникальным идентификатором, чтобы впоследствии эти сообщения можно было объединить.

Ускорьте процесс логирования

Излишнее логирование может снизить производительность. Сообщения должны куда-то записываться — на диск, в консоль или в удаленную систему. Перед записью строки должны быть объединены и отформатированы. Для ускорения процесса логирования используйте параметризованные методы ведения журнала и асинхронные аппендеры.

Вы обнаружите, что конкатенация строк происходит очень медленно и может иметь разрушительные последствия в циклах, чувствительных к производительности. Когда конкатенированная строка передается методу ведения журнала, конкатенация производится вне зависимости от

уровня детализации, поскольку оценка аргументов осуществляется до их передачи в метод. Фреймворки для логирования предусматривают механизмы, позволяющие отложить конкатенацию строк до тех пор, пока в ней не возникнет необходимость. Некоторые фреймворки сохраняют сообщения журнала в замыканиях (closures), которые оцениваются только при запросе строки журнала, в то время как другие предусматривают поддержку параметризованных сообщений.

Например, в Java есть три способа конкатенации строк в вызовах метода ведения журнала, два из которых конкатенируют строковый параметр перед вызовом метода `trace`:

```
while(messages.size() > 0) {
  Message m = message.poll();

  // Эта строка конкатенируется даже при отключенной трассировке!
  log.trace("got message: " + m);

  // Эта строка тоже конкатенируется при отключенной трассировке.
  log.trace("got message: {}".format(m));

  // Эта строка конкатенируется только при включенной трассировке.
  // Так быстрее.
  log.trace("got message: {}", m);
}
```

В последнем вызове используется параметризованная строка, оценка которой будет производиться только при фактической записи строки журнала.

Регулировать степень влияния на производительность можно и с помощью *аппендеров*. Аппендеры позволяют направлять журналы в разные места назначения: в консоль, файл или удаленный агрегатор журналов. Обычно аппендеры по умолчанию работают в вызывающем потоке аналогично вызову функции `print`. *Асинхронные* аппендеры регистрируют сообщения в журнале, не блокируя потоки выполнения: это повышает производительность, поскольку коду приложения не нужно ждать, когда записи будут внесены в журналы. *Пакетные* (batching) аппендеры буферизуют сообщения журнала в памяти перед их записью на диск, тем самым повышая пропускную способность записи. Страничный кэш операционной системы также повышает пропускную способность, выступая в качестве буфера. Хотя асинхронная и пакетная записи

улучшают производительность, они могут привести к потере сообщений журнала в случае сбоя в работе приложения, поскольку при этом не все журналы гарантированно будут сброшены на диск.

Имейте в виду, что изменение степени детализации и конфигурации журнала может устранить состояния гонки и ошибки благодаря замедлению работы приложения. Если после повышения степени детализации журнала с целью отладки вы обнаружите, что ошибка исчезла, знайте, что причиной этого может быть само изменение метода ведения журнала.

Не регистрируйте конфиденциальные данные

Будьте осторожны при работе с конфиденциальными данными. Сообщения журнала не должны содержать такие личные данные, как пароли, токены безопасности, номера кредитных карт или адреса электронной почты. Это может показаться очевидным, однако в данном случае легко допустить ошибку, поскольку зарегистрированный URL-адрес или HTTP-ответ может содержать информацию, защиту которой агрегаторы журналов не обеспечивают.

Большинство фреймворков поддерживают замену строк и сокрытие содержащихся в них данных на основе правил: настройте их, однако не полагайтесь на это как на единственное средство защиты. Лучше будьте параноиком, ведь регистрация конфиденциальных данных может создать угрозу безопасности и нарушить правила конфиденциальности.

Метрики

Добавьте в приложение метрики для отслеживания его действий. Метрики представляют собой числовой эквивалент журналов и позволяют измерять поведение приложения. Как долго длился запрос? Сколько элементов находится в очереди? Какое количество данных было записано на диск? Измерение поведения приложения помогает обнаруживать проблемы и упрощает процесс отладки.

Существуют три распространенных типа метрик: счетчики, измерители и гистограммы. Они схожи, но могут обозначать несколько разные вещи в зависимости от конкретной системы мониторинга. *Счетчики* (counters) измеряют число наступлений того или иного события. Используя счетчик попаданий в кэш и счетчик запросов, вы можете рассчитать процент кэш-попаданий. Значения счетчиков могут только увеличиваться или обнуляться при перезапуске процесса (они *монотонно увеличиваются*). *Измерители* (gauges) отражают значения того или иного параметра в конкретный момент времени: значения могут повышаться или понижаться подобно показаниям спидометра или индикатора уровня топлива в баке автомобиля. Измерители предоставляют такую статистику, как размер очереди, стека или карты. *Гистограммы* позволяют распределять события по диапазонам размерных показателей. Каждый диапазон имеет счетчик, значение которого увеличивается всякий раз, когда соответствующий показатель события попадает в этот диапазон. Как правило, гистограммы используются для измерения времени выполнения запросов или размера полезной нагрузки.

Производительность системы часто измеряется с помощью так называемых процентилей, например 99-го процентиля, обозначаемого как *P99*. Системе с задержкой P99 2 миллисекунды требуется 2 миллисекунды или меньше для обработки 99 % получаемых запросов. Процентили выводятся из гистограмм. Чтобы сократить объем отслеживаемых данных, некоторые системы требуют от пользователя указать интересующие его процентили. Если система по умолчанию отслеживает 95-й процентиль, а вашим целевым уровнем качества обслуживания (SLO) является P99, измените настройки соответствующим образом.

Метрики приложений агрегируются в такие централизованные *системы наблюдения* (мониторинга), как Datadog, LogicMonitor или Prometheus. Наблюдаемость — это концепция теории управления, которая определяет, насколько легко можно определить состояние системы по ее выводам. Системы наблюдения призваны облегчить определение состояния работающего приложения путем предоставления информационных панелей и инструментов мониторинга агрегированных метрик. Информационные панели сообщают операторам о том, что происходит в системе, а инструменты мониторинга выводят оповещения, исходя из значений показателей.

Кроме того, метрики используются для автоматического масштабирования системы вверх или вниз. *Автомасштабирование* часто имеет место в средах, которые обеспечивают динамическое распределение ресурсов. Например, облачные хосты могут автоматически регулировать количество запущенных экземпляров, отслеживая показатели нагрузки. Автомасштабирование наращивает ресурсы сервера, когда это необходимо, а когда данная потребность исчезает, сокращает их с целью экономии средств.

Чтобы отслеживать SLO, использовать системы наблюдения и пользоваться преимуществами автомасштабирования, вы должны все измерять. Метрики отслеживаются с помощью стандартной библиотеки метрик, предоставляемой большинством фреймворков для создания приложений. Ваша задача как разработчика — обеспечить доступность важных метрик для систем наблюдения.

Используйте стандартные библиотеки метрик

Несмотря на то что счетчики, измерители и гистограммы реализуются довольно легко, не стоит создавать собственную библиотеку метрик. Нестандартные библиотеки — это сущий кошмар с точки зрения обслуживания. Стандартные библиотеки легко интегрировать с любыми приложениями. Скорее всего, ваша компания отдает предпочтение какой-то конкретной библиотеке метрик — в таком случае используйте именно ее; если нет — предложите выбрать одну из библиотек.

Большинство систем наблюдения предоставляют клиентские библиотеки метрик на разных языках. Чтобы показать, как выглядят метрики, мы будем использовать клиент StatsD в простом веб-приложении, написанном на языке Python. Все библиотеки метрик очень похожи между собой, поэтому вы можете смело применить следующее описание к библиотеке, используемой вами.

Веб-приложение на языке Python в листинге 4.1 включает четыре метода: `set`, `get`, `unset` и `dump`. Методы `set` и `get` просто устанавливают и извлекают значения карты, хранящейся в базе сервиса. Метод `unset`

удаляет пары «ключ — значение» из карты, а метод `dump` преобразует данные карты в формат JSON и возвращает ее.

Листинг 4.1. Пример приложения Python Flask, использующего клиентскую библиотеку метрик StatsD

```
import json
from flask import Flask, jsonify
from statsd import StatsClient

app = Flask(__name__)
statsd = StatsClient()
map = {}

@app.route('/set/<k>/<v>')
def set(k, v):
    """ Sets a key's value. Overwrites if key already exists. """
    map[k] = v
    statsd.gauge('map_size', len(map))

@app.route('/get/<k>')
def get(k):
    """ Returns key's value if it exists. Else, None is returned. """
    try:
        v = map[k]
        statsd.incr('key_hit')

        return v
    except KeyError as e:
        statsd.incr('key_miss')
    return None

@app.route('/unset/<k>')
def unset(k):
    """ Deletes key from map if it exists. Else, no-op. """
    map.pop(k, None)
    statsd.gauge('map_size', len(map))

@app.route('/dump')
def dump():
    """ Encodes map as a JSON string and returns it. """
    with statsd.timer('map_json_encode_time'):
        return jsonify(map)
```

В этом примере для отслеживания успешных и неудачных попыток извлечения значения ключа используются счетчики `key_hit` и `key_miss` в методе `get` с `statsd.incr`. Таймер `statsd.timer` измеряет, сколько

времени требуется для преобразования данных карты в формат JSON: данные будут учтены во временной гистограмме. Сериализация — дорогостоящая операция, предполагающая интенсивное использование процессора, поэтому она тоже требует измерения. Измеритель `statsd.gauge` определяет текущий размер карты. Для отслеживания размера карты мы могли бы использовать методы `increment` и `decrement` с целью увеличения и уменьшения значения счетчика, однако код с использованием измерителя меньше подвержен ошибкам.

Фреймворки для создания веб-приложений, такие как Flask, как правило, сами рассчитывают множество метрик. Большинство из них способны подсчитывать все коды состояния HTTP для каждого вызова метода веб-сервиса, а также рассчитывать длительность выполнения HTTP-запросов. Фреймворки являются отличным источником бесплатных метрик: просто настройте фреймворк для вывода в вашу систему наблюдения. Помимо всего прочего, благодаря этим измерениям ваш код станет чище.

Измеряйте все

Измерения не требуют больших затрат, поэтому вам следует проводить их как можно чаще. В частности, измеряйте:

- пулы ресурсов;
- кэши;
- структуры данных;
- операции с интенсивным использованием ЦП;
- операции с интенсивным вводом-выводом;
- размер данных;
- исключения и ошибки;
- удаленные запросы и ответы.

Используйте измерители для определения размера пулов ресурсов. Обратите особое внимание на пулы потоков и пулы соединений. Большие размеры пулов указывают на то, что система зависла или не справляется с нагрузкой.

Подсчитывайте количество кэш-попаданий и кэш-промахов. Изменения в соотношении этих показателей влияют на производительность приложений.

Определяйте размер ключевых структур данных с помощью измерителей. Ненормальный размер структуры данных указывает на то, что происходит нечто странное.

Измеряйте длительность операций, предполагающих интенсивное использование ЦП. Обращайте особое внимание на чрезмерно дорогие операции сериализации данных. Простое преобразование структуры данных в формат JSON часто является самой затратной операцией в коде.

Дисковые и сетевые операции ввода-вывода — медленные и непредсказуемые: используйте таймеры для измерения их длительности. Измеряйте размер данных, с которыми работает ваш код, и размеры полезной нагрузки *удаленного вызова процедур* (RPC). Используя гистограммы, следите за размером данных, генерируемых для ввода-вывода, чтобы определить размеры данных 99-го процентиля. Большой размер данных влияет на объем потребляемой памяти, скорость ввода-вывода и использование диска.

Считайте все исключения, коды ошибок и случаи ввода данных в неправильном формате. Контроль за ошибками позволяет легко генерировать предупреждения, когда что-то идет не так.

Отслеживайте запросы к вашему приложению. Чрезмерно высокое или низкое количество запросов — явный признак наличия проблемы. Пользователи хотят, чтобы системы реагировали как можно быстрее, поэтому вам необходимо измерить задержку. Следите за временем отклика с целью выявления признаков замедления работы системы.

Выделите время на ознакомление с принципом работы своей библиотеки метрик. Метод расчета той или иной метрики не всегда бывает очевидным: многие библиотеки используют выборки. Выборка обеспечивает высокую производительность и сокращает использование диска и памяти, однако снижает точность измерений.

Трассировки

Все разработчики знают о трассировке стека, но существует не самый известный вид трассировки под названием *распределенная трассировка вызовов*. Один-единственный вызов API фронтенда может породить сотни нисходящих RPC-вызовов, адресованных различным службам. Распределенная трассировка вызовов позволяет объединить все эти нисходящие вызовы в один граф. Распределенная трассировка полезна для отладки, измерения производительности, понимания зависимостей и анализа стоимости системы, то есть для ответа на вопросы вроде: «Какие API являются наиболее дорогими в обслуживании?», «Какие заказчики обходятся дороже всего?» и т. д.

RPC-клиенты используют библиотеку трассировки для присвоения своему запросу идентификатора трассировки вызова. Последующие RPC-вызовы нисходящих служб снабжаются тем же идентификатором. Затем службы сообщают о вызовах, которые они получают, вместе с идентификатором трассировки и другой информацией, такой как теги метаданных и время обработки. Специальная система записывает данные и объединяет трассировки вызова по идентификатору. Обладая всей информацией, система трассировки может предоставить распределенные графы вызовов.

Идентификаторы трассировки вызовов обычно автоматически передаются через оболочки RPC-клиента и сервисные сетки. Убедитесь в передаче необходимых состояний при вызове других сервисов.

Конфигурация

В приложениях и сервисах должны быть предусмотрены параметры, позволяющие разработчикам или SRE-инженерам (Site Reliability Engineering, инжиниринг надежности сайтов) настраивать поведение этих программ при выполнении. Применение передовых методов конфигурирования упростит эксплуатацию вашего кода. Не стремитесь к излишней креативности: используйте стандартный формат конфигурации, задавайте разумные умолчания, проверяйте вводимые параметры и по возможности избегайте динамической конфигурации.

Для задания параметров конфигурации могут использоваться:

- файлы в таких простых и удобочитаемых форматах, как INI, JSON или YAML;
- переменные среды;
- флаги командной строки;
- пользовательский *предметно-ориентированный язык* (DSL);
- язык, на котором написано приложение.

Удобочитаемые для человека файлы конфигурации, переменные среды и флаги командной строки применяются чаще всего. Файлы используются тогда, когда требуется задать много значений или предполагается управление версиями конфигураций. Переменные среды легко устанавливаются в сценариях, а среды легко поддаются исследованию и логированию. Флаги командной строки легко устанавливаются и отображаются в таких списках процессов, как `ps`.

DSL оказываются полезны, когда конфигурация предполагает использование программируемой логики, например циклов `for` или операторов `if`. Конфигурация на основе DSL обычно применяется в том случае, если приложение написано на дружественном DSL языке, например Scala. Используя DSL вместо полноценного языка программирования, разработчики могут предусмотреть обходные пути для выполнения сложных операций и ограничить параметры конфигурации безопасными значениями и типами, что является важным фактором с точки зрения безопасности и производительности системы при запуске. Однако DSL сложно анализировать с помощью стандартных парсеров, а это затрудняет обеспечение совместимости с другими инструментами.

Задание параметров конфигурации с помощью языка, на котором написано приложение, как правило, имеет место тогда, когда приложение написано на языке сценариев, например на Python. Использование кода для создания конфигурации — мощный инструмент, но также и опасный: настраиваемая логика препятствует обзору той конфигурации, которую видит приложение.

При настройке конфигурации не стремитесь к излишней креативности

Системы конфигурации должны быть скучными. Если оператора вызвали на работу в 3 часа ночи, ему необязательно помнить синтаксис Tcl для изменения значения тайм-аута.

Соблазн внедрить инновации в систему конфигурации вполне понятен. Процесс конфигурации знаком всем, а в простых системах конфигурации часто не хватает таких полезных функций, как подстановка переменных, операторы `if` и т. д. Многие творческие люди из лучших побуждений потратили огромное количество времени на создание сложных систем конфигурации. Но, к сожалению, чем более изощренной является система, тем более причудливы ее ошибки. Не стремитесь к излишней креативности при настройке конфигурации — используйте самый простой из работоспособных подходов. Файл статической конфигурации в едином стандартном формате является идеальным вариантом.

Большинство приложений настраиваются с помощью файла статической конфигурации. Изменение этого файла в процессе работы приложения никак на него не влияет: чтобы изменения вступили в силу, приложение нужно перезапустить. Системы динамической конфигурации используются тогда, когда приложение необходимо перенастроить без перезапуска. Динамическая конфигурация, как правило, хранится в специальной службе конфигурации, к которой обращается приложение при изменении значений. Альтернативным способом обновления параметров динамической конфигурации является периодическая проверка локального файла конфигурации на предмет наличия обновлений.

Из-за сложности, свойственной динамической конфигурации, ее использование обычно не является оправданным, поскольку требует учета всех последствий изменения различных параметров в процессе работы. Кроме того, становится труднее отслеживать такие данные, как время изменения параметра, автор изменения и предыдущее значение, а вся эта информация может иметь решающее значение при устранении

операционных проблем. Динамическая конфигурация также может добавлять внешние зависимости от других распределенных систем. Несмотря на кажущуюся примитивность, перезапуск процесса с целью обновления конфигурации обычно является более предпочтительным подходом с эксплуатационной и архитектурной точек зрения.

Тем не менее существует несколько ситуаций, которые требуют использования именно динамической конфигурации. Детальность журнала часто является динамическим параметром: операторы могут изменить уровень логирования на более высокий, например на DEBUG, когда происходит нечто странное. Повторный запуск процесса при проявлении странного поведения может повлиять на то самое поведение, которое вас интересует, а изменение уровня логирования запущенного процесса позволяет подробно изучить его поведение без перезапуска.

Регистрируйте и проверяйте все параметры конфигурации

Регистрируйте все (несекретные) параметры конфигурации сразу после запуска, чтобы отразить то, что видит приложение. Разработчики и операторы иногда неправильно понимают, где должен быть размещен файл конфигурации и как происходит объединение нескольких таких файлов. Логирование параметров конфигурации позволяет пользователям выяснить, видит ли приложение ожидаемую конфигурацию.

Всегда проверяйте загружаемые параметры конфигурации. Проводите проверку только один раз и как можно раньше (сразу после загрузки конфигурации). Убедитесь в том, что для параметров заданы правильные типы: например, для номера порта задано целое число. Удостоверьтесь также, что значения имеют логический смысл: проверьте границы, длину строк, допустимые значения перечисления и т. д. Скажем, значение `-200` представляет собой целое число, но не является допустимым номером порта. Чтобы гарантировать приемлемость значений параметров, воспользуйтесь преимуществами систем конфигурации, обладающих надежными системами типов.

Предусмотрите значения по умолчанию

Если пользователю придется настраивать большое количество параметров, это усложнит процесс эксплуатации вашей системы. Предусмотрите разумные умолчания, чтобы пользователи смогли без труда запустить приложение. Используйте сетевые порты с номерами выше 1024 по умолчанию (порты с номерами, меньшими 1024, являются привилегированными), если порты не сконфигурированы. Используйте временный каталог системы или домашний каталог пользователя, если пути к каталогам не указаны.

Группируйте связанные параметры конфигурации

Конфигурация приложения может чрезмерно усложниться, особенно если речь идет о форматах «ключ — значение», которые не поддерживают вложенную конфигурацию. Используйте стандартный формат, например YAML, допускающий вложение. Группировка связанных свойств упрощает организацию и обслуживание конфигурации.

Объединяйте тесно связанные между собой параметры, такие как длительность тайм-аута и единица измерения, в единую структуру, чтобы их связь была очевидной, и приучите оператора объявлять значения атомарно. Вместо того чтобы определять `timeout_duration=10` и `timeout_units=second`, используйте `timeout=10s` или `timeout: { duration: 10, units = second }`.

Относитесь к конфигурации как к коду

Принцип восприятия *конфигурации как кода* (CAC, configuration as code) гласит, что к конфигурации следует относиться с такой же строгостью, с какой вы относитесь к коду. Ошибки конфигурации могут иметь катастрофические последствия. Одно некорректное целое число или отсутствующий параметр могут нарушить работу приложения.

Чтобы сделать изменение конфигурации безопасным, необходимо контролировать ее версии, проводить проверку, тестирование, сборку и публикацию. Храните конфигурацию в системе управления версиями вроде Git, чтобы отслеживать историю ее изменений. Проверяйте изменения конфигурации так же, как вы это делаете в случае изменения кода. Убедитесь в том, что конфигурация использует правильный формат и соответствует ожидаемым типам и границам значений. Создавайте и публикуйте пакеты конфигурации. Подробнее о доставке конфигурации мы поговорим в главе 8.

Поддерживайте чистоту файлов конфигурации

Чистый файл конфигурации проще понять и изменить: удаляйте неиспользуемую конфигурацию, применяйте стандартное форматирование и интервалы. Не копируйте конфигурацию из других файлов вслепую — это пример *карго-культа*, который предполагает копирование какой-либо вещи без понимания ее сути или устройства. Частые итерации усложняют поддержание аккуратного вида конфигурации, а неправильная конфигурация способна привести к остановке производственного процесса.

Не редактируйте развернутую конфигурацию

Избегайте ручного редактирования конфигурации на конкретном компьютере. Разовые изменения конфигурации перезаписываются при последующих развертываниях, и тогда становится неясно, кто внес изменения, а компьютеры с аналогичной конфигурацией начинают в конечном итоге различаться.

Как и в случае с поддержанием чистоты файлов конфигурации, устоять перед соблазном отредактировать файл конфигурации вручную в рабочей среде бывает очень сложно, однако иногда избежать этого просто невозможно. Если вы редактируете конфигурацию вручную при решении производственной проблемы, позаботьтесь о дальнейшем сохранении внесенных изменений в системе управления версиями.

Инструменты

Работоспособные системы поставляются вместе с инструментами, которые помогают операторам их эксплуатировать. Операторам может потребоваться выполнить массовую загрузку данных, произвести восстановление, сбросить состояние базы данных, инициировать выбор лидера или перенести назначение разделов с одной машины на другую. Системы должны быть обеспечены инструментами, помогающими операторам решать стандартные задачи.

Написание инструментов требует совместных усилий. В некоторых случаях от вас могут ожидать написания и предоставления операционных инструментов. Организации с сильными командами SRE-инженеров также могут создавать инструменты для ваших систем. Как бы то ни было, вам следует взаимодействовать со своей операционной группой, чтобы понимать ее потребности.

SRE-инженеры обычно предпочитают инструменты на основе интерфейса командной строки и API с самоописанием, так как они облегчают процесс написания сценариев. Работу инструментов со сценариями легко автоматизировать. Если вы планируете создавать инструменты на основе пользовательского интерфейса, абстрагируйте логику в общую библиотеку или службу, способную использовать инструменты на основе интерфейса командной строки. И относитесь к инструментам в своей системе как к любому другому коду: следуйте стандартам чистого кодирования и проводите тщательное тестирование.

Вполне вероятно, что у вашей компании уже есть набор инструментов. Например, зачастую для веб-разработки используется стандартный фреймворк: интегрируйте свои инструменты в стандартные доступные вам фреймворки. Возможно, имеется унифицированная консоль управления — в таком случае предполагается, что все инструменты будут интегрированы в эту консоль. Если у компании есть инструменты на основе интерфейса командной строки, спросите, стоит ли интегрировать в них ваши инструменты. Сотрудники привыкают к существующим интерфейсам, поэтому интеграция может упростить работу с вашими инструментами.

КАК КОМПАНИЯ AMAZON ОБРУШИЛА ИНТЕРНЕТ

28 февраля 2017 года Крис находился в офисе в комнате для совещаний, когда заметил, что программа для планирования конференций Zoom перестала работать. Не придав этому особого значения, спустя несколько минут он снова оказался за своим столом и заметил странности в поведении нескольких крупных сайтов. В этот момент ему сообщили, что у Amazon Web Services (AWS) возникли проблемы с хранилищем S3. От сервиса Amazon зависит работа многих крупных веб-сайтов, а Amazon сильно зависит от S3. Возникшая проблема затронула практически весь интернет. Сервис Twitter наполнился сообщениями вроде «Похоже, нелетная погода» и «Пора домой».

В конце концов компания Amazon опубликовала уведомление с описанием случившегося, после того как операционная группа исследовала подсистему биллинга. Оказалось, что инженер ввел команду для удаления небольшого количества машин из соответствующего пула в хранилище S3, однако при указании количества узлов допустил опечатку. В результате из пула узлов было удалено намного больше машин, чем предполагалось, что вызвало перезапуск нескольких других критически значимых подсистем. Это привело к многочасовому отключению сервиса Amazon, от чего пострадали многие другие ведущие компании.

В уведомлении Amazon был краткий, но емкий комментарий: «В связи с произошедшим событием мы вносим несколько изменений. Несмотря на то что сокращение емкости является ключевой операционной практикой, в данном случае использовался инструмент, который допускал слишком резкое сокращение. Мы изменили этот инструмент для замедления данного процесса. Мы также приняли меры безопасности, предотвращающие удаление ресурсов, способное понизить емкость любой подсистемы ниже минимально необходимого уровня. Это позволит не допустить наступления аналогичного события в будущем при вводе ошибочных данных. Кроме того, мы проверяем другие операционные инструменты с целью принятия аналогичных мер безопасности».

Что следует и чего не следует делать

Следует	Не следует
Устранять ошибки на этапе компиляции, а не во время выполнения	Использовать исключения для управления логикой приложения
По возможности использовать неизменяемые объекты	Использовать возвращаемые значения для обработки исключений
Проверять входные и выходные данные	Перехватывать исключения, которые вы не можете обработать
Ознакомиться с отчетом проекта OWASP	Создавать многострочные журналы
Использовать инструменты поиска ошибок, средства проверки типа или подсказки типа	Регистрировать в журналах секретные или конфиденциальные данные
Очищать ресурсы после обработки исключений (особенно сокеты, указатели на файлы и память)	Редактировать вручную конфигурацию на компьютере
Оснащать код метриками	Хранить пароли и секретные данные в файлах конфигурации
Создавать свое приложение настраиваемым	Создавать собственные форматы конфигурации
Проверять и логировать все параметры конфигурации	Использовать динамическую конфигурацию, если этого можно избежать

Повышение уровня

Теме написания работоспособного кода посвящено не так много книг, однако во многих изданиях по программной инженерии можно найти отдельные главы. Например, глава 8 книги *Code Complete* (Microsoft Press, 2004)[1] Стива Макконнелла (Steve McConnell) посвящена защитному программированию. В главах 7 и 8 книги *Clean Code* (Pearson, 2008)[2] рассказывается об обработке ошибок и границах. С этого вполне можно начать.

[1] *Макконнелл С.* Совершенный код.

[2] *Макконнелл С.* Чистый код. — СПб.: Питер, 2021.

В Сети опубликовано множество статей о защитном программировании, исключениях, логировании, конфигурации и инструментах. Особенно полезным ресурсом является библиотека *Amazon Builders' Library* (https://aws.amazon.com/builders-library/).

Книга *Building Secure & Reliable Systems* (O'Reilly Media, 2020), написанная группой SRE-инженеров компании Google, — кладезь полезных советов, особенно с точки зрения обеспечения безопасности. *Site Reliability Engineering* (O'Reilly Media, 2016)[1] — каноническая книга, в которой освещаются вопросы, связанные с обеспечением надежности сайтов. Она в меньшей степени посвящена теме *написания* работоспособного кода, но с ней все же стоит ознакомиться, чтобы получить представление о сложном мире промышленной эксплуатации программного обеспечения. Обе книги доступны в интернете бесплатно, но их также можно приобрести в печатном виде.

[1] *Бейер Б., Джоунс К., Петофф Дж., Мерфи Р.* Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2023.

5 Управление зависимостями

В марте 2016 года тысячи JavaScript-проектов перестали компилироваться из-за пропажи всего одного пакета `left-pad`. Он представлял собой библиотеку с единственным методом, который просто дополнял строки слева до определенной ширины. От этого пакета зависело несколько фундаментальных библиотек на JavaScript, а от тех, в свою очередь, зависело множество проектов. Из-за большой популярности транзитивных зависимостей тысячи кодовых баз с открытым исходным кодом и коммерческих баз критически зависели от одной довольно тривиальной библиотеки. Когда `left-pad` был удален из NPM (менеджера пакетов JavaScript), программисты столкнулись с большими проблемами.

Добавление зависимости в существующий код кажется легким решением. *Don't repeat yourself* («Не повторяйся»), или DRY, — общепринятый принцип в программировании. Почему всем нам нужно писать собственный пакет `left-pad`? Драйверы базы данных, фреймворки приложений или пакеты машинного обучения — существует множество примеров библиотек, которые нет нужды писать с нуля. Однако при использовании зависимостей вы рискуете столкнуться с такими неприятностями, как конфликтующие изменения, циклические зависимости, конфликты версий или отсутствие контроля. При работе следует учитывать риски и думать о том, как снизить вероятность возникновения упомянутых ситуаций.

В этой главе мы рассмотрим основы управления зависимостями, а также обсудим главный кошмар каждого инженера-программиста — ад зависимостей.

Основы управления зависимостями

Прежде чем начать говорить о проблемах и рекомендованных методах, познакомимся с общими концепциями зависимостей и управления версиями.

Зависимость — это некий код, на который опирается ваш код. Момент, когда зависимость необходима, например во время компилирования, тестирования или выполнения, называется *областью действия зависимости*.

Зависимости объявляются в файлах управления пакетами или файлах сборки: на Java это фреймворки Gradle или Maven, на Python — `setup.py` или `requirements.txt`, на JavaScript — NPM `package.json`. Вот фрагмент файла `build.gradle` из проекта на Java:

```
dependencies {
    compile 'org.apache.httpcomponents:httpclient:4.3.6'
    compile 'org.slf4j:slf4j-api:1.7.2'
}
```

Проект зависит от клиентской библиотеки HTTP версии 4.3.6, а также библиотеки интерфейса прикладного программирования (API) SLF4J версии 1.7.2. Каждая зависимость объявляется вместе с областью действия `compile`, следовательно, зависимости необходимы для компиляции кода. Для каждого пакета установлена определенная версия: для `httpclient` — версия 4.3.6, для `slf4j` — версия 1.7.2. Пакеты с версиями используются для контроля зависимостей и решения проблем при появлении разных версий одного и того же пакета (подробнее данную тему мы рассмотрим позже).

Хорошая схема управления версиями содержит версии с нижеперечисленными качествами.

- **Уникальность.** Версии не должны использоваться повторно. Артефакты распространяются, кэшируются и извлекаются только с по-

мощью автоматизированных процессов. Никогда не публикуйте измененный код в существующей версии.

- **Совместимость.** Версии должны помогать пользователям и инструментам понимать приоритет версий. *Приоритет* используется для разрешения конфликтов, когда сборка зависит от нескольких версий одного и того же артефакта.
- **Информативность.** В версиях различаются предварительный и выпущенный (релиз) коды, связываются номера сборки с артефактами, а также устанавливаются устойчивость и совместимость ожиданий.

Git-хеши или коммерческие версии, например «десертные» версии на Android (Android Cupcake, Android Froyo) и версии с животными на Ubuntu (Trusty Tahr, Disco Dingo), уникальны, но не соответствуют качествам совместимости и информативности. То же самое касается и монотонно возрастающего номера версии (1, 2, 3): название будет уникальным и совместимым, но неинформативным.

Семантическое управление версиями

Для пакетов в предыдущем примере используется схема управления версиями, которая называется *семантическим управлением версиями* (SemVer). Это одна из распространенных схем управления версиями: ее официальная спецификация доступна по ссылке https://semver.org/. Спецификация определяет три номера версий: MAJOR, MINOR и PATCH (патч-версию иногда называют *микроверсией*), или МАЖОРНАЯ, МИНОРНАЯ и ПАТЧ. Номера версий объединены в одну версию MAJOR.MINOR.PATCH. У библиотеки `Httpclient` версии 4.3.6 значения мажорной, минорной и патч-версий указаны цифрами 4, 3 и 6 соответственно.

Семантические версии являются уникальными, совместимыми и информативными. Каждый номер версии используется только однажды, и его можно сопоставить путем просмотра слева направо (версия 1.13.7 выпущена раньше версии 2.14.1). В них есть информация о совместимости разных версий и версии-кандидате или о номере сборки.

Мажорная версия 0, которая считается предварительной, используется для быстрых итераций; о гарантиях совместимости речь не идет.

Разработчики могут изменять API и нарушать старый код, добавляя новый обязательный параметр или удаляя публичный метод. Начиная с мажорной версии 1 ожидается, что проект будет гарантировать следующее:

- увеличение патч-версии для исправления обратно совместимых ошибок;
- увеличение минорной версии для добавления функций, не нарушающих обратную совместимость;
- увеличение мажорной версии для обратно несовместимых изменений.

SemVer также выделяет предварительную версию с помощью буквы «a», добавляя ее после версии исправления, или патч-версии. Последовательности цифр и букв, разделенные точками, используются в именах предварительных версий (2.13.7-alpha.2). В предварительные версии могут вноситься изменения, которые не отразятся на мажорной версии. Во многих проектах используются сборки версий-кандидатов (release candidate, RC). Первые пользователи версии-кандидата могут найти в ней ошибки до официального выпуска. Предварительным версиям дают составные имена, например 3.0.0-rc.1. Финальная версия-кандидат становится версией для выпуска путем повторного релиза версии без суффикса rc. Все предварительные версии заменяются окончательной (в нашем примере это 3.0.0). Дополнительную информацию об алгоритме управления выпусками вы можете получить в главе 8.

Номера версий сборки добавляются после метаданных основной и предварительной версий: 2.13.7-alpha.2+1942. Номер сборки в имени версии помогает разработчикам и инструментам находить журналы сборки для любой из скомпилированных версий.

Схема SemVer позволяет использовать *подстановочные знаки* диапазона версий (2.13.*). Поскольку SemVer гарантирует совмещение минорной и патч-версий, то сборки должны сохранять работоспособность даже в том случае, если исправленные и обновленные версии добавляются в сборку автоматически.

Транзитивные зависимости

Файлы сборки или менеджеры пакетов показывают прямые зависимости проекта. Но прямые зависимости — лишь часть того, что используется в системах сборок или пакетов. Обычно это зависимости от библиотек, и тогда их называют *транзитивными зависимостями*. Отчет о зависимостях демонстрирует полностью разрешенное *дерево зависимостей*, или *граф зависимостей*. Большинство систем сборки и управления пакетами могут создавать отчеты о зависимостях. Продолжая предыдущий пример, рассмотрим систему сборки `dependencies output`:

```
compile - Compile classpath for source set 'main'.
+--- org.apache.httpcomponents:httpclient:4.3.6
|    +--- org.apache.httpcomponents:httpcore:4.3.3
|    +--- commons-logging:commons-logging:1.1.3
|    \--- commons-codec:commons-codec:1.6
\--- org.slf4j:slf4j-api:1.7.2
```

Дерево зависимостей показывает зависимости, используемые в сборке для компиляции проекта. Отчет состоит из нескольких слоев, в которых указываются зависимости зависимостей, и т. д. Библиотека `httpclient` использует три транзитивные зависимости: `httpcore`, `commons-logging` и `commons-codec`. Напрямую проект не зависит от этих библиотек, но связывается с ними через `httpclient`.

Понимание транзитивных зависимостей является важной частью управления зависимостями. Добавление зависимости может показаться незначительным изменением, но если у библиотеки 100 зависимостей, то теперь ваш код будет зависеть от 101 библиотеки. Любое изменение зависимости может повлиять на ваш код. Убедитесь в том, что вы понимаете, как можно получить дерево зависимостей для отслеживания и решения конфликтов зависимостей.

Ад зависимостей

Если вы спросите любого программиста об аде зависимостей, то услышите множество печальных историй. Конфликтующие версии одной библиотеки или несовместимое обновление библиотеки могут нарушить

сборку или вызвать ошибку при выполнении. Наиболее частыми причинами ада зависимостей становятся циклические зависимости, «алмазные» зависимости и конфликты версий.

Предыдущий отчет о зависимостях был простым. Более реалистичный отчет укажет на конфликты версий и покажет вам, что на самом деле представляет собой ад зависимостей:

```
compile - Compile classpath for source set 'main'.
+--- com.google.code.findbugs:annotations:3.0.1
|    +--- net.jcip:jcip-annotations:1.0
|    \--- com.google.code.findbugs:jsr305:3.0.1
+--- org.apache.zookeeper:zookeeper:3.4.10
|    +--- org.slf4j:slf4j-api:1.6.1 -> 1.7.21
|    +--- org.slf4j:slf4j-log4j12:1.6.1
|    +--- log4j:log4j:1.2.16
|    +--- jline:jline:0.9.94
|    \--- io.netty:netty:3.10.5.Final
\--- com.mycompany.util:util:1.4.2
     \--- org.slf4j:slf4j-api:1.7.21
```

Дерево показывает три прямые зависимости: `annotations`, `zookeeper` и `util`. Все библиотеки зависят от других библиотек — это транзитивные зависимости. В отчете появляются две версии `slf4j-api`. Библиотека `util` зависит от `slf4j-api` версии 1.7.21, однако `zookeeper` зависит от `slf4j-api` версии 1.6.1.

Образуется «алмазная» зависимость (рис. 5.1).

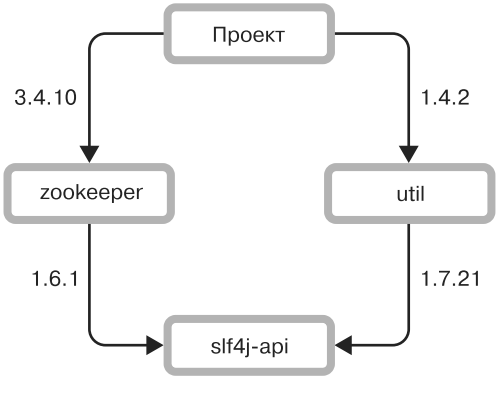

Рис. 5.1. «Алмазная» зависимость

Проект не может одновременно использовать две версии одной библиотеки, так что системе сборки нужно выбрать одну из них. В отчете о зависимостях Gradle варианты версий показываются с такими аннотациями:

```
|    +--- org.slf4j:slf4j-api:1.6.1 -> 1.7.21
```

Примечание `1.6.1 -> 1.7.21` означает, что `slf4j-api` был обновлен до версии 1.7.21 во всем проекте, чтобы избежать конфликта версий. Zookeeper может некорректно работать с другими версиями `slf4j-api`, особенно если связанная с ним зависимость `slf4j-log12` не будет обновлена. Обновление *должно* работать корректно, так как номер версии зависимости мажорной версии Zookeper остается прежним (SemVer гарантирует обратную совместимость в рамках одной мажорной версии). На самом деле совместимость очень важна. В проектах номера версии очень часто ставятся без проверки совместимости, а автоматизация не всегда гарантирует полную совместимость изменений. Несовместимые изменения встраиваются в минорные версии или патч-версии, нанося вред кодовой базе.

Более неприятными являются *циклические*, или *круговые*, *зависимости*, когда библиотека транзитивно зависит сама от себя (A зависит от B, при этом B зависит от C, которая зависит от A, что показано на рис. 5.2).

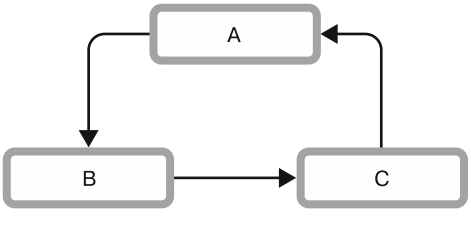

Рис. 5.2. Циклическая зависимость

Циклические зависимости служат примером проблемы, где трудно отличить причину от следствия (проблема курицы и яйца): при обновлении одной библиотеки возникает ошибка в работе другой библиотеки. Вспомогательные и служебные проекты обычно появляются в циклических зависимостях. К примеру, библиотека обработки естественного языка (или NLP) зависит от служебной библиотеки для получения доступа к функции парсинга строк; другой разработчик случайно добавляет библиотеку NLP в качестве зависимости служебной библиотеки для получения доступа к ее методу стемминга.

ЗАГАДОЧНАЯ ИСТОРИЯ С GOOGLE COLLECTIONS

Все библиотеки Java заархивированы в файлах JAR. Во время выполнения Java будет искать классы во всех JAR-файлах в своем параметре classpath. Такой подход будет отлично работать до тех пор, пока не появятся несколько архивов JAR с разными версиями одного класса.

В LinkedIn был инструмент под названием Azkaban — обработчик рабочего процесса. Он позволял разработчикам загружать пакеты кода и планировать их выполнение в программе Hadoop. Azkaban был написан на Java и не изолировал пути к классам, а это значит, что загруженный код работал не только с собственными зависимостями, но и с зависимостями Azkaban. В какой-то момент при работе стала появляться ошибка вызова несуществующих методов класса — `NoSuchMethodErrors`. Как ни странно, команда Криса видела якобы отсутствующие методы в загруженных пакетах кода. Однако в данном случае существовала определенная закономерность: все отсутствующие методы были из популярной библиотеки Guava от Google.

Guava имеет ряд полезных функциональных возможностей, а также упрощает использование весьма неудачных библиотек коллекций на Java. Команда считала, что библиотеки Azkaban и загруженные пакеты кодов могут конфликтовать друг с другом. Однако все оказалось еще сложнее: Azkaban не использовал Guava. Потом разработчики поняли, что библиотека Guava изначально произошла от другой библиотеки, которую использовал Azkaban, — библиотеки `google-collections`. Azkaban создал эту библиотеку для двух классов: `ImmutableMap` и `ImmutableList`. Компилятор Java находил в `google-collections` ссылки на классы быстрее, чем Guava, а затем пытался вызвать методы, которых не было в ранней версии библиотеки.

В конце концов команда изолировала пути к классам, и впоследствии Azkaban перестал добавлять файлы JAR в среду выполнения кода. В какой-то степени это решало основную проблему, однако иногда ошибки все равно появлялись. Затем команда обнаружила, что пакеты, которые часто выдавали ошибку, содержали зависимости как `google-collections`, так и Guava. Система сборки не могла определить, что `google-collections` представляла собой устаревшую версию Guava, так что две библиотеки в системе сборки вызывали ту же проблему, что и зависимости Azkaban. Потребовалось провести внушительный объем работ по рефакторингу кода, отвлекая разработчиков от их основной деятельности. Стоят ли несколько вспомогательных методов коллекций этих усилий?

Избегание ада зависимостей

Когда-нибудь вы точно столкнетесь с адом зависимостей. Вы не можете не использовать зависимости, однако каждая зависимость влияет на весь проект. Спросите себя, а не преувеличиваете ли вы ценность зависимости и стоит ли ее добавлять в код?

- Вам точно нужна функциональность?
- Как хорошо поддерживается зависимость?
- Если что-то пойдет не так, вы сможете исправить зависимость? Вызовет ли у вас это какие-либо трудности?
- Вам точно нужно добавлять в код зависимость?
- Как часто из-за зависимости возникают обратно несовместимые ошибки?
- Как хорошо вы и ваша команда понимаете зависимость?
- Легко ли написать код в одиночку?
- Как именно лицензируется ваш код?
- Каково соотношение используемого и неиспользуемого кода в зависимости?

Когда вы захотите добавить зависимости в свой код, обратите внимание на следующие рекомендации.

Изолирование зависимостей

Вам не нужно отказываться от управления зависимостями, чтобы создавать и упаковывать системы. Вы можете копировать, выполнять вендоринг или затенять зависимый код. Копирование кода в проект приводит к увеличению изоляции (устойчивости) взамен на автоматизацию управления зависимостями. Вы можете сами выбирать, какой код хотите использовать, однако вам придется управлять скопированным кодом.

Многие разработчики придерживаются принципа DRY, который не позволяет коду дублироваться. Будьте практичны: если копирование

кода помогает избежать большой или неустойчивой зависимости, копируйте его (лицензия на программное обеспечение позволяет это делать).

Лучше всего копирование кода работает с небольшими и устойчивыми фрагментами. У копирования вручную целых библиотек есть свои недостатки: в процессе может потеряться история версий, и в таком случае при каждом обновлении кода нужно будет опять его копировать. Код после *вендоринга* использует особые инструменты для управления историей и обновлениями при встраивании библиотек в базу кода. В папках vendor находятся полноценные копии библиотек. Такие инструменты, как `git-subtree` и `git-vendor`, помогают управлять папками vendor в кодовой базе. Некоторые упаковочные системы, например Go, имеют встроенную поддержку таких папок.

Затенение зависимостей также позволяет изолировать некоторые зависимости. Затенение автоматически перемещает зависимость в другое пространство имен. Это делается для того, чтобы избежать конфликтов: например, `some.package.space` становится `shaded.some.package.space`. Таким способом очень легко предотвратить зависимость библиотек от приложений. Сначала затенение использовалось в экосистеме Java, однако спустя время оно приобрело широкое применение. Другие языки программирования, например Rust, используют при работе аналогичные концепции.

Затенение — сложная техника, которую следует использовать с особой осторожностью. Никогда не представляйте объекты затененной зависимости в общедоступных API, поскольку разработчикам придется создавать объекты в пространстве затененного пакета (`shaded.some.package.space.Class`). Затенение используется для скрытия зависимости. Пользователям библиотеки очень сложно, а иногда и вовсе невозможно создать затененный объект. Помните, что затененные зависимости могут привести разработчиков в замешательство, так как обычно в выходных файлах сборки имена пакетов отличаются. Мы рекомендуем применять затенение зависимостей только при создании библиотек с широко используемыми зависимостями, из-за которых могут возникать конфликты.

Намеренное добавление зависимостей

Все используемые библиотеки объявите явно как зависимости. Не используйте методы и классы из транзитивных зависимостей, даже если при работе не возникает ошибок и конфликтов. Библиотеки могут менять свои зависимости даже при изменении версии на уровне патча. Если во время обновления будет удалена транзитивная зависимость, присутствующая в вашем коде, выскочит ошибка.

Проект, зависящий только от библиотеки `httpclient` (см. предыдущий пример), не должен явно использовать классы в `httpcore`, `commons-logging` и `commons-codec` (зависимости `httpclient`). В противном случае необходимо объявить прямую зависимость от библиотеки.

При управлении зависимостями не полагайтесь только на интегрированную среду разработки. Всегда явно объявляйте используемые зависимости в файлах сборки. Интегрированная среда разработки часто сохраняет зависимости в собственных конфигурациях проекта, никак не связанных с механизмом сборки. Несоответствия между интегрированной средой разработки и файлами сборки позволят вашему коду работать в интегрированной среде разработки, но не в актуальной сборке кода и наоборот.

Закрепление версии

Явно указывайте номер версии каждой зависимости: такая практика называется *закреплением версий*. Номера незакрепленным версиям будут автоматически заданы системой сборки или системой управления пакетами, однако доверять все решения системе сборки — не лучшая идея. Устойчивость кода будет нарушаться при изменении версий зависимостей во время последовательных сборок.

В следующем примере объявляется список зависимостей библиотеки Go с закрепленными версиями:

```
require (
    github.com/bgentry/speakeasy v0.1.0
    github.com/cockroachdb/datadriven v0.0.0-20190809214429-80d97fb3cbaa
    github.com/coreos/go-semver v0.2.0
```

```
    github.com/coreos/go-systemd v0.0.0-20180511133405-39ca1b05acc7
    github.com/coreos/pkg v0.0.0-20160727233714-3ac0863d7acf
    ...
)
```

Теперь сравним предыдущий пример с фрагментом зависимостей Apache Airflow, в котором используются три способа управления версиями:

```
flask_oauth = [
    'Flask-OAuthlib>=0.9.1',
    'oauthlib!=2.0.3,!=2.0.4,!=2.0.5,<3.0.0,>=1.1.2',
    'requests-oauthlib==1.1.0'
]
```

Библиотека `requests-oauthlib` явно закреплена за 1.1.0. Зависимость `FlaskOAuthlib` закреплена за версией 0.9.1 или любой версией выше. Библиотека `oauthlib` особенная: она работает с версиями 1.1.2 и выше, но не работает с версиями от 3.0.0, а также версиями 2.0.3, 2.0.4 и 2.0.5. Версии от 2.0.3 до 2.0.5 не используются из-за известных ошибок или несовместимости версий.

Ограничение допустимого диапазона версий — это некий компромисс между закреплением версий и разрешением неограниченного диапазона версий. Система установления зависимостей может решать конфликты и обновлять зависимости, однако возможность совершать кардинальные изменения ограниченна. При этом любая незакрепленная версия будет включать в себя не только последние исправления ошибок, но и недавно допущенные ошибки, логику работы и даже несовместимые изменения.

Даже если вы закрепите все прямые зависимости, в транзитивных зависимостях могут быть подстановочные символы. Версии транзитивных зависимостей можно закрепить путем создания полного манифеста всех разрешенных зависимостей и их версий. Манифесты зависимостей имеют множество имен: это фиксация требований в Python, создание файла `Gemfile.locks` в Ruby и `Cargo.locks` в Rust. Системы сборки используют манифесты для получения одинаковых результатов при каждом выполнении. Если разработчики хотят изменить версию, то они повторно создают манифесты. Выполнение коммитов манифестов вместе с кодом позволяет и отслеживать изменения любой зависимости, и предотвращать ошибки.

ПОЧЕМУ AIRFLOW FLASK_OAUTH ТАКОЙ ЗАПУТАННЫЙ?

Замысловатая зависимость в предыдущем фрагменте кода пытается исправить проблему зависимости, унаследованную Airflow от библиотеки `Flask-OAuthlib`. Библиотека `Flask-OAuthlib` обладает собственными зависимостями от `oauthlib` и `requests-oauthlib`, из-за чего и появляются ошибки. Разработчики `Flask-OAuthlib` разрешили ограниченный диапазон для `oauthlib` и `requests-oauthlib` в попытке решить проблему, однако им потребовалось время для того, чтобы выпустить исправленную версию. Но Airflow уже сломался, и дожидаться релиза `Flask-OAuthlib` не было возможности. В качестве временного исправления в инструмент Airflow скопировали блок зависимостей библиотеки `Flask-OAuthlib`, сопроводив действия комментарием: «Мы открепим их только тогда, когда выйдет новая версия `Flask-OAuthlib`». Спустя 18 месяцев изменение все еще никто не отменил. Это хитрость, к которой приходится прибегать при исправлении проблем, связанных с зависимостями.

Сужение области действия зависимостей

Область действия зависимостей, о которой говорилось ранее, определяет, когда в цикле сборки используется зависимость. В области действия существует своя иерархия: зависимости времени компиляции используются во время выполнения кода, но зависимости выполнения кода — только при выполнении кода и не используются во время компиляции кода. Зависимости тестирования используются только при тестировании, их применение не обязательно при работе уже опубликованного кода.

Для каждой зависимости используйте как можно более узкую область действия. Вы можете объявлять все зависимости с областью действия во время компиляции кода, однако мы не советуем так делать. Узкая область действия помогает избежать ошибок и конфликтов, а также уменьшает размер двоичных файлов, используемых во время выполнения.

Защитите себя от циклических зависимостей

Никогда не используйте циклические зависимости. Они приводят к странному поведению системы сборки и проблемам с порядком развертывания. Системы сборки будут сначала работать, но затем начнут выдавать ошибки. Ошибки появятся и в приложениях.

Защитите себя от циклических зависимостей с помощью инструментов сборки. Многие системы сборки имеют средства обнаружения циклических зависимостей, которые в случае нахождения такой зависимости вас предупредят. Если система сборки не предупреждает о циклических зависимостях, можно воспользоваться специальными плагинами.

Что следует и чего не следует делать

Следует	Не следует
Использовать семантическое управление версиями	Использовать хеши коммитов в качестве номеров версий
Использовать закрепление диапазонов версий зависимости	Добавлять зависимость, если ее польза не превышает затраченных усилий
Использовать генератор отчетов о транзитивных зависимостях	Использовать транзитивные зависимости непосредственным образом
Скептически относиться к добавлению новых зависимостей	Вводить циклические зависимости
Определять область действия ваших зависимостей	

Повышение уровня

С проблемой конфликтов зависимостей или несовместимыми изменениями сталкиваются многие пользователи. Все эти проблемы можно описать двумя словами — *ад зависимостей* (у многих систем есть свои варианты ада зависимостей — ад DDL, ад JAR, «Каждый раз, когда мне требуется использовать систему pip»). Несмотря на то что управлять зависимостями чрезвычайно сложно, книг по этой теме написано очень мало — в интернете вы можете найти объяснения и обсуждения только

отдельных систем. Для общего понимания взгляните на статью в Википедии об аде зависимостей и пройдите по предложенным там ссылкам.

Для получения краткой и понятной спецификации семантического управления версиями см. https://semver.org/. Для Python есть похожая схема — см. https://www.python.org/dev/peps/pep-0440/. Обе системы управления версиями широко используются и достойны того, чтобы потратить на их изучение некоторое время. Существует и множество других систем управления версиями. Кроме того, вы можете столкнуться с артефактами, которые используют разные системы управления версиями в одном проекте. В соответствии с законом Парето, если вы только начинаете работать, мы не рекомендуем вам углубляться в семантическую сторону версии (конечно, за исключением тех случаев, когда это является основной частью вашей работы или вам требуется дополнительная информация для решения проблемы). Информация, предложенная в данной главе, позволяет выполнять большую часть повседневных дел.

Описанные концепции управления версиями можно применять как к библиотекам, так и к сервисным API. О версиях API подробнее будет рассказано в главе 11.

6 Тестирование

Написание, выполнение и исправление тестов могут показаться достаточно рутинной работой, однако ее важность невозможно преуменьшить. Из-за плохих тестов разработчики тратят дополнительные ресурсы, поскольку такие тесты не только не приносят никакой пользы, но и могут негативно повлиять на стабильность всего набора тестов. В этой главе вы научитесь проводить эффективное тестирование. Мы рассмотрим, для чего тесты используются и какими они бывают, поговорим о существующих инструментах тестирования, ответственном тестировании и о том, как бороться с недетерминизмом.

Широкое применение тестов

Большинство разработчиков знают о главной функции тестов — проверять, работает код или нет. Но тесты служат и другим целям: они защищают код от будущих изменений, которые непреднамеренно могут повлиять на его поведение; заставляют разработчиков использовать собственные API; документируют то, как должны взаимодействовать между собой различные компоненты кода; служат площадкой для экспериментов.

В основном тесты подтверждают то, что код или программное обеспечение работает правильно. Непредсказуемое поведение вызывает у пользователей и разработчиков множество проблем. Изначально тесты подтверждают то, что в работе кода нет никаких ошибок. Затем запускаются тесты, защищающие код от внедрения новых изменений. Когда тестирование проходит неудачно, приходится гадать, намеревался ли разработчик изменить поведение кода или это была ошибка.

Написание тестов заставляет разработчиков думать об интерфейсе и реализации собственной программы. Как правило, сначала программисты работают со своим кодом в тесте. Новый код обычно имеет шероховатости, а при тестировании можно выявить, например, неудобный дизайн интерфейса и исправить его на ранних стадиях. Тестирование также демонстрирует запутанность разработки: *спагетти-код* или код с большим количеством зависимостей очень сложно тестировать. Написание тестов вынуждает разработчиков создавать продуманный и простой код, а также улучшать разделение обязанностей и уменьшать сильную связанность.

Побочные эффекты чистого кода при тестировании настолько сильны, что применение *разработки через тестирование* (TDD) стало распространенным явлением. TDD — это практика написания тестов перед написанием кода. Сначала пишется тест, а затем код, способный выполнить тестирование. TDD заставляет разработчиков думать о поведении кода и его интерфейсе перед тем, как начинать писать сам код.

Тесты также служат документацией, иллюстрирующей предполагаемую работу с кодом. Документация — это первое, что опытный разработчик начнет читать для того, чтобы понять незнакомую базу кода. Наборы тестов — отличная площадка для экспериментов. Разработчики выполняют тесты с отладчиками, привязанными к поэтапному выполнению кода. При обнаружении ошибок или возникновении вопросов, касающихся поведения кода, могут быть добавлены новые тесты.

Виды тестирований

Существуют десятки различных типов тестов и методик тестирования. Наша цель — не охватить тему целиком, а рассмотреть наиболее распространенные типы: юнит-тесты, интеграционные, системные и приемочные, а также тесты производительности.

Юнит-тесты (модульные) проверяют фрагмент кода, например метод или поведение. Юнит-тесты, как правило, быстрые, небольшие и сфокусированные. Скорость важна по причине того, что чаще всего такие тесты выполняются на компьютерах разработчиков. Компактные тесты, которые проверяют что-то одно, упрощают нахождение и понимание причины ошибки.

Интеграционные тесты проверяют работу нескольких компонентов в группе. Если вы создаете ряд объектов, которые взаимодействуют между собой в тесте, то с большой вероятностью вы пишете интеграционный тест. Обычно такое тестирование выполняется медленно и требует сложной настройки. Разработчики нечасто проводят такое тестирование, поскольку на получение обратной связи нужно много времени. Интеграционные тесты могут устранить проблемы, которые трудно устранить при использовании юнит-тестов.

НЕКОТОРЫЕ РЕШЕНИЯ ОЧЕВИДНЫ ТОЛЬКО В РЕТРОСПЕКТИВЕ

Несколько лет назад Дмитрий решил купить посудомоечную машину. Он читал отзывы, ходил по магазинам, внимательно изучал все характеристики, сравнивал варианты — и наконец определился с моделью. Продавец, который настаивал на том, что Дмитрию нужно походить по залу и посмотреть имеющийся товар, проверил наличие на складе выбранной посудомоечной машины и уже приготовился оформить заказ, но в тот момент, когда его палец завис над клавишей Enter, внезапно остановился.

— Эта посудомоечная машина точно встанет на вашей кухне?

— Да, встанет.

— И у вас есть шкафчик, который открывается под углом в 90 градусов к тому месту, где будет установлена посудомоечная машина? И его дверца достает до дверцы посудомоечной машины?

— Ну да, есть у меня такой шкафчик.

— Ох, понятно, — сказал продавец и убрал руку от клавиатуры. — Вам нужно выбрать другую модель посудомоечной машины.

У модели, выбранной Дмитрием, на дверце была ручка, которая не позволяла бы нормально открывать шкафчик. Идеально работающая посудомоечная машина и идеально открывающийся шкафчик являются несовместимыми вещами. Понятно, что продавец уже сталкивался с подобной проблемой. Решением оказалось приобретение аналогичной модели со скрытой ручкой.

Системные тесты проверяют всю систему. Сквозные, или end-to-end, потоки работ запускаются для имитации реального взаимодействия с пользователем. Подходы к автоматизации системного тестирования различаются. Некоторые компании требуют, чтобы перед выпуском проект проходил системное тестирование — то есть чтобы все компоненты тестировались и выпускались одновременно, синхронно. Другие компании поставляют такие большие системы, что выпуск компонентов никак нельзя синхронизировать: они проводят масштабное интеграционное тестирование и дополняют его проведением непрерывного синтетического мониторинга производственных тестов. Скрипты *синтетического мониторинга* выполняются для имитации регистрации пользователя, поиска, покупки товара и т. д. Синтетический мониторинг требует инструментов, позволяющих биллинговым, бухгалтерским и другим системам отличать эти производственные испытания от реальной работы.

Тесты производительности, такие как нагрузочные и стресс-тесты, измеряют производительность системы при различных настройках. *Нагрузочный тест* измеряет производительность при разной нагрузке: например, во время работы системы при одновременном обращении 10, 100 или 1000 пользователей. *Стресс-тесты* доводят нагрузку до максимума и показывают, что происходит с системой при чрезвычайной нагрузке. Такие тесты полезны для планирования мощности системы и определения целей уровня обслуживания.

Приемочные тесты выполняются клиентом или его доверенным лицом: именно так проверяется соответствие ПО условиям приемки. Данный вид тестирования широко распространен в корпоративной среде, где официальное приемочное тестирование является частью дорогостоящего контракта. *Международная организация по стандартизации (The International Standards Organization, ISO)* в качестве своего стандарта безопасности требует проводить приемочное тестирование для подтверждения явных требований со стороны бизнеса; профессиональные аудиторы запрашивают подтверждение документации как для требований, так и для соответствующих тестов. Менее формальные приемочные тестирования, которые проводятся в организациях с не столь строгими требованиями регуляторов рынка, представляют собой вариации на тему: «Я кое-что изменил в проекте. Скажите, вас все устраивает?»

ТЕСТИРОВАНИЕ НА ПРАКТИКЕ

В этой главе мы рассмотрели тестовые принципы многих успешных проектов с открытым исходным кодом. В каких-то проектах отсутствовали определенные виды тестов, в то время как в других часто встречалось смешанное тестирование, например модульное и интеграционное. Важно понимать, что означают эти виды тестирования и какие различия между ними существуют. Тем не менее не слишком зацикливайтесь на том, чтобы сразу сделать все идеально. В успешных проектах по результатам тестов принимаются реалистичные практичные решения, и вы должны поступать так же. Если видите возможность улучшить тест или набор тестов, воспользуйтесь ею! Не зацикливайтесь на именах или категориях и не вините себя, если делаете что-то неправильно: энтропия ПО — мощная сила (см. главу 3).

Инструменты тестирования

Инструменты тестирования делятся на несколько категорий: инструменты для написания тестов, платформы тестирования, инструменты для контроля качества кода. *Инструменты для написания тестов*, например библиотеки заглушек, помогают писать понятные и эффективные тесты. *Платформы тестирования* позволяют запускать тест и моделировать цикл теста от начала до конца. Платформы тестирования сохраняют результаты тестов, взаимодействуют с системами сборки и оказывают другую поддержку. *Инструменты оценки качества кода* используются для анализа уровня сложности кода, поиска ошибок с помощью статического анализа и нахождения ошибок в архитектуре кода. Такие инструменты анализа обычно являются частью этапа сборки или компиляции.

Каждый инструмент, добавленный в вашу систему, имеет свои особенности, и вы должны их понимать. Инструмент может зависеть от других библиотек, что увеличивает нагрузку и усложняет систему. Некоторые инструменты влияют на скорость выполнения теста. Избегайте использования сторонних инструментов, пока не убедитесь в том, что

сложности, с которыми вам и вашей команде придется столкнуться, приведут к нужному результату, а также в том, что ваша команда согласна на это.

Библиотеки заглушек

Библиотеки заглушек, как правило, используются в юнит-тестах, особенно в объектно-ориентированном коде. Очень часто код зависит от внешних элементов, например от систем, библиотек или объектов. В таком случае все внешние зависимости заменяются *заглушками*, которые имитируют интерфейс, предоставляемый реальной системой. Они реализуют необходимую для тестирования функциональность, выдавая в ответ жестко запрограммированную информацию.

Заглушка внешних зависимостей делает модульные тесты быстрыми и узконаправленными. Заглушка удаленных систем позволяет тестам обходить вызовы сети, упрощает настройку и помогает избегать медленных операций. Методы и объекты заглушки дают возможность разработчикам писать модульные тесты для проверки одного конкретного поведения.

Заглушки предотвращают загромождение кода приложения специфичными для тестов методами, параметрами и переменными. Характерные для тестов изменения сложно поддерживаются, затрудняют чтение кода и вызывают ошибки (не добавляйте в свои методы булев параметр `isTest`!). Заглушки помогают разработчикам получать доступ к защищенным методам и переменным, не нарушая структуру самого кода.

Хотя использование заглушек и полезно, не стоит ими увлекаться. Заглушки со сложной логикой могут сделать ваш тест хрупким и сложным для понимания. Начните с базовых встроенных заглушек в юнит-тестах и не используйте общий класс заглушек до тех пор, пока не потребуется повторять логику заглушек в разных тестах.

Злоупотребление заглушками является признаком плохого кода. Каждый раз, когда вам захочется использовать заглушку, подумайте о том, можно ли реорганизовать код так, чтобы была возможность удалить зависимость от имитируемой системы. Отделение логики вычислений и преобразования данных от кода ввода-вывода помогает упростить процесс тестирования и делает программу менее хрупкой.

Платформы тестирования

Платформы тестирования позволяют писать и выполнять тесты. Вы можете найти платформы, которые помогают координировать и выполнять модульные и интеграционные тесты, а также тесты производительности и даже тесты пользовательского интерфейса. Платформы тестирования выполняют следующее:

- управляют методами теста `setUp()` и `tearDown()`;
- управляют выполнением и оркестрацией теста;
- создают отчет о результате теста;
- предоставляют дополнительные инструменты, например методы с утверждением;
- выполняют интеграцию с инструментами покрытия кода.

Методы `setUp()` и `tearDown()` позволяют программистам указывать такие шаги, как настройка структуры данных или очистка файлов, которые нужно выполнять до или после каждого выполнения теста. Во многих платформах тестирования вы можете встретить несколько вариантов этих методов: все это нужно настраивать перед каждым выполнением теста или группы тестов. Перед применением методов `setUp()` и `tearDown()` прочтите соответствующую документацию и убедитесь в том, что правильно их используете. Не думайте, что методы `tearDown()` будут работать при любых обстоятельствах. Например, метод не сработает в том случае, если при выполнении теста возникнет ошибка, которая приведет к завершению всего процесса тестирования.

Платформы тестирования помогают регулировать скорость и изоляцию тестов с помощью инструментов оркестрации. Тесты могут выполняться как последовательно, так и параллельно. Последовательные тесты выполняются один за другим. Такой способ выполнения считается более безопасным, так как в этом случае тесты с меньшей вероятностью могут повлиять друг на друга. Параллельное выполнение происходит быстрее, однако увеличивает вероятность ошибок.

Платформы тестирования можно настроить так, чтобы они всегда запускали новый процесс выполнения между выполнением тестов. Таким

образом вы изолируете тесты, поскольку каждое тестирование будет начинаться с самого начала. Помните, что запуск новых процессов — операция затратная. В разделе «Детерминизм в тестировании» вы узнаете больше об изолировании тестов.

Отчеты о тестировании помогают программистам проводить отладку неудачных сборок. В отчетах содержится подробная информация о том, какие тесты закончились успешно, а какие выдали при выполнении ошибку или вообще были пропущены. Когда тест выдает ошибки, в отчетах указывается, что именно стало причиной ошибки. В отчетах также содержатся журнал событий и трассировка стека для каждого теста, благодаря чему программисты могут быстро выполнять отладку сбоев. Но будьте внимательны: не всегда можно сразу понять, где именно хранятся результаты теста, так как краткий отчет выводится на консоль, а вся информация записывается на диск. Если не можете найти отчет, рекомендуем проверить папки тестирования и сборки.

Инструменты проверки качества кода

Используйте такие инструменты, как *линтеры*, чтобы сделать ваш код более качественным. Линтеры отвечают за статический анализ и проверку стиля кода. Данный инструмент показывает сложность или покрытие тестов.

Статические анализаторы кода занимаются поиском типичных ошибок, таких как открытые дескрипторы файлов или использование неустановленных переменных. Статические анализаторы особенно важны для динамических языков, например для Python или JavaScript, в которых нет функции, отвечающей за обнаружение синтаксических ошибок. Анализаторы ищут известные явные проблемы в структуре кода и выделяют часть кода, в которой возможны ошибки. Однако статические анализаторы могут ошибаться, поэтому вам нужно относиться с долей сомнения к ошибкам и проблемам, о которых сообщают такие инструменты. С помощью аннотаций кода можно отменять ложные срабатывания анализатора: в таком случае инструмент будет игнорировать определенные ошибки.

Инструменты проверки стиля кода гарантируют, что весь исходный код отформатирован в едином стиле путем проверки максимального количества символов в каждой строке, стилей camelCase или snake_case, правильности отступов и т. д. Единый стиль помогает разным программистам работать над одной кодовой базой. Мы настоятельно рекомендуем настроить интегрированную среду разработки так, чтобы все правила стилей применялись к коду автоматически.

Инструменты определения сложности кода защищают его от слишком сложной логики, вычисляя *цикломатическую сложность* или, условно говоря, количество путей в вашем коде. Чем выше сложность кода, тем труднее будет его протестировать и тем больше ошибок может в нем содержаться. Цикломатическая сложность в программе увеличивается с размером базы кода, так что высокий общий показатель сложности не всегда означает что-то плохое. Однако резкое увеличение сложности должно вызывать обеспокоенность.

Инструменты покрытия кода измеряют количество строк, которое было проверено тестами. Если ваше изменение снижает покрытие кода, нужно писать больше тестов. Убедитесь в том, что тесты проверяют все внесенные в код изменения. Стремитесь к тому, чтобы покрытие составляло 65–85 % всего кода. Не забывайте, что сам по себе параметр покрытия не является показателем качества программы тестирования: данный параметр может вводить разработчика в заблуждение как при высоких, так и при низких показателях. Тестирование автоматически сгенерированного кода, например скаффолдинга или сериализации класса, может выдать результат с обманчиво низким уровнем покрытия. И наоборот, упорное стремление создавать модульные тесты для достижения полного покрытия кода не гарантирует того, что ваш код будет безопасно интегрироваться.

Иногда разработчики зацикливаются на показателе качества кода. Сам факт того, что инструмент указывает на проблемы качества кода, не означает, что проблема есть и ее нужно сразу же исправлять. Будьте прагматичными при работе с кодовыми базами, не прошедшими проверку качества. Не ухудшайте состояние кода, но избегайте масштабных проектов по очистке кода. В качестве руководства используйте раздел «Технический долг» из главы 3 — так вы будете понимать, когда следует решать проблемы, связанные с качеством кода.

Пишем собственные тесты

Вы ответственны за правильную работу созданного вашей командой кода. Напишите несколько собственных тестов и не ждите, что кто-то другой будет выполнять очистку кода за вас. Во многих компаниях есть специальные подразделения *обеспечения качества* (QA) программного обеспечения, имеющие различные обязанности, в том числе такие:

- написание тестов белого и черного ящика;
- написание тестов производительности;
- выполнение интеграционных, приемочных или системных тестов;
- обеспечение и поддержка инструментов тестирования;
- поддержка среды тестирования и инфраструктуры;
- определение формальных процессов сертификации и выпуска тестов.

Команды обеспечения качества помогают убедиться в том, что код стабилен, однако после выполнения всех тестов не прекращайте работать с кодом. Команды обеспечения качества перестали писать модульные тесты. Если вы работаете в компании, где есть команда обеспечения качества, узнайте, за что она отвечает. Если она является частью вашей основной команды, то, скорее всего, вы можете встретиться со специалистами отдела контроля качества на собраниях по планированию скрам-разработки и спринтов (больше информации о гибкой методологии разработки см. в главе 12). Если вы работаете в централизованной организации, то для контакта с командой обеспечения качества вам придется подавать официальный запрос или открыть тикет.

Пишите чистые тесты

Относитесь к написанию тестов с такой же ответственностью, как и к написанию любого другого кода. Тесты вводят зависимости и с течением времени нуждаются в обслуживании и рефакторинге. Написанные недостаточно чистым кодом тесты требуют больших затрат, что замедляет их дальнейшее улучшение или изменение. К тому же такие тесты нестабильны и с меньшей вероятностью выдают надежные результаты.

Документируйте, как тесты работают, как запускаются и для какой цели были написаны. Не дублируйте код и избегайте жестко закодированных значений. Для разделения задач, а также для поддержания связанности и разделения программ тестирований при работе используйте передовые практики проектирования.

При работе сосредоточьтесь на тестировании основных функций кода, а не на способах его реализации. Так вы упростите последующий рефакторинг кодовой базы, потому что после внесенных изменений тесты все равно будут выполняться. Если тестируемый код слишком тесно связан с определенными элементами реализации, изменения в основной части кода приведут к сбоям в работе теста. Но эти сбои не означают наличия ошибок в коде, они лишь сообщают об изменении фрагментов кода, и эта информация не представляет никакой ценности для разработчика.

Зависимости тестов рекомендуется хранить отдельно от зависимостей кода. Если для выполнения тестов необходима библиотека, не заставляйте всю кодовую базу зависеть от этой библиотеки. Большинство систем сборки и упаковки позволяют хранить зависимости тестов в отдельном месте, так что по возможности пользуйтесь данной функцией.

Не переусердствуйте с тестированием

Не пишите много тестов. Очень легко потерять представление о том, какие тесты стоит писать. Пишите тесты, которые будут выдавать ошибки, имеющие значение. Не пытайтесь написать тест с большим покрытием лишь для того, чтобы увеличить площадь покрытия кода. Нет смысла в тестировании оберток баз данных, библиотек сторонних разработчиков или назначений базовых переменных, даже если это увеличивает площадь покрытия кода. Сосредоточьтесь на тестах, которые с большей вероятностью могут повлиять на код.

Тест, выдающий ошибку, должен сообщать разработчику о важных изменениях в поведении программы. Тест, выдающий ошибку при внесении в код незначительных изменений, создает дополнительную работу для программиста и понижает его восприимчивость к подобным ошибкам. Если код работает исправно, тесты исправлять не нужно.

Не воспринимайте покрытие кода как обязательство, относитесь к нему как к желательному показателю. Высокое покрытие кода не гарантирует корректности кода. Выполнение кода теста учитывается при расчете покрытия, но это не означает, что это полезное увеличение параметра покрытия. Даже в кодовых базах, где покрытие составляет 100 %, могут возникать критические ошибки. Попытка добиться высокого покрытия не приведет ни к чему хорошему.

Не создавайте вручную тесты для автоматически сгенерированного кода, например для скаффолдинга веб-фреймворков или клиентов OpenAPI. Если инструментальные средства покрытия кода не настроены на игнорирование автоматически сгенерированного кода, то код будет помечен как непротестированный. В данном случае вам нужно изменить настройки инструментальных средств покрытия кода. Генераторы кода очень тщательно тестируются при их создании, так что повторное тестирование сгенерированного кода будет пустой тратой времени. Конечно, когда вы вручную изменяете сгенерированный код, его следует повторно тестировать. Если в какой-то момент вам понадобится протестировать сгенерированный код, постарайтесь придумать способ, которым можно добавить тесты в генератор кода.

Сосредоточьте усилия на самых значимых тестах. Написание и поддержание тестов требуют времени. Если смотреть на написание тестов с точки зрения затрат, то больше всего выгоды вы получите от наиболее значимых тестов. Используйте матрицу рисков, чтобы найти области, на которых следует сконцентрироваться. *Матрица рисков* определяет риск в виде влияния сбоя и вероятности сбоя.

На рис. 6.1 представлен пример матрицы рисков. Вероятность сбоя измеряется по оси *Y*, а влияние — по оси *X*. Пересечение вероятности и влияния сбоя определяет возможный риск.

Наличие тестов позволяет сместить риски кода вниз таблицы — чем больше тестов, тем меньше вероятность ошибки. Сосредоточьтесь на областях кода с высокой вероятностью или с высоким влиянием сбоя. Не стоит проводить тестирование кода с низкими рисками или временного кода, например для проверки концепции.

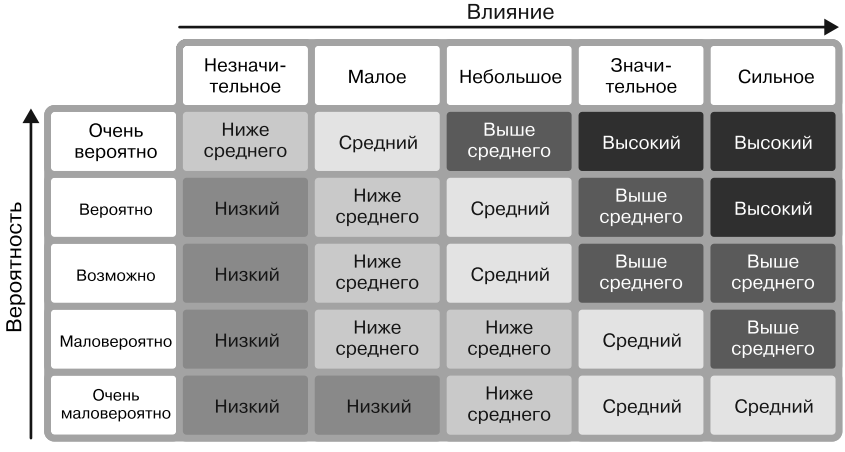

Рис. 6.1. Матрица рисков

Детерминизм в тестировании

Детерминированный код всегда выдает один и тот же результат для одних и тех же входных данных. Однако *недетерминированный код* может выдавать разные результаты для одних и тех же входных данных. Юнит-тест, вызывающий удаленный веб-сервис через сетевой сокет, является недетерминированным — если нет соединения, он выдаст ошибку. Недетерминированные тесты — проблема, с которой сталкиваются многие программисты. Важно понимать, чем именно недетерминированные тесты плохи, как их можно исправить и что делать, чтобы не писать такие тесты.

Недетерминированные тесты понижают тестовое значение. Сбои в тестировании (известные как *нестабильные тесты*) очень тяжело повторить или отладить, так как ошибка не возникает при каждом запуске теста. Вы не знаете, где допущена ошибка: в тесте или самом коде. Поскольку нестабильные тесты не дают никакой информации, разработчики могут их игнорировать, из-за чего в результате получается неправильно работающий код.

Сбои в тестировании следует исправлять сразу же после их обнаружения. Несколько раз запустите нестабильный тест, чтобы спровоцировать

сбой. В интегрированных средах разработки имеются функции, позволяющие выполнять тест по несколько раз. Вы также можете запускать цикл тестирований в оболочке операционной системы. После того как вам удастся получить ошибку, устраните ее путем исправления самой ошибки или избавления от недетерминизма.

Недетерминизм часто возникает из-за неправильного обращения с остановкой потока, тайм-аутом и генератором случайных чисел. Тесты, создающие побочные эффекты или зависящие от удаленных систем, тоже могут быть причиной недетерминизма. Избегайте сетевых вызовов, выполняйте очистку кода после тестирований, сделайте время и случайность детерминированными.

Начальное значение генератора случайных чисел

Генераторы случайных чисел должны получать начальное значение, определяющее случайные числа, которые вы будете получать. По умолчанию генераторы случайных чисел используют системные часы в качестве начального значения. Значение часов системного времени меняется, поэтому два запуска теста с генератором случайных чисел дадут разные результаты — это недетерминизм.

Начальное значение генераторов случайных чисел с константой используется для того, чтобы заставить генератор выдавать одну и ту же последовательность при каждом тестировании. Тесты с генераторами случайных чисел, которые используют в качестве начального числа константу, всегда заканчиваются успешно или всегда выдают ошибку.

Не вызывайте удаленные системы в юнит-тестах

Для удаленных систем требуются сетевые вызовы, а они обычно нестабильны. У сетевых вызовов может наступить тайм-аут, что вносит недетерминированность в модульные тесты. Тест можно запустить сто раз, но на сто первый появится ошибка из-за тайм-аута вызова

сети. Удаленные системы ненадежны: их можно отключить, перезапустить и заморозить. Если удаленная система повреждена, тест выдаст ошибку.

Без использования медленных удаленных вызовов юнит-тесты остаются быстрыми и переносимыми. Эти два параметра особенно важны для юнит-тестов, так как программисты используют их достаточно часто. Модульные тесты, зависящие от удаленных систем, перестают быть переносимыми, так как устройство, на котором выполняется тестирование, должно иметь доступ к удаленной системе. При этом удаленные системы чаще всего находятся во внутренних средах интеграционных тестов, к которым нелегко получить доступ.

Вы можете отказаться от использования удаленных систем в юнит-тестах с помощью заглушек или рефакторинга кода, так как удаленные системы нужны только для интеграционного тестирования.

Внедрение часов

Код, зависящий от определенного интервала времени, при неправильной работе может вызвать недетерминизм. Внешние факторы, например сетевая задержка или скорость центрального процессора, влияют на продолжительность операций, а системные часы работают независимо. Код с задержкой 500 мс считается неустойчивым. Тест пройдет, если код выполняется за время, меньшее или равное 499 мс, и выдаст ошибку, если код выполняется больше 501 мс. Статические методы системных часов, например `now` или `sleep`, оповещают о том, что ваш код зависит от времени. Для контроля времени, которое код видит в тесте, внедрите часы и не используйте статические методы времени.

Проблема показана на примере класса `SimpleThrottler` из языка Ruby. Когда счетчик операций превышает граничное значение, а часы не были внедрены, `SimpleThrottler` вызывает метод `throttle`:

```
class SimpleThrottler
  def initialize(max_per_sec=1000)
    @max_per_sec = max_per_sec
    @last_sec = Time.now.to_i
    @count_this_sec = 0
  end
```

```
  def do_work
    @count_this_sec += 1
    # ...
  end

  def maybe_throttle
    if Time.now.to_i == @last_sec and @count_this_sec > @max_per_sec
      throttle()
      @count_this_sec = 0
    end
    @last_sec = Time.now.to_i
  end

  def throttle
    # ...
  end
end
```

Мы не можем со стопроцентной уверенностью утверждать, что условие `maybe_throttle` из примера выше будет выполнено. Если производительность тестовой машины упала или операционная система неверно спланировала расписание тестирования, выполнение двух последовательных операций займет неопределенное время.

Чтобы избежать таких проблем, реализуйте возможность внедрения часов. Встраиваемые часы позволят вам использовать заглушки для точного управления течением времени в ваших тестах.

```
class SimpleThrottler
  def initialize(max_per_sec=1000, clock=Time)
    @max_per_sec = max_per_sec
    @clock = clock
    @last_sec = clock.now.to_i
    @count_this_sec = 0
  end

  def do_work
    @count_this_sec += 1
    # ...
  end

  def maybe_throttle
    if @clock.now.to_i == @last_sec and @count_this_sec > @max_per_sec
      throttle()
      @count_this_sec = 0
    end
```

```
    @last_sec = @clock.now.to_i
  end

  def throttle
    # ...
  end
end
```

Данный подход называется *внедрением зависимостей*. Он позволяет тестам переопределять поведение часов путем внедрения заглушки в параметр часов. Заглушка может возвращать целые числа, которые вызовут условие `maybe_throttle`. Обычный код использует системные часы по умолчанию.

Избегайте остановок потока и тайм-аутов

Разработчики часто используют функцию `sleep()` или тайм-ауты для прекращения тестирования на отдельном потоке, процессе или устройстве, прежде чем тестирование проверит правильность результатов. У такой техники есть одна проблема: предполагается, что отдельный поток закончит свою работу в определенное время, однако разработчик не может быть в этом полностью уверен. Если языковая виртуальная машина или интерпретатор выполняют сборку мусора либо операционная система решает приостановить процесс, выполняющий тест, то (иногда) ваши тесты будут завершаться неудачей.

Кроме того, приостановка выполнения или длительные тайм-ауты в тесте замедляют его выполнение, а следовательно, и весь процесс разработки и отладки. Если у вас есть тест, находящийся в режиме ожидания 30 минут, то минимум времени, которое потребуется для выполнения тестов, — 30 минут. Если у вас предусмотрен большой (или не очень) тайм-аут, существует вероятность того, что тесты зависнут.

Прежде чем добавить приостановку выполнения или установить тайм-аут в тесте, посмотрите, сможете ли вы реструктурировать тест так, чтобы не допустить недетерминизма. Если нет, ничего страшного, но постараться стоит. При тестировании асинхронного или параллельного кода не всегда можно добиться детерминизма.

Закрывайте сетевые сокеты и дескрипторы файлов

Многие тесты приводят к утечке ресурсов операционной системы, потому что разработчики считают тесты недолговечными и предполагают, что после завершения теста операционная система удалит всю информацию. Однако очень часто при выполнении разных тестов платформа тестирования использует один и тот же процесс, следовательно, утечка системных ресурсов, например сетевых сокетов или дескрипторов файлов, не может быть сразу же устранена.

Утечка ресурсов также вызывает недетерминизм. В операционных системах есть ограничение на количество сетевых сокетов и дескрипторов файлов, так что при утечке большого количества ресурсов операционные системы будут отклонять новые запросы. Тест, который не может получить доступ к новым сетевым сокетам или дескрипторам файлов, выдаст ошибку. Утечка сетевых сокетов также нарушает работу тестов, использующих один и тот же порт. Даже если тесты выполняются последовательно, второй тест не сможет привязаться к порту, так как порт открылся, но не закрылся после предыдущего теста.

Используйте стандартизованные техники управления ресурсами для ограниченных ресурсов, например конструкции `try — with — resource` или блоки. Ресурсы, которые используются в нескольких тестах, должны быть закрыты с помощью методов `setUp()` и `tearDown()`.

Привязка к порту 0

Тесты не должны привязываться к определенному сетевому порту. Статическая привязка порта вызывает недетерминизм: тест, успешно работающий на одном устройстве, на другом может выдать ошибку, если порт будет занят. Привязка всех тестов к одному порту — распространенная практика в случае, если тесты проводятся поочередно, а не параллельно. Ошибки тестирования будут недетерминированными, так как порядок выполнения тестов не всегда одинаков.

Вместе этого привяжите сетевые соединители к порту 0 — в таком случае операционная система будет автоматически выбирать свободный порт. Тесты могут получить выбранный порт и использовать это значение.

Создание уникальных путей к файлам и базам данных

Тесты не должны записываться в статически определенные места. При сохранении данных вы сталкиваетесь с такой же проблемой, как и при привязке сетевого порта. Постоянные пути к файлам и постоянное расположение баз данных будут нарушать работу тестов.

Создавайте уникальные имена файлов, баз данных, таблиц и путей к каталогам. Такие имена позволяют запускать несколько тестов параллельно, поскольку все они будут записываться в разные места. Многие языки программирования предоставляют служебные библиотеки для создания временных каталогов (к примеру, на Python это `tempfile`). Вы также можете добавлять уникальный универсальный идентификатор путям к файлам.

Изолируйте и очищайте сохранившееся состояние теста

Тесты, в которых не очищается состояние, являются причиной недетерминизма. Состояние есть везде, где хранятся данные, — в памяти или на диске. Глобальные переменные, например счетчики, соответствуют общему состоянию в памяти, а базы данных и файлы — общему состоянию на диске. Тест, добавляющий запись в базу данных и утверждающий, что эта строка существует, выдаст ошибку, если другой тест будет производить запись в ту же самую таблицу. Если запустить только один тест с пустой базой данных, он пройдет успешно. Сохранившееся состояние заполняет пространство диска, и это нарушает работу среды тестирования.

Среду интеграционного теста очень сложно настроить, так как подобные тесты обычно проводятся в паре. Многие тесты выполняются параллельно: они записываются в одни и те же хранилища данных. Будьте осторожны при работе с подобными средами тестирования, так как совместное использование ресурсов может привести к неожиданному поведению тестов. Тесты могут влиять на скорость и производительность друг друга. Хранение нескольких тестов в одном и том же хранилище данных может

привести к тому, что тесты будут конфликтовать между собой. Чтобы избежать конфликта, следуйте нашим указаниям из предыдущего раздела «Создание уникальных путей к файлам и базам данных».

Независимо от того, был тест успешным или выдал ошибку, вы должны очищать состояние: не позволяйте тестам с ошибками загружать систему. Используйте методы `setUp()` и `tearDown()` для удаления файлов теста, а также для очистки баз данных и состояния тестов между их выполнением. Проводите повторную сборку среды тестирования между запусками наборов тестов, чтобы очистить тестовые машины от сохранившегося состояния. Такие инструменты, как контейнеры или виртуальные машины, позволяют легко отключать целые машины и создавать новые. Помните, что удаление и запуск виртуальных устройств происходят намного медленнее, чем запуск методов `setUp()` и `tearDown()`, так что эти инструменты лучше всего использовать с большими наборами тестов.

Не полагайтесь на порядок тестов

Тестирование не должно зависеть от порядка выполнения тестов. Зависимость от порядка обычно возникает в том случае, когда при выполнении одного теста запоминаются данные, а при выполнении следующего предполагается, что сохраненные данные уже есть. Это может привести к следующим проблемам.

- Если первое тестирование заканчивается сбоем, то второе также не будет выполнено.
- Невозможно выполнять тесты параллельно, так как вы не можете запустить второй тест без выполненного первого.
- Изменения в первом тесте могут нарушить выполнение второго.
- Изменения в исполнителе тестов могут привести к изменению порядка выполнения тестов.

Для разделения логики между тестами используйте методы `setUp()` и `tearDown()`. В методе `setUp()` укажите данные для каждого теста и очистите данные в разделе `tearDown()`. Сброс состояния перед каждым запуском не позволит тестам мешать друг другу в том случае, если их состояние изменится.

Что следует и чего не следует делать

Следует	Не следует
Использовать тесты для воспроизведения ошибок	Игнорировать затраты на внедрение новых инструментов тестирования
Использовать инструменты создания заглушек для упрощенного написания модульных тестов	Зависеть от других при написании тестов для себя
Использовать инструменты качества кода для проверки покрытия, форматирования и сложности	Писать тесты для увеличения покрытия кода
Использовать начальные значения генераторов случайных чисел в тестах	Полагаться только на покрытие кода как на показатель его качества
Закрывать сетевые сокеты и дескрипторы файлов в тестах	Использовать режимы ожидания и тайм-аутов там, где без этого можно обойтись
Создавать уникальные пути к файлам и базам данных в тестах	Вызывать удаленные системы в модульных тестах
Очищать сохранившееся состояние теста между тестированиями	Создавать зависимость от порядка выполнения тестов

Повышение уровня

По тестированию ПО написано много (объемных) книг. Но мы предлагаем вместо чтения обстоятельных учебников использовать определенные методики тестирования.

Unit Testing[1] Владимира Хорикова (Vladimir Khorikov) (Manning Publications, 2020) — это книга, которую вы можете прочитать, если захотите больше узнать о передовых методах тестирования. В издании рассматривается концепция модульного тестирования, а также его общие паттерны и антипаттерны. Несмотря на название, в книге также говорится и об интеграционном тестировании.

В книге Кента Бека (Kent Beck) *Test-Driven Development*[2] (Addison-Wesley Professional, 2002) подробно рассматривается полезный навык

---

[1] *Хориков В.* Принципы юнит-тестирования. — СПб.: Питер, 2022.

[2] *Бек К.* Экстремальное программирование: разработка через тестирование. — СПб.: Питер, 2022.

разработки через тестирование, или TDD. Если вы попадете в компанию, практикующую TDD, данная книга обязательна к прочтению.

В книге Эндрю Ханта и Дэвида Томаса (Andrew Hunt, David Thomas) *The Pragmatic Programmer*[1] (Addison-Wesley Professional, 1999) вы можете полистать главу о тестировании на основе свойств. Мы не стали освещать этот способ тестирования, однако, если вы захотите расширить свои знания и возможности, данный вид тестирования — отличный вариант для изучения.

В книге Элизабет Хендриксон (Elisabeth Hendrickson) *Explore It!* (Pragmatic Bookshelf, 2013) рассматривается исследовательское тестирование, позволяющее изучить код. Если вы работаете со сложным кодом, эта книга будет очень полезна.

[1] *Хант Э., Томас Д.* Программист-прагматик.

7 Ревью кода

Большинство команд предпочитают проводить ревью изменений кода перед его объединением. Правильный и качественный подход к рецензированию кода помогает программистам любого уровня повышать свой профессионализм, а также способствует общему пониманию кодовой базы. Некачественный подход к рецензированию кода препятствует внедрению чего-то нового, замедляет работу и приводит к росту недовольства.

Ваша команда ожидает, что вы будете участвовать в проведении ревью кода (code review) и в качестве рецензента, и в качестве разработчика. При рецензировании кода у вас может проявиться синдром самозванца или эффект Даннинга — Крюгера (мы рассматривали их в главе 2). Вполне естественно, что при ревью кода вы можете испытывать как тревогу, так и самоуверенность, однако со всем можно справиться, если использовать нужные навыки и практики.

В этой главе объясняется, чем полезно рецензирование кода, а также как стать не только хорошим рецензентом, но и хорошим разработчиком. Вы познакомитесь с разными способами выполнения ревью кода, а также узнаете, каким образом нужно реагировать на обратную связь. Затем мы посмотрим на этот процесс с другой стороны и обсудим, как стать хорошим рецензентом.

Зачем нужны ревью кода

Качественное рецензирование кода всегда высоко ценится. У ревью кода есть вполне очевидные достоинства — вы можете узнать об ошибках или сделать код простым. Однако преимущества такой проверки выходят за рамки участия программиста в автоматизированных тестах и контроле качества кода. Хорошие рецензии кода могут служить учебным материалом, распространять информацию, создавать документацию о выполненных решениях, а также предоставлять записи об изменениях для обеспечения безопасности кода.

Ревью кода используется в качестве учебного материала для обучения вашей команды. Вы можете узнать много нового из обратной связи, которую получите из ревью кода. Рецензенты могут посоветовать вам библиотеки и практики программирования, о которых вы еще не слышали. Вы также можете столкнуться с запросами на рецензирование кода от более опытных коллег и узнать больше о кодовой базе, а также научиться писать промышленный код (см. главу 4 с подробной информацией о написании кода для продакшена). Кроме того, участие в ревью кода — хороший способ познакомиться со стилем программирования вашей команды.

Обзор изменений кодовой базы гарантирует, что хотя бы два человека знакомы с каждой ее строчкой. Общее понимание кодовой базы помогает команде улучшать код, двигаясь в одном направлении. Остальные члены вашей команды знают, какие изменения вы вносите в код, поэтому в случае ошибки команда может обратиться за помощью не только к вам, но и к другому человеку. Любой вызванный на помощь разработчик может предоставить вам краткую информацию о том, когда и какой код был изменен. То, что команда будет обладать информацией о кодовой базе, означает, что вы можете взять отпуск, не беспокоясь о поддержке своего кода.

Комментарии к ревью кода также являются частью документации: они объясняют, почему были приняты те или иные решения. Не всегда понятно, почему код написан именно таким образом. Ревью кода выступает

архивом принятых решений. Наличие предыдущих ревью предоставляет разработчику письменную историю изменения кодовой базы.

Ревью могут даже потребоваться для обеспечения безопасности и соответствия нормативным требованиям. Политика безопасности и соответствия нормативным требованиям предписывает выполнять ревью кода для предотвращения умышленного изменения кода отдельным программистом.

Однако воспользоваться всеми преимуществами ревью кода вы сможете только в том случае, если все члены команды работают в атмосфере доверия. В такой атмосфере рецензенты оставляют полезную обратную связь, а разработчики воспринимают критику и всегда открыты для чего-то нового. Необдуманная и поспешная обратная связь не представляет никакой ценности и только замедляет работу разработчиков. Задержка в выполнении ревью тормозит совершенствование и изменение кода. Без правильной культуры ревью кода между разработчиками могут возникнуть разногласия, способные привести даже к распаду команды!

Принятие ревью кода

Изменения кода готовятся, отправляются, проверяются, а затем подтверждаются и внедряются. Сначала разработчики готовят свой код для отправки. Когда код готов, они создают запрос на проверку и отправляют все подготовленные изменения. В этот момент рецензенты получают уведомление о сформированной заявке. Если между рецензентом и разработчиком налажена связь, то происходит двустороннее обсуждение кода, после чего в него вносятся все принятые изменения. После этого ревью кода утверждают и включают изменения в кодовую базу.

Подготовьте свой запрос на ревью

Правильно подготовленный запрос на ревью позволяет другим разработчикам быстро понять, что именно вы делаете, а также упрощает составление полезной и конструктивной обратной связи. Следуйте рекомендациям, представленным в главе 3: каждое изменение кода

должно быть небольшим; разделите работу над функциями и рефакторингом на разные ревью; а также составляйте описательное сообщение о фиксации. Добавляйте комментарии и тесты. Не привязывайтесь к коду, который отправляете на проверку: иногда после ревью он может сильно измениться.

Добавьте заголовок и описание, имена рецензентов, а также укажите решенную проблему, которую надо рассмотреть в ходе ревью кода. Заголовок и описание — это не сообщение о фиксации. В заголовке и описании запроса должны быть указаны информация о тестировании внесенных изменений, ссылки на ресурсы и выноски по открытым вопросам или деталям разработки. Например:

```
Рецензенты: agupta, csmith, jshu, ui-ux
Название: [UI-1343] Исправление исчезнувшей ссылки в шапке меню
Описание:

# Краткое содержание
В шапке меню отсутствует ссылка на раздел "О нас". При нажатии кнопки
меню ничего не происходит. Исправлено путем добавления правильного
атрибута href.
Для проверки изменений добавлено тестирование Selenium.

# Список
Этот запрос на включение изменений:
- [x] Добавил новые тестирования
- [] Изменил публичные API
- [] Включает дизайн-документ
```

В данном примере запроса на ревью используются популярные техники. В тексте упоминаются не только отдельные рецензенты, но и вся команда UI/UX. В названии указывается ошибка, которую нужно устранить (UI-1343). Использование соглашения о стандарте форматирования ссылок на проблемы позволяет интегрировать системы, которые автоматически связывают номера для отслеживания проблем с ревью кода. Это полезно при последующем обращении к старым проблемам.

Описание также является частью шаблона ревью кода, и оно было включено в репозиторий. В некоторых репозиториях есть шаблоны описания с указанием всей необходимой для рецензентов информации об изменениях. Например, может потребоваться отдельная проверка изменения, влияющего на общедоступный API.

Снижение рисков с помощью черновика ревью

Многим программистам лучше всего думается во время написания кода. Создание черновика — хороший способ продумать все изменения, не потратив при этом времени на написание тестов, изменение кода или добавление документации. Вы можете отправить *черновик ревью* (draft review) на проверку и узнать, есть ли у вас какие-либо недочеты или ошибки. В таком случае вы создаете запрос, после чего в короткий срок получаете обратную связь от членов команды. Данный способ не даст вам увязнуть в ошибках и спасет от лишней работы.

Во избежание путаницы указывайте в запросе тип работы: черновик кода или незавершенный (WIP) код. Многие команды обсуждают черновики, так что обычно перед названием кода указано DRAFT или WIP. Некоторые платформы для ревью кода имеют встроенную поддержку: например, у GitHub есть «черновой запрос на включение» (draft pull request). Как только станет понятно, что в вашем черновике не так много ошибок, вы сможете перевести его из состояния «черновик», завершив разработку, выполнив соответствующие тестирования и составив документацию, а также отшлифовав детали. Четко определите, когда ваш код готов к полноценному рецензированию, а затем подготовьте запрос на ревью, как описано в предыдущем разделе.

Не отправляйте ревью кода для запуска тестирования

Очень часто в больших проектах используются сложные инструменты тестирования. Новому разработчику может быть трудно понять, как запускать соответствующие тесты. Некоторые стараются обходить эту проблему и отправляют ревью кода для запуска системы непрерывной интеграции, однако это плохая практика.

Отправка ревью кода будет слишком расточительным способом инициировать выполнение теста. В этом случае ревью заполнит очередь тестов, что заблокирует обзоры тех проектов, которым действительно

нужно пройти тест перед слиянием. Коллеги по ошибке могут принять ваш запрос на ревью как срочный. Система непрерывной интеграции запустит полный пакет тестов, в то время как вам нужны были только тесты с проверкой внесенных вами изменений.

Изучите возможность локального запуска тестов. Проводить отладку теста, выдавшего ошибку, лучше локально, а не в системе непрерывной интеграции. Вы не сможете подключать отладчики или свободно получать информацию об отладке на удаленных машинах. Настройте локальную среду тестирования и узнайте, как запускать только необходимые вам тесты. Сделайте ваш цикл программирования и тестирования быстрым — в таком случае вы сразу будете знать, если внесенные в код изменения нарушают работу кода. Затратив на это время в самом начале, вы сэкономите свое время в будущем. Это поможет вам также наладить связь с коллегами.

Прогон больших изменений

После внесения в код больших изменений его следует прогнать. Прогон — это личная встреча, на которой разработчик на общем экране показывает выполнение кода и знакомит коллег с внесенными изменениями. Прогоны помогают команде принять изменения и позволяют обменяться идеями.

Перед собранием разошлите коллегам соответствующую документацию и код, чтобы они могли с ними ознакомиться перед встречей. Выделите на это достаточно времени, а не предоставляйте коллегам новую информацию за час до прогона.

Начните прогон с предыстории изменений. Иногда может потребоваться беглый обзор проектной документации. Затем выведите изображение на экран и в ходе повествования перемещайтесь по коду в интегрированной среде разработки. Лучше всего проводить демонстрацию с самого начала: от загрузки страницы, вызова API и запуска приложения — и до завершения выполнения. Расскажите об основных концепциях, лежащих в основе новых моделей, для чего они используются и как вписываются в приложение.

Не пытайтесь склонить коллег к тестированию кода прямо во время прогона: они должны приберечь свои комментарии для рецензирования. Прогон выполняется, чтобы члены команды поняли, для чего вносятся изменения, а также чтобы предложить участникам несколько моделей для тестирования кода.

Не ассоциируйте себя с кодом

Разработчик не всегда с легкостью может принять критику кода со стороны. Держите эмоциональную дистанцию — комментарии касаются не вас, а кода, который на самом деле даже и не ваш, а всей команды. Если вы получаете большое количество правок и предложений, это не значит, что вы не справились: это знак того, что рецензент работает над вашим кодом и предлагает варианты по его улучшению. Получать большое количество комментариев, особенно не являясь опытным разработчиком, вполне нормально.

Рецензенты могут попросить внести изменения, которые кажутся неважными или которые можно внести позже. Старайтесь сохранять непредвзятость и понимать причину появления замечаний. Здраво воспринимайте критику и будьте готовы менять код на основе полученной обратной связи.

Развивайте эмпатию, но не терпите грубости

Каждый человек общается по-своему, однако нельзя терпеть грубости. Всегда имейте в виду, что краткие и точные слова для одного человека могут быть грубыми и бестактными для другого. Относитесь к рецензентам с пониманием, но дайте им знать, если их комментарии кажутся необоснованными или грубыми. Когда дискуссия затягивается или становится бессмысленной, обсуждение с глазу на глаз может помочь прояснить ситуацию и прийти к решению. Если вам по каким-либо причинам будет неловко, обратитесь к своему руководителю.

Не соглашаясь с каким-либо предложением, старайтесь обсудить это с командой. Сначала оцените собственную реакцию. Вы защищаете

свой код только из-за того, что это вы его написали, или из-за того, что методы, которые вы использовали, намного лучше предложенных? Объясните свою точку зрения. Если вы все еще не можете прийти к согласию, то спросите у руководителя, что вам делать дальше. Разные команды по-разному справляются с конфликтами, возникающими из-за ревью кода: одни полагаются на заказчика, другие — на техлида. Всегда придерживайтесь соглашений команды.

Проявляйте инициативу

Не стесняйтесь просить других людей протестировать ваш код. Очень часто рецензенты завалены запросами на проверку кода и уведомлениями о запросах, так что ваше обращение может затеряться во множестве срочных дел. Если вы не получили никакой обратной связи, свяжитесь с командой (но без особой настойчивости). Получив обратную связь, не тяните с ответом, если не хотите, чтобы проверка вашего кода заняла несколько недель.

Вносите изменения в код сразу же, как получите на это разрешение. Не стоит оставлять проведенное рецензирование кода без внимания — ваши коллеги могут ждать, когда вы внесете изменения, или могут захотеть изменить код после слияния. Если вы будете затягивать время, ваш код придется переделывать и исправлять. Иногда преобразование может нарушить логику кода, из-за чего потребуется повторно выполнить ревью кода.

Выполнение ревью кода

При получении запроса на ревью хороший рецензент делит процесс на несколько этапов: сначала изучает запрос и определяет его срочность и сложность, а затем выделяет время для просмотра внесенных изменений. Начните рецензирование с чтения кода: чтобы понять причину изменений, задавайте вопросы. После дайте обратную связь и закончите анализ кода. Сочетание данного метода с несколькими передовыми практиками значительно улучшит сделанные вами ревью.

Рассмотрение запроса на ревью

Ваша работа в качестве рецензента начинается именно в тот момент, когда вы получаете уведомление о запросе. Начните свою работу с сортировки запросов на проверку. Некоторые изменения нужно срочно проанализировать, однако большая часть запросов на рецензирование не требует столь быстрого выполнения. Если срочность выполнения непонятна, обратитесь к разработчику. Не забывайте учитывать размер и сложность изменений. На проверку больших изменений потребуется много времени, однако если вам нужно проверить маленькое изменение, то лучше рассмотреть его быстро, чтобы не задерживать коллег.

У быстро работающих команд в работе может быть большое количество ревью. Вам не нужно анализировать каждое изменение. Постарайтесь сосредоточиться на изменениях, которые вам понятны и находятся в знакомом для вас коде.

Выделите время для ревью

Рецензирование кода аналогично оперативной работе (подробнее в главе 9): объем и частота выполнения ревью непредсказуемы. Когда вы получаете новый запрос на ревью, не спешите бросать то, чем занимались до этого. Вы можете сильно снизить свою продуктивность, если отвлечетесь от основной задачи.

Выделите в своем графике время, когда будете заниматься только рецензированием кода. Так вы сможете спокойно сосредоточиться на основной работе. Это улучшит качество вашей обратной связи — вы не будете спешить и переживать, что нужно вернуться к незаконченным делам.

Для больших ревью кода может потребоваться дополнительное время. Если вы получили код, на проверку которого уйдет больше пары часов, сформируйте отдельную задачу для отслеживания анализа кода. Вместе с руководителем выделите время на планирование спринта (подробнее о гибкой разработке читайте в главе 12).

Вникайте в изменения

Не начинайте ревью с написания комментариев: сначала прочтите все и задайте разработчику несколько вопросов. Ревью ценится намного выше, когда рецензент сам пытается разобраться в предлагаемых изменениях. Старайтесь понять, зачем нужны изменения, как ведет себя код и как он будет себя вести после внесения изменений. Рассмотрите также долгосрочные последствия предложенных проекта API, структур данных и других ключевых решений.

Понимание мотивации изменений поможет понять их смысл. Иногда в результате вы можете решить, что в изменениях нет никакого смысла. Сравнив код до и после внесения изменений, вы проверите правильность принятого решения, а также рассмотрите другие возможности для внесения изменений.

Давайте исчерпывающую обратную связь

Дайте обратную связь о корректности предложенных изменений и их внедрении, о возможности сопровождения, удобочитаемости и безопасности. Укажите на код, который нарушает руководство по стилю, трудно читается или вводит в заблуждение. Читайте тесты и ищите ошибки, чтобы проверить правильность кода.

Спросите себя, как бы вы внедрили изменения, и рассмотрите альтернативные идеи. Если общедоступные API меняются, то подумайте о том, как это отразится на совместимости и предложенных изменениях кода (больше информации вы найдете в главе 8). Представьте, что другой программист может неправильно понять или использовать код, и поищите способ так изменить код, чтобы не допустить подобной ситуации.

Поразмышляйте о том, какие библиотеки и службы можно использовать для внесения изменений. Для поддержки кода предложите паттерны из главы 11. Ищите уязвимости из топ-10 по версии OWASP (https://owasp.org/www-project-top-ten/), например внедрение SQL-кода, утечку конфиденциальных данных и межсайтовый скриптинг.

Не будьте слишком резкими — пишите комментарии так, словно вы сидите рядом с разработчиком и объясняете ему все лично. Ваши комментарии должны быть вежливыми и содержать как «что», так и «почему».

```
Убедитесь, что значение `port` будет ≥ нулю, и увеличьте значение
InvalidArgumentException, если условие не выполнено. Значение порта
не может быть отрицательным.
```

Отмечайте положительные стороны

Совершенно естественно, что при ревью кода вы нацеливаетесь на поиск ошибок, но отзыв не должен состоять только из негатива. Отмечайте и положительные моменты! Если при чтении кода вы узнаете что-то новое, то в обратной связи сообщите об этом разработчику. Если в процессе рефакторинга устраняются ошибки в коде или новые тесты делают будущие изменения менее рискованными, напишите положительный комментарий. Даже об изменении, которое вам не нравится, можно написать что-то хорошее — а если совсем не за что, то просто похвалите за стремление и приложенные усилия.

> *Очень интересное изменение. Я понимаю желание перенести код очереди в стороннюю библиотеку, но я не хочу добавлять новую зависимость. Код достаточно простой и отлично справляется с поставленными задачами. Обязательно сообщите, если я неправильно понимаю мотивацию. Надеюсь обсудить более подробно.*

Разница между проблемами, предложениями и придирками

Не все комментарии имеют одинаковый уровень важности. Серьезные проблемы требуют большего внимания, чем нейтральные предложения и незначительные придирки.

Не стесняйтесь отмечать огрехи форматирования, однако дайте понять, что это некритическое замечание. Обычно к такому комментарию добавляется префикс nit (от nitpick — «придирка»):

```
Nit: Двойной пробел
Nit: Здесь и далее используйте стили snake_case для методов
и PascalCase для классов
Nit: Название метода лично мне кажется странным. Как насчет
maybeRetry(int threshold)?
```

Если одна и та же проблема форматирования повторяется несколько раз, не нужно указывать в комментарии все случаи — укажите только на первое появление ошибки и посоветуйте исправить ее во всем коде. Никто не любит, когда ему раз за разом говорят одно и то же, так как в этом нет необходимости. Если вам приходится часто придираться к форматированию, спросите, настроены ли в проекте соответствующие инструменты линтинга. В идеале такую работу должны делать программы.

Если вы видите, что ваш отзыв состоит из одних придирок и только малая часть — реально полезные комментарии, остановитесь и просмотрите код еще раз. Изменения стиля — это часть рецензирования, но не его главная цель. Для получения дополнительной информации об инструментах линтинга и чистоте кода обратитесь к главе 3.

Помечайте предложения, которые кажутся вам полезными и которые не нужно утверждать. При написании комментариев добавьте пометки «необязательно» (optional), «принять или нет» (take it or leave it) или «неблокируемый» (nonblocking). Оформите отличия предложений от изменений, которые действительно хотите внести, иначе разработчику будет сложно разобраться.

Не штампуйте ревью

Вы постоянно будете испытывать на себе давление со стороны, обязывающее вас поскорее завершить и отправить ревью. Это могут быть требования срочных изменений, постоянные напоминания коллег, слишком объемные для обработки изменения или кажущиеся вам незначительными правки. Если вы эмпатичный человек, то тоже будете пытаться быстро дать ответ на запрос рецензирования, потому что точно знаете, каково это — долго ждать обратной связи.

Не поддавайтесь искушению поспешно утвердить ревью кода. Механическое и формальное написание отзывов только навредит. Ваши коллеги будут считать, что вы знакомы с изменением и понимаете, для чего оно внедряется, поэтому в будущем ответственность за код может лечь и на ваши плечи. Разработчик подумает, что вы посмотрели и одобрили его код. Если вы не можете правильно расставить приоритеты в работе, вообще не рецензируйте код.

Желание штамповать ревью может быть знаком того, что изменение слишком большое и его надо рассматривать в рамках нескольких запросов. Не стесняйтесь просить своих коллег разбить один большой запрос на несколько маленьких. Разработчики часто увлекаются и вносят изменения на пару тысяч строк. Странно ожидать, что столь большое изменение можно проверить за один раз. Если вы считаете, что выполнить прогон кода будет более эффективным подходом, сообщите об этом.

Не ограничивайте себя веб-инструментами рецензирования

Обычно рецензирование выполняется в специальном пользовательском интерфейсе, например интерфейсе запроса на включение на GitHub. Не забывайте, что вы все так же можете тестировать рецензируемый код или вносить в него предлагаемые изменения и поэкспериментировать с ними в локальной среде.

Локальная проверка кода позволит вам рассмотреть предложенные изменения в вашей интегрированной среде разработки. Если вы сталкиваетесь с объемными изменениями, то работа в веб-интерфейсах может вызвать некоторые проблемы. Интегрированные среды разработки, а также инструменты анализа на базе ПК упрощают просмотр вносимых изменений.

Вы также можете запустить локальный код и написать собственные тесты для проверки того, что код работает правильно. Отладчик можно подключить к работающему коду для лучшего понимания работы кода. Возможно, даже получится инициировать сценарий сбоя, чтобы лучше проиллюстрировать комментарии в ревью.

Не забывайте выполнять ревью тестов

Очень часто рецензенты не обращают должного внимания на тесты, особенно если изменение проводится длительный период времени. Тесты должны проверяться так же, как и основной код. Чаще всего при ревью стоит начинать с чтения тестов, потому что именно там содержится информация о том, для чего был написан код и что от него ожидается.

Обязательно проверьте чистоту кода, а также возможность будущих улучшений. Ищите неудачные фреймворки тестирования: неправильный порядок выполнения, недостаток изоляции или удаленные системные вызовы. Для знакомства с полным списком лучших практик тестирования и ошибок, на которые нужно обращать внимание, просмотрите главу 6.

Делайте выводы

Не становитесь тем, кто будет на корню пресекать все улучшения. Помогите разработчикам, которые отправляют свой код на проверку. Проводите анализ кода быстро, не настаивайте на абсолютной правильности, не увеличивайте количество изменений, а в обратной связи четко укажите, какие моменты являются критически важными. Не позволяйте разногласиям встать между вами и разработчиком.

Указывайте на важность качества кода, однако не делайте из этого критерия непреодолимый барьер. Это противоречие при просмотре списка изменений обсуждается в статье *Engineering Practices Documentation* от Google (https://google.github.io/eng-practices/) (для обозначения списка предлагаемых изменений в Google используют термин CL):

> *Если CL не идеален, но его текущий уровень наполненности изменениями способствует улучшению общего состояния кода системы, рецензентам стоит одобрить такой CL.*

Учитывайте масштаб изменений. По мере чтения вы найдете способы улучшить код и у вас появятся идеи насчет новых функций, но не стоит

настаивать на внесении ваших изменений во время текущего ревью. Откройте тикет для улучшения кода и отложите эту работу на будущее. Сохранение жестких рамок увеличит скорость и сделает изменения постепенными.

Вы можете завершить ревью, пометив его как «Запрос на изменения» (Request Changes) или «Одобрено» (Approved). Если у вас много комментариев, может понадобиться краткое резюме отчета. Если вы запрашиваете изменения, укажите, что именно следует изменить для одобрения. Например:

```
Изменения хорошие. Есть несколько недочетов, но у меня только одна
просьба — исправьте работу с портами. Код кажется ненадежным.
Подробности в комментариях.
```

Если вы и автор кода не можете прийти к единому мнению, обратитесь к третьему лицу — другому эксперту, который поможет разрешить ваши разногласия.

Что следует и чего не следует делать

Следует	Не следует
Убедиться в том, что тесты и линтеры пройдены, прежде чем запрашивать ревью	Отправлять запрос на проверку только ради запуска системы непрерывного развертывания
Выделять время на ревью кода, расставлять приоритеты так же, как и при любой другой работе	Штамповать ревью кода
Не молчать, если обратная связь кажется грубой или неуместной	«Влюбляться» в свой код и принимать критику близко к сердцу
Помогать рецензенту, предоставив контекст для изменений	Обращать внимание на небольшие детали кода, пока не составите полную картину всех изменений
При написании рецензии не ограничиваться поверхностными вопросами стиля	Слишком придираться
Для понимания сложных изменений использовать все доступные инструменты, а не только интерфейс ревью кода	Позволять лучшему стать врагом хорошего
Проверять тесты	

Повышение уровня

Code Review Developer Guide от Google по ссылке https://google.github.io/eng-practices/review/ представляет собой хороший пример корпоративной культуры ревью кода. Только помните, что это руководство написано специально для Google. Терпимость вашей компании к рискам, инвестиции в автоматизированные проверки качества и выбор в качестве приоритета скорости или стабильности могут привести к другой культуре ревью.

В любом случае ревью кода — это особый вид предоставления и получения обратной связи. Книга Дугласа Стоуна и Шейлы Хин (Douglas Stone, Sheila Heen) *Thanks for the Feedback: The Science and Art of Receiving Feedback Well* (Penguin Books, 2014) — отличное пособие по тому, как стать хорошим рецензентом и разработчиком.

8 Доставка программного обеспечения

Вы должны понимать, каким образом ваш код попадает к пользователям. Понимание процесса доставки поможет справляться с задачами и контролировать внедрение изменений. Вам необязательно принимать в этом участие лично: процесс может быть автоматизирован или выполняться релиз-инженерами, но этапы между `git commit` и реальным трафиком не должны быть для вас загадкой.

Программное обеспечение считается доставленным, когда оно стабильно работает в продакшен-среде и клиенты активно его используют. Доставка состоит из нескольких этапов, таких как выпуск, развертывание и выгрузка. В данной главе описываются этапы доставки ПО клиентам, стратегии ветвления в управлении версиями (которые влияют на выпуск программного обеспечения) и передовые наработанные методики.

Этапы доставки ПО

К сожалению, для этапов доставки нет определений, принятых в качестве производственного стандарта. В зависимости от того, с кем вы разговариваете, такие понятия, как *выпуск* и *развертывание*, могут обозначать разные этапы процесса доставки ПО. Участники вашей команды могут называть весь процесс доставки, начиная от упаковки и заканчивая выгрузкой, *выпуском*. Они могут назвать *выпуском* упаковку артефакта, а открытие доступа для скачивания артефакта — *публикацией*. Пока

функция не включена в продакшен-код, нельзя сказать, что произошел *выпуск*, в то время как все перед этим действием относится к *развертыванию*.

Одна из целей данной главы — рассмотреть четыре этапа доставки программного обеспечения: *сборку*, *выпуск*, *развертывание* и *выгрузку*. Все этапы представлены на рис. 8.1. Сначала необходимо выполнить *сборку* программного обеспечения в пакеты. Сборка должна быть неизменяемой, а также должна получить определенный номер версии. Затем необходимо выполнить *выпуск (релиз)* пакета. Примечания к релизу и журналы изменений объекта обновляются, а пакеты программ публикуются в централизованном репозитории. Артефакты опубликованных релизов должны быть *развернуты* в предпродакшен- и продакшен-средах. Пользователи еще не могут работать с развернутым программным обеспечением, так как оно только что было установлено. После этого этапа программное обеспечение *выгружается* путем перехода пользователей на новое программное обеспечение. После завершения выгрузки считается, что программное обеспечение доставлено.

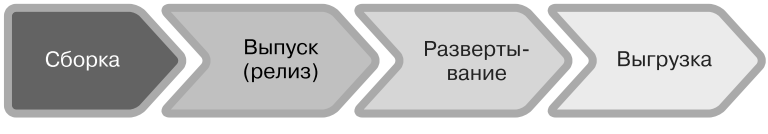

Рис. 8.1. Этапы доставки ПО

Процесс доставки является частью большого цикла разработки ПО. После развертывания собирается обратная связь, ведется поиск ошибок, оцениваются новые требования к продукту. Начинается добавление новых функциональных средств, а затем новая сборка.

Для каждого этапа доставки разработан набор передовых методик. Они помогут вам быстро и качественно доставлять программное обеспечение. Однако перед тем, как мы подробно рассмотрим этапы, поговорим о стратегиях ветвления в системах управления версиями. Стратегии ветвления определяют, где фиксируются изменения кода и как сопровождается выпуск кода. Правильно выбранная стратегия ветвления позволит сделать доставку ПО простой и предсказуемой, а неподходящая стратегия ветвления усложнит процесс доставки продукта.

Стратегии ветвления

Дистрибутивы создаются из кода в системах управления версиями. *Магистраль* (trunk, main, mainline) содержит в себе основную версию кодовой базы со всей историей изменений. Из магистрали «прорезаются» *ветки* (brunch) для изменения кода: с некоторыми ветками можно работать параллельно и объединять все готовые изменения в магистраль. Разные стратегии ветвления показывают, сколько должны существовать ветви, как они связаны с выпускаемыми версиями ПО и как изменения распространяются на разные ветки. Существуют две основные модели: магистральная разработка и разработка на основе функциональных веток.

При использовании *магистральной разработки* программисты не работают на магистрали. Ветки создаются для написания небольшой функции, исправления ошибки или обновления.

На рис. 8.2 изображена стратегия магистральной разработки. Функциональная ветка создается в ветви feature-1 и соединяется с trunk. Ветвь bug-1 была создана для исправления ошибок. Ветвь release вырезана, поэтому разработчики решили добавлять исправления ошибок в ветвь release-1.0.

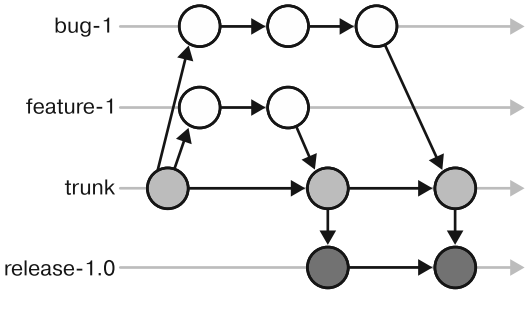

Рис. 8.2. Ветви магистральной разработки

Магистральная разработка работает лучше всего в том случае, когда ветви быстро (за несколько дней, а может, и часов) соединяются с магистралью и не распределяются между разработчиками. Чаще всего процесс соединения известен как *непрерывная интеграция (CI)*. Поскольку

о внесенных изменениях узнают сразу все разработчики, снижается вероятность значительного расхождения в их действиях. Синхронизация кодовых баз разработчиков позволяет выявлять ошибки и несовместимости интеграции на ранних этапах. Однако ошибки в магистрали могут затормозить работу всех разработчиков. Для предотвращения таких ситуаций перед объединением веток с магистралью проводятся быстрые автоматизированные тесты. У команд часто имеются явные процессы для обнаружения неисправностей в магистрали: ожидается, что магистраль всегда должна быть работоспособной, так как релизы происходят очень часто.

При использовании *разработки на основе функциональных веток* большинство разработчиков одновременно работают с долго существующими функциональными ветками, где каждая ветка связана с каким-то функциональным элементом в продукте. Поскольку функциональные ветки существуют достаточно продолжительное время, разработчикам нужно *включать изменения из магистрали в свой код*, чтобы функциональная ветка не слишком расходилась с основным кодом. Ветки остаются стабильными благодаря контролю в тот момент, когда происходит объединение изменений. При подготовке релиза в его ветку включаются функциональные ветки. Ветки релиза проходят тестирование, а функциональные продолжают дорабатываться. Пакеты обычно проходят сборку на основе стабильных выпускаемых веток.

Использование разработки на основе функциональных веток — обычное явление. Такой подход применяется, когда магистраль слишком нестабильна для релиза, но разработчики не хотят прекращать добавлять функционал в ожидании стабилизации. Разработка на основе функциональных ветвей более распространена при работе с ПО для розничной продажи, где разные заказчики пользуются различными версиями. В сервис-ориентированных системах обычно используется магистральная разработка.

Самым популярным подходом к функциональным веткам является *Gitflow*, описанный Винсентом Дрисеном в 2010 году. Gitflow использует ветки разработки, исправлений и релиза. Ветка разработки — главная, с которой объединяются другие ветки. При подготовке релиза соответствующая ветка отсоединяется от ветви разработки. Разработка продолжается в функциональных ветках во время стабилизации выпуска.

Все релизы стабилизируются и объединяются в магистраль. Считается, что магистраль всегда в состоянии готовности, так как в ней находятся только стабилизированные релизы. Если магистраль нестабильна из-за наличия ошибок, то они немедленно отправляются в ветку исправлений, не дожидаясь стандартного релиза. Исправления вносятся в ветку исправлений и объединяются не только с магистралью, но и с веткой разработки.

На рис. 8.3 представлен пример Gitflow с двумя функциональными ветками: feature-1 и feature-2. Функциональные ветки существуют долго, выполняются коммиты и объединения с веткой разработки. В ветке разработки два релиза включаются в магистраль. Ветвь исправлений используется для исправления ошибок, найденных в магистрали. Исправления также добавляются в ветку разработки, чтобы функциональные ветки могли их использовать.

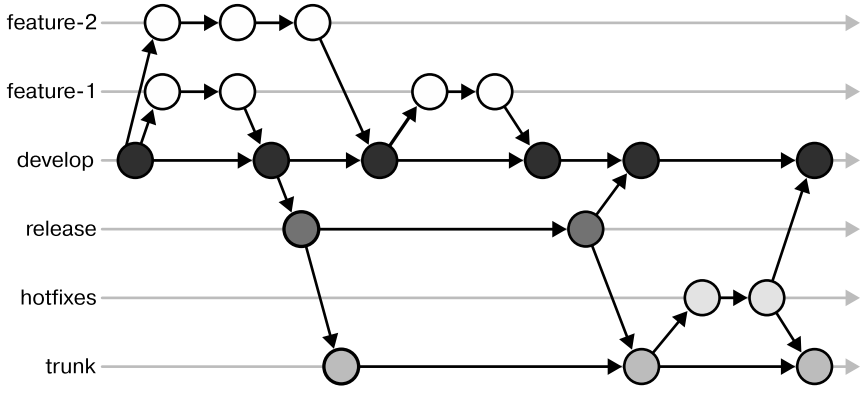

Рис. 8.3. Gitflow с использованием разработки на основе функциональных веток

Поймите стратегию ветвления, используемую вашей командой, и придерживайтесь ее. Стратегии ветвления определяют, когда выходят обновления, устанавливают ожидания от тестов, определяют варианты устранения неполадок и количество версий, в которые следует перенести изменения. Многие компании разрабатывают внутренние инструменты, помогающие руководить процессами системы управления версиями. Эти скрипты будут автоматически ветвиться, объединяться и отмечаться тегами.

Если вам не нужны долговечные функциональные ветки, придерживайтесь стратегии магистральной разработки. Управление функциональными ветками постоянно усложняется. По правде говоря, Дрисен изменил свой первый пост в блоге Gitflow, чтобы не казалось, что он поощряет использование Gitflow для ПО, которое нужно постоянно интегрировать и поставлять.

Этап сборки

Выполнять сборку пакетов ПО следует до начала этапа доставки. Сборка программного обеспечения состоит из множества шагов, таких как разрешение и связывание зависимостей, запуск линтеров, компилирование, тестирование и упаковка ПО. Большинство шагов этапа сборки также используются в разработке: они были рассмотрены в главах 3–6. В этом разделе мы сосредоточимся на продукте сборки — пакетах.

Пакеты создаются для каждого отдельного релиза, так что не нужно выполнять сборку ПО на каждом устройстве, на котором оно работает. Предварительная сборка пакетов формирует более однообразное ПО, чем компиляция и запуск кода на каждой отдельной машине с использованием собственной среды разработки и уникальной комбинации инструментов.

Если ПО предназначено для нескольких платформ или сред, то во время сборки создается несколько пакетов. На этапе сборки очень часто создают пакеты ПО для разных операционных систем, архитектур ЦП и сред выполнения языка программирования. Мы уверены, что вам встречались такие имена пакетов на Linux:

- `mysql-server-8.0_8.0.21-1_amd64.deb`;
- `mysql-server-8.0_8.0.21-1_arm64.deb`;
- `mysql-server-8.0_8.0.21-1_i386.deb`.

Все пакеты MySQL были созданы для одной версии MySQL, однако каждый настроен для разных архитектур: AMD, ARM и Intel 386.

Содержимое и структура пакетов отличаются друг от друга. В пакетах могут находиться двоичный или исходный код, зависимости, настройки,

примечания к релизу, необходимая документация, медиа, лицензии, контрольные суммы или даже образы виртуальных машин. Все библиотеки упакованы в подходящие для языка форматы, например *JAR*, *wheel* и *crates*. Большая часть из них представляет собой заархивированные библиотеки, организованные в соответствии со спецификацией. Пакеты приложений создаются в виде ZIP-архивов, TAR-архивов или установочных пакетов (`setup.exe` или файлов в формате `.dmg`). Контейнерные или машинные пакеты позволяют создавать не только ПО, но и среду, в которой оно будет работать.

Сборка пакета определяет, какое программное обеспечение будет выпущено. Плохая упаковка затрудняет развертывание и отладку ПО. Во избежание таких проблем всегда проверяйте версии пакетов, а также разделяйте их в соответствии с типами ресурсов.

Версии пакетов

У всех пакетов должны быть определенная версия и уникальный идентификатор. Уникальный идентификатор позволяет операторам и разработчикам связать работающее приложение с исходным кодом, набором функций или документацией. Без управления версиями вы не будете знать, как пакет себя поведет. Если вы не уверены в том, какую систему управления версиями выбрать, выбирайте семантическое управление версиями. Большинство пакетов связаны с той или иной формой семантического управления версиями (см. главу 5).

Упаковывайте разные ресурсы по отдельности

Программное обеспечение — это не просто код. Настройки, схемы, медиа и языковые пакеты (или переводы) также являются частью ПО. Различные ресурсы имеют разную частоту выпуска, разное время сборки, а также обладают разными требованиями к тестированию.

Различные ресурсы нужно упаковывать отдельно: так у вас будет возможность изменять их, не затрагивая целостности всего программного пакета. Отдельная упаковка ресурсов позволяет каждому типу ресурсов обладать собственным циклом выпуска, который может выполняться независимо от других ресурсов.

Если вы отправляете заказчику полностью готовое приложение, то конечным пакетом будет метапакет — пакет пакетов. Если вы отправляете веб-сервис или самообновляемое приложение, можете отправлять пакеты по отдельности, позволяя выполнять конфигурацию и перемещение отдельно от остального кода.

ПАКЕТЫ В PYTHON

У Python длинная и весьма запутанная история управления пакетами, что делает его отличным примером для изучения. The Python Packaging Authority (PyPA) — отдел, отвечающий за упаковку в Python, — опубликовал документ *An Overview of Packaging for Python* (https://packaging.python.org/overview/), в котором авторы пытаются рационализировать варианты пакетирования Python. На рис. 8.4 и 8.5, созданных Махмудом Хашеми и используемых в обзоре PyPA, показаны различные варианты упаковки Python.

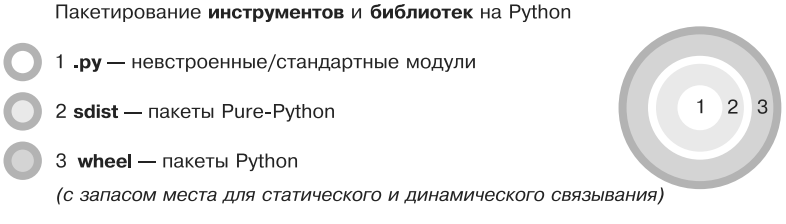

Рис. 8.4. Варианты упаковки инструментов и библиотек, доступных на Python

В основе луковичной архитектуры библиотеки Python лежит простой исходный файл `.py`. Следующий уровень, *sdist*, является группой файлов `.py` — модулей, собранных в архивы `.tar.gz`. Несмотря на то что пакеты sdist включают в себя весь код Python для модуля, в них не содержится скомпилированный код, который в определенный момент может понадобиться пакету. Затем совершается переход на следующий уровень. Кроме необработанного кода Python, *wheel* включают в себя скомпилированные собственные библиотеки, написанные на C, C++, Fortran, R или любом языке, от которого может зависеть пакет Python.

На рис. 8.5 показан многоуровневый способ пакетирования приложений. Данный вариант включает в себя языковые среды выполнения, образы виртуальных машин и аппаратное обеспечение. *Пакеты PEX* содержат код на Python, а также его зависимости от библиотек. *Anaconda* представляет собой систему для управления всеми установленными библиотеками, а не только теми, от которых зависит приложение. Утилиты *Freeze* объединяют в пакеты не только библиотеки, но и среду выполнения Python. *Образы*, *контейнеры* и *виртуальные машины* пакетируют операционную систему и образы дисков. В некоторых случаях в качестве метода пакетирования используют отправку комплектующих вместе с установленными пакетами приложений, системными библиотеками или операционной системой.

Рис. 8.5. Варианты упаковки, доступные для приложений на Python

И хотя некоторые из перечисленных вариантов упаковки специфичны для Python, во многих языках есть похожие модели. Внешние круги, например образы машин и контейнеров, не зависят от языка программирования и сопоставляются с каждым слоем используемого вами стека упаковки. Понимание особенностей вашей системы упаковывания позволит избежать проблем с развертыванием.

Этап релиза

После релиза ПО становится доступным для пользователей и обеспечивается развертывание — следующий этап поставки. В зависимости от типа и размера ПО, а также пользовательского опыта процессы выпуска будут различаться. Релиз внутреннего веб-сервиса может состоять только из одного шага: публикации пакета ПО в общем репозитории пакетов. Релизам для широкого использования требуются публикация артефактов, обновление документации и заметок о релизе и взаимодействие с пользователем.

Управление релизом — это мастерство публикации стабильного, хорошо документированного ПО с предсказуемой частотой. Правильное управление релизом должно обеспечить удобство клиентов. У сложного ПО, над которым работает несколько команд, чаще всего будет релиз-менеджер. Релиз-менеджеры координируют весь процесс разработки, от тестирования и проверки функций до процедур безопасности, основной документации и т. д.

Понимание механизма управления релизами поможет вам работать над процессом выпуска ПО с большей эффективностью. Возьмите на себя ответственность за публикацию своего ПО, часто выпуская обновления пакетов. Будьте предсказуемы в расписании релизов, а также публикуйте заметки о релизе и журналы изменений вместе с новыми релизами.

Несите ответственность за релизы

Отвечайте сами за выпуск своего программного обеспечения. Даже если в вашей компании уже есть команды релиз-инжиниринга или использования ПО, вы должны знать, когда и в каком состоянии ваше ПО попадает к конечному пользователю. Команды по выпуску и использованию ПО могут оказать помощь в настройке инструментов, дать советы по методам, которые стоит использовать, рассказать об автоматизации рутинной работы и ведения отчетности, но при этом они не знают ваш код настолько хорошо, как знаете его вы. В конце концов, именно вы несете ответственность за корректное развертывание и функционирование вашего ПО.

Убедитесь в том, что ваш код корректно работает в средах тестирования, внимательно следите за расписанием выхода релизов, изучайте возможные варианты, а также выбирайте правильный подход к разработке и реализации своего приложения. Если выпускается только половина приложения или в рабочей среде появляется ошибка, вам необходимо выяснить, почему это произошло, а также принять возможные меры для предотвращения подобных ситуаций в будущем.

Публикуйте пакеты в репозитории релизов

Обычно пакеты релиза публикуются в репозитории пакетов или просто помечаются тегами и выкладываются в систему VCS, например в Git. Хотя работать может любой подход, мы советуем публиковать пакеты в специально созданном для этого репозитории. В репозитории пакетов находятся артефакты релизов для пользователей. Docker Hub, GitHub Release Pages, PyPI и Maven Central — все это публичные репозитории. Многие компании выпускают и публикуют внутреннее ПО в частных репозиториях.

Репозитории пакетов позволяют сделать артефакты релиза (еще одно выражение для обозначения развертываемого пакета) доступными к развертыванию. Репозитории также играют роль архивов — там вы можете найти артефакты прошлых релизов, которые пригодны для отладки, отката или поэтапного развертывания. Содержимое пакетов записывается и открыто для просмотра. Удобный поиск позволяет находить зависимости, информацию о версии, а также дату публикации релиза: при устранении неполадок подобная информация имеет огромное значение. Репозитории релизов созданы с учетом требований развертывания, так что новый релиз могут скачивать одновременно тысячи пользователей.

Системы управления версиями, такие как Git, тоже могут использоваться в качестве репозитория релизов. Такой подход, например, в Go: вместо общего репозитория пакетов зависимости там выражаются через Git URI (обычно репозитории GitHub).

Репозиториями релизов могут быть и системы VCS, хотя они для этого не предназначены. VCS имеют не так много полезных функций поиска

и развертывания и не подходят для больших развертываний, так как могут часто перегружаться. Когда одни и те же устройства систем управления версиями будут обрабатывать запросы на переход от разработчиков, запросы развертывания и предоставления инструментов, это начнет отражаться на развертываниях в продакшене. Если вы обнаружите, что делаете релиз на основе VCS, убедитесь в способности системы справиться с нагрузкой. Совместное использование одной системы как для выпуска, так и для развертывания может стать причиной множества проблем использования, потому что требования к развертыванию и разработке ПО сильно отличаются. Разработчики делают частые небольшие коммиты и чуть реже — отладку.

Развертывание проверяет код на многих устройствах одновременно. На производительность могут влиять требования к развертыванию и то, какие инструменты использует разработчик, особенно если применяется одно физическое устройство или один и тот же репозиторий.

Оставляйте релизы неизменными

После публикации никогда не изменяйте или не перезаписывайте выпускаемый пакет. Неизменность релизов гарантирует то, что все приложения, работающие с определенными версиями, будут полностью идентичны. Идентичные пакеты релиза позволяют разработчикам понимать и оценивать код, лежащий в основе приложения, а также то, как он должен себя вести. Пакеты с номером версии, которые впоследствии изменяются, ничем не лучше пакетов без указания версий.

Публикуйте релизы часто

Публикуйте версии как можно чаще. Чувство безопасности, которое дают редкие релизы, обманчиво. Вам кажется, что большой разрыв между релизами помогает полностью протестировать изменения, однако на практике все иначе: частые релизы помогают создать более стабильное программное обеспечение, которое в случае обнаружения ошибок намного проще исправить. За каждый цикл публикуется меньше изменений и исправлений, и риск возникновения ошибок в каждом релизе

очень маленький. Когда ошибка попадает в среду эксплуатации, для отладки требуется выполнить небольшое количество изменений. При этом разработчики все еще помнят свой код, что значительно упрощает и ускоряет исправление ошибки.

Программное обеспечение с автоматизированной публикацией и развертыванием пакетов должно выполнять выпуск после каждого коммита. Для больших программных продуктов, развертывание которых представляет некоторую сложность, сбалансируйте частоту релизов со стоимостью выпуска, развертывания и обслуживания и со скоростью принятия ПО пользователями.

Относитесь к планированию релизов серьезно

С помощью *планирования релизов* определяется, как часто будет выпускаться ПО. У некоторых проектов есть весьма предсказуемый временной график с выпуском обновлений ежеквартально или ежегодно. Другие проекты выпускаются после завершения внедрения и проверки определенных функций. Некоторые проекты выпускаются тогда, когда этого захочет команда разработчиков. Очень часто внутренние системы предприятий публикуют релизы при каждом коммите. Вне зависимости от вида или стиля выпуска четко определите расписание релизов. Опубликуйте расписание и сообщайте пользователем о новых релизах.

Публикуйте журнал изменений и примечаний к релизу

Журналы изменений проекта и примечания к релизу помогают пользователям и вашей команде поддержки понимать, что именно находится в выпуске. В *журналах изменений проекта* перечислены все обработанные тикеты и коммиты, сделанные в релизе. Для автоматизации создания журнала изменений отслеживайте все изменения в сообщениях о завершении транзакции или метках системы отслеживания проблем.

Примечания о релизе — это список новых функций и исправлений ошибок, находящийся в выпуске. Обычно журналы изменений просматривает служба поддержки и разработчики, а примечания о релизе читают конечные пользователи.

ПРОЦЕСС ВЫПУСКА APACHE FOUNDATION

Apache Software Foundation (ASF) предоставляет рекомендации и ресурсы для проектов с открытым исходным кодом. Если вы хотите реальный пример, то лучше всего рассмотреть руководство по выпуску ASF.

Для запуска каждого релиза проекта ASF назначается релиз-менеджер. Релизы включают в себя пакет исходного кода и бинарный пакет. Релиз-менеджеры помечают артефакты с помощью криптографического ключа — так пользователи могут убедиться в том, что загруженные пакеты точно принадлежат Apache. Для обнаружения повреждений данных также добавляются контрольные суммы. Релизы включают в себя файлы LICENSE и NOTICE с объявлением различных лицензий и авторских прав на программное обеспечение; исходные файлы содержат заголовки лицензий.

Затем релиз-менеджер «вырезает» кандидата на релиз. Сначала создаются пакеты, после чего члены управляющего комитета проекта (project management committee, PMC) голосуют за принятие или непринятие кандидата на релиз. Предполагается, что члены PMC перед голосованием проверят окончательные артефакты: правильность контрольных сумм, все подписи, а также то, что ПО проходит приемочные тесты. После принятия артефакты публикуются на странице https://downloads.apache.org/. После выпуска релиз-менеджер делает оповещение в списке рассылки Apache, а также обновляет сайт проекта, добавляя примечания о релизе, журналы изменений проекта и новую документацию.

Для того чтобы увидеть процесс выпуска полностью, посетите страницу https://www.apache.org/dev/#releases/, а также страницу выпуска Apache Spark, где вы сможете ознакомиться с подробными инструкциями (https://spark.apache.org/release-process.html).

Этап развертывания

Развертывание программного обеспечения — это процесс получения пакетов ПО там, где оно должно работать. Механизмы развертывания отличаются друг от друга: например, развертывание мобильных приложений будет отличаться от развертывания ПО для ядерных реакторов. Однако при всех различиях в основе лежат одни и те же принципы.

Автоматизируйте развертывание

Выполняйте развертывание ПО с помощью готовых скриптов, а не вручную. Автоматизированное развертывание является предсказуемым процессом, так как поведение скриптов можно воспроизвести и контролировать версиями. Операторы могут предсказать поведение при развертывании в случае ошибки.

Скрипты менее склонны делать ошибки, нежели люди, и они избавляют разработчиков от желания добавлять в систему изменения или вручную копировать пакеты во время процесса развертывания. Очень сложно изменять состояние существующей машины. Два разных развертывания одного и того же ПО могут привести к непоследовательному поведению, которое впоследствии будет трудно отладить.

Развитая автоматизация приводит к *непрерывной доставке*. При непрерывной доставке люди исключены из процесса. Упаковка, тестирование, развертывание и даже выгрузка полностью автоматизированы. Развертывание выполняется с определенной частотой — например, каждый день, каждый час или постоянно. Благодаря непрерывной доставке команды могут быстро публиковать новые функции для пользователей, а также получать от них обратную связь. Успешный процесс непрерывной доставки требует использования автоматизированного тестирования (см. главу 6), а также автоматизированных инструментов и клиентской базы, способной обрабатывать быстрые изменения.

Мы рекомендуем автоматизировать процесс развертывания с помощью готовых инструментов. Очень просто начать работать с пользовательскими скриптами развертывания, однако достаточно быстро они становятся громоздкими и неудобными в использовании. Готовые решения, например Puppet, Salt, Ansible и Terraform, специально созданы для

автоматизации развертывания и интегрируются с уже существующими инструментами.

Вы можете обнаружить, что полностью автоматизировать процесс развертывания невозможно, — и в этом нет ничего страшного. Развертывание, зависящее от действий человека, никогда нельзя полностью автоматизировать. Просто сделайте все возможное, чтобы ограничить проблемные задачи, и автоматизируйте то, что получается.

Делайте развертывания атомарными

Обычно скрипт установки состоит из нескольких шагов. Не думайте, что каждый шаг при каждом выполнении будет успешным: вполне реальны ситуации, когда заканчивается место на диске, а компьютеры в самый неподходящий момент перегружаются или сообщают пользователю, что у того нет необходимых прав доступа к определенным файлам. Частично развернутое приложение может привести к сбою будущих развертываний, поэтому выполняйте атомарное развертывание. Частично развернутое приложение никогда не должно заменять предыдущее успешное развертывание. У вас также должна быть возможность установить один и тот же пакет на один и тот же компьютер несколько раз, даже если предыдущие процессы установки были неудачными.

Самый простой способ сделать развертывание атомарным — установить ПО не в то место, где установлена старая версия. Ничего перезаписывать не надо! После установки пакета вы увидите, что ярлык или ссылка могут атомарно переключаться. У установки пакета в другое место есть и еще одно преимущество: откатить изменения будет намного проще, просто снова указав старую версию. В некоторых случаях на одном устройстве могут одновременно работать разные версии одного и того же ПО!

Проводите независимое развертывание приложений

Одной из распространенных проблем в ПО с большим количеством взаимодействующих приложений или служб является порядок развертывания, когда развертывание одного приложения зависит от обновления другого приложения. Для выполнения обновления разработчики

просят операторов развернуть одно приложение перед другим или перевести несколько систем в автономный режим. Избегайте создания очереди развертывания. Требования к порядку развертываний замедляют процесс, так как приложения должны ждать друг друга. Когда обновления двух приложений взаимосвязаны, требования к порядку чаще всего приводят к возникновению конфликта.

Создавайте приложения, которые могут развертываться независимо друг от друга. Программное обеспечение, не зависящее от порядка развертывания, должно быть совместимо и с предыдущими, и с последующими версиями. К примеру, протоколы связи обязаны взаимодействовать как со старыми, так и с новыми версиями. Подробнее о совместимости — в главе 11.

Если зависимости от порядка развертывания не удается избежать, используйте методы выгрузки, описанные ниже. Развертывание с отключенными изменениями и их последующая активация в определенном порядке будут проходить быстрее и легче, чем принудительное выполнение порядка развертывания.

РАЗВЕРТЫВАНИЕ С ПОМОЩЬЮ WIKI

Раньше в сети LinkedIn веб-сервисы выпускали вручную. Разработчики и инженеры собирались вместе перед релизами и выясняли, какие сервисы и изменения настроек нужно развернуть; при этом на странице wiki была записана вся информация о развертывании.

Релиз-менеджеры разбивали сервисы, которые предстоит развернуть, на несколько этапов. Разработчики не особо контролировали совместимость, так что некоторые сервисы нужно было развертывать раньше других. В некоторых случаях сервисы вызывали друг друга, из-за чего появлялась циклическая зависимость. В одном развертывании было больше 20 этапов.

Однажды вечером, когда планировалось развертывание, все вошли в канал IRC-чата, чтобы наблюдать за процессом. Инженеры, отвечающие за надежность сервисов, копировали артефакты на

рабочие устройства, перезапускали сервисы, запускали сценарии миграции базы данных... Развертывание шло шаг за шагом, и вот уже наступило утро.

Ситуация складывалась не лучшим образом. Развертывание оказалось медленным и дорогостоящим, так как его нужно было выполнять вручную. Автоматизация, выходящая за рамки обычных сценариев, была очень сложной. Сбой при развертывании одного сервиса мог привести к сбою всего развертывания. Разработчикам приходилось вручную проверять развертывание перед тем, как переходить к следующему этапу. Процесс развертывания шел весьма напряженно и утомительно.

В конце концов в LinkedIn запретили применять порядок развертывания. Сервисы должны были поддерживать независимое развертывание, иначе их не разрешалось публиковать для пользователей. Из-за этого запрета у разработчиков появилось много работы: пришлось вносить изменения во все сервисы, инструменты, тесты и процессы развертывания. Однако в итоге процесс развертывания был полностью автоматизирован. Разработчики могли выпускать ПО в удобном для них темпе — даже по несколько раз в день. Инженерам, отвечающим за надежность сервисов, больше не нужно было следить за развертыванием, так что теперь все работники спали на несколько часов больше.

Этап выгрузки

Как только новый код пройдет процесс развертывания, вы можете его включить (выгрузить). Полноценный переход на новый код — очень рискованный шаг. Никакое количество проведенных тестирований не сможет устранить возможность появления ошибок, а выгрузка кода сразу для всех пользователей может привести к сбою. Вместо этого лучше внедрять изменения дозированно с дальнейшим отслеживанием показателей их работоспособности.

Существует множество методов выгрузки: флаги функций, шаблон Circuit Breaker («Автоматический выключатель»), тайный запуск (dark launch), канареечный выпуск и сине-зеленое развертывание. *Флаги*

функций позволяют контролировать то, какой процент пользователей получает доступ к одной ветке кода по сравнению с другой. Шаблон *Circuit Breaker* при возникновении проблем автоматически переключает ветви кода. *Тайный запуск*, *канареечный выпуск* и *сине-зеленое развертывание* позволяют одновременно запускать несколько развернутых версий ПО. При правильном использовании эти методы снижают риск возникновения опасных изменений. Однако не увлекайтесь использованием слишком изощренных стратегий выгрузки, так как они значительно усложняют работу. Операторы и разработчики должны одновременно поддерживать несколько версий кода и отслеживать, какие функции включены, а какие — выключены. Используйте сложные стратегии выгрузки только при работе с большими изменениями.

Контролируйте выгрузки

Отслеживайте такие показатели работоспособности, как частота ошибок, время отклика и потребление ресурсов во время работы нового кода. Контроль может осуществляться как вручную, так и автоматически. Расширенные конвейеры развертывания автоматически развертывают изменения для множества пользователей, а также откатывают изменения согласно статистике. Но даже в полностью автоматизированном процессе люди должны следить за статистикой и ходом выгрузки. Решения об увеличении и снижении темпов также принимают люди, которые занимаются анализом логов и метрик.

Заранее определитесь с общими показателями метрик. *Индикаторы уровня обслуживания* (service level indicator, SLI), подробно рассматриваемые в главе 9, представляют собой отслеживание показателей работоспособности сервиса для обнаружения ухудшения работы. Подумайте, что именно вы хотели бы видеть в логах и метриках в качестве подтверждения корректности внесенных изменений. Убедитесь в том, что нужные вам действия на самом деле реализуются.

Помните: ваша работа не завершается после выполнения коммита, и она все еще не закончена после развертывания кода. Не начинайте праздновать, пока логи не покажут, что ваши изменения работают исправно.

Увеличивайте темпы с флагами функций

Флаги функций, иногда называемые *переключателями функциональности* или *разбиением кода*, позволяют разработчикам контролировать выпуск нового кода для пользователей. Код обернут оператором `if`, который проверяет флаг (установленный динамическим сервисом или статической конфигурацией) для определения того, какую ветвь кода следует запустить.

Такие флаги могут быть булевыми значениями включения-выключения, списками разрешений, функциями линейного изменения на основе процентов или даже небольшими функциями. Булево значение переключает доступ к функции для всех пользователей. В списках разрешений указаны функции для определенных пользователей. Функции линейного изменения позволяют разработчикам постепенно включать функцию для разных пользователей. Сначала функцию проверяют на тестовых учетных записях компании, затем пробуют на реальном клиенте, а после проводят пошаговый выпуск на основе процентов. Функция динамически оценивает флаги на основе входных параметров, которые передаются во время запросов.

Код с флагами функций, который может изменять состояние, требует отдельного внимания. Базы данных не очень часто контролируются флагами функций. Новый и старый код постоянно работают с одними и теми же таблицами. Код должен иметь прямую и обратную совместимости. При отключении функций состояние не должно пропадать. Любые изменения состояния, произошедшие в период отключения-включения функции для пользователя, должны учитываться. Некоторые изменения, например изменения базы данных, не подходят для постепенного развертывания и должны постоянно отслеживаться. По возможности изолируйте данные флагов функций, проводите тестирование кода во всех состояниях и пишите скрипты для очистки данных тех функций, которые пришлось откатить.

Обязательно удаляйте флаги, которые больше не используются или достигли предела использования. Код, содержащий слишком много флагов функций, труден для понимания и даже может вызывать ошибки. Например, отключение ранее активной функции, которая долгое время находилась во включенном состоянии, может привести к полному

хаосу. Работа с флагами функций требует дисциплины. Чтобы не забыть удалить флаги, советуем создавать тикеты. Как и при рефакторинге, удаляйте флаги постепенно.

Флаги функций иногда могут использоваться для A/B-тестирования — измерения поведения пользователей при внедрении новой функции. A/B-тестирование подходит в том случае, если все пользователи сгруппированы статистически значимым образом. Проводите A/B-тестирование, только если система флагов создает для вас тестовые сегменты, а тестирование проводит специалист по обработке и анализу данных.

Защитите код с помощью автоматического выключателя

Большинством флагов функций управляют люди. Автоматические выключатели — это особый тип флагов функций, контролируемых операционными событиями, такими как резкое увеличение времени отклика или нештатные ситуации. Такие выключатели обладают уникальными характеристиками: они двоичные (вкл./выкл.), постоянные и автоматизированные.

Автоматические выключатели используются для предотвращения снижения производительности. Если пороговое значение времени отклика превышено, то определенные функции автоматически отключаются. Отключение может также произойти, если в логах наблюдается аномальное поведение — нештатная ситуация или изменение уровня детализации журнала.

Они также используются для защиты от неустраняемых серьезных повреждений. Приложения, в которых совершаются необратимые действия, например отправка сообщения или перевод денег с банковского счета, используют автоматические выключатели, если непонятно, нужно ли дальше продолжать действие. Так же могут себя защитить и базы данных: они переключаются в режим «только для чтения». При обнаружении ошибок с диском многие базы данных делают это автоматически.

Проводите развертывание версий параллельно

Новые версии сервисов можно развертывать параллельно со старыми. Пакеты могут находиться на одном компьютере, но могут разворачиваться и на отдельных устройствах. Параллельное развертывание позволяет увеличивать нагрузку постепенно: риски снижаются, а в случае нештатной ситуации действие можно отменить и откатить назад. Часть входящих вызовов сервиса переносится на новую версию с помощью переключателя, подобного флагам функций, однако этот переключатель располагается на входе в приложение — обычно это балансировщик нагрузки или прокси. Канареечное и сине-зеленое развертывания — два наиболее распространенных способа параллельного развертывания.

Канареечное развертывание используется для сервисов, которые занимаются обработкой большого объема трафика и развертываются на многих устройствах. Новая версия сначала развертывается на ограниченном количестве устройств, и на нее перенаправляется небольшое количество пользователей. На рис. 8.6 показано, что версия 1.1 канареечного развертывания получает около 1 % всего входящего трафика. Как канарейки в шахтах (их использовали для обнаружения опасных газов), канареечное развертывание предназначено для раннего предупреждения в новых версиях приложения. С вышедшими из строя «канарейками» работает малая часть пользователей, которых в случае обнаружения ошибок можно быстро вернуть к старой версии.

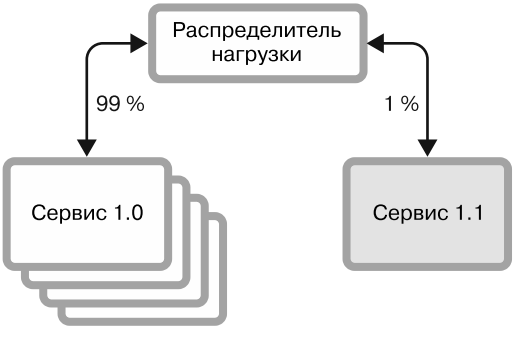

Рис. 8.6. При канареечном развертывании балансировщик нагрузки направляет часть входящего трафика в новое развертывание

При *сине-зеленом развертывании* запускаются две разные версии приложения: активная и пассивная. На рис. 8.7 показан пассивный кластер (синий) с версией 1.0, а также активный кластер (зеленый) с версией 1.1. Развертывание новой версии выполняется в пассивной среде. Когда процесс завершен, трафик перенаправляется на новую версию: таким образом, новая версия становится активной, а старая — пассивной. Как и при канареечном развертывании, если в новой версии есть некоторые проблемы, то весь трафик можно снова направить на старую версию. В отличие от канареечного развертывания, трафик направляется атомарным образом, а синие и зеленые среды практически идентичны. Когда релиз считается стабильным, пассивная среда удаляется из облачной среды.

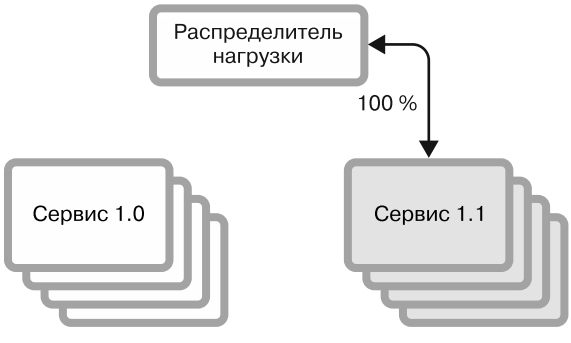

Рис. 8.7. При сине-зеленом развертывании сервис 1.0 сохраняется в качестве запасного варианта на случай сбоя сервиса 1.1

Сине-зеленое развертывание полезно тогда, когда трафик нельзя разделить на подмножества или когда параллельное выполнение разных версий невозможно. В отличие от канареечного развертывания, здесь каждая среда должна обрабатывать весь пользовательский трафик. В аварийных сценариях, когда всех пользователей нужно перенести из некорректно работающей системы, возможность быстро настроить и запустить параллельную системную среду чрезвычайно важна.

Как и флаги функций, параллельное развертывание, взаимодействующее с базой данных и кэшем, требует к себе особого внимания. Версии

приложения не должны конфликтовать друг с другом. Кроме того, должна быть обеспечена прямая и обратная совместимость всех схем. Более подробно эта тема обсуждается в главе 11.

Запуск в тайном режиме

Флаги функций, канареечное и сине-зеленое развертывания позволяют развертывать код для множества пользователей и предоставляют механизмы смягчения последствий при возникновении непредвиденных проблем. Запуск в тайном режиме, который иногда называется *затенением трафика (traffic shadowing)*, открывает код для реального трафика, не делая его видимым для пользователей. Даже если скрытый код плохой, пользователь об этом не узнает.

Программное обеспечение, запускаемое в тайном режиме, по-прежнему включено и код все еще вызывается, однако результаты отбрасываются. Тайные запуски помогают разработчикам узнать о поведении ПО в эксплуатационной среде с минимальным воздействием на пользователя. Используйте такие запуски каждый раз, когда выпускаете очень сложные изменения. Это особенно полезно при проверке систем миграции.

При запуске в тайном режиме прокси-блок приложения располагается между трафиком в реальном времени и самим приложением. Прокси-блок дублирует запросы для тайной системы. Ответы на идентичные запросы от обеих систем сравниваются, а различия между ними записываются. Пользователям отправляются ответы только от выпущенной в эксплуатацию системы. Данный метод позволяет операторам наблюдать сервис в условиях реального трафика без участия клиентов. Говорят, что система находится в режиме тайного считывания (dark reads), когда ей отправляется только считываемый трафик и никакие данные не изменяются. Если трафик отправляется в систему, а система использует независимое хранилище данных, то система находится в режиме тайной записи (dark writes). На рис. 8.8 показаны оба режима.

Поскольку операции для одного запроса выполняются дважды — в эксплуатируемой системе и в тайном режиме, — следует задуматься о том,

как избежать ошибок, связанных с дублированием данных. Трафик в тайной системе не нужно учитывать в пользовательской аналитике, и таких ситуаций, как двойной учет данных, быть не должно. Вы можете пометить запросы, которые не нужно учитывать, путем изменения заголовков — так можно обозначить тайный трафик. Некоторые сервисные сети, например Istio, а также шлюзы API, к примеру Gloo, имеют встроенную поддержку этих операций.

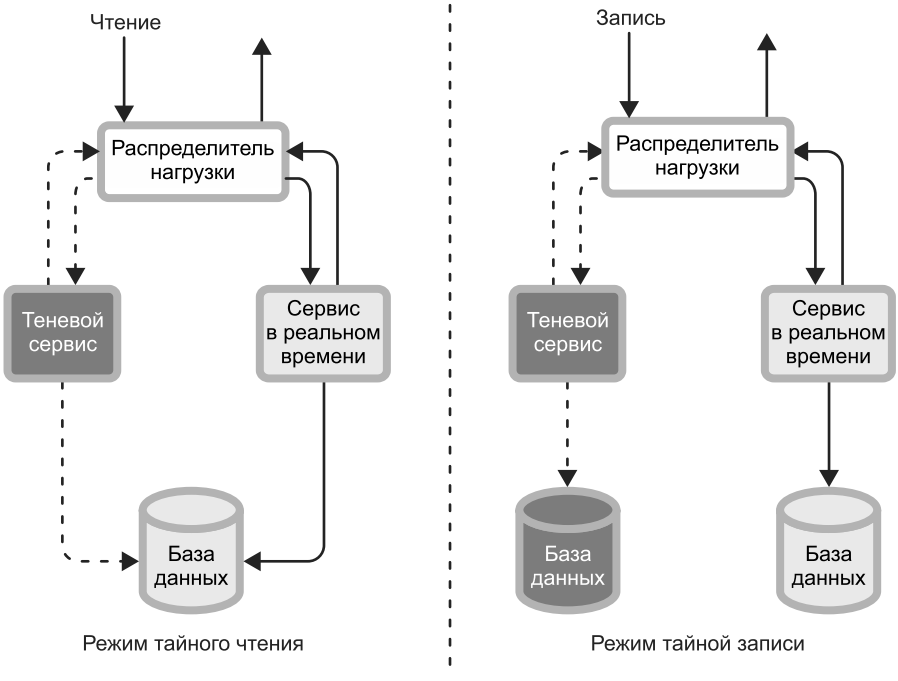

Рис. 8.8. Режимы тайного чтения и записи

Тайный запуск позволяет делать множество интересных вещей. Инструмент Diffy с открытым исходным кодом отправляет тайный трафик трем экземплярам бэкенд-сервиса: два из них содержат рабочую версию кода, а один — нового кандидата на релиз. Diffy сравнивает отклики новой и старой версий и выявляет различия, после чего сравнивает отклики двух старых версий для выявления не определенного ранее шума. Это позволяет Diffy автоматически выявлять будущие различия и удалять ошибочное срабатывание.

НУЖНО БОЛЬШЕ ТАЙНОЙ РАЗРАБОТКИ

Из-за серии организационных изменений, произошедших в период быстрого роста, один из ключевых сервисов Twitter перестал справляться с нагрузкой. У сервиса накопился внушительный технический долг, однако ему продолжали поступать новые запросы на дополнительные функции. Каждое изменение, отправляемое в рабочую среду, было рискованным из-за сложностей с тестированием сервиса, и время от времени возникали критические ситуации. Инженер, работавший над всем этим в одиночку, был просто в ужасе, так как на отладку и исправление проблем уходило много времени. Частые ошибки замедляли выход новых изменений, что приводило к росту бэклога функций. Сочетание всех этих проблем не позволяло привести процесс разработки к нормальному ритму, увеличивало нагрузку и усложняло выполнение рефакторинга — и все это на фоне непрекращающегося пополнения кодовой базы новыми функциями.

После очередного организационного нововведения к команде Дмитрия присоединился инженер с описанным выше сервисом. Оценив сложившееся положение, тимлид объявил ситуацию недопустимой: команде *пришлось* прекратить разработку функций и заняться решением проблемы технического долга. У инженера, работающего с системой, было множество идей по поводу улучшения сервиса, однако даже самое небольшое изменение могло привести к неожиданным проблемам в эксплуатируемом коде.

Было решено в первую очередь сосредоточиться на стабилизации системы, находящейся в эксплуатации, и применить для этого тайную запись. Инженеры использовали инструмент Diffy: в течение двух недель сравнивали объекты в потоке, а не ответы HTTP. У сервиса теперь был вариант обхода: команда могла позволить себе делать новую версию неограниченное количество времени, при этом анализируя всевозможные непредвиденные различия в данных. Можно было запускать изменения в тайном режиме, получать из пользовательского трафика данные о пограничных состояниях, фиксировать их, затем добавлять тесты для решения возникающих проблем, а после повторять попытку.

Набор тестов увеличился, цикл разработки ускорился, а у команды появилось больше уверенности в выпусках. Инженер, ответственный за сервис, рассказывал, что они почувствовали себя так, будто гора свалилась с плеч. Сейчас изменения в кодовой базе происходят быстрее, будь то рефакторинг, повышение производительности или даже добавление новых функций. Применение тайной записи кардинально изменило рабочий процесс в лучшую сторону.

Что следует и чего не следует делать

Следует	Не следует
По возможности использовать магистральную разработку и непрерывную интеграцию	Выпускать пакеты без указания версий
Использовать инструменты VCS для управления ветками	Размещать настройки, схемы, изображения и языковые пакеты вместе
Всегда работать вместе с командами по выпуску и использованию ПО: они помогут правильно настроить процессы для вашего приложения	Перекладывать всю ответственность на релиз-менеджеров и группы по разработке проекта
Обязательно публиковать примечания к релизу и журналы изменений	Использовать VCS для распространения ПО
Постоянно уведомлять пользователей о публикации нового релиза	Изменять релизы после их выгрузки
Использовать инструменты для автоматизации развертывания	Выгружать ПО без контроля результатов
Всегда постепенно вносить изменения с флагами функций	Создавать зависимость от порядка развертывания
Обязательно использовать автоматические выключатели для предотвращения серьезных неполадок в приложениях	
Обязательно использовать теневой трафик и запуск в тайном режиме при работе с серьезными изменениями	

Повышение уровня

В книге Эммы Джейн Хогбин Уэстби (Emma Jane Hogbin Westby) *Git for Teams* (O'Reilly Media, 2015) дается подробная информация о стратегиях ветвления, полезная даже в том случае, если вы не работаете с Git.

В книге Джеза Хамбла и Дэвида Фарли (Jez Humble, David Farley) *Continuous Delivery*[1] (Addison-Wesley Professional, 2010) подробно

[1] *Хамбл Дж., Фарли Д.* Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий.

рассматриваются темы, затронутые в этой главе. Если на подготовку релиза у вас уходит много времени, обязательно почитайте эту книгу. Для получения краткой информации обратитесь к главе 8 книги *Site Reliability Engineering* (O'Reilly Media, 2016) от Google, где подробно рассматривается разработка релизов.

В книге Майкла Нейгарда (Michael T. Nygard) *Release It!*[1] (Pragmatic Bookshelf, 2018) речь идет об операциях, которые мы обсуждаем в главах 8 и 9, и много внимания уделяется паттернам проектирования ПО для операций — об этом мы говорили в главе 4. Книгу настоятельно рекомендуем разработчикам, имеющим дело с веб-сервисами.

Хорошим бесплатным ресурсом с рекомендациями по доставке ПО является *Builder's Library* от Amazon. Библиотека располагается по адресу https://aws.amazon.com/builders-library/. Там вы можете найти записи о непрерывной доставке, автоматическом развертывании и откатах.

[1] *Нейгард М.* Release it! Проектирование и дизайн ПО для тех, кому не всё равно. — СПб.: Питер, 2021.

9 Дежурство

Во многих компаниях есть дежурные инженеры, которые первыми реагируют на различные инциденты, идет ли речь о вопросах по эксплуатации или о специальных запросах поддержки. Отделив процесс решения углубленных задач от вопросов по сопровождению ПО, бо́льшая часть команды может сосредоточиться на разработке, в то время как дежурные инженеры будут обрабатывать непредсказуемые проблемы эксплуатации и задачи поддержки. Хороших дежурных инженеров высоко ценят как коллеги, так и руководство, и они могут быстро сделать карьеру благодаря возможности получать новые знания и налаживать отношения.

В этой главе рассматриваются базовые знания и навыки, необходимые для работы дежурным инженером. Мы объясним, как работают дежурные инженеры, и расскажем о важных навыках их работы. А также подробно рассмотрим инцидент из реальной практики — так вы получите пример того, как нужно обрабатывать подобные ситуации. Работа дежурным инженером может стать причиной выгорания, поэтому в конце главы мы поговорим о том, почему не стоит брать на себя роль героя.

Даже если вы работаете в компании, где нет работы для дежурного инженера, не пропускайте эту главу. Знание сути такой работы поможет вам в любой экстренной ситуации.

Схема работы дежурного

Дежурные инженеры работают посменно согласно расписанию. Продолжительность работы дежурным может варьироваться от одного дня до 1–2 недель, и в этой роли по очереди выступают все квалифицированные разработчики. Молодым специалистам, которые только начали свою работу и еще не обладают необходимыми навыками, часто предлагают стать напарником старшего коллеги на несколько смен, чтобы понять основы.

При составлении расписания дежурств обычно предусматривают работу основного и дополнительного дежурного разработчика. Дополнительный дежурный работает только в том случае, если основной не может откликнуться на запрос. (Само собой, основные дежурные, постоянно перекладывающие работу на дополнительных, в коллективах долго не задерживаются.) В некоторых компаниях предусмотрена сложная структура реагирования на инциденты: сначала о проблеме узнает команда поддержки, затем информация передается инженерам по сопровождению и уже потом — команде разработчиков.

Большую часть рабочего времени дежурные тратят на обработку запросов поддержки, например, просматривают и составляют отчеты об ошибках, снимают вопросы о работе и использовании ПО. Дежурные рассматривают запросы, сортируют их и отвечают на самые срочные.

Однако каждый дежурный когда-нибудь сталкивается с серьезным сбоем системы (критической проблемой рабочего ПО). О любом инциденте дежурный узнает из сообщения системы автоматизированного контроля или от инженера службы поддержки.

Когда происходит инцидент, дежурные разработчики получают экстренное уведомление. Раньше для этого использовались пейджеры, но сегодня оповещения поступают по электронной почте, в чаты, по телефону и пр. Обязательно внесите в контакты номер службы оповещения, если вы, как и мы, не отвечаете на звонки с незнакомых номеров!

Все дежурные смены должны начинаться и заканчиваться передачей информации. Предыдущий дежурный рассказывает о произошедших

инцидентах и предоставляет контекстную информацию по еще не закрытым задачам. Если дежурный добросовестно относится к работе, передать информацию для него не составит труда.

Важные навыки дежурного

Работа дежурного может быть напряженной. К счастью, для работы с инцидентами и запросами поддержки не требуется получения отдельных навыков. Дежурный должен быть всегда доступен и обязан следить за возникновением инцидентов. Необходимо уметь правильно расставлять приоритеты, чтобы выполнять наиболее важные дела в первую очередь. Особое значение имеет умение четко и понятно объяснять задачи и действия. При решении проблемы необходимо отмечать, что и как делается. В этом разделе мы дадим вам несколько советов, которые помогут повысить эти навыки.

Будьте доступны

«Ваше лучшее качество — ваша доступность». При работе дежурным вам предстоит отвечать на запросы и реагировать на происшествия. Не игнорируйте просьбы и не пытайтесь прятаться. Всегда будьте готовы к тому, что к вам могут обратиться, и примите тот факт, что во время дежурства вы не сможете полностью погрузиться в свою привычную работу.

В некоторых случаях дежурные инженеры должны находиться рядом с компьютером в режиме 24/7. Это не значит, что нужно всю ночь просидеть возле монитора в ожидании инцидента. Это означает, что дежурного можно вызвать и он сможет отреагировать на запрос даже в ущерб своим личным планам. В больших компаниях используется принцип «следуй за солнцем», когда дежурными становятся разработчики из разных часовых поясов. Узнайте о требованиях к дежурным инженерам в своей команде и старайтесь не попадать в такие ситуации, когда нельзя ответить на вызов.

Однако доступность не означает, что при получении запроса вы должны немедленно бросать все дела. Во многих случаях достаточно просто подтвердить получение запроса, а также указать примерное время, когда вы сможете им заняться: «Сейчас я помогаю другому человеку. Могу я заняться вашим запросом через 15 минут?» От дежурного инженера ожидают быстрого ответа, а не быстрого решения проблемы.

Будьте внимательны

Информация, относящаяся к работе дежурного, может поступать из разнообразных источников, таких как чат, электронная почта, звонки, сообщения, тикеты, журнал событий, средства мониторинга или даже встречи. Обращайте внимание на эти источники, чтобы быть достаточно компетентными при устранении проблем.

Заранее читайте замечания к релизу, просматривайте чаты и электронную почту, в которых может появиться оперативная информация, например, о развертывании ПО или изменении настроек. Периодически проверяйте чаты, в которых команды по эксплуатации сообщают об изменениях. Читайте протоколы собраний и уделите особое внимание выдержкам из скрам-совещаний с данными по текущим инцидентам и обслуживанию в течение дня. Постоянно держите информационные панели (дашборды) открытыми в фоновом режиме или на соседнем мониторе, чтобы контролировать основные показатели системы при работе в нормальном состоянии. Когда произойдет инцидент, вы увидите, какие графики выглядят странно.

Создайте список всех необходимых ресурсов на случай экстренной ситуации, в котором будут прямые ссылки на важные информационные панели, перечень задач для ваших сервисов, инструкции по доступу к журналу событий, ссылки на важные чаты, а также руководство по устранению ошибок. Рекомендуется создать отдельную папку для информации по дежурству и постоянно ее обновлять. Поделитесь созданным списком со всеми членами команды, чтобы ваши коллеги тоже могли его использовать или изменять.

Расставляйте приоритеты

Сначала работайте над первоочередными задачами. По мере завершения или блокировки задач продвигайтесь по списку от более приоритетных задач к менее приоритетным. Во время работы вам будут поступать новые запросы и оповещения. Быстро сортируйте поступающие запросы: либо отложите их, либо, если ситуация экстренная, начните действовать. Если новый запрос приоритетнее работы, которую вы делаете в данный момент, но не является критическим, постарайтесь завершить текущую работу или доведите ее до логической точки.

Одни запросы поддержки являются срочными, в то время как другие можно обработать в течение недели. Если вы не можете определить срочность запроса, спросите себя, что от него зависит, и таким образом определите приоритетность. В случае несогласия с заказчиком по поводу приоритетности запроса обсудите проблему с руководителем.

Работа дежурных классифицируется по приоритету: П0, П1, П2 и т. д. Распределение работ по приоритетам помогает определять срочность задач. Названия категорий и их значения различаются от компании к компании, но в категории П0 находятся самые объемные задачи. Например, определить приоритет можно по списку приоритетов поддержки Google Cloud (https://cloud.google.com/support/docs/best-practice#setting_the_priority_and_escalating/):

- П1: критическое воздействие — сервис непригоден для использования;
- П2: высокое воздействие — сервис практически непригоден для использования;
- П3: среднее воздействие — сервис частично пригоден для использования;
- П4: низкое воздействие — сервис пригоден для использования.

Индикаторы уровня обслуживания, *цели* и *соглашения* помогают расставить приоритеты в работе. Такие индикаторы уровня обслуживания (service level indicators, SLI), как, например, частота ошибок, задержка запросов или количество запросов в секунду, позволяют оценить работоспособность приложения. Цели уровня обслуживания (service level objectives, SLO) определяют целевой уровень качества обслуживания. Если частота ошибок принимается в качестве индикатора SLI, то, например, желаемым значением SLO может быть значение менее 0,001 %. Соглашение об уровне обслуживания (service level agreement, SLA) — это соглашение о том, что происходит, когда требования SLO не соблюдаются. (За нарушение соглашения об уровне обслуживания компания обычно возвращает заказчику деньги. Иногда заказчик может расторгнуть контракт с компанией.) Изучите SLI, SLO и SLA для своих приложений. Индикаторы SLI укажут на самые важные метрики. SLO и SLA помогут расставить приоритеты для инцидентов.

Общайтесь четко и по делу

При решении оперативных задач определяющее значение имеет четкая коммуникация. Если все происходит очень быстро, недопонимание может стать причиной серьезных проблем. Чтобы общаться четко и по делу, будьте прямолинейны, вежливы, отзывчивы и обстоятельны.

Под шквалом большого количества срочных задач и постоянных вопросов разработчики быстро становятся раздражительными. В этом нет ничего странного — такова человеческая натура. Отвечая на запросы, будьте вежливы и терпеливы. Вас могут отвлекать уже десятый раз за день, но помните, что для ожидающего вашего ответа человека это первое взаимодействие с вами.

Обменивайтесь короткими фразами. Вам может показаться, что говорить прямо неудобно и невежливо, но на самом деле это не так. Краткость гарантирует, что ваше сообщение прочитают и поймут правильно. Если вы не знаете ответа, так и скажите. Быстро реагируйте на поступающие запросы.

Вы не обязаны в своих ответах сразу давать решение. Сообщите человеку, что увидели его запрос, и убедитесь в том, что правильно поняли суть проблемы.

> **Спасибо за обращение!** *Хочу уточнить: сервис входа в систему выдает код отклика 503 от сервиса профилей? Вы же говорите не про авторизацию? Это два разных сервиса, но их названия очень похожи, и их часто путают.*

Иногда статус нужно обновлять. В обновлениях вы должны указывать то, что выяснили или сделали с момента последнего обновления, а также то, что собираетесь предпринять дальше. Каждый раз, проводя обновление, сообщайте примерное время следующего обновления:

> *Я проверил сервис входа в систему. Не вижу никаких сильных отклонений в частоте ошибок, но сейчас посмотрю логи и вернусь. Ждите новой информации через час.*

Фиксируйте свою работу

Записывайте, что делаете во время работы. Каждая задача, над которой вы работаете во время дежурства, должна быть отмечена в системе отслеживания ошибок или в журнале дежурств команды. Фиксируйте свой прогресс во время дежурства, записывая каждое обновление в отдельный тикет. В нем укажите шаги, благодаря которым вы уменьшили или устранили проблему: так у вас будет уже записанное решение на случай, если такая же или похожая проблема опять возникнет.

Отслеживание прогресса работы позволит вам вспомнить, на чем вы остановились, если вам придется прерваться. Следующий дежурный также сможет просмотреть состояние текущей работы, прочитав ваши записи, и любой, у кого вы попросите помощи, сумеет оценить ситуацию из записей в журнале. Зафиксированные запросы и инциденты складываются в базу с возможностью поиска, которой могут пользоваться другие дежурные.

Некоторые компании используют чат-каналы, например в мессенджере Slack, для оказания поддержки и разрешения проблем. Чат — отличное место общения, однако с архивом переписки в чате в дальнейшем трудно работать, поэтому не забывайте резюмировать информацию в заявке или в документе. Не бойтесь перенаправлять запросы в службу поддержки по соответствующим каналам. Говорите четко и по делу: «Я собираюсь заняться этой задачей прямо сейчас. Можно ли открыть тикет, чтобы моя работа учитывалась при оценке нагрузки на нашу службу поддержки?»

По завершении задачи закрывайте ее, чтобы висящие выполненные заявки не отвлекали дежурных и не искажали показатели их нагрузки. Перед закрытием заявки уточните у заказчика и убедитесь, что проблема точно решена. Если заказчик не отвечает, сообщите, что из-за отсутствия ответа вы собираетесь закрыть заявку в течение 24 часов.

Всегда добавляйте метки времени в свои записи: они помогут операторам при устранении неполадок. Информация о том, что сервис был перезапущен в 13:00, очень поможет, если клиенты начнут сообщать о задержках в 13:05.

Обработка инцидентов

Обработка инцидентов — главная обязанность дежурного. Большинство разработчиков считают, что обработка инцидентов состоит в устранении проблемы, возникшей при эксплуатации. Конечно, решить проблему нужно, однако при появлении критического инцидента главная задача — смягчить последствия и восстановить работу. Вторая цель — собрать информацию для анализа того, как и почему инцидент произошел. Определить причину инцидента и устранить основную проблему — это третья цель.

Обработка инцидентов состоит из следующих пяти шагов.

1. **Сортировка.** Инженеры должны найти проблему, оценить ее серьезность и определить, кто именно может ее устранить.
2. **Координация.** Команды (и потенциальные заказчики) должны узнать об инциденте. Если дежурный не может сам решить проблему, он должен предупредить тех, кто может это сделать.
3. **Снижение рисков.** Инженеры должны как можно быстрее стабилизировать ситуацию. Снижение рисков не является долгосрочным решением: вы только откладываете решение проблемы. Чтобы снизить риски, можно переключиться в другую среду, откатить выпуск, отключить неправильно работающие функции или добавить аппаратные средства.
4. **Решение проблемы.** После снижения рисков у инженеров есть немного времени, чтобы подумать и поработать над решением проблемы. Инженеры продолжают рассматривать инцидент, определяют и устраняют основные проблемы. Инцидент исчерпывается после разрешения самой проблемы.
5. **Дальнейшие действия.** Происходит поиск первопричины инцидента. Если инцидент был серьезным, то проводится ретроспектива и создается постмортем. Создаются задачи для предотвращения основной причины (или причин) инцидента. Команды ищут любые пробелы в процессах, инструментах или документации проекта. Инцидент считается незавершенным до тех пор, пока не будут выполнены все поставленные задачи.

Перечисленные этапы могут показаться абстрактными. Для прояснения ситуации рассмотрим реальный инцидент и поговорим обо всех

этапах его обработки. Инцидент возникает в тот момент, когда данные не загружаются в хранилище. *Хранилище данных* — это база данных, предназначенная для подготовки отчетов и машинного обучения. Оно может стабильно работать благодаря частым обновлениям в системе обмена сообщениями. Коннекторы читают сообщения из системы потоковой передачи и вносят их в хранилище. Данные из хранилища используются всеми командами компании как для внутренних отчетов, так и для отчетов заказчиков, машинного обучения, отладки приложений и многого другого.

Сортировка

Чтобы определить приоритет проблемы, оцените возможный ущерб: сколько людей она затрагивает и какой вред приносит. Воспользуйтесь шкалой приоритетов, принятой в вашей компании, а также определениями SLO/SLA, чтобы присвоить проблеме правильный приоритет. Это можно выполнить с помощью индикаторов SLI и метрики, которая стала причиной оповещения.

РЕАЛЬНЫЙ ПРИМЕР

Когда в системе обмена сообщениями обнаруживаются данные, которых нет в хранилище данных, команда по сопровождению получает оповещение. Запускается первый этап обработки инцидента — *сортировка*, включающая в себя признание проблемы и определение ее влияния для правильного определения приоритета. Дежурный инженер подтверждает получение заявки и начинает исследовать проблему, чтобы правильно определить ее приоритет. Так как в оповещении написано, что в таблицах, которые используются для создания отчетов заказчиков, отсутствуют данные, проблема считается высокоприоритетной.

Этап сортировки подходит к концу. Инженер подтвердил заявку и определил приоритетность проблемы, но не пытался ее решить. Он просто посмотрел, какие таблицы затронула проблема.

Если вы не можете определить серьезность проблемы, обратитесь за помощью к другим специалистам. На этапе сортировки у вас нет времени доказывать, что вы можете справиться со всеми проблемами самостоятельно, — время очень ценно.

Сортировка не устраняет проблему. Пользователи будут испытывать неудобства все то время, пока вы боретесь с неполадками. Оставьте поиск причин для этапов снижения рисков и решения проблемы.

Координация

Координация начинается с поиска ответственного лица. Если инцидент имеет низкую приоритетность, его координирует сам дежурный. В случае серьезных инцидентов ответственность на себя берет *руководитель ликвидации инцидентов*. Он следит за тем, кто какую работу выполняет и на каком этапе сейчас решается проблема.

РЕАЛЬНЫЙ ПРИМЕР

Дежурный инженер переходит к этапу координации. Он отправляет в чат сообщение о том, что в данных таблиц для клиентов есть пробелы. Беглый осмотр показывает, что коннектор, загружающий данные в хранилище данных, работает корректно, а логи не сообщают о каких-либо сбоях.

Дежурный инженер просит помощи у разработчиков коннекторов, а также привлекает к работе инженера, имеющего опыт работы с коннекторами. Дежурный сообщает о каждом шаге менеджеру по технической поддержке, который сейчас выступает как менеджер инцидентов. Команда отправляет компании электронное письмо, уведомляя всех об отсутствии в хранилище данных для нескольких таблиц. Менеджер инцидентов работает с системой управления аккаунтами и выполняет действия для размещения уведомления в сервисе информирования клиентов.

Как только кто-то берет на себя ответственность, все должны быть уведомлены об инциденте. Свяжитесь со всеми, кто должен работать над этой проблемой. Заинтересованные стороны, например менеджеры по технической поддержке, продакт-менеджеры или специалисты службы поддержки, также должны знать об инциденте. Пострадавших пользователей нужно оповестить о случившемся через сервис информирования клиентов, по электронной почте, через Twitter и т. д.

В такой ситуации приходится вести множество параллельных бесед, что затрудняет контроль происходящего, — в данном случае сложно отследить статус проблемы. При работе с крупными инцидентами создается этакий командный пункт (war room) для облегчения коммуникации. Это виртуальные или реальные пространства, которые используются для координации реагирования на инциденты. К командному пункту присоединяются все заинтересованные стороны.

Отслеживайте письменное общение в определенном месте, например в чате или системе тикетов. Общение помогает всем работающим над проблемой контролировать прогресс, избавляет вас от необходимости постоянно отвечать на вопросы о состоянии, предотвращает дублирование работы и позволяет другим вносить полезные предложения. Перед тем как что-то сделать, поделитесь планами с коллегами. Сообщайте о своей работе, даже если работаете в одиночку: позже к вашей работе может кто-нибудь присоединиться и журнал очень пригодится. Кроме того, подробная запись поможет впоследствии восстановить хронологию.

Снижение рисков

На этом этапе вашей целью является снижение влияния проблемы. Снижение рисков — это не решение проблемы, а только уменьшение наносимого ей урона. На устранение проблемы может уйти много времени, в то время как на снижение рисков требуется пара мгновений.

Обычно снижение рисков инцидентов происходит путем отката выпуска программного обеспечения до последней корректной версии или отвлечения внимания от проблемы. В зависимости от ситуации снижение рисков может включать в себя отключение флага функции, удаление устройства из накопителя или откат развернутой службы.

РЕАЛЬНЫЙ ПРИМЕР

Когда уведомления отправлены, инженеры начинают заниматься снижением рисков. Принимается решение перезапустить коннектор, чтобы посмотреть, не освобождается ли он из тупика выполнения, однако проблема остается. Дамп стека показывает, что коннектор читает и десериализует сообщение. Машина, на которой работает коннектор, имеет загруженный на 100 % центральный процессор, поэтому инженерам кажется, что коннектор зависает на большом или поврежденном сообщении, из-за чего в процессе десериализации нагрузка ЦП возрастает до предела.

Инженеры пытаются как-то снизить риски и запускают второй коннектор с исправными потоками. Всего есть 30 потоков, но инженеры не понимают, в каких потоках передаются «плохие» сообщения. Для нахождения некорректно работающих потоков инженеры запускают бинарный поиск: организуется выполнение половины потоков, после чего набор корректируется в зависимости от поведения коннектора. Вскоре команда находит некорректно работающий поток. Коннектор перезапускается со всеми исправными потоками, и их табличные данные восстанавливаются. Влияние проблемы теперь ограничено одним потоком и одной таблицей.

В идеале программное обеспечение должно иметь runbook для решения проблем. *Runbook* — это пошаговая инструкция для решения распространенных проблем и выполнения действий вроде перезапуска или отката. Убедитесь в том, что знаете, где находятся runbook и руководство по поиску и устранению неисправностей.

Соберите все возможные данные, пока работаете над устранением проблемы. После снижения рисков мало шансов, что проблема повторится. Быстрое сохранение данных телеметрии, трассировки стека, дампов кучи, журналов событий и скриншотов информационных панелей поможет при отладке и анализе причин проблемы.

Пытаясь снизить риски, вы часто будете находить пробелы в метриках, наборах инструментов и настройках. Вы можете обнаружить, что там

отсутствуют важные показатели и стоят неправильные разрешения или что системы настроены некорректно. Помечайте каждый такой пробел — так вы упростите себе задачу при поиске и устранении неисправностей. На следующем этапе откройте тикеты и устраните обнаруженные пробелы.

Решение проблемы

После снижения рисков инцидент больше не является задачей с высоким приоритетом. Можно уделить время устранению неполадок и решению проблемы. В нашем примере приоритет был снижен после восстановления всех клиентских потоков коннектора, когда у инженеров появилось немного времени на исследование проблемы.

РЕАЛЬНЫЙ ПРИМЕР

Инженеры работают над решением проблемы. В одном потоке появилось некорректное сообщение, поэтому таблица хранилища данных для этого потока не получает данные.

Инженеры удаляют все исправные потоки из исходного коннектора, чтобы попытаться воспроизвести проблему. Теперь можно видеть сообщение о зависании коннектора, поэтому они вручную читают сообщение с помощью инструмента командной строки. Все кажется нормальным.

В этот момент на команду снисходит озарение: а как так получилось, что инструмент командной строки может десериализовать сообщение, а коннектор не может? Видимо, коннектор активирует какой-то код, который не использует инструмент командной строки, — например, причудливый десериализатор даты. Десериализатор даты определяет тип данных заголовка сообщения (это целое число, строка, дата и т. д.). По умолчанию инструмент командной строки не выводит заголовки сообщений. Инженеры повторно запускают инструмент с заголовком сообщения и узнают, что у некорректного сообщения в заголовке есть единственный ключ, но без значения.

Ключ к заголовку указывает на то, что заголовок сообщения вводится инструментом *управления производительностью приложения* (APM). Управление производительностью приложения находится внутри приложений и сообщает разработчикам о поведении приложения: об использовании памяти, загрузке ЦП и трассировке стека. Однако инженеры не знали, что APM-демон ставит заголовки во все сообщения.

Команда обращается за внешней поддержкой. Инженеры технической поддержки потоковой передачи сообщают, что в инструменте командной строки наблюдается ошибка: она не выводит заголовки сообщений, в которых имеется строка с нулевым байтом в конце. Инженеры считают, что в заголовках есть байты, из-за которых не совершается передача сообщений.

Команда принимает решение проверить эту теорию и отключает десериализацию заголовков в коннекторе. Данные для последней оставшейся без данных таблицы загружаются в хранилище данных. Теперь все таблицы обновлены, и начинается проверка качества данных. Команда уведомляет инженеров технической поддержки о решении проблемы, а также обновляет информацию в сервисе информирования клиентов.

На этапе решения проблемы сосредоточьтесь на том, что нужно решить в первую очередь, и начните работать над проблемами, обнаруженными на этапе снижения рисков. Большие технические проблемы отложите на следующий этап.

Используйте научный метод для устранения неполадок и технических проблем. В главе 12 книги *Site Reliability Engineering* от Google предлагается *гипотетико-дедуктивная* модель научного метода. Изучите проблему, определите причину, затем выполните проверку и исправьте. Если исправление помогло, то проблема устранится; если нет — проделайте всю работу с самого начала. В нашем примере команда применила этот научный метод в тот момент, когда появилась гипотеза, что у коннектора возникли определенные проблемы с десериализацией, из-за чего данные не доходили до таблицы. Инженеры просмотрели данные и провели эксперимент с бинарным поиском, после чего нашли некорректно работающий поток. Если бы у них ничего не получилось, им пришлось бы выдвигать новую гипотезу.

В идеальной ситуации вы можете поместить некорректно работающий элемент программы в своеобразный карантин и подробно изучить его поведение. В нашем примере с коннектором инженеры изолировали неисправный поток на отдельном коннекторе. На этапе решения проблемы вашей целью являются понимание проблемы и дальнейшая попытка ее воспроизвести. В этот момент используйте все данные, которые у вас есть: метрики, трассировки стека, журналы действий, дампы кучи, уведомления об изменениях, тикеты и каналы связи.

Как только у вас появится четкое представление о симптомах, диагностируйте проблему путем поиска причин. Диагностика — это поиск, так что для ее выполнения можно обратиться к алгоритмам поиска. Для небольших задач подойдет линейный поиск с проверкой компонентов от начала и до конца. При работе с большими системами используйте алгоритм типа «разделяй и властвуй» или бинарный поиск, который также называется *разделением пополам*. Найдите точку посередине стека вызовов и посмотрите, где находится проблема. Если проблема расположена выше середины по потоку, то выберите новый компонент на отрезке выше средней точки; если проблема находится ниже средней точки, сделайте наоборот. Продолжайте повторять, пока не найдете тот компонент, в котором возникает проблема.

Затем проверьте свою теорию. Проверка — это не исправление: вы еще не решаете проблему. Посмотрите, можете ли вы контролировать некорректное поведение. Вы можете его воспроизвести? Можете изменить настройки, чтобы устранить проблему? Если да, то у вас получилось найти причину. Если нет, значит вы устранили только одну из причин сбоя. Теперь вернитесь в начало, проверьте проблему и сформулируйте новый диагноз. Когда инженеры из примера решили, что они уточнили проблему до десериализации заголовка, они проверили свою теорию путем отключения десериализации заголовка в настройках коннектора.

После успешного завершения проверки гипотезы можно выбирать оптимальный вариант исправления проблемы. Вполне вероятно, что вам потребуется всего лишь изменить настройки. Часто исправление ошибки необходимо написать, протестировать и после запустить. Примените выбранное решение и получите подтверждение, что проблема устранена. Следите за метриками и журналами до тех пор, пока не убедитесь в том, что все в порядке.

Дальнейшие действия

РЕАЛЬНЫЙ ПРИМЕР

Менеджер по разработке, ответственный за коннектор, создает план *дальнейших действий*, а дежурный инженер пишет черновик постмортема и назначает дату его рассмотрения. В процессе написания постмортема открывается заявка на получение информации о том, почему APM использует заголовки сообщений и почему коннектор не может их десериализовать, а также почему инструмент командной строки не может выводить заголовки с нулевыми строками.

Инцидент — это серьезно, поэтому команде следует предпринять дальнейшие действия. Цель данного этапа проста: инженеры должны извлечь из случившегося полезные уроки и новые знания и научиться предотвращать повторение инцидента. Пишется и рассматривается постмортем, после чего открываются задачи для предотвращения нового инцидента.

ПРИМЕЧАНИЕ

Термин «постмортем» взят из области медицины, где проводятся и документируются вскрытие и обследование трупа. К счастью, в нашем случае все не так страшно. Альтернативным термином является «ретроспектива», но это слово мы используем для других действий постфактум, например для ретроспективы спринта. Подробнее об этом — в главе 12.

Дежурный инженер, работающий с инцидентом, пишет постмортем, в котором должна быть отображена следующая информация: что случилось, что было изучено и что нужно сделать для предотвращения повторения инцидента. Существует множество паттернов написания постмортема. Хорошим вариантом станет шаблон от Atlassian (https://www.atlassian.com/incident-management/postmortem/templates/). В нем

с использованием примеров описываются подготовка, ошибки, влияние, обнаружение, реагирование, откаты, временные шкалы, первопричины, извлеченные уроки и дальнейшие действия.

Важнейшим разделом постмортема является *анализ первопричины* (root-cause analysis, RCA). Данный анализ выполняется с помощью использования пяти «почему». Метод достаточно прост: постоянно спрашивайте «почему?». Возьмите любую проблему и спросите, почему она возникла. Когда получите ответ, спросите «почему?» еще раз. Продолжайте спрашивать до тех пор, пока не узнаете первопричину. Значение «пять» здесь очень примечательно — чтобы узнать первопричину большинства проблем, нужно сделать примерно пять шагов.

Проблема: из хранилища данных пропали данные

1. Почему? Коннектор не загружал данные в хранилище данных.
2. Почему? Коннектору не удавалось десериализовать входящие сообщения.
3. Почему? У входящих сообщений были некорректные заголовки.
4. Почему? APM вставлял в сообщения заголовки.
5. Почему? У APM такое поведение было установлено по умолчанию без ведома разработчика.

В данном примере первопричиной была случайная настройка заголовка сообщения APM.

ПРИМЕЧАНИЕ

> «Анализ первопричины» — термин довольно популярный, но вводящий в заблуждение. Очень редко инциденты возникают из-за одной проблемы. На практике пять «почему» могут привести к обнаружению нескольких причин. И в этом нет ничего странного — просто делайте пометки и документируйте всю информацию.

После того как постмортем написан, один из руководителей назначает дату обзорной встречи со всеми заинтересованными лицами. Обзор ведет автор постмортема, и участники внимательно слушают каждый раздел отчета. В ходе обсуждения автор документа добавляет туда недостающую информацию и новые задачи.

В стрессовых ситуациях очень легко выйти из себя и начать обвинять других людей. Постарайтесь предоставить конструктивную обратную связь. Деликатно укажите на области, которые стоит улучшить, однако не обвиняйте разработчиков или целые команды во всех проблемах. Высказывание «Питер не отключил заголовки сообщений» обязательно вызовет у человека чувство вины, в то время как высказывание «Изменения настроек заголовков сообщений не проходят ревью кода» определит область улучшения. Не делайте главной целью постмортема обвинение конкретных людей.

В хорошем постмортеме «решение» идет отдельно от обзорной встречи. Решение, то есть попытка выяснить, как лучше решить проблему, занимает очень много времени и отвлекает от главной цели постмортема — обсудить проблему и составить список задач. «У сообщения был некорректный заголовок» — это проблема; «Некорректные сообщения должны отправляться в очередь недоставленных сообщений» — это решение. Все решения должны происходить при определении дальнейших действий.

После встречи для обсуждения постмортема необходимо выполнить задачи, поставленные на этапе определения дальнейших действий. Если задачи закреплены за вами, то вам нужно поговорить с руководителем и с командой постмортема, для того чтобы правильно расставить приоритеты. Инцидент не должен закрываться до тех пор, пока не будут выполнены все поставленные задачи.

Изучение старых документов постмортема — отличный способ научиться чему-то новому. Некоторые компании даже публикуют постмортемы, из которых каждый желающий может узнать для себя много полезного. Дан Луу собрал целую коллекцию таких документов (https://github.com/danluu/post-mortems). Вы также можете попробовать найти группы чтения постмортемов в своей компании, когда команды собираются для

группового обзора документов постмортемов. Отдельные команды используют старые постмортемы для моделирования проблем при обучении инженеров.

Предоставление поддержки

Когда дежурные инженеры не заняты инцидентами, они обрабатывают запросы поддержки. Запросы поддержки поступают как от сотрудников компании, так и от клиентов и могут быть очень разными: от простого «Привет, не подскажете, как это работает?» до вопросов по поводу сложных проблем и устранения неполадок. Большая часть запросов представляет собой отчеты об ошибках, вопросы о логике функционирования приложения или технические вопросы о том, как использовать программное обеспечение.

Запросы в службу поддержки поступают по довольно стандартной схеме. Вы подтверждаете получение запроса и задаете дополнительные вопросы, чтобы убедиться в правильном понимании проблемы. Начав работать над проблемой, оцените, когда вы уже сможете что-то о ней сказать, например: «Я сообщу вам дополнительную информацию в 17:00». Начните разбираться в проблеме, в процессе оповещая человека, приславшего запрос. Придерживайтесь стратегий решения конфликтов и снижения риска, которые мы рассматривали ранее. Когда вы посчитаете, что проблема решена, попросите отправившего запрос подтвердить это. Закройте запрос. Ниже представлен пример:

[15:48] Сумит: Я получаю отзывы клиентов о долгой загрузке страниц.

[16:12] Джанет: Сумит, привет! Спасибо, что написал. Можно чуть подробнее? Ты можешь дать мне пару идентификаторов клиентов, сообщивших о проблеме, а также указать страницы, на которых возникла проблема? Наша информационная панель не показывает никаких серьезных задержек.

[17:15] Сумит: Идентификаторы клиентов 1934 и 12305. Страницы: главная страница операций (/ops) и диаграмма APM (/ops/apm/

dashboard). Согласно отчетам, загрузка страниц занимает больше 5 секунд.

[17:32] Джанет: Отлично, спасибо! Завтра к 10 утра жди дополнительную информацию.

[8:15] Джанет: Думаю, я знаю, в чем причина. Вчера днем проводилось техническое обслуживание базы данных, на которой как раз работает информационная панель АРМ. Это отразилось на главной странице операций, так как на ней тоже показываются сведения. Техническое обслуживание закончилось в районе 20:00. Клиент больше не сталкивается с проблемой долгой загрузки?

[9:34] Сумит: Отлично! Только что получил обратную связь от нескольких клиентов. Страницы загружаются намного быстрее.

Этот пример иллюстрирует многие методы, описанные в разделе «Важные навыки дежурного». Джанет, дежурный инженер, *обращает внимание* на сообщение и *выделяет на него время*. На первое сообщение она отвечает в течение 30 минут. Джанет *четко озвучивает* интересующие ее вещи, задает уточняющие вопросы для понимания проблемы и ее влияния на других, чтобы правильно *расставить приоритеты*. Джанет указывает предположительное время, когда у нее будет достаточно информации для того, чтобы разобраться с проблемой. Как только Джанет кажется, что проблема решена, она *подводит итог проделанной работы*: описывает причину проблемы, а затем просит обратную связь для подтверждения того, что вопрос закрыт.

Предоставление поддержки может отвлекать, так как ваша «настоящая» работа — это программирование. Воспринимайте работу в поддержке как возможность научиться чему-то новому. Вы увидите, как ПО вашей команды используется на практике, что иногда оно выдает ошибки или путает пользователей. Отвечая на запросы поддержки, вы познакомитесь с частями кода, с которыми до этого вам не удавалось поработать: вам придется хорошенько подумать и поэкспериментировать. Вы заметите закономерности, вызывающие проблемы, которые помогут вам создавать более качественное программное обеспечение в будущем. Выполнение задач поддержки сделает из вас хорошего инженера. К тому же вы сможете помочь другим людям, улучшить свою репутацию и завести новые знакомства. Быстрые, четкие и качественные ответы поддержки очень ценятся.

Не пытайтесь быть героем

В этой главе мы много говорили о том, что не нужно уклоняться от обязанностей дежурного инженера и запросов поддержки. Однако не впадайте в другую крайность — не берите на себя слишком много. Выполняя работу дежурного инженера, вы можете испытать большое удовлетворение, когда коллеги будут благодарить вас за помощь, а руководители — хвалить за активное участие в решении важных вопросов. Однако серьезная нагрузка и большой объем работы могут привести к выгоранию.

Для некоторых инженеров переход в «пожарный» режим работы становится рефлекторным по мере того, как они набираются опыта. Талантливый инженер-«пожарный» может быть находкой для команды: все будут знать, что в случае какой-то проблемы они просто к нему обратятся и он все сделает сам. Однако в зависимости от такого инженера нет ничего хорошего. «Пожарные», которых привлекают для решения каждой проблемы, превращаются в постоянных дежурных. А переработки и большая ответственность приводят к выгоранию. Эти инженеры сталкиваются и с проблемами в своей основной работе, так как их постоянно от нее отвлекают. Команды, полагающиеся на «пожарных», не смогут набраться собственного опыта и не приобретут необходимых навыков. Это может привести даже к такой ситуации, когда команда не начнет заниматься решением серьезной проблемы, ожидая, что сейчас придет тот самый инженер и все решит.

Если вам кажется, что вы единственный человек в команде, способный разобраться с проблемой, или вы постоянно участвуете в устранении каких-либо инцидентов, не будучи при этом дежурным, вы превращаетесь в «героя». Поговорите с управляющим или техническим руководителем проекта о том, как найти баланс и привлечь к работе обученных людей. Если в вашей команде есть «герой», узнайте, чему вы можете у него научиться и какую часть его работы можете взять на себя. Дайте этому человеку знать, что вы тоже пытаетесь помочь: «Джен, спасибо. Вообще я хотел бы попробовать разобраться с этим самостоятельно, набраться опыта... Если я не справлюсь, то попрошу у тебя помощи позже, хорошо?»

Что следует и чего не следует делать

Следует	Не следует
Добавить номер службы поддержки в свои контакты	Игнорировать предупреждения
Использовать категории приоритетов, SLI, SLA и SLO для определения приоритета инцидента	Пытаться искать и устранять неисправности во время сортировки
Выполнять этапы сортировки, координации, снижения рисков и решения проблем и определять дальнейшие действия для критических инцидентов	Оставлять проблему без внимания во время поиска ее первопричины
Использовать научный метод для устранения ошибок	Искать виновного во время постмортемов
Задать пять вопросов «почему» при работе с инцидентом	Бояться закрывать запросы в службу поддержки, на которые вы не получили ответа
Подтверждать получение запроса поддержки	Спрашивать у людей, обращающихся в службу поддержки, какой приоритет у запроса, — вместо этого спросите о влиянии проблемы
Устанавливать временные рамки и периодически обновлять информацию	Пытаться быть героем, который должен исправить все ошибки
Подтверждать факт устранения проблемы перед закрытием запроса	
Отправлять запросы поддержки по соответствующим каналам связи	

Повышение уровня

Пять этапов реагирования на инциденты взяты из статьи *Increment* с названием *What Happens When the Pager Goes Off?* (https://increment.com/on-call/when-the-pager-goes-off/). В статье приводится большое количество цитат и деталей того, как разные компании работают с инцидентами.

Иногда разработчикам приходится самим определять желаемое значение количественной оценки работы сервиса. Если в какой-то момент

вы понимаете, что отвечаете и за это, то мы рекомендуем вам прочитать главу 4 книги Google *Site Reliability Engineering*.

В главах 11, 13–15 книги *Site Reliability Engineering* рассматриваются дежурство, реагирование на чрезвычайные ситуации, урегулирование инцидентов и проведение постмортемов. Мы в нашей книге поделились самыми важными сведениями для начинающих инженеров, однако в издании Google вы сможете найти более подробную информацию по данным темам.

10 Процесс технического проектирования

Когда требуется внести изменения, большинство начинающих инженеров сразу же приступают к написанию кода. Вначале такой метод погружения подходит, однако однажды вы столкнетесь с настолько объемной задачей, что этот вариант не сработает и вам придется подумать о техническом проекте.

Процесс технического проектирования позволяет согласовать проект крупных изменений. Работа над проектом делится на два вида деятельности: уединенную работу по технике высокой продуктивности (deep work) и групповые обсуждения. Исследования, мозговой штурм и написание документации относятся к работе по технике высокой продуктивности. Обсуждение проекта и проектной документации — это групповые обсуждения. Результатом становится проектный документ.

В этой главе мы рассмотрим расширенную версию процесса проектирования для работы со значительными изменениями. Процесс может казаться медленным и пугающим: некоторые инженеры до сих пор помнят, как сложные многосоставные процессы проектирования шли наперекосяк. Однако можно уменьшить масштаб для небольших изменений, а проблему изложить в трех предложениях, а не в эссе на несколько абзацев. Разделы шаблона проектирования могут оказаться неактуальными, несколько раундов обратной связи — ненужными, а обзор от других команд — невостребованным. Со временем вы разовьете

ощущение оптимального количества усилий для решения сложных проблем. Но поначалу лучше подстраховаться: спросите совета у руководителя и расскажите ему о своих планах. При правильном выполнении участие в работе по техническому проектированию и руководство ею очень ценятся и хорошо вознаграждаются.

Конус процесса технического проектирования

Разработка программного обеспечения не является линейным процессом от проведения исследований и мозгового штурма до составления документации и утверждения проекта. Она больше похоже на спираль, в которой чередуются индивидуальная работа в технике высокой продуктивности и совместная работа, и на каждом новом этапе проект улучшается (рис. 10.1).

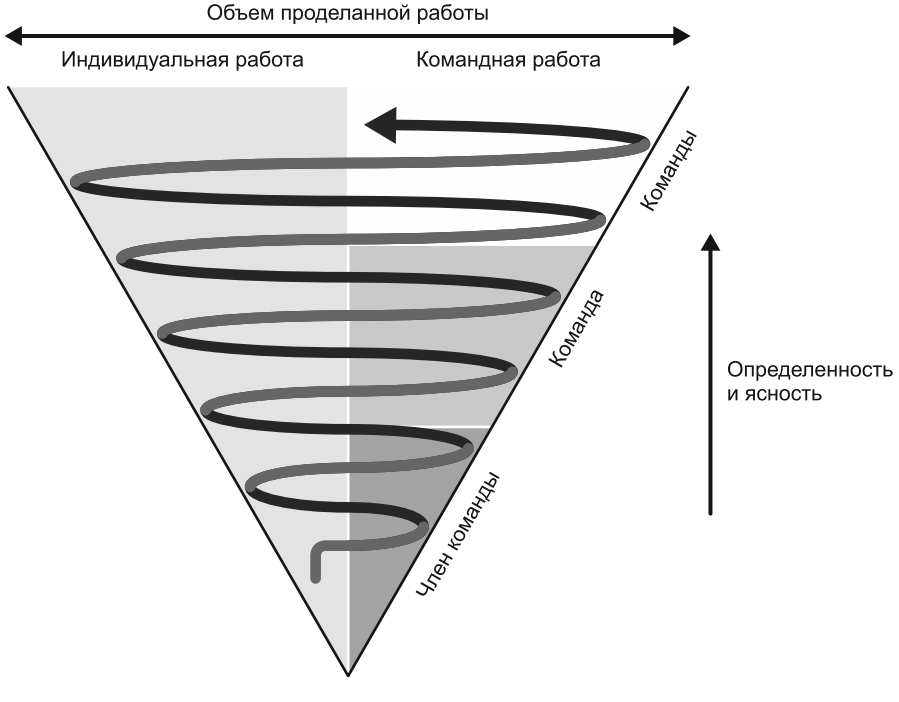

Рис. 10.1. Спираль процесса проектирования

С каждой итерацией проектный документ становится более подробным. Вместе с уверенностью автора в решении растет и объем работы, которую необходимо выполнить, — эксперименты, доказательства, тестирование. Со временем увеличивается и количество людей, с которыми проводились консультации по поводу проекта.

Работа начинается с основания конуса, когда вы еще не определились с областью возникновения проблемы, требованиями и возможными решениями. На этом этапе у вас еще нет решения, в котором вы точно будете уверены.

Во время исследований вы переключаетесь между самостоятельной работой и групповыми обсуждениями с членами команды или экспертами в области, которую исследуете. Вы проводите мозговой штурм и экспериментируете. Цель состоит в том, чтобы, получая информацию, повышать уверенность в выбранном пути решения и ясность процесса.

В результате все исследования, эксперименты и мозговой штурм приведут вас к одному из вариантов проекта. После проверки варианта теми, с кем вы работали до этого, вы пишете технический проект. По мере его написания обнаруживается, что вы многого еще не знаете. Вы создаете несколько прототипов, на которых проверяете варианты, и получаете ответы на свои вопросы. В конце концов выбираете один из жизнеспособных вариантов. Проводите дополнительные исследования и спрашиваете мнение экспертов. Затем оформляете черновик проектного документа.

Стрелка в конусе начинает закручиваться вверх. Теперь вы уверены, что точно понимаете область проблемы. Ваши прототипы придают вам уверенности в том, что вы выбрали правильное решение. У вас также есть проектное предложение, которое вы уже готовы показать другим. Вы обсуждаете его со своей командой, получая обратную связь. Затем рассматриваете, обсуждаете и обновляете проектный документ.

Сейчас вы находитесь на верхнем уровне в широкой части конуса. Вы достаточно вложились в проект и уверены в своем подходе. Вы распространяете свой вариант проекта по всей компании: служба

безопасности, служба эксплуатации, смежные команды и архитекторы тоже должны знать о внесенных изменениях. Подобное информирование требуется не только ради получения обратной связи, но и для того, чтобы у сотрудников обновилась их ментальная модель работы системы.

После одобрения проекта начинается его реализация, однако проект все еще не закончен. На этапе реализации вы столкнетесь со множеством сюрпризов. Если в код вашей команды будут внесены большие изменения, вам потребуется обновить свой проектный документ.

Размышляем о проекте

В основе воронки процесса проектирования лежит исследование. Прежде чем приступать к разработке проекта, вам необходимо понять область возникновения проблемы и требования к итоговому проекту. При проведении исследования нужно размышлять, экспериментировать и дискутировать. Исследование — это одновременно и индивидуальная, и командная работа.

Определите проблему

Вашей первой задачей будут определение и понимание проблемы (или проблем), которую вам нужно решить. Чтобы решить проблему, важно определить ее границы: может быть, вы обнаружите, что никакой проблемы нет или что ее не нужно решать.

Начните с общения с заинтересованными сторонами и спросите их мнение о том, в чем именно заключается проблема. Заинтересованной стороной могут быть ваш руководитель, члены команды, продакт-менеджеры — не все они будут воспринимать проблему одинаково.

При общении с заинтересованными сторонами лучше всего описать проблему своими словами. Спросите, совпадает ли ваше понимание проблемы с их пониманием. Если проблем несколько, поинтересуйтесь, какие из них кажутся более приоритетными.

«Что произойдет, если мы не решим эту проблему?» — важный вопрос. Когда получите ответ, уточните, является ли приемлемым результат выполнения. Вы узнаете, что на самом деле многие проблемы не нужно решать.

Получив достаточно замечаний о проблеме от разных заинтересованных сторон, постарайтесь обобщить информацию и определить проблему. Не принимайте на веру описание проблемы и всегда обдумывайте, что вам говорят. Особое внимание обратите на область проблемы: что в нее включено и что можно было бы включить. Не пытайтесь рассматривать все проблемы, предложенные заинтересованными сторонами. Не бойтесь отбрасывать изменения с низким приоритетом. Напишите формулировку проблем — и той, которая находится в рамках обозначенной области, и той, которая выходит за рамки. Подтвердите свое понимание с помощью обратной связи.

Первоначальный запрос функциональной возможности может выглядеть так:

> *Менеджеры отдела снабжения хотят видеть номера каталогов и номера страниц для каждого товара, отображающегося на странице запросов. Отображение номера каталога упростит повторный заказ товара в тот момент, когда все запасы товаров подходят к концу. Мы можем использовать подрядчиков для сканирования всех каталогов, а также можем воспользоваться моделью машинного обучения для сопоставления отсканированных изображений с описаниями товаров в базе данных.*

Этот запрос вызывает множество вопросов у продакт-менеджера.

- Как менеджеры отдела снабжения размещают заказы?
- Может ли товар отображаться в нескольких каталогах?
- Как пользователи работают с сервисом без этой функции?
- Какие есть проблемные моменты у данного решения?
- Какой из проблемных моментов оказывает наибольшее влияние на бизнес?

Ответы на эти вопросы могут привести к тому, что придется пересмотреть постановку задачи:

> *Менеджерам отдела снабжения нужен простой способ сортировки товаров, когда они заканчиваются. В настоящее время они поддерживают сопоставление идентификаторов товарных позиций, которые мы генерируем, с именем поставщика и единицей складского учета в таблицах Excel и перекрестными ссылками на них. Переход с нашего ПО на Excel для поиска и заказа товаров у поставщика происходит очень медленно и приводит к ошибкам.*
>
> *У одной единицы складского учета может быть несколько поставщиков. Менеджерам желательно иметь доступ ко всем из них для минимизации затрат. В настоящее время из-за ограничения таблиц они отслеживают лишь одного поставщика для каждого товара.*
>
> *Менеджеры отдела снабжения определяют свои приоритеты в следующем порядке: точность данных, время размещения заказа, минимизация стоимости заказа.*
>
> *Несколько продавцов предлагают онлайн-каталоги, и примерно половина продавцов предлагают онлайн-покупки.*

Уточненное описание проблемы приводит вас к принятию совершенно другого решения, отличного от начального. Инженер сосредоточивается на проблеме и расставляет приоритеты. Предложенные решения, например работа подрядчиков или использование машинного обучения, должны быть отвергнуты. В список потенциальных решений проблемы включается информация об онлайн-каталогах.

Проводите собственные исследования

Не нужно сразу переходить от определения проблемы к оформлению «финального» проекта. Рассмотрите альтернативные решения, компромиссы, проведите исследования. Созданный вами проект должен быть не первой пришедшей в голову, а лучшей идеей.

В интернете можно найти множество полезной информации. Посмотрите, как подобные проблемы решались другими разработчиками.

У многих компаний есть блоги, в которых описывается то, как они решают проблемы или внедряют функциональные возможности. В то время как блоги компаний по большей части являются пиар-акцией, описывая упрощенную архитектуру и опуская сложные ситуации, записи в блогах остаются отличным способом получить представление о том, что в подобной ситуации делают ваши коллеги по профессии. Вы можете написать авторам в социальных сетях или по электронной почте — так у вас появится возможность получить больше деталей, опущенных в записях блога.

Отраслевые конференции — это еще один вариант, где можно проверить свои догадки. Обычно презентации и записи выступлений загружаются в интернет. Не забывайте и об академических исследованиях и технической документации: используйте список литературы в конце статей, в котором можно найти много полезных материалов для дальнейшего изучения.

Поговорите с экспертами в той области, которую изучаете: в первую очередь обратитесь к экспертам из вашей компании, однако не ограничивайтесь только своими коллегами. Вы узнаете, что множество авторов блогов или статей, а также авторы докладов стремятся рассказать о своей работе больше. Просто не забывайте о том, что нельзя разглашать конфиденциальную информацию о работе и компании при общении с третьими лицами.

Наконец, мыслите критически. Не все, что вы читаете в интернете, — хорошая идея. Особенно распространенной ошибкой является перенос найденного в интернете решения похожей, но не идентичной вашей проблемы без малейших изменений. Но ведь при всем сходстве ваши проблемы — это не проблемы Google (даже если вы работаете в Google).

Проводите эксперименты

Пишите черновик кода, запускайте тесты и экспериментируйте. Пишите черновики API и частичные реализации. Запускайте тестирования производительности или пользовательские тестирования A/B, чтобы понять, как ведут себя системы и пользователи.

Эксперименты добавят уверенности в ваших идеях и решениях, приведут к компромиссам в проекте и помогут определить область проблемы. В ходе экспериментов вы сможете представить, как пользователи будут использовать ваш код. Поделитесь своим кодом с членами своей команды и получите обратную связь.

Не привязывайтесь к экспериментальному коду. Он предназначен для составления представления об идее, после чего должен быть удален или переписан. Сосредоточьте все усилия на визуализации и проверке своей идеи. Не пишите тесты и не тратьте время на улучшение экспериментального кода: вам нужно получить как можно больше информации за небольшое количество времени.

Не торопитесь

Для создания качественного проекта нужно творческое настроение. Не ждите, что у вас получится создать весь проект за один раз. Делайте перерывы, меняйте обстановку и наберитесь терпения.

Над проектом нужно долго думать. Вы не можете заниматься проектом периодически по 15 минут: выделите несколько часов, в течение которых сможете полностью сосредоточиться на вопросе. Пол Грэм (Paul Graham) написал эссе под названием *Manager's Schedule, Maker's Schedule* (http://www.paulgraham.com/makersschedule.html). В нем рассказывается, как ценно непрерывное время работы для «творцов», то есть именно для вас. Определите, в какое время суток у вас самая сильная концентрация, и выделите это время в своем расписании. Крис предпочитает работать после ланча в полной тишине, а у Дмитрия пик продуктивности приходится на раннее утро. Найдите время, когда чувствуете себя лучше всего, и ничем его не занимайте.

Перерывы — это убийцы работы в технике высокой продуктивности. Закройте все чаты, отключите электронную почту и уведомления на телефоне. Если планируете сменить обстановку, убедитесь, что все необходимые инструменты — блокнот, бумага, ручка — у вас с собой.

Вы не должны «проектировать» все свое свободное время. Вашему мозгу тоже необходимо отдыхать. Делайте перерывы, давая себе возможность

отвлечься и расслабиться. Прогуляйтесь, заварите чай, почитайте или напишите что-нибудь, нарисуйте диаграммы.

Проектирование — это круглосуточная работа, поэтому нужно запастись терпением. Ваш мозг всегда обдумывает идеи. Интересные мысли будут приходить к вам в течение всего дня (и даже когда вы спите).

Нерегламентированный подход к проектированию не означает, что вы сможете работать вечно. У вас есть сроки поставки, когда вы должны встретиться с членами команды. Проект-спайки — это хороший способ справиться с противоречием между творческой работой и сроками. *Спайк* — это термин из экстремального программирования, использующийся для описания ограниченного по времени исследования. Определение спайк-задачи в спринте дает вам пространство для углубленных размышлений, позволяя не беспокоиться о других задачах.

Написание проектных документов

Проектные документы дают возможность четко сформулировать ваши идеи. Процесс написания документации проекта структурирует ваше мышление и выявляет слабые места идеи. Однако документирование идей не всегда проходит так, как запланировано. Чтобы создавать полезные проектные документы, сосредоточьтесь на важных изменениях, не забывайте о целях и пользователях, практикуйтесь в написании документов и обновляйте их.

Последующие изменения документов

Не каждое изменение требует разработки проектной документации и тем более формального процесса рассмотрения проекта. В вашей организации могут быть собственные рекомендации по этому поводу, но при их отсутствии воспользуйтесь нижеприведенными тремя критериями, которые помогут решить, нужно или нет создавать проектную документацию:

- проекту потребуется как минимум месяц инженерных работ;
- изменение отразится в будущем на расширении и обслуживании ПО;
- изменение сильно влияет на другие команды.

С первым критерием все понятно: если на реализацию проекта уйдет какое-то время, то лучше заранее потратить определенное время на создание проектного документа. Так вы убедитесь в том, что все делаете правильно.

Второй требует некоторых пояснений. Какие-то изменения внедряются быстро, однако имеют долгосрочные последствия. Например, это может быть внедрение части инфраструктуры — уровня кэширования, сетевого прокси или системы хранения данных. Это может быть новый общедоступный API или что-то, касающееся безопасности. Существуют быстрые способы добавить решение той или иной проблемы, однако такие решения, как правило, приводят к неочевидным последствиям в будущем. Написание проектного документа и его рассмотрение дадут вам возможность найти вероятные проблемы и решить их. Обзор проектной документации также гарантирует, что вся команда будет понимать, почему добавляются те или иные изменения, что позволит избежать неожиданностей в будущем.

Для изменений, которые сильно влияют на другие команды, тоже нужно создавать проектную документацию. Команды должны понимать, что вы делаете, чтобы впоследствии дать вам обратную связь и принять внедренные изменения. Изменения с широким охватом требуют проведения ревью кода или рефакторинга, что может повлиять на другие проекты. Из вашего проектного документа команды узнают о будущих изменениях.

Знайте, зачем вы пишете

Вам может казаться, что проектные документы рассказывают другим о том, как работает программный компонент. Однако основная функция проектного документа выходит за рамки просто документирования. Проектная документация — это инструмент, помогающий вам думать, получать обратную связь, оповещать вашу команду, обучать новых инженеров и управлять планированием проекта.

Написание документации позволит вам узнать то, о чем вы не знали до этого (просто поверьте нам). Заставляя себя записывать идеи по проектированию, вы исследуете область проблемы и оформляете свои

мысли. Это бурный процесс, но его стоит пройти, чтобы лучше понять свой проект и его компромиссы. Создав проектную документацию, вы сможете разложить все мысли по полочкам, и процесс обсуждения проекта станет более продуктивным.

Проще всего попросить обратную связь для проекта, оформленного в письменном виде. Такие документы можно распространить между разными людьми, и они смогут читать их и отвечать вам в свободное время. Даже когда обратная связь минимальна, распространение проектного документа позволяет информировать команду.

Распространение знаний о проекте поможет другим представить себе точную модель того, как работает система. В дальнейшем команда придет к лучшим решениям по проекту и его внедрению, а дежурные инженеры правильно поймут поведение системы. Инженеры могут использовать проектные документы своих коллег и учиться чему-то новому.

Проектная документация будет особенно полезна для новых членов команды. Без документации инженерам придется самим изучать код, рисовать схемы и узнавать информацию от старших инженеров. Чтение проектных документов — намного более эффективное занятие.

В конце концов, руководители используют проектную документацию для планирования проектов. Во многих проектных документах отмечены ключевые моменты и этапы реализации, необходимые для завершения проекта. Наличие записей по определенному проекту облегчает координацию, если проектом одновременно занимаются несколько команд.

Учимся писать

Инженеров, считающих себя плохими писателями, перспектива заняться сочинительством может напугать. Однако не стоит бояться: письмо — это навык, который развивается практикой. Пользуйтесь любой возможностью попрактиковаться в письме — пишите проектные документы, электронные письма, комментарии к ревью кода: главное, старайтесь писать четко и по делу.

Четкий стиль написания облегчит вам жизнь. Письмо — это способ передачи информации с некоторыми потерями: вы озвучиваете коллегам идею, но они в своей голове воссоздают ее не так, как видите вы. Хороший текст позволяет точно передать нужную информацию. Умение писать четко и по делу также способствует росту карьеры. Хорошо и понятно написанный документ быстро распространяется в больших группах людей, в том числе и среди руководителей, — а на хороших авторов всегда обращают внимание.

Перечитайте написанное с точки зрения целевой аудитории: неважно, что вы понимаете написанный текст, главное, чтобы его понимала ваша аудитория. Будьте лаконичны. Чтобы понять точку зрения аудитории, читайте то, что писали другие люди. Подумайте, как бы вы изменили их текст: что можно убрать, а что добавить. Поищите в своей компании хороших авторов документации и попросите их оставить обратную связь о документе, который написали вы. Больше материалов по написанию документов можно найти в разделе «Повышение уровня» в конце главы.

Разработчики, не являющиеся носителями языка, иногда пугаются письменного общения. Разработка ПО — это международный бизнес. Очень редко бывает так, что все в команде говорят на своем родном языке. Однако не позволяйте языковому барьеру помешать вам писать проектную документацию. Не беспокойтесь об ошибках в грамматике: важнее четкое и ясное выражение мыслей.

Поддерживайте актуальность проектной документации

До этого мы говорили о проектных документах как об инструменте для представления и разработки проекта от начала до реализации. Как только вы принимаетесь за реализацию, проектная документация превращается из предложений в документы, описывающие, как реализуется программное обеспечение, — это *живые (динамические) документы*.

При переходе от предложения к документу часто возникают две распространенные ошибки. Первая ошибка — документ предложения забывается и больше не обновляется. Но разработка может отойти от первоначального плана, из-за чего документ будет вводить в заблуждение будущих пользователей. Вторая ошибка — документ обновляется, из-за чего теряется история предложений: будущие разработчики не смогут просмотреть обсуждения, которые привели к определенным решениям, и могут повторить ошибки прошлого.

Следите за тем, чтобы документы находились в актуальном состоянии. Если ваши проектные документы и предложения — это два разных документа (например, PEP для Python и Документация Python), то вам нужно постоянно обновлять документ с учетом разработанных предложений. Следите также за тем, чтобы другие члены команды обновляли документы, пока вы проводите ревью кода.

Контролируйте версии проектной документации. Хорошим способом управления версиями является хранение версий проектных документов в том же репозитории, где хранится код. Ревью кода может быть использован в качестве проверки комментариев к проекту. Советуем вам обновлять документы по мере развития кода. Однако не всем нравится читать проектную документацию в Markdown или AsciiDoc: если вы предпочитаете работать в wiki, Google Docs или Word, откройте всю историю документа вместе с обсуждениями.

Использование шаблонов проектной документации

Проектный документ должен описывать текущий проект кода, мотивацию принятых изменений, возможные и предлагаемые решения. В нем должна быть подробная информация о предполагаемом решении: структурные графики, важные алгоритмические детали, публичные API, схемы, компромиссные альтернативные варианты, предположения и зависимости.

Не существует единого шаблона для проектных документов, однако вся проектная документация для систем с открытым исходным кодом позволяет увидеть то, как записывалась информация об изменениях. В разделе «Повышение уровня» вы найдете полезные ссылки на Предложение по улучшению Python, Предложение по улучшению Kafka, а также Рабочее предложение (RFC) по Rust. Если в вашей команде есть шаблон проектного документа, то используйте его, если нет — попробуйте представленный ниже вариант.

- Введение.
- Текущее состояние и контекст.
- Мотивация для изменений.
- Требования.
- Возможные решения.
- Выбранное решение.
- Проектирование и архитектура.
  - Диаграмма системы.
  - Изменения UI/UX.
  - Изменения кода.
  - Изменения API.
  - Изменения уровня хранения данных.
- План тестирования.
- План выпуска.
- Нерешенные вопросы.
- Приложение.

Введение

Расскажите о проблеме и объясните, почему именно ее нужно устранить. Предоставьте краткое описание предлагаемого изменения длиной один абзац и рекомендации, указывающие различным читателям — инженерам по безопасности, инженерам по эксплуатации, специалистам по данным — на соответствующие разделы.

Текущее состояние и контекст

Опишите проектирование, которое вы изменяете, а также определитесь с используемой терминологией. Объясните, как системы реагируют на элементы с неочевидными именами, например: «Feedler — это система регистрации пользователей. Она создана на основе инфраструктуры Rouft, которая обеспечивает обработку рабочих процессов с учетом их состояния». Расскажите о том, как вы решаете проблему в настоящее время. Вы используете какие-нибудь временные решения? Какие у них недостатки?

Мотивация для изменений

Команды разработчиков, как правило, имеют больше проектов, чем в состоянии вести одновременно. Так почему же именно эту проблему нужно решать и почему именно сейчас? Опишите пользу, которую принесет устранение проблемы. Свяжите ее с потребностями бизнеса. «Мы сможем наполовину сократить потребность в памяти» звучит не так убедительно и надежно, как «Сократив потребность в памяти на 50 %, мы сможем избавиться от причины, из-за которой нашим ПО не хотели пользоваться. Так мы сможем расширить рынок и увеличить продажи». Однако будьте осторожны с обещаниями!

Требования

Перечислите все требования, которым должно соответствовать решение проблемы. Все требования можно разделить на несколько видов.

- **Требования пользователей.** Они обычно составляют большую часть всех требований. Требования пользователей определяют характер изменений с точки зрения конечного пользователя.
- **Технические требования.** К решению выдвигаются жесткие требования. Технические требования обычно вызваны соображениями согласованности или строгими внутренними рекомендациями, например: «Должно поддерживать MySQL для уровней хранения» или «Должно соответствовать техническим требованиям OpenAPI для работы с нашим шлюзом приложений». Кроме того, могут определяться цели и задачи оказания услуг.

- **Требования безопасности и соответствия.** Их можно рассматривать как требования пользователей или технические требования, однако, как правило, они выделяются отдельно для возможности обеспечить обсуждение требований безопасности. Очень часто в них идет речь о политике хранения данных и доступа к ним.
- **Иные.** Могут включать в себя временные сроки, бюджет и другие важные моменты.

Возможные решения

Обычно существует несколько возможных решений проблемы. Написание данного раздела в проектной документации может быть полезным как для вас, так и для читателя: вам нужно не только разрабатывать основное решение, но и продумывать альтернативные идеи. Опишите альтернативные варианты, а затем объясните, почему не выбрали эти решения. Описание возможных решений может выглядеть как ответ на вопрос: «Почему мы не сделали так?» Если вы отклонили потенциальные решения по несостоятельным причинам, это может вызвать недоумение читателей. Они также могут предложить альтернативные варианты, которые вы не рассматривали.

Выбранное решение

Опишите решение, на котором вы остановились. Описание должно быть намного более подробным в сравнении с тем, что давалось во введении. Можете использовать диаграммы, графики и рисунки для визуализации изменений. Если ваше решение состоит из нескольких этапов, то и в этом, и в последующих разделах объясните развитие решения по этапам.

Проектирование и архитектура

Обычно бóльшую часть документа занимает раздел проектирования и архитектуры: все технические детали, на которые стоит обратить внимание, находятся в нем. Выделите интересующие вас детали разработки,

например используемые библиотеки и фреймворки, паттерны разработки и любые отклонения от практик, принятых в компании. Данный раздел должен включать в себя функциональные схемы компонентов, поток вызовов и поток данных, интерфейс системы, код, API и прототипы схем.

Диаграмма системы

Вставьте диаграмму, на которой отобразите основные компоненты и их взаимодействие. Объясните, что именно изменилось: выделите новые или замененные компоненты либо создайте две диаграммы, показывающие ситуацию до и после изменений. Диаграмма должна быть подписана и сопровождаться текстом, знакомящим читателя со всеми изменениями.

Изменения UI/UX

Если ваш проект вносит изменения в пользовательские интерфейсы, нужно создавать мокапы. Используйте заглушки, чтобы пройтись по потоку действий пользователя. При отсутствии у изменений визуального компонента в данном разделе можно рассказать об опыте работы разработчиков с созданной вами библиотекой или о том, как пользователь может использовать инструмент командной строки. Цель — продумать, как с вашим изменением люди в будущем будут работать.

Изменения кода

Опишите свой план реализации. Сосредоточьте внимание на том, что, как и когда следует изменить в уже существующем коде. Опишите также новые абстракции, если их нужно внедрить в код.

Изменения API

Документируйте новые API и изменения в уже существующих. Обсудите прямую и обратную совместимости и управление версиями. Включите в данный раздел обработку ошибок: предоставьте полезную информацию при обнаружении некорректных входных данных, нарушений ограничений или непредвиденных ошибок.

Изменения уровня хранения данных

Объясните, как были изменены технологии хранения данных. Рассмотрите новые базы данных, макеты файлов и файловых систем, поисковые индексы и процессы преобразования данных. Включите в этот раздел диаграммы изменений и примечания об их обратной совместимости.

План тестирования

Не нужно заранее определять каждый тест: объясните, как вы планируете тестировать изменения. Обсудите генерирование тестовых данных, выделите варианты использования, которые нужно охватить, а также рассмотрите библиотеки и стратегии тестирования, которые планируете использовать. Укажите, как вы будете проверять изменения на соответствие требованиям безопасности.

План выпуска

Опишите стратегии, которых будете придерживаться во время работы, чтобы избежать сложных требований к порядку развертывания. Задокументируйте флаги функций: их нужно будет установить для управления развертыванием. Сообщите, будете ли вы использовать паттерны развертывания, рассмотренные нами в главе 8. Подумайте, как обнаружить неработающие изменения и выполнить откат при возникновении ошибок.

Нерешенные вопросы

Перечислите вопросы, на которые у вас пока нет ответов. Это хороший способ получить совет от читателей, а также указать «известные неизвестные».

Приложение

В данном разделе перечислите остальные детали, представляющие интерес. Можете добавить ссылки на дополнительную литературу и иные документы, имеющие отношение к теме.

Совместная работа над проектом

Конструктивное сотрудничество с командой поможет вам создать качественный проект. Однако работать вместе не всегда просто. Разработчики — люди весьма самоуверенные. Понимание и объединение обратной связи в понятный проект — это очень сложный процесс. Работайте над проектом сообща в рамках процессов проектирования вашей команды, информируйте коллег подробно и заранее, чтобы избежать неожиданностей, используйте обсуждения проекта для реализации мозгового штурма.

Понимайте процесс обзора проекта в вашей команде

Обзор проекта уведомляет разработчиков архитектуры о предстоящих крупных изменениях, а также дает потенциальным клиентам возможность оставить обратную связь. В каких-то компаниях приняты более жесткие формальные правила проверки проекта, в других — менее жесткие. Советы по архитектурному обзору и запросы на принятие решения — два распространенных паттерна.

Архитектурные обзоры — формальный и более сложный процесс. Проекты должны быть одобрены заинтересованными лицами, например операторами или сотрудниками службы безопасности. Для этого создается проектный документ и иногда требуется провести несколько встреч или презентаций. Из-за большой траты времени архитектурные обзоры проводятся только для крупных или рискованных изменений.

Начинайте писать код до начала окончательного утверждения проекта. Потратьте время на создание прототипов и «спайков» проверок концепции: так вы сможете повысить свою уверенность в правильности создаваемого проекта и проложить более короткий путь к выпуску в эксплуатацию. Однако не стоит выходить за рамки обычной проверки концепции. Скорее всего, после получения обратной связи вам придется изменить код.

Мы называем другой тип процесса рассмотрения проекта *запросом на принятие решения*, или RFD (не путайте с рабочим предложением, или RFC). Термин RFD не особо распространен, но описать его можно

так: RFD — это краткий внутригрупповой обзор для быстрого принятия обсуждаемого решения, для чего не требуется полный обзор. Инженер, отправивший запрос на принятие решения, рассылает краткий отчет с описанием решения, которое нужно принять. Коллеги обсуждают варианты, вносят свой вклад в проект и принимают решение.

Конечно же, существуют и другие паттерны архитектурных обзоров. Важно лишь, чтобы вы понимали, каким процессам следует ваша команда. Пропуск этапа обзора проекта может привести к тому, что на последнем этапе появятся ошибки и проект не получится завершить. Узнайте, кто должен поставить подпись под вашим проектом и кто уполномочен принимать решения.

Не удивляйте коллег

Знакомьте коллег с вашим проектным предложением постепенно, еще в процессе работы над ним. Неудачный результат работы будет более вероятным, если готовый проектный документ — это первое, что другие команды и технические руководители узнают о вашей работе. Каждая сторона имеет свои взгляды и свои интересы и может резко отреагировать на сюрприз — появление проекта, в котором она не имела права голоса.

Вместо этого, когда занимаетесь первоначальным исследованием, получите обратную связь от других команд и руководителей: вы сможете улучшить свою проектную документацию, сообщив другим о проекте, над которым работаете, и тем самым заинтересовав их. Специалисты, которых вы вовлекаете в процесс на ранних этапах работы, позже смогут оказать поддержку вашему проекту.

Сеансы получения обратной связи не обязательно должны быть формальными или запланированными. Непринужденные разговоры за ланчем, в коридоре или перед началом совещания — это нормально, даже предпочтительно. Ваша цель — рассказать другим людям о том, что вы делаете, предоставить им возможность оставить обратную связь, а также сделать так, чтобы они начали думать о вашей работе.

Сообщайте коллегам о своих успехах. Делитесь новостями на статус-собраниях или стендапах. Продолжайте вести непринужденные разговоры. Рассказывайте командам, которые работали вместе с вами

над проектом, о предстоящих изменениях. Не забывайте уведомлять команды поддержки, команды контроля качества и команды по эксплуатации. Применяйте инклюзивный подход: привлекайте других людей к мозговому штурму и выслушивайте их мысли и идеи.

Мозговой штурм с обсуждением проекта

Обсуждение проекта поможет понять область проблемы, поделиться своими знаниями, найти компромиссные решения и улучшить проект. Мозговой штурм носит неформальный характер. Обычно обсуждения проходят в начале процесса проектирования, когда проблема уже изучена, но еще не решена. Хорошо, если вы создадите черновик проектной документации, даже если в ней и будет множество открытых вопросов или пробелов. Проводите мозговой штурм в несколько этапов с разными участниками, акцентируя внимание на разных аспектах проекта.

Мозговой штурм может проводиться от двух до пяти раз. Организуйте масштабные сеансы мозгового штурма, если проблема порождает множество вопросов и противоречий. На обычные дискуссии приглашайте небольшое количество участников, чтобы встреча проходила более непринужденно.

Встречи для обсуждения проекта изначально должны планироваться как достаточно продолжительные — часа на два. Хорошим идеям нужно время «настояться». Постарайтесь не отвлекаться от обсуждения — дождитесь, пока идеи закончатся или участники устанут. Чтобы прийти к конечному выводу, может понадобиться несколько встреч.

Перед сеансом мозгового штурма разработайте свободную повестку и включите в нее описание проблемы, охватываемую ею область, предлагаемый проект или проекты, а также возможные компромиссы и открытые вопросы. Ожидается, что участники заранее прочитают повестку, поэтому делайте ее краткой. Цель состоит в том, чтобы участники обладали достаточной информацией для начала обсуждения проекта.

На самой встрече не пытайтесь навязывать ее жесткую структуру — людям нужно свободно менять темы, чтобы исследовать идеи. Используйте интерактивную доску, а не слайды, и по возможности старайтесь говорить без заранее подготовленного текста (однако можете опираться на заметки).

Ведение участниками записей во время мозгового штурма может отвлекать. Некоторые команды официально назначают ответственного человека, который записывает все, что звучит на встречах. Убедитесь, что эта роль поочередно достается всем членам команды, иначе «записывающий» человек не сможет внести свой вклад в обсуждение. Можно оставлять заметки на доске, а при наличии интерактивной доски можно делать скриншоты во время встречи. После встречи напишите резюме на основании сделанных скриншотов и того, что запомнилось. Отправьте сделанные заметки участникам мозгового штурма и другим членам команды.

Вносите вклад в проект

Вы должны вносить вклад в работу всей команды, а не только в свою собственную. Как и в случае с ревью кода, будучи новым сотрудником и участвуя в разработке проекта, вы можете чувствовать себя неловко: вам может казаться, что вы не в состоянии чем-либо помочь старшим разработчикам с их проектом. Чтение ревью проекта и участие в мозговых штурмах может отвлекать, однако все равно уделяйте немного времени этим делам. Ваше участие улучшит проект, создаваемый командой, а вы научитесь чему-то новому.

Когда вы присоединяетесь к разработке проекта, не бойтесь вносить предложения и задавать вопросы. Применяйте те же самые рекомендации, которые мы давали вам относительно ревью кода. Думайте о проекте. Учитывайте безопасность, масштаб, производительность и возможность исправления ошибок. Обратите особое внимание на то, как данный проект связан с вашими областями знания. Общайтесь четко и ясно, ведите себя корректно.

Очень важно не только вносить предложения, но и задавать вопросы. Вопросы помогают расти в профессиональном плане. Вполне вероятно, что вы не единственный человек, который задается вопросом по поводу того или иного решения, так что ваши вопросы могут помочь не только вам, но и вашим коллегам. К тому же эти вопросы могут натолкнуть команду на новые идеи или позволят обнаружить проблемы, незаметные на этапе разработки.

Что следует и чего не следует делать

Следует	Не следует
Использовать шаблоны проектных документов	Привязываться к экспериментальному коду: он изменится
Читать блоги, статьи и презентации других разработчиков, чтобы узнавать новое	Рассматривать только одно решение проблемы
Относиться с долей сомнения ко всему, что читаете	Позволять языковому барьеру мешать вам писать
Экспериментировать с кодом	Забывать обновлять проектные документы, если разработка отходит от первоначального плана
Практиковаться писать четко и ясно	Отказываться от участия в командном обсуждении проекта
Осуществлять контроль версий проектной документации	
Задавать вопросы насчет проектов ваших коллег по команде	

Повышение уровня

Ричард Хикки (Richard Hickey), создатель языка программирования Clojure, составил «полевой отчет» по проектированию программного обеспечения в своей лекции *Hammock Driven Development* (https://youtu.be/f84n5oFoZBc/). Выступление Хикки — одно из лучших введений в запутанный процесс проектирования программного обеспечения.

Чтобы увидеть реальный процесс разработки, советуем вам изучать большие проекты с открытым исходным кодом. Предложения по улучшению Python (https://github.com/python/peps/), предложения по улучшению Kafka (https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Improvement+Proposals/) и Рабочее предложение (RFC) по Rust (https://github.com/rust-lang/rfcs/) являются хорошими примерами реального проектирования.

Если хотите получить подробное представление о внутренних процессах проектирования, в блоге WePay есть пост *Effective Software Design Documents* (https://wecode.wepay.com/posts/effective-software-design-documents). В нем описывается подход WePay к проектированию и то, как данный

подход развивался с течением времени. Шаблон проекта, использованный WePay, доступен на GitHub по ссылке (https://github.com/wepay/design_doc_template/).

Книга Э. Б. Уайта и Уильяма Странка (E. B. White, William Strunk) *The Elements of Style* (Auroch Press, 2020) стала канонической рекомендацией ясного изложения мыслей. Мы также настоятельно рекомендуем вам прочитать книгу Уильяма Зиснера (William Zissner) *On Writing Well* (Harper Perennial, 2016). Обе книги могут улучшить вашу стилистику письма. Пол Грэм (Paul Graham), один из основателей Y Combinator, написал на данную тему два эссе: *How to Write Usefully* (http://paulgraham.com/useful.html) и *Write Like You Talk* (http://www.paulgraham.com/talk.html).

11 Создание эволюционной архитектуры

Волатильность требований, или изменение требований клиентов — это неизбежная проблема в проектах по разработке ПО. Когда контекст и требования к продукту меняются, должен измениться и сам продукт. Однако изменение требований может привести к нестабильности или даже к прекращению разработки ПО.

При работе с волатильностью требований руководители используют процессы последовательной разработки, например гибкий подход к планированию (рассмотрим в следующей главе). Вы можете внести свой вклад в удовлетворение меняющихся требований, создав *эволюционную архитектуру*. Она избегает сложности — врага возможности эволюционировать.

В этой главе вы научитесь методам, которые помогут вам упростить программное обеспечение и, следовательно, облегчить его возможность эволюционировать. Как это ни парадоксально, добиться простоты в программном обеспечении может быть непросто: без сознательных усилий код станет запутанным и сложным. Мы начнем с описания сложности и того, как она приводит к жесткой и запутанной кодовой базе. Затем покажем вам принципы проектирования, снижающие сложность. Наконец, мы переведем эти принципы проектирования в конкретные передовые практики API и слоев обработки данных.

Понимание сложности

В своей книге *A Philosophy of Software Design* (Yaknyam Press, 2018) Джон Оустерхаут (John Ousterhout), профессор информатики из Стэнфордского университета, пишет: «Сложность — это все, что связано со структурой системы и что затрудняет ее понимание и модификацию». Согласно Оустерхауту, сложные системы обладают двумя свойствами: высокой степенью *зависимости* и *неясности*. Мы добавим третье свойство — высокую *инертность*.

Высокая степень *зависимости* приводит к тому, что ПО будет опираться на API или поведение другого кода. Конечно, зависимости нельзя избежать, и в некоторых случаях она даже желательна, однако всегда нужно помнить о балансе. Каждая новая зависимость только усложняет будущие изменения кода. Системы с высокой степенью зависимости сложно изменять, так как у них имеются сильная *связанность* и увеличение *количества изменений*. Связанность описывает модули, сильно зависящие друг от друга. Это является причиной увеличения количества изменений, так как даже при внедрении одного небольшого изменения потребуется изменять зависимости. Продуманная структура API и ограничение на возможное количество использований абстрактных понятий помогут по максимуму избежать зависимости и увеличения количества изменений.

Повышение *неясности* затрудняет для программистов прогнозирование побочных эффектов изменений, поведения кода и ухудшает понимание, куда именно необходимо внести изменения. Неясности требуют дополнительного времени для изучения кода, и разработчики с большей вероятностью могут случайно что-то повредить. *Божественные объекты*, «знающие» слишком много, глобальное состояние, повышающее количество побочных эффектов, чрезмерная запутанность связей, вносящая неясности в код, а также *действия на расстоянии*, влияющие на поведение удаленных частей программы, — это признаки неясности. API с четкой структурой и стандартными шаблонами проектирования уменьшают неясность.

Свойство, которое мы добавили в список Оустерхаута, — *инертность*: это тенденция ПО оставаться в использовании и постоянно обнов-

ляться. Код, используемый в быстрых экспериментах, легко удаляется и обладает низкой инертностью. Сервис, поддерживающий множество важных бизнес-приложений, обладает высокой инертностью. Затраты на поддержку системы с повышенной сложностью будут со временем возрастать, поэтому системы с высокой инертностью или большим количеством изменений необходимо упрощать. Системы с низкой инертностью или небольшим количеством изменений можно оставить на текущем уровне сложности (до тех пор, пока вы не откажетесь от их использования).

Не всегда получается избавиться от сложности, однако у вас есть возможность выбрать место, где она будет сконцентрирована. Совместимые с предыдущими версиями изменения (их мы обсудим позже) могут упростить использование кода, но способны отрицательно повлиять на его написание. Уровни косвенных обращений для снижения связанности подсистем уменьшают зависимость, но повышают неясность. Хорошо подумайте о том, когда, где и как управлять сложностью.

Проектирование с учетом эволюционности

Сталкиваясь с неизвестностью будущих требований, инженеры чаще всего выбирают один из двух подходов: или пытаются понять, что от них может потребоваться, или создают абстракции, которые облегчат будущее изменение кода. Не надо играть в подобные игры: оба подхода приведут к увеличению сложности. Будьте проще. Кстати, этот принцип известен как KISS — *Keep It Simple, Stupid (делай проще, тупица)* — благодаря любви ВМС США к акронимам. Воспользуйтесь принципом KISS, чтобы не забывать о простоте. Вы всегда можете усложнить простой код позже, когда появится реальная необходимость во внедрении изменений.

Простейший способ сделать код простым — не писать его. Скажите себе, что *вам это не понадобится* (принцип You Ain't Gonna Need It, YAGNI). Когда вы пишете код, пользуйтесь принципом наименьшего удивления и инкапсуляцией. Эти принципы проектирования упростят дальнейшее развитие вашего кода.

Вам это не понадобится

YAGNI — это кажущийся обманчиво простым принцип программирования, в основе которого лежит мысль: «Не создавайте то, что вам не нужно». Разработчики очень часто не придерживаются данного принципа, особенно когда нервничают, зациклены или обеспокоены каким-либо аспектом своего кода. Трудно сказать, что может вам понадобиться, а что будет бесполезным. Каждая неверная догадка — это уйма потраченного времени и сил: код продолжает тормозить, его нужно поддерживать и тестировать, разработчики должны в нем разобраться.

Но есть возможность обойтись без ненужной разработки. Для этого избегайте преждевременной оптимизации, слишком гибких абстракций и функциональных возможностей продукта, которые не обязательны для *минимально жизнеспособного продукта* (minimum viable product, MVP). MVP — это продукт, обладающий минимальным набором функций, необходимых для получения обратной связи от пользователей.

Преждевременная оптимизация возникает в тот момент, когда разработчик добавляет в код оптимизацию, но необходимость ее добавления при этом не доказана. Классический сценарий выглядит так: разработчик видит область кода, которую можно сделать быстрой или более масштабируемой благодаря добавлению сложной логики или таких архитектурных уровней, как кэши, сегментированные базы данных или очереди. Разработчик оптимизирует код перед доставкой клиентам, до того как кто-нибудь начнет им пользоваться. После отправки разработчик понимает, что оптимизация была не нужна. Однако оптимизация уже не удаляется, а сложность кода только увеличивается. Многие улучшения производительности кода требуют повышения сложности. К примеру, кэш будет работать очень быстро, но окажется несовместимым с базовыми данными.

Еще одним соблазном может стать использование гибких абстракций: архитектура плагинов, оболочки интерфейсов, а также обобщенные структуры данных, например пары «ключ — значение». Иногда разработчикам бывает очень сложно сопротивляться желанию добавить в код гибкие абстракции. Разработчики думают, что в таком случае при

появлении новых требований они смогут легко адаптироваться. Однако у абстракции есть своя цена — реализация заключена в жесткие рамки, которые впоследствии начнут ограничивать разработчика. Применение гибких абстракций затрудняет чтение и понимание кода. Посмотрите на этот интерфейс распределенной очереди:

```
interface IDistributedQueue {
  void send(String queue, Object message);
  Object receive(String queue);
}
```

`IDistributedQueue` выглядит довольно просто: вы отправляете и получаете сообщения. Но что если очередь будет поддерживать и ключи, и значения для сообщений (как это делает Apache Kafka), и гарантию доставки сообщений (это делает Amazon Simple Queue Service)? Разработчикам нужно сделать выбор: интерфейс должен быть объединением всех функций из всех очередей сообщений или пересечением всех функций? Объединение всех функций создаст интерфейс, в котором каждому методу потребуется разная реализация. Пересечение всех функций создаст интерфейс, имеющий недостаточно функций для того, чтобы он мог быть полезен. Лучше всего использовать реализацию функций напрямую: вы можете провести рефакторинг кода позже, если посчитаете, что вам нужна поддержка другой функции.

Лучший способ сделать ваш код гибким — уменьшить его объем. Когда у вас появится желание что-либо добавить, спросите себя, а точно ли вам это понадобится? Если нет, то не следует ничего менять в коде. Данная практика называется *мюнцинг*: она помогает сохранять аккуратность и адаптивность ПО.

Вам может показаться хорошей идеей добавление новых функций продукта. Разработчики попадают в эту ловушку, ошибочно считая, что новые интересные функции понадобятся многим пользователям и функции при этом будет легко реализовать в коде. И еще разработчики думают, что после добавления новых функций код по-прежнему будет аккуратным! Однако на разработку и поддержку любой функции требуется много времени, а уверенности в том, что функция на самом деле будет полезной, у вас нет.

Создание минимально жизнеспособного продукта позволит не гадать, что вам на самом деле нужно. MVP дает возможность протестировать идею без затрат сил и времени на ее реализацию.

Конечно, у подхода YAGNI есть свои недостатки. Со временем вы наберетесь опыта и научитесь с большей точностью предсказывать, когда нужно что-то оптимизировать. До тех пор используйте оболочки интерфейса в тех местах, куда, как вам кажется, следует добавить оптимизацию. Например, если вы создаете новый формат файла и считаете, что позже вам понадобится его распаковать или зашифровать, то в заголовке укажите кодировку, но реализуйте только нераспакованную кодировку. В будущем вы можете добавить еще больше сжатия, а благодаря заголовку новый код легко сможет прочитать старые файлы.

Принцип наименьшего удивления

Принцип наименьшего удивления очень понятный: не удивляйте пользователей. Создавайте функции, поведение которых совпадает с ожиданиями пользователей. Функции с высокой кривой обучения или странным поведением разочаровывают пользователей. Не стоит удивлять и разработчиков: необычный код трудно понять, из-за чего возникают сложности. Вы можете сохранить особенность своего кода, но используйте при этом стандартные паттерны и библиотеки и не используйте неявные знания.

Все неочевидные вещи, которые разработчик должен знать для того, чтобы использовать API, и которые не являются частью этого API, относятся к *неявным знаниям*. API, требующие неявных знаний, приводят разработчиков в замешательство, вызывают ошибки и завышают кривую обучаемости. Есть два распространенных препятствия для применения неявных знаний: скрытые требования к упорядочиванию выполнения и аргументам.

Требования к упорядочиванию определяют последовательность выполнения действий в программе. Частым препятствием служит упорядочивание методов: метод А должен вызываться перед методом Б, но

API позволяет первым вызвать метод Б, и разработчик будет сильно удивлен, получив сообщение об ошибке времени исполнения. Документирование требований к упорядочиванию — хорошая идея, однако лучше этим не заниматься. Избегайте упорядочивания вызовов методов путем вызова методами подметодов:

```
pontoonWorples() {
  if(!flubberized) {
    flubberize()
  }
  // ...
}
```

Существуют другие способы, позволяющие избежать соблюдения порядка, такие как объединение всех методов в один, использование паттерна проектирования под названием *Строитель*, использование системы типов, а также метода `pontoonWorples` только с `lubberizedWorples`, а не со всеми `Worples`, и т. д. Все это лучше, чем требовать, чтобы пользователь знал о скрытых требованиях. Вы можете сделать так, чтобы в имени метода имелось предупреждение для разработчиков: например, назовите метод `pontoonFlubberizedWorples()`. Как бы странно это ни звучало, но короткие имена увеличивают когнитивную нагрузку. Лучше всего использовать длинные и детализированные имена, которые являются более простыми для понимания и в которых описывается суть метода или переменной.

Скрытые требования к аргументам появляются тогда, когда сигнатура метода подразумевает диапазон допустимых входных данных намного шире, чем метод может принять на самом деле. К примеру, принятие `int` при разрешении чисел только от 1 до 10 является скрытым ограничением. Требование того, чтобы определенное поле значения было задано в объекте JSON, также подразумевает наличие неявных знаний у пользователя. Требования к аргументам должны быть точными и явными. Используйте определенные типы, с помощью которых вы сможете отобразить необходимые ограничения. Если вы используете гибкие типы, например JSON, то для описания объекта можете применять схемы JSON. По крайней мере, объявляйте требования к аргументам в документации, если их нельзя сделать программно видимыми.

Наконец, используйте стандартные библиотеки и паттерны разработки. Реализация собственного метода извлечения квадратного корня — это неожиданно и удивительно, а использование встроенного метода `sqrt()` языка — нет. Подобное правило применяется и к паттернам разработки: используйте идиоматический стиль кода и шаблоны разработки.

Инкапсулируйте знания предметной области

Программное обеспечение изменяется вместе с требованиями со стороны бизнеса. Инкапсулируйте знания предметной области, группируя ПО на основе предметной области: бухгалтерского учета, формирования счетов, доставки и т. д. Сопоставление компонентов ПО с бизнес-областями позволяет сделать изменение кода более целенаправленным и чистым.

Инкапсулированные функциональные области всегда устремляются к *высокой связности* и *слабой связанности* — желаемым характеристикам. ПО с высокой связностью и слабой связанностью легче развивать, так как изменения имеют небольшой радиус действия. *Связность* кода считается высокой тогда, когда взаимосвязанные методы, классы и переменные находятся рядом друг с другом в модулях или пакетах. Код со слабой *связанностью* — самодостаточный: при изменении его логики не нужно изменять другие программные компоненты.

Разработчики часто рассматривают ПО с точки зрения уровней: фронтенд, средний уровень и бэкенд. Многоуровневый код сгруппирован в соответствии с технической областью, при этом весь код пользовательского интерфейса находится в одном месте, а все сохраняемые объекты — в другом. Группировка кода согласно техническим областям подходит при работе в пределах одной бизнес-области, однако при расширении бизнес-областей данный метод становится бесполезным. Вокруг каждого из уровней собираются отдельные команды, и затраты на координацию увеличиваются, так как любое изменение бизнес-логики отражается на всех уровнях. При использовании горизонтальных уровней общего доступа очень легко смешать бизнес-логику разных функциональных областей и тем самым усложнить код.

Определение границ предметной области и инкапсуляция знаний предметной области — это и наука, и искусство одновременно. Существует целый архитектурный подход, называющийся *предметно-ориентированным проектированием* (DDD). Данный принцип сопоставляет бизнес-концепции с программным обеспечением. Применение полноценного DDD требуется только в самых сложных ситуациях. Однако знакомство с этой концепцией поможет вам принимать более обоснованные проектные решения.

Эволюционные API

При изменении требований необходимо изменять API — общие интерфейсы между блоками кода. Несмотря на то что изменить API легко, очень сложно сделать это правильно. Многие мелкие изменения могут привести к большой проблеме в будущем. Даже незначительные изменения API способны нарушить совместимость версий. Если в API вносится изменение, нарушающее совместимость, то клиенты выдадут ошибку. Но сначала это будет неочевидно, особенно если были внесены изменения, выдающие ошибку не во время компиляции, а во время выполнения. Небольшие, четко определенные, совместимые и получающие жесткие номера версий API намного более просты в использовании.

Не увеличивайте размеры API

Небольшие API легче понимать и улучшать. При работе с большими API у разработчиков возникает высокая когнитивная нагрузка: большой API означает большой объем кода, который нужно постоянно документировать, поддерживать и отлаживать. Каждый новый метод или область применения расширяют API и привязывают к определенному паттерну использования.

Воспользуйтесь принципом YAGNI: добавляйте только то, что нужно в данный момент времени. При начальной загрузке API с помощью фреймворка или генератора необходимо удалить все неиспользуемые поля или методы.

Методы API со множеством полей должны иметь разумные значения по умолчанию. В таком случае разработчики смогут сосредоточиться на нужных полях, зная, что остальным полям передаются значения по умолчанию, и большие API покажутся маленькими.

Предоставляйте строго определенные сервисные API

Развивающиеся системы имеют четко определенные схемы запросов и ответов, которые поддерживают версии и имеют четкие контракты совместимости. Определения схем должны быть доступны, чтобы их можно было использовать в автоматическом тестировании как клиентского кода, так и серверного (больше информации — в разделе «Упаковывайте разные ресурсы по отдельности» в главе 8).

Для определения сервисных API используйте стандартные инструменты. Строго определенный сервис будет объявлять свои схемы, методы запросов и ответов, а также собственные исключения. Экосистема OpenAPI используется для RESTful-сервисов, остальные сервисы задействуют на протокол сериализации данных, язык Thrift или *язык описания интерфейса* (IDL). Строго определенные интерфейсы сервисных API облегчают проверку во время компиляции, а также обеспечивают синхронизацию клиентов, серверов и документации.

Инструменты описания интерфейса поставляются совместно с генераторами кода, которые преобразуют описание сервисов в клиентский или серверный код. Таким образом можно создавать документацию. Инструменты тестирования могут использовать язык описания интерфейса для создания заглушек и mock-данных. В некоторых инструментах есть функция обнаружения, с помощью которой можно найти сервисы, узнать, что их поддерживает, и показать, как они используются.

Если в вашей компании уже выбран фреймворк определения API, то используйте его: поиск «самого лучшего» фреймворка требует большой работы по функциональной согласованности. Если ваша компания до сих пор использует REST API в ручном режиме и интерфейсы JSON дополняются в самом коде без формальной спецификации, то лучшим

выбором станет OpenAPI, поскольку ее можно модернизировать на основе уже существующих сервисов REST и в таком случае не требуется серьезных миграций для внедрения.

Изменения API должны быть совместимыми

Поддержание совместимости изменений позволяет версиям клиента и сервера развиваться независимо друг от друга. Существуют две формы совместимости: прямая и обратная.

Благодаря изменениям *с прямой совместимостью* клиенты могут использовать новую версию API вместе со старой версией сервиса. Вебсервис, работающий с API версии 1.0, но способный принимать вызовы от клиента, работающего с API версии 1.1, будет иметь прямую совместимость.

Изменения *с обратной совместимостью* прямо противоположны: новым версиям библиотек или сервисов не требуется обновленный клиентский код. Изменение является обратно совместимым, если код, который использовался для версии API 1.0, будет компилироваться или выполняться для версии 1.1.

Посмотрите на простой, типа Hello World, сервисный API gRPC, определенный с помощью протокола сериализации данных:

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

message HelloRequest {
  string name = 1;
  int32 favorite_number = 2;
}

message HelloReply {
  string message = 1;
}
```

В сервисе `Greeter` есть метод под названием `SayHello`, которому поступает `HelloRequest` и который возвращает `HelloReply` с забавным сообщением `favorite_number`. Числа возле каждого поля — порядковые

номера полей. Протокол сериализации данных внутри ссылается на поля с помощью чисел, а не строк.

Давайте представим, что мы собираемся отправить приветствие по электронной почте. Для этого нам нужно добавить в код поле электронной почты:

```
message HelloRequest {
  string name = 1;
  int32 favorite_number = 2;
  required string email = 3;
}
```

Это изменение является *обратно несовместимым*, так как старые клиенты не предоставили электронную почту. Клиент вызывает метод `SayHello` со старым `HelloRequest`; сообщение будет отсутствовать, так что сервис не сможет обработать запрос. Обратную совместимость можно поддержать путем удаления ключевого слова или пропуска писем в том случае, если адрес электронной почты не указан.

Обязательные к заполнению поля стали такой проблемой эволюционности, что их удалили из третьей версии протокола сериализации данных. Кентон Варда, основной автор второй версии протокола сериализации данных, сказал: «Использование ключевого слова required[1] в протоколе сериализации данных обернулось большой ошибкой» (https://capnproto.org/faq.html#how-do-i-make-a-field-required-like-in-protocol-buffers). Во многих других системах есть обязательные к заполнению поля, так что всегда помните: «Обязательное поле — это навсегда».

Мы можем создать *прямо несовместимое* изменение путем изменения настроек `HelloRequest`. В документации протокола сериализации данных указано: «Если в ваше поле можно поместить отрицательные значения, используйте тип данных `sint32`», так что мы меняем тип `favorite_number` соответственно:

```
message HelloRequest {
  string name = 1;
  sint32 favorite_number = 2;
}
```

[1] Для обязательных полей. — *Примеч. пер.*

Изменение типа `int32` на `sint32` порождает прямую и обратную несовместимость. Клиент с новым `HelloRequest` будет кодировать `favourite_number` с помощью другой схемы сериализации, нежели клиент со старой версией сервиса `Greeter`, и именно поэтому сервис не сможет его проанализировать. Новая версия `Greeter` не сможет анализировать и сообщения от пользователей, использующих старые версии.

Чтобы значение `sint32` было совместимо с будущими версиями, нужно добавить новое поле. Буферы протокола позволяют переименовывать поля, пока номер поля остается прежним.

```
message HelloRequest {
  string name = 1;
  int32 _deprecated_favorite_number = 2;
  sint32 favorite_number = 3;
}
```

Код сервера должен обрабатывать оба поля до тех пор, пока осуществляется поддержка старых клиентов. Выполнив развертывание, вы можете отслеживать частоту, с которой клиенты используют старое поле, и удалять его после обновления клиентов или истечения срока по графику депрекации.

В нашем примере используется протокол сериализации данных, так как он поддерживает строгую систему типов, а совместимость типов очень просто показать и обосновать. Такие же проблемы возникают в других контекстах, включая «простые» REST-сервисы. Когда ожидания клиента и сервера в отношении контента разнятся, то вне зависимости от того, какой формат используется, будут возникать разнообразные ошибки. К тому же вам придется следить не только за полями сообщений: изменения в *семантике* сообщения или логике того, что происходит при определенных событиях, тоже могут быть прямо или обратно несовместимыми.

Версии API

Поскольку API постоянно развиваются, нужно решить, каким именно образом следует обеспечить совместимость разных версий. Обратно и прямо совместимые изменения взаимодействуют с предыдущими и будущими версиями API. Иногда это очень сложно поддерживать по

причине появления таких *излишеств*, как логика работы с устаревшими областями. Менее строгая гарантия совместимости позволяет вносить большие изменения.

Когда-нибудь вы захотите изменить API так, чтобы он больше не был совместим со старыми клиентами, и решите добавить новое поле. Управление версиями API означает, что при добавлении изменений вы используете новую версию. Старые клиенты могут использовать старые версии API. Отслеживание версий помогает поддерживать связь с клиентами: они могут сообщить вам, с какой версией работают, а вы можете продвигать новую версию с совершенно новыми функциями.

Обычно все версии API управляются с помощью API-шлюза или сервисной сети. Адресованные определенным версиям запросы отправляются в соответствующий сервис: запросы версии 2 отправляются в сервис версии 2.x.x, а запросы версии 3 — в сервис версии 3.x.x. При отсутствии шлюза клиенты инициируют удаленный вызов процедур непосредственно к узлам сервиса, определенным для конкретной версии, или один экземпляр сервиса запускает несколько версий внутри себя.

Назначение версий API имеет свою цену. Старые мажорные версии необходимо постоянно поддерживать, а исправления ошибок следует переносить во все предыдущие версии. Разработчикам нужно отслеживать, какие функции поддерживаются в той или иной версии. При отсутствии инструментов управления версиями вся работа по поддержке версий может быть переложена на инженеров.

Будьте прагматичны при использовании методик управления версиями. Семантическое управление версиями, рассмотренное в главе 5, является распространенной схемой определения версий API, однако многие компании задают версии API, используя даты или иные числовые схемы. Номера версий могут быть указаны в путях URI ресурсов, параметрах запроса или в заголовке HTTP Accept. По поводу этих подходов существует множество мнений и допускается немало компромиссов. Полагайтесь на то, что в вашей компании считается стандартным, а если стандарта нет, посоветуйтесь с руководителем, как лучше поступить.

Храните версию документации вместе с API. Разработчикам, работающим с прошлыми версиями вашего кода, потребуется точная и понятная

документация. Пользователи действительно оказываются сбиты с толку, встречая ссылку на другую версию документации и обнаруживая, что используемый ими API не совпадает с указанным. Отправка документации по API в основной репозиторий кода позволяет синхронизировать документацию и код. Запросы на включение кода помогут обновить документацию при изменении API.

Управление версиями API наиболее ценно, когда клиентский код трудно изменить. Обычно у разработчиков нет необходимого контроля над версиями кода клиентов, поэтому особенно важными являются версии клиентских API. Если ваша команда контролирует сервис и клиентский код, то вы можете работать без внутреннего управления версиями API.

Эволюция данных

API не такие долговечные, как сохраненные данные: после каждого обновления клиентского или серверного API работа завершается. Данные должны меняться по мере изменения приложений. Эволюция данных охватывает весь спектр: от простых изменений схемы, таких как добавление или удаление столбца, до преобразования записей с использованием новых схем, исправления ошибок, преобразования записей в зависимости от новой бизнес-логики или масштабных миграций из одной базы данных в другую.

Изолирование баз данных и использование явных схем помогают сделать эволюцию данных более удобной в управлении. Если у вас изолированная база данных, то все, о чем вам нужно думать, — это влияние изменений на ваше приложение. Схемы дают защиту от чтения и записи некорректных данных, а благодаря автоматизированной миграции схем все изменения схем становятся предсказуемыми.

Изолируйте базы данных

Совместно используемые базы данных очень трудно улучшать, что приводит к потере *автономности*, то есть возможности для разработчика или команды вносить независимые изменения. Вы не сможете безопасно изменять схемы или даже читать и записывать данные, не беспокоясь

о том, как все используют вашу базу данных. По мере того как схемы становятся неформальными, архитектура кода оказывается более хрупкой. Отдельные базы данных упрощают внедрение изменений.

Доступ к изолированным базам данных имеет только одно приложение, в то время как к совместно используемым базам данных доступ имеют несколько приложений (рис. 11.1).

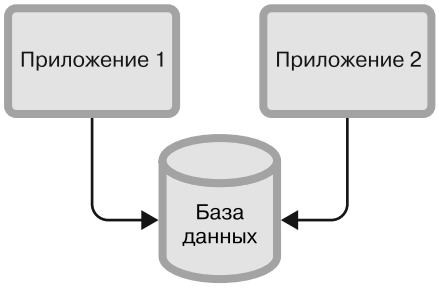

Рис. 11.1. Совместно используемая база данных

У совместно используемых баз данных есть несколько проблем. Совместное использование баз данных приложениями может перерасти в прямую зависимость от данных друг друга. Приложения являются контрольной точкой для базовых данных, с которыми они работают. Вы не можете применять бизнес-логику к необработанным данным перед их предоставлением и легко перенаправлять запросы в новое хранилище данных во время миграции, если запросы обходят ваше приложение. Если несколько приложений выполняют запись, значение (семантика) данных могут отличаться, что затрудняет их понимание при считывании. Данные приложения не защищены, поэтому другие приложения могут изменить их неожиданным образом. Схемы не изолированы, и каждое изменение схемы одного приложения может повлиять на схемы другого приложения. Производительность тоже не является изолированной, так что, если одно приложение будет перегружать базу данных, это повлияет на производительность других приложений. В некоторых случаях безопасность может быть также нарушена.

У изолированных баз данных есть только одно считывающее и записывающее приложение (рис. 11.2). Остальной трафик перенаправляется через удаленные вызовы процедур.

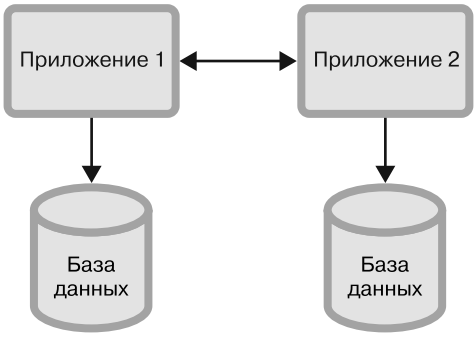

Рис. 11.2. Изолированные базы данных

Изолированные базы данных обусловливают гибкость и изолированность, которых нет в совместно используемых базах данных. При работе с изолированными базами данных вам нужно думать только о собственном приложении в тот момент, когда вы вносите изменения в схему базы. Производительность базы зависит от того, как вы ее используете.

Существуют моменты, когда совместно используемая база данных предоставляет больше преимуществ. При разделении монолита на части совместное использование базы станет промежуточным шагом перед переносом всех данных в новую изолированную базу данных. Управление множеством баз данных связано с высокими расходами. На раннем этапе лучше всего использовать несколько баз на одном и том же устройстве. Однако предварительно убедитесь, что совместно используемые базы данных в итоге будут изолированы, разделены или заменены.

Используйте схемы

Наличие жестко заданных и предварительно определенных столбцов и типов данных, а также требовательных к ресурсам процессов для их обновления привело к появлению популярного *слабоструктурированного*, или *бессхемного*, способа управления данными. Многие современные хранилища данных позволяют хранить объекты JSON и им подобные без предварительного объявления структуры. Бессхемные данные — не буквально данные без схем (тогда данные нельзя было бы использовать). Скорее, у слабоструктурированных данных неявная схема, которая предоставляется или выводится во время чтения.

Однако при работе мы обнаружили, что слабоструктурированный подход имеет несколько серьезных проблем со сложностью и целостностью данных. При структурированном подходе с использованием схем применяется строгая типизация, которая уменьшает запутанность, а следовательно, и сложность вашего приложения. Кратковременная простота обычно не стоит того, чтобы идти на компромисс с неясностью. Как и сам код, иногда данные описываются так: «записываем один раз, считываем много раз». Для упрощения считывания используйте схемы.

Может сложиться впечатление, что без схем вносить изменения проще: вы просто начинаете или прекращаете записывать поля по мере необходимости для обновления данных. Но на самом деле слабоструктурированные данные только усложняют внедрение изменений, так как вы не знаете, что именно вы нарушаете при обновлении данных. Данные быстро превращаются в кашу из различных записей. Разработчики, бизнес-аналитики, специалисты по обработке и анализу данных стараются постоянно быть в теме, и, например, у специалиста по обработке и анализу данных выдастся сложный день, когда он соберется проанализировать такие данные в формате JSON:

```
{"name": "Fred", "location": [37.33, -122.03], "enabled": true}
{"name": "Li", "enabled": "false"}
{"name": "Willa", "location": "Boston, MA", "enabled": 0}
```

Определение явных схем ваших данных обеспечит стабильность приложения и даст возможность в дальнейшем эти данные использовать. Явные схемы позволяют проверять работу данных во время их записи. Скорость анализа данных с использованием явных схем высока. Схемы помогают обнаружить изменения с прямой или обратной несовместимостью. Специалисты по обработке и анализу данных знают, какой результат они получат при использовании данных из следующей таблицы.

```
CREATE TABLE users (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  latitude DECIMAL,
  longitude DECIMAL,
  enabled BOOLEAN NOT NULL
);
```

Жестко заданные явные схемы имеют свои недостатки: иногда их трудно изменить. Это сделано специально. При использовании схем нужно на

мгновение остановиться и подумать о том, каким образом будут переноситься существующие данные и как это повлияет на последующих пользователей.

Не прячьте слабоструктурированные данные внутри структурированных. Возможно, вы захотите пойти по легкому пути и вставить строку JSON в поле с именем `data` или определить карту строк для хранения пар «ключ — значение». В таком случае вы только зря потратите время, так как столкнетесь со всеми проблемами явных схем.

В некоторых случаях применение слабоструктурированного подхода будет более полезным. Если вашей основной целью является быстрое продвижение — возможно, вы еще не знаете, что вам нужно, когда быстро проводите обновления или когда старые данные практически или совсем не имеют ценности, — слабоструктурированный подход позволит «срезать углы». Некоторые данные заведомо неоднородны: в одних записях есть такие поля, которых нет в других. Переход от данных с явной схемой к данным с неявной схемой также хорошо использовать при переносе данных; на какое-то время вы можете преобразовать данные в слабоструктурированные, что упростит переход к новой явной схеме.

Автоматизация миграции схем

Изменение схемы базы данных опасно. Небольшое изменение, например добавление индекса или удаление столбца, может привести к остановке работы всей базы данных или приложения. Управление изменениями базы данных путем выполнения команд *языка описания структур данных* (database description language, DDL) в самой базе данных может привести к ошибкам. Схемы базы данных в разных средах различаются, а их состояние остается неопределенным: в таком случае неизвестно, что, когда и кем было изменено, как и неизвестно влияние этих действий на производительность. Сочетание изменений, подверженных ошибкам, и вероятности потратить время впустую представляет собой взрывную смесь.

Инструменты управления схемой базы данных помогают уменьшить подверженность изменений ошибкам. Автоматизированные инструменты

полезны в двух случаях: они заставляют вас отслеживать историю схемы, а также предоставляют возможности для переноса схемы из одной версии в другую. Для отслеживания изменений схемы советуем использовать автоматизированные инструменты баз данных и поддерживать связь со своей командой для управления обновлением схемы.

Обычно история схемы хранится в нескольких файлах, которые характеризуют каждое изменение схемы с момента ее создания до настоящего времени. Отслеживание изменений, вносимых с помощью DDL, в файлах помогает разработчикам понять, как именно схема изменялась. Файлы расскажут о том, кто именно вносил то или иное изменение, когда и по какой причине это было сделано. Запросы на включение предоставляют возможность для обзора схем и линтинга.

Мы можем взять пользовательскую таблицу из раздела «Используйте схемы» и поместить ее в файл определенной версии для инструмента миграции схем, например Liquibase:

```
--liquibase formatted sql
--changeset criccomini:create-users-table
CREATE TABLE users (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  latitude DECIMAL,
  longitude DECIMAL,
  enabled BOOLEAN NOT NULL
);
--rollback DROP TABLE users
```

Определим команду `ALTER` в отдельном блоке:

```
--changeset dryaboy:add-email
ALTER TABLE users ADD email VARCHAR(255);
--rollback DROP COLUMN email
```

Библиотека Liquibase может использовать файлы для понижения или повышения версии схем с помощью интерфейса командной строки:

```
$ liquibase update
```

Если Liquibase ссылается на пустую базу данных, то выполнятся команды `CREATE` и `ALTER`. Если Liquibase ссылается на базу данных, где уже была выполнена команда `CREATE`, то выполнится только команда `ALTER`. Инструменты, подобные Liquibase, часто отслеживают текущую версию

схемы базы данных в специальных таблицах метаданных, расположенных в самой БД. В таком случае не удивляйтесь, если обнаружите таблицы с такими именами, как `DATABASECHANGELOG` или `DATABASECHANGELOGLOCK`.

В предыдущем примере команда Liquibase запускается с помощью командной строки — обычно это делает *администратор базы данных*. Некоторые команды автоматизируют выполнение с помощью хука коммита или веб-интерфейса пользователя.

Не нужно связывать жизненные циклы приложения и базы данных. Привязанность миграции схемы к развертыванию приложения может привести к неожиданным последствиям. Изменения схемы чувствительны и могут иметь серьезные последствия для производительности. Разделение миграции базы данных и развертывания приложения позволяет контролировать то, когда именно внедряются изменения схемы.

Liquibase — всего лишь один из инструментов, способных управлять миграцией базы данных. Существуют и другие, например Flyway или Alembic. Большинство *объектно-реляционных отображений* (object-resource mappers, ORM) поставляются вместе с инструментами миграции схем. Если в вашей компании уже используется подобный инструмент, обращайтесь к нему, но если нет — обсудите с командой, какой именно выбрать. После того как сделаете выбор, используйте систему миграции базы данных для всех изменений. Если вы не будете применять эту систему, вся польза инструмента утратится, поскольку реальная ситуация часто отличается от тех, что складываются при отслеживании данных.

Для работы с базами данных существуют более сложные инструменты. Например, gh-ost от GitHub или pt-online-schema-change от Percona помогают администраторам базы данных вносить большие изменения в схему без снижения производительности. С помощью инструментов Skeema и Shift компании Square вы можете углубленно управлять версиями: «разбирать» схемы баз данных и автоматически создавать изменения. Подобные инструменты помогают сделать обновление базы данных безопасным.

В большинстве инструментов миграции имеется функция, позволяющая откатывать изменение миграции. Функция отката может сделать не так

много, поэтому будьте осторожны при ее использовании. К примеру, откат удаления столбца приведет к восстановлению столбца, но данные столбца при этом не восстановятся! Намного полезнее проводить резервное копирование таблицы перед ее удалением.

Из-за постоянного и масштабного характера изменений в компаниях существуют специальные подгруппы, отвечающие за правильность внесенных изменений. В них могут находиться как администраторы баз данных, операторы и SRE-инженеры, так и старшие инженеры, уже имеющие опыт работы с подобными инструментами и знакомые как с последствиями для производительности, так и с проблемами, которые могут возникнуть при работе с определенными приложениями. У таких подгрупп всегда можно научиться чему-то новому — это отличный ресурс для понимания особенностей систем хранения данных.

Поддерживайте совместимость схем

Записанные на диск данные имеют те же проблемы с совместимостью, что и API. Подобно API, устройства чтения и записи данных могут изменяться независимо, могут быть разными программами и находиться на разных устройствах. Как и в случае с API, у данных есть схемы с именами и типами полей. Любое изменение схем, не соответствующее прямой или обратной совместимости, может привести к ошибке в работе приложений. Для обнаружения несовместимых изменений используйте проверки совместимости схем, а также устройства обработки и передачи данных для изолирования внешних и внутренних схем.

Разработчики считают, что базы данных — это та часть реализации, которая скрыта от других систем. Полностью инкапсулированные базы данных идеальны, однако на практике используются не так часто. Даже если рабочая база данных скрыта приложением, данные чаще всего передаются в хранилища данных.

Хранилище данных — это база данных, используемая для целей анализа и отчетности. Компании создают конвейер *извлечения, преобразования* и *загрузки (extract, transform, load, ETL)* данных, который извлекает данные из рабочих БД, преобразует и загружает их в хранилище данных (рис. 11.3).

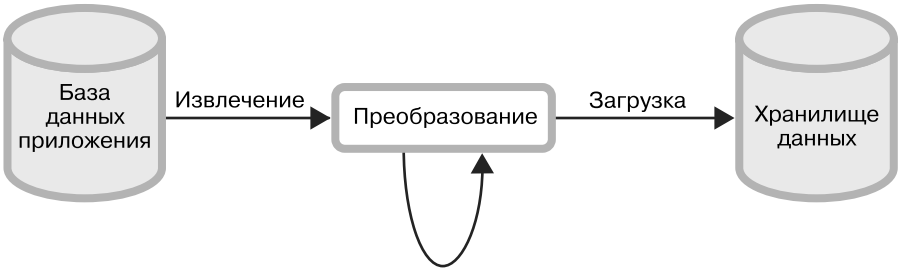

Рис. 11.3. Конвейер извлечения, преобразования и загрузки данных

Конвейеры ETL строго зависят от схем базы данных. Удаление одного столбца в рабочей среде базы данных может привести к остановке работы всего конвейера. Даже если удаление столбца никак не влияет на работу конвейера, пользователи ниже по потоку выполнения могут использовать поле для отчетов, моделей машинного обучения или специализированных запросов.

Другие системы тоже могут зависеть от ваших схем баз данных. *Захват изменения данных* (CDC) — это архитектура на основе событий, преобразующая операции вставки, обновления и удаления в сообщениях для последующих пользователей. Вставка в таблицу «members» может активировать сообщение, которое служба электронной почты использует для отправки письма новому пользователю. Подобные сообщения для API остаются неявными, а внесение обратно несовместимых изменений схемы может нарушить работу остальных сервисов.

Конвейеры хранилищ данных и последующие пользователи должны быть защищены от изменений, нарушающих обратную совместимость схем. Перед внедрением изменений в рабочую среду убедитесь, что они безопасны. Лучше всего проводить проверку совместимости на ранних этапах работы — например, во время выполнения коммита кода путем проверки операторов DDL. Выполнение операторов DDL в среде тестирования поможет дополнительно защитить изменения. Если хотите убедиться в исправной работе других сервисов, запустите операторы DDL и интеграционные тесты.

Вы можете защитить внутренние схемы с помощью экспорта *продуктов данных*, которые отделяют внутренние схемы от последующих пользователей. Продукты данных связывают внутренние схемы

с пользовательскими; команда разработчиков владеет рабочей базой данных и опубликованными продуктами данных. Отдельные продукты данных, которые могут быть просто представлениями базы данных, помогают командам поддерживать совместимость с потребителями данных без необходимости замораживать внутренние схемы базы данных.

Что следует и чего не следует делать

Следует	Не следует
Запомнить принцип YAGNI: «Вам это не понадобится»	Писать много ненужных абстракций
Использовать стандартные библиотеки и паттерны обновления	Писать методы со скрытым порядком или скрытыми требованиями к аргументам
Использовать язык определения интерфейсов для определения вашего API	Удивлять разработчиков необычным кодом
Создавать версии внешних API и документации	Добавлять в API несовместимые изменения
Изолировать друг от друга базы данных приложений	Быть догматичными в отношении управления внутренними версиями API
Определять явные схемы для всех ваших данных	Внедрять слабоструктурированные данные в строковое или байтовое поле
Использовать средства миграции для автоматизации управления схемой базы данных	
Поддерживать совместимость схем, если пользователи ниже по потоку выполнения используют ваши данные	

Повышение уровня

Подробную информацию об эволюционных архитектурах вы можете найти в книге *Building Evolutionary Architectures*[1] Нила Форда, Ребекки Парсонс и Патрика Куа (Neal Ford, Rebecca Parsons, Patrick Kua) (O'Reilly Media, 2017). Если хотите узнать больше об эволюционных

[1] *Форд Н., Парсонс Р., Куа П.* Эволюционная архитектура. Поддержка непрерывных изменений. — СПб.: Питер, 2019.

базах данных и API, мы также рекомендуем эту книгу. В издании кратко затрагивается и тема предметно-ориентированного программирования — более подробно она рассматривается в книге *Implementing Domain Driven Design*[1] Вона Вернона (Vaughn Vernon) (Addison-Wesley Professional, 2013).

В начале главы мы привели цитату из замечательной книги Джона Оустерхаута *A philosophy of software design* (Yaknyam Press, 2018). Советуем вам ее прочитать, чтобы больше узнать о сложности и управлении сложностью.

Elements of Clojure Зака Телмена (Zach Tellman) (lulu.com, 2019) — прекрасная книга, состоящая всего из четырех глав: «Имена», «Идиомы», «Перенаправление» и «Компоненты». В ней дается краткая и понятная информация об этих четырех элементах, которая поможет вам создавать эволюционные архитектуры (даже если вы никогда не будете работать с Clojure).

У Ричарда Хикки (Richard Hickey) есть прекрасное выступление под названием *Simple Made Easy* (https://www.youtube.com/watch?v=oytL881p-nQ). Хикки рассуждает о простоте, легкости, сложности, а также о способах создания хорошего ПО. Обязательно к просмотру!

В книге Жамак Дегани (Zhamak Dehghani) *Data Mesh: Building a Next Generation Data Architecture* подробно рассматриваются data-продукты.

В книге Мартина Клеппмана (Martin Kleppman) *Designing Data-Intensive Applications*[2] (O'Reilly Media, 2017) обсуждаются вопросы эволюции данных, схем, языков определения интерфейсов и захвата изменения данных. Эта книга — классика программирования, и мы советуем обязательно ее прочитать.

---

[1] *Вернон В.* Реализация методов предметно-ориентированного проектирования..

[2] *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2023.

12 Гибкое планирование

Процесс разработки ПО должен планироваться и отслеживаться. Для эффективной работы вашим коллегам следует знать, над чем вы работаете, поэтому командам нужно отслеживать прогресс — так они смогут планировать будущую работу и корректировать направление деятельности по мере обнаружения новой информации в процессе разработки. Без заранее спланированного процесса выполнение проектов затягивается, требования извне смещают фокус работы, а оперативные вопросы отвлекают разработчиков.

Гибкая (Agile) разработка — это методика разработки программного обеспечения, которая часто используется для быстрой доставки ПО. Понимание основного принципа деятельности и целей общих процессов гибкой разработки, таких как планирование спринта, ежедневные стендапы, обзоры и ретроспективы, поможет вам использовать все это с пользой. В текущей главе вы познакомитесь с основами гибкого планирования, а также с ключевыми практиками Scrum (широко распространенного метода гибкой разработки), так что сразу сможете взяться за дело.

Манифест Agile

Перед тем как разбираться в методах гибкой разработки, нужно понять философию этой методологии. Она появилась в 2001 году как результат сотрудничества лидеров предыдущих процессов разработ-

ки, таких как экстремальное программирование, Scrum, разработка, управляемая функциональностью, и прагматичное программирование. Создатели написали Манифест Agile (https://agilemanifesto.org/), в котором перечислили базовые ценности и принципы, лежащие в основе методики.

> *В процессе непосредственной разработки ПО и помощи другим программистам в их проектах мы постоянно открываем для себя более совершенные методы разработки. Благодаря проделанной работе мы смогли осознать, что:*
>
> - **люди и взаимодействие** *важнее процессов и инструментов;*
> - **работающий продукт** *важнее исчерпывающей документации;*
> - **сотрудничество с заказчиком** *важнее согласования условий контракта;*
> - **готовность реагировать на изменения** *важнее следования первоначальному плану.*
>
> *То есть, не отрицая важности того, что написано справа, мы все-таки больше ценим то, что слева.*

Манифест может показаться немного странным, однако в нем затрагиваются важные моменты. Методики гибкой разработки сосредоточены на взаимодействии с коллегами по команде и клиентами, на принятии и внедрении изменений и на фокусировке внимания на пошаговом улучшении проекта. Обычно гибкую разработку противопоставляют каскадной модели, которую перестали широко использовать тогда, когда проекты начали планироваться с самых первых шагов.

Иронично, но как только методика гибкой разработки стала популярной, некоторые компании просто-таки захватили «специалисты» — настоящие ниндзя с черным поясом по Agile: появилось множество сертификатов и консультантов по процессам. Люди были одержимы «правильным» способом «выполнять гибкую разработку», часто в ущерб самому главному принципу: «Люди и взаимодействие важнее процессов и инструментов».

Фреймворки планирования в гибкой разработке

Скрам (Scrum) и Канбан (Kanban) являются двумя самыми востребованными методологиями гибкой разработки. *Скрам*, более популярная методика, рекомендует работу небольшими фиксированными промежутками времени с контрольными точками для корректировки планов. Разработка делится на промежутки, называемые *спринтами*. Длительность спринта может быть разной, но обычно составляет две недели. В самом начале спринта команда проводит встречу для *планирования спринта* и разделения работы. После планирования разработчики берутся за дело. Работа отслеживается в *пользовательских историях* или *задачах*. Прогресс можно посмотреть в системе *отслеживания ошибок* или *обработки заявок*. Каждый день проводится стендап, на котором обсуждаются новости и сообщается о возникших ошибках. После спринта проходит *ретроспектива*, где анализируется вся проделанная работа, обсуждаются результаты, рассматриваются основные метрики, а также происходит согласование процесса. *Ретроспективы* передают данные процессу планирования следующего спринта, создавая контур обратной связи от плана к разработке, затем к ретроспективе, а затем обратно к плану.

Система Канбан не использует спринты фиксированной длины, как Скрам. Вместо этого Канбан определяет этапы рабочего процесса, через которые проходят все рабочие элементы, такие как бэклог, планирование, внедрение, реализация, тестирование, развертывание и выгрузка. Команды очень часто настраивают этапы Канбан в зависимости от своих потребностей. Этот подход ограничивает количество *работ, находящихся в процессе выполнения* (WIP), устанавливая лимит задач на каждом из этапов. Из-за ограничения количества тикетов команды должны завершать существующие задачи, прежде чем приступать к новой работе. Канбан-доски — это панели с вертикальными колонками для каждого из этапов работы. Задачи, представленные в виде окошка с названием, перемещаются между колонками в момент, когда их статус изменяется. Канбан-доски позволяют отслеживать работу в процессе и указывать на существующие проблемы, например на застой работ на определенном этапе. Если доска показывает, что бóльшая часть работ уже долгое время стоит на этапе тестирования, то команда должна немного изменить свой

план: часть действий по разработке перемещается в бэклог, а для работы с тестированием выделяются несколько инженеров. Больше всего Канбан подходит для команд, которые занимаются не длительными масштабными проектами, а обработкой входящих запросов — например, для групп инженеров службы поддержки или SRE-инженеров.

Очень часто команда выбирает одну методологию, но не может реализовать ее идеально. Например, команда выбирает для себя Канбан и ее методы, но при этом берет какие-то практики из Скрам, другие практики Канбан изменяет, третьи просто отбрасывает. Независимо от того, использует ли ваша компания Скрам, Канбан, смесь двух методик под названием *Скрамбан (Scrumban)* (и такое реально!) или любой другой вариант гибкой разработки, процесс планирования должен быть направлен на разработку качественного ПО для удовлетворения потребностей заказчиков. Сосредоточьтесь на поставленных целях, а не на технике. Экспериментируйте и измеряйте результаты, оставьте то, что работает, и отбросьте все остальное.

Скрам

Большая часть команд разработчиков ПО практикуют ту или иную форму Скрам, поэтому вам нужно понять, как именно эта методика работает. Обычно планирование начинается с подготовки. Разработчики и продакт-менеджер создают новые *пользовательские истории*. Тикеты из *бэклога* сортируются. Сложность историй оценивается в единицах под названием *story point*, затем они разбиваются на *задачи*. Более крупные истории разрабатываются и исследуются с помощью *спайков*. Во время планирования спринта команда выбирает, какие именно истории нужно будет завершить в течение следующего спринта, и использует story point для предотвращения чрезмерной нагрузки.

Пользовательские истории

Пользовательская история — это особая заявка, в которой запрос выражается с точки зрения конечного пользователя: «Как <пользователь>, я <хочу>, <чтобы>». Например: «Как администратор, я хочу предоставить бухгалтерам право просматривать входящие выписки по счетам».

Написание ориентированных на пользователя историй позволяет разработать и создать продукт, который будет иметь для пользователя наивысшую ценность.

Распространенный вариант неправильного применения пользовательских историй — попытка втиснуть в историю обычное описание задачи, например: «Как разработчик, я хочу обновить плагин шейдера до версии 8.7» или «Как пользователь, я хочу, чтобы политика конфиденциальности находилась внизу страницы». В подобных историях нельзя понять главную причину изменений. Зачем нужно обновлять плагин шейдера, что от этого изменится и кому это нужно? Политика конфиденциальности нужна пользователю или тому, кто ответствен за соблюдение требований? Если вы собираетесь писать не задачи, а истории, пишите правильно.

Кроме названия и описания, у историй есть и другие атрибуты. Двумя наиболее распространенными являются *критерий приемки* и *оценка*. Оценки пользовательской истории представляют собой предложения о действиях, которые нужно совершить для реализации истории. Критерии приемки позволяют разработчикам, продакт-менеджерам, тестировщикам ПО и пользователям понимать друг друга. Попробуйте написать явные тесты для каждого критерия приемки.

- На странице прав администратора выводится «выписка по счетам».
- Пользователи без прав администратора с разрешением на просмотр «выписок по счетам» могут просматривать все выписки по счетам аккаунта.
- Для пользователей без прав администратора кнопка Редактировать на странице выписок по счетам аккаунта будет скрыта.
- Пользователи без прав администратора не могут редактировать выписки по счетам.
- Пользователи без прав администратора с правом редактирования и просмотра выписок по счетам могут как редактировать, так и просматривать выписки по счетам.

Чаще всего небольшие истории используются в качестве тикета, а большие истории связаны с тикетами по разработке или подзадачами. Если историю нужно доработать, то проводится спайк. Спайк — это

ограниченное по времени исследование, позволяющее закончить истории. В результате проведения спайка появляются проектные документы, делается выбор между приобретением или собственной разработкой проекта, оцениваются компромиссы и т. д. Для получения дополнительной информации о проектировании читайте главу 10.

Задачи

Иногда одну историю нужно разделить на небольшие задачи, чтобы оценить, сколько времени уйдет на реализацию всей истории. В таком случае работа делится между несколькими разработчиками и далее отслеживается процесс реализации. Хороший способ разбить работу на несколько частей — писать подробные описания. Прочитайте описание и попытайтесь найти все задачи.

> *Нам нужно добавить параметр* `retry` *в* `postProfile`. *На данный момент профили не обновляются, когда происходит тайм-аут сети. Мы, вероятно, захотим ограничить количество повторных попыток и добавить экспоненциальную выдержку, чтобы блокировка не была слишком долгой. Нам следует поговорить с продакт-менеджером и узнать, как долго он готов ждать размещения профиля.*
>
> *После реализации нужно запустить модульные и интеграционные тесты. В интеграционных тестах необходимо воссоздать реальные тайм-ауты для проверки корректной работы выдержки.*
>
> *После тестирования следует провести развертывание сначала в тестовой среде, затем в эксплуатационной. В эксплуатационной среде нужно разделить трафик и постепенно увеличивать количество повторных попыток, так как* `postProfile` *очень чувствителен к изменениям.*

Такая простая вещь, как добавление параметра `retry`, состоит из множества шагов: согласования спецификации с продакт-менеджерами, программирования, проведения модульного и интеграционного тестирования, развертывания и выгрузки. Разделение работы на несколько задач помогает отслеживать все выполняемые шаги.

Стори поинты

Производительность команды измеряется в общепринятых единицах (измерение ведется в часах, днях или в уровнях сложности выполняемой задачи) под названием стори поинты (story point, еще называют баллами истории). Производительность спринта — это количество разработчиков, умноженное на количество стори поинтов на каждого разработчика. Например, если в команде 4 разработчика и на каждого из них приходится 10 стори поинтов, то производительность составит 40 баллов. Пользовательские истории также могут оцениваться в стори поинтах. Количество стори поинтов в спринте не должно превышать общей производительности спринта.

Многие команды в качестве единицы измерения используют время, где один стори поинт — это один рабочий день. Если в качестве единицы измерения выбран день, то учитывается и работа помимо выполнения основных задач, например встречи, перерывы, ревью кода, а рабочий день составляет 4 часа.

Другие команды в качестве стори поинтов рассматривают сложность задачи, используя «метод размера футболок»: 1 поинт — очень маленькая (XS), 2 поинта — маленькая (S), 3 поинта — средняя (M), 5 поинтов — большая (L), 8 поинтов — очень большая (XL). Звучит знакомо? Это последовательность Фибоначчи! Увеличение количества стори поинтов в соответствии с последовательностью Фибоначчи позволяет разрешить некоторые разногласия по поводу трех и трех с половиной поинтов. Пропуски между стори поинтами заставляют команду задуматься о том, сколько поинтов стоит присвоить задаче. Увеличение пропусков при работе со сложными задачами объясняет неточность оценки в большом проекте.

Концепция гибкой разработки не одобряет оценки, основанные на времени. Специалисты-практики утверждают, что при встречах создается эмоциональная привязанность и они не представляются чем-то сложным. Иные варианты оценок помогают упростить выражение неопределенности. Может показаться, что для изменения одного метода требуется небольшой объем работы, но если этот метод невероятно сложен, то

времени и сил уйдет немало. Мысленно легче сказать, что «это задача средней сложности», чем «это займет у меня три полных рабочих дня».

Чаще всего в качестве единицы измерения стори поинтов используется соотношение времени и сложности, а также общая полезность. Мы не нашли никаких аргументов в пользу применения той или иной единицы измерения с точки зрения продуктивности, поэтому рекомендуем вам использовать то, что больше подходит для вашей команды.

Оценка стори поинтов — дело очень субъективное, и чаще всего с оценкой люди ошибаются. Использование *относительных величин* является одним из способов повышения точности оценивания. Относительная величина определяется путем высчитывания стори поинтов для уже выполненной задачи, а затем сравнения выполненной задачи с текущей. Если в текущей задаче предстоит сделать меньше работы, она получит меньшее количество стори поинтов; если нужно сделать больше работы, то логично, что она получит больше стори поинтов. В случае схожести задач их стоит оценить одинаковым количеством баллов. Иногда используются такие процессы, как *покер планирования*: даже если вы не принимаете в них участия, после просмотра уже выполненной работы вы получите представление о ценности стори поинтов вашей команды.

Обработка бэклога

Обработка бэклога, или *груминг* (в значении обрезки деревьев), происходит до совещаний по планированию. Бэклог — это список возможных историй. Обработка используется для того, чтобы он оставался актуальным и с правильно расставленными приоритетами. Продакт-менеджеры вместе с менеджерами по разработке (иногда и с разработчиками) просматривают невыполненную работу. В список добавляются новые истории, убираются старые, обновляется статус незавершенных историй, а работа с высоким приоритетом перемещается в начало списка. Упорядоченный и грамотно составленный бэклог помогает обсуждать нужные вещи на совещаниях по планированию.

Планирование спринта

Совещание по планированию спринта проводится после окончания подготовительных работ. На совещаниях по планированию работают все вместе: инженеры и продакт-менеджеры решают, какой работе нужно уделить особое внимание. Расставляются приоритеты историй и определяется, какая из них будет соответствовать производительности спринта.

Производительность спринта рассчитывается исходя из того, какой объем работы был выполнен в предыдущих спринтах. При определении производительности спринта также учитывается, кто из членов команды из нее уходит, кто приходит в команду, у кого запланирован отпуск или дежурство.

Главной особенностью спринта является его длительность — обычно она составляет две недели. Короткие спринты делают работу выполнимой. При проведении небольших спринтов командам приходится разбивать крупные задачи на несколько более мелких. Разбивка задачи дает возможность работать над проектом одновременно нескольким разработчикам. Маленькие циклы разработки с частыми стендапами и обзорами позволяют выявлять ошибки раньше.

После завершения планирования спринт считается зафиксированным. Если во время спринта всплывает новая работа, то не стоит сразу за нее браться — лучше перенести в бэклог и запланировать ее выполнение на следующий спринт. Фиксация спринтов позволяет сосредоточиться на запланированной работе и обеспечить предсказуемость процесса. Если же во время спринта новая работа все-таки начинается, то ожидается, что во время ретроспективы команда рассмотрит причины ее возникновения и попытается уменьшить объем незапланированной работы для будущих спринтов.

Однако строгое соблюдение практики планирования спринта встречается нечасто. Большинство команд тщательно выбирают, что им делать: некоторые выполняют предварительную работу на совещании по планированию спринта, а у других команд нет продакт-менеджера и распределение работы ложится на плечи самих разработчиков.

Многие команды не задействуют пользовательские истории, а выбирают задачи или сообщения об ошибках в более открытом формате. Не думайте, что у всех команд процесс планирования спринта проходит одинаково.

Стендапы

После завершения планирования спринта начинается основная работа, и команда проводит стендап, который также называется скрам-встречей, или собранием. Стендапы информируют всех о вашем прогрессе, делают вас более сосредоточенным и ответственным, а также дают команде возможность отреагировать на все, что ставит цели спринта под угрозу.

Обычно стендапы — это 15-минутные встречи, проходящие каждое утро: достаточно короткие, чтобы отстоять их не составляло большого труда (хотя стоять и необязательно). В ходе встречи члены команды по очереди берут слово и рассказывают о том, над чем они работали с момента предыдущего стендапа, над чем планируют работать и нашли ли какие-нибудь ошибки или проблемы, способные поставить выполнение спринта под угрозу. Несмотря на то что чаще всего подобные встречи происходят очно, некоторые команды практикуют асинхронные стендапы. В асинхронном стендапе появившееся обновление ежедневно отправляется в чат-бот или групповым электронным письмом.

Стендап служит проверкой системы, что можно сравнить со взглядом на приборную панель вашего автомобиля: вы видите, что у вас достаточно бензина и лампочка «проверьте двигатель» не горит. Быстро озвучьте появившиеся обновления: на стендапе не стоит заниматься решением проблем. Постарайтесь говорить только о самом важном, но при этом задавайте любые интересующие вас вопросы. Сообщайте, если столкнулись с чем-то новым: ошибками, неожиданным поведением и т. д. Обсуждение ваших «открытий» может происходить на *парковке* (не в буквальном смысле, конечно).

При проведении синхронных стендапов старайтесь на них не опаздывать. Если стендап связан с обновлением статуса тикетов или проблем,

попробуйте заранее обновить эти тикеты. Узнавая, какие обновления вносят другие члены команды, вы должны искать способы снижения рисков возникновения проблем при завершении спринта: если кто-то говорит, что на выполнение задачи у него уходит больше времени, чем планировалось, помогите этому человеку решить данную проблему (конечно, если располагаете свободным временем).

Дискуссии «на парковке» происходят после общих встреч. Это позволяет сделать стендапы короткими, а также помогает убедиться в том, что обсуждения актуальны для всех членов команды. Когда кто-то говорит: «Оставим это до парковки», — это значит, что обсуждение стоит продолжить после окончания стендапа.

Нет ничего страшного, если вы пропустите стендап из-за того, что он не совпадает с вашим рабочим расписанием. Обратитесь к руководителю и узнайте, как вы можете поделиться своими новостями и узнать о других. Асинхронные стендапы «пропускаются» очень редко.

Есть множество вариантов стендапов и скрам-встреч. Вы можете услышать такие выражения, как scrum of scrums или Scrumban. Scrum of scrums — это модель, в которой лидер каждой отдельной скрам-встречи выбирается для участия в следующей скрам-встрече, где представители всех команд собираются вместе, чтобы сообщить о прогрессе своей команды и выявить взаимозависимости между разными командами. Модель scrum of scrums распространена в деятельности по использованию ПО, когда каждая команда отправляет инженера (обычно дежурного) на скрам-встречу по эксплуатации, чтобы быть в курсе проблем использования ПО. Scrumban — это гибрид методологий Скрам и Канбан. Вам нужно понимать, как работает ваша команда и компания в целом, и в своей работе придерживаться этих установок.

Обзоры

Обзоры проводятся между спринтами. Обычно они разделены на два этапа: демонстрация и обзор проекта. Во время демонстрации каждый член команды рассказывает о своем прогрессе в спринте, после чего

команда оценивает текущий спринт в сравнении с его целью. Если спринт прошел успешно, то все поставленные цели будут выполнены, а также будут завершены все истории.

Структуры обзорных совещаний сильно отличаются. Для некоторых команд основным на встрече является демонстрация, в то время как другие команды сосредоточиваются на обзоре проекта. Но у многих команд обзоров нет вообще. Если в компании проводятся обзоры, относитесь к ним со всей серьезностью: оставляйте полезную обратную связь и не преуменьшайте значения проделанной работы. Польза, которую вы получаете от обзоров, равна вложенным в них усилиям.

На одну неделю спринта обычно приходится не более одного часа обзора — таким образом, у двухнедельного спринта будет двухчасовой обзор. Все собираются за столами или в конференц-зале для демонстраций и по очереди показывают, над чем работали. Встреча носит неформальный характер. После этого цели спринта анализируются и оцениваются на предмет их выполнения.

Не стоит тратить слишком много времени на подготовку к обзору спринта: пары минут для определения того, что вы собираетесь показывать, будет достаточно. Убедитесь, что статусы ваших тикетов актуальны. Демонстрации носят неформальный характер, поэтому старайтесь избегать официальных презентаций или речей.

Обзоры отмечают командные успехи, позволяют ощутить сплоченность коллектива, предоставляют возможность обратной связи и поддерживают актуальность информации о прогрессе команд. На обзоре у команды появляется шанс честно поговорить о том, как идет работа. Не все разработчики трудятся над одними и теми же проектами, так что обзоры помогают участникам команды узнать, чем занимаются другие. Синхронизация действий членов команды позволяет обмениваться отзывами и отмечать хорошо сделанную работу: так укрепляется командный дух. Во время обзоров проекта разработчики определяются с тем, какие задачи уже выполнены, а что еще только предстоит сделать. Обнаруженные проблемы команда может обсудить на ретроспективе спринта.

Ретроспективы

Один из 12 принципов Манифеста Agile звучит так: «Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы». На ретроспективе команды придерживаются этого принципа.

При проведении ретроспективы команда собирается вместе и обсуждает объем выполненной работы, а также то, что не получилось сделать за время, прошедшее с предыдущей ретроспективы. Обычно ретроспективная встреча состоит из трех этапов, таких как беседа, определение приоритетов и принятие решений.

Лидер, или *скрам-мастер*, начинает встречу с вопроса к участникам команды о том, что получилось сделать во время последнего спринта, а что — нет. На встрече высказываются все, а скрам-мастер записывает сказанное в виде списка на доске или документа, который раздается всем участвующим. Команда обсуждает значимость элементов, сработавших не так, как ожидалось: от какого из них больше всего проблем? В конце команда проводит мозговой штурм, пытаясь найти решение для основных проблем.

Не бойтесь что-то менять. Метод гибкой разработки должен быть пластичным, и об этом тоже говорится в манифесте: «Люди и взаимодействие важнее процессов и инструментов». Перед каждой встречей уделите немного времени размышлениям о том, что именно может помочь вашей команде стать лучше. Выскажите свои мысли на ретроспективе.

Часто ретроспективы путают с обзорами. В обзорах в центре внимания работа, выполненная во время спринта, в то время как ретроспективы сосредоточены на процессах и инструментах. Обычно ретроспективы проводятся между спринтами и сразу после обзорных совещаний. Многие команды перед началом спринта объединяют обзор, ретроспективу и планирование спринта в одну встречу. Такой способ допустим, если каждый из этапов — обзор, ретроспектива и планирование спринта — проходит отдельно.

Ретроспективы также являются причиной того, почему существует множество вариантов методики гибкой разработки. Командам

рекомендуется часто пересматривать методы разработки ПО. Постоянный пересмотр гарантирует то, что две команды не будут использовать гибкую разработку одинаковым способом.

Дорожные карты

Двухнедельные спринты — это отличный способ сделать небольшой или средний проект, однако для выполнения масштабных проектов необходимо проводить более тщательное планирование. Заказчики устанавливают сроки работы, которых должны придерживаться разработчики. Специалистам нужно понимать, каким командам требуется больше инженеров. Крупные проекты следует делить на части, планировать и координировать.

При долгосрочном планировании руководители используют дорожные карты. Обычно дорожные карты разбиты на кварталы: январь — март, апрель — июнь, июль — сентябрь и октябрь — декабрь.

Перед началом каждого квартала проводится планирование. Менеджеры по разработке ПО, продакт-менеджеры, разработчики и другие участники проекта собираются, чтобы обсудить работу и намеченные цели. Обычно планирование состоит из серии встреч и нескольких этапов обсуждений.

В 11-м томе издания *The Papers of Dwight David Eisenhower* можно прочитать такие слова Дуайта Эйзенхауэра: «В ходе подготовки к бою я всегда считал, что планы — ничто, планирование — все». То же самое можно сказать и о дорожных картах. Мы никогда не видели, чтобы дорожная карта на год или квартал полностью совпала с реальностью, но это и не главное. Дорожные карты должны побуждать всех думать о том, что команда создает в долгосрочной перспективе, а намеченный в дорожной карте план не строго обязателен к исполнению. На ближайшие кварталы нужно строить конкретные планы, на отдаленные — более общие планы с возможностью корректировки. Не обманывайте себя, думая, что план на квартал будет полностью выполнен.

В отличие от спринтов, которые фиксированы, в дорожные карты можно вносить изменения. Требования заказчика могут измениться, из-за чего

возникнут новые проблемы. В этой ситуации помогут планирование спринта, обзоры и ретроспективы, так как они позволяют менять то, что запланировано. При изменении дорожных карт ключевое значение имеет коммуникация. Команды должны заранее знать о том, что работа изменяется или останавливается.

Многие команды проходят через ежегодные циклы планирования: в четвертом квартале руководители пытаются спланировать работу на четыре квартала следующего года. Годовое планирование, как правило, представляет собой только обсуждение и разговоры о будущем. Тем не менее часто оно влияет на распределение ресурсов или фактическую численность сотрудников — на корпоративном языке так обозначается то, где будут работать вновь принятые на работу инженеры. Обсуждаются обычно большие проекты, на которые команда потратит большую часть рабочего времени. Не переживайте, если проект, над которым вы работаете, не упоминается в ходе годового планирования: в самом конце совещания задайте руководителю вопрос о том, на каком этапе находится ваш проект.

Что следует и чего не следует делать

Следует	Не следует
Постоянно выпускать небольшие обновления	Зацикливаться на «правильном» использовании гибкой разработки
Указывать подробные критерии приемки для историй	Бояться менять процессы гибкой разработки
Браться только за такую работу в спринте, которую сможете довести до конца	Делать из обычного описания задач истории
Разбивать большую работу на части, если вы понимаете, что не сможете закончить ее в рамках спринта	Забывать отслеживать проектирование и планирование работы
Использовать стори поинты для оценки работы	Добавлять работу после начала спринта, если фиксированная работа еще не выполнена
Использовать относительные величины и принцип «размера футболок» при оценке проекта	Слепо следовать процессам

Повышение уровня

Из большинства книг по гибкой разработке вы не узнаете ничего нового. В них во всех подробностях рассматриваются сам подход и множество его вариантов. Книги больше подходят для руководителей проектов, а вам мы советуем изучать онлайн-ресурсы.

У Манифеста Agile, который упоминался в этой главе, есть страница под названием *Principles behind the Agile Manifesto* (http://agilemanifesto.org/principles.html). Познакомьтесь с принципами для лучшего понимания самой философии Agile.

Статьи на Atlassian (https://www.atlassian.com/agile/) станут хорошим источником практической информации. Там вы сможете прочитать обо всем: начиная от управления проектами и планирования дорожных карт и заканчивая DevOps в Agile. Если ваша команда чаще использует Канбан, а не Скрам, статьи о Канбан будут для вас бесценны.

13 Взаимодействие с менеджментом

Хорошие рабочие отношения с менеджером-управленцем (далее мы будем называть таких руководителей просто менеджерами) способствуют карьерному росту, уменьшают стресс и даже помогают создавать и поставлять надежное ПО. Для качественной работы необходимо добиться взаимопонимания с начальством. Вы должны понимать, что требуется вашему руководителю, чтобы суметь это предоставить. Точно так же руководитель должен понимать ваши потребности, чтобы помочь вам в работе.

Из этой главы вы узнаете, как можно выстроить эффективные взаимоотношения с руководством. Мы кратко расскажем о работе менеджера — что и как он делает. Рассмотрим также общие процессы управления. Очень часто инженеры сталкиваются с такими сокращениями, как 1:1, PPP и OKR, и с определенными терминами, например performance review, но при этом не знают, что они обозначают и для чего используются. Мы покажем, как из всей этой информации можно извлечь максимальную пользу. В конце вы найдете несколько советов по концепции управления вверх, а также раздел о том, как стоит вести себя с плохими менеджерами. К концу главы вы будете обладать набором инструментов и методик для выстраивания эффективных отношений.

Что делают менеджеры

Менеджеры всегда присутствуют на собраниях, но что на самом деле они *делают*, многим непонятно. Менеджеры по разработке ПО работают

с людьми, продуктом и самим процессом. Они собирают команды, набирают и обучают инженеров, а также следят за отношениями внутри коллектива. Планируют и координируют разработку продуктов. Они также могут участвовать в технической части разработки ПО — в проведении ревью кода или создании архитектуры кода, — однако хорошие менеджеры редко сами пишут код. Они следят за эффективностью всей команды. То есть они управляют всем, работая на разных уровнях: на верхнем — с вышестоящим начальством, на среднем — с другими менеджерами и на нижнем — со своей командой.

Они управляют вверх с помощью выстроенных отношений и коммуникации с руководителями, занимающими высокие должности. Менеджеры — связующее звено между директорами, принимающими решения на самом верху, и рядовыми инженерами. Так называемый восходящий менеджмент имеет решающее значение для получения ресурсов (денег и специалистов) и обеспечения того, чтобы команду признали, оценили и услышали.

Менеджеры работают на среднем уровне, сотрудничая с другими менеджерами. У менеджера есть две команды: одна — это группа людей, которой он управляет, а вторая — коллеги-менеджеры. Пул равноправных менеджеров работает вместе, чтобы команды согласованно достигали общих целей компании. Поддерживание отношений, четкая коммуникация и совместное планирование позволяют обеспечить комфортную и эффективную работу команд.

Менеджеры управляют своей командой и отслеживают прогресс текущих проектов, устанавливают цели и предоставляют обратную связь, обеспечивают прозрачность приоритетов, нанимают и увольняют работников, а также поддерживают командный дух.

Коммуникация, цели и процессы роста

Менеджеры создают процессы, которые обеспечивают постоянную работу команд и отдельных разработчиков. В главе 12 мы подробно рассмотрели структуру процессов гибкой разработки, ориентированную на командную работу. В данном разделе вы познакомитесь с методами,

которые будете использовать для поддержания хороших отношений с непосредственным руководителем.

Методы «один на один» (1:1) и «прогресс, планы, проблемы» (PPP) используются для коммуникации и оповещений, в то время как стратегия целей и ключевых результатов (OKR) и методология performance review предназначены для управления целями и ростом. Больше пользы эти методы принесут вам, если вы будете знать, как правильно их применять.

Один на один (1:1)

Ваш руководитель должен запланировать еженедельные или раз в две недели встречи с вами один на один. *Встречи 1:1* — это специальное время, когда вы можете обсудить с руководителем насущные темы и общие проблемы и выстроить продуктивные долгосрочные отношения. Встречи 1:1 — хорошо известная практика, но она зачастую не подходит в качестве сессии устранения неполадок или проверки текущего статуса задач.

Вы должны задать повестку такой встречи и много говорить. Кратко обсудите с менеджером намеченную повестку. Сохраняйте заметки о встречах и обращайтесь к ним во время разговора с менеджером. Менеджер может по мере надобности добавлять собственные вопросы и темы, однако на встречах формата 1:1 внимание в первую очередь должно уделяться не повестке начальника, а вашей.

Мы затронули два важных аспекта таких встреч, давайте сформулируем их более кратко: именно вы устанавливаете повестку дня; встреча один на один не предназначена для обновления статуса текущих задач. Если вы не будете придерживаться этих установок, то потратите несколько часов не на продуктивные разговоры, а впустую. Используйте следующие подсказки, пытаясь найти темы для обсуждения.

- **Ситуация в целом.** Какие вопросы у вас есть о деятельности компании? Имеются ли вопросы, касающиеся организационных изменений?

- **Обратная связь.** Что можно было бы улучшить? Что вы думаете о процессе планирования работы команды? С какой самой большой технической проблемой вы сталкиваетесь? Что бы вы хотели сделать из того, что не можете сделать сейчас? Какая у вас самая большая проблема? Какая, на ваш взгляд, самая большая проблема у компании? С какими препятствиями сталкиваетесь вы и ваши коллеги?
- **Карьера.** Что менеджер может вам посоветовать относительно вашего карьерного роста? Что вам следует делать лучше? Какие навыки вы хотели бы приобрести? Если у вас имеются долгосрочные цели, то как вам их достичь?
- **Личное.** Что нового произошло в вашей жизни? О каких личных проблемах должен быть уведомлен менеджер?

Встречи 1:1 помогают добиться взаимопонимания. Совершенно нормально немного поговорить на отвлеченные темы — например, о вашей кошке, о любви менеджера к ярким кроссовкам или погоде. Вы работаете над выстраиванием отношений, которые намного глубже, чем просто обмен готового кода на зарплату. Тем не менее, хотя разговоры на отвлеченные и личные темы важны, однако не позволяйте всем встречам 1:1 превращаться в дружеские посиделки.

Если вам не назначили встречу 1:1, узнайте, проводятся ли они в принципе. Практика таких встреч общепринятая, но не все ее придерживаются. Уточните у менеджера, как ему будет удобнее проводить встречи. Некоторые менеджеры предпочитают встречи 1:1 по «ленивому» принципу: время назначает другое лицо, а не они. Если встреча не запланирована, то менеджеры считают, что тем для обсуждения нет. Однако бóльшую часть времени темы для разговоров будут браться из того длинного списка, который мы предложили ранее.

Если менеджер постоянно отменяет встречи, обязательно поговорите с ним об этом. Управление командой — это часть работы менеджера, а часть управления — это общение с ее участниками. Оправдания, что «у меня нет времени», недостаточно. Если менеджер не может найти время для встречи, проблему нужно решать. Сообщения о повторных отменах встреч могут быть важным сигналом для менеджера. Разговор не должен сводиться к конфронтации: это обратная связь, в которой нуждается ваш менеджер.

ЗАБЫТЫЕ 1:1

Однажды Дмитрий принимал участие в реорганизации, из-за чего разные команды перемешались между собой. В итоге под его руководством оказалось 20 человек, часть из которых он знал и даже дружил с ними уже несколько лет, но некоторых видел впервые. Предстояла большая работа: познакомиться с новыми людьми, разработать дорожную карту, улучшить старые системы и заняться созданием новых. Через год Дмитрий обнаружил, что, будучи с командой в дружеских отношениях, он забыл о проведении регулярных встреч с одним из своих подчиненных. Он бы и не узнал об этом, если бы работодатель не сообщил, что данный инженер хочет перейти в другую команду. Среди причин такого решения было то, что подчиненному хотелось работать с менеджером, который участвовал бы в развитии его карьеры. Дружеские отношения не являются заменой встреч 1:1. Менеджер, слишком занятый для проведения 1:1, может быть слишком занят и для того, чтобы быть менеджером!

Вы можете рассчитывать на подобные встречи и с теми, кто не является вашим менеджером. Обратитесь к специалистам, у которых, как вам кажется, вы можете чему-то научиться. Встречи 1:1 с более опытными коллегами будут особенно полезны в том случае, если в вашей компании нет программы наставничества. Встречи 1:1 также помогают знакомиться с различными подразделениями компании. Убедитесь, что у вас подготовлена повестка, чтобы не превращать встречу в «обновление статуса текущих задач».

PPP

Отчеты в формате PPP широко используются для обновления статуса. Обновление статуса нужно не для учета вашего времени, а для того, чтобы менеджер мог найти проблемы, с которыми вам следует разобраться, и связать вас с нужными людьми. Обновления статуса помогают найти темы для встреч 1:1 и задуматься о том, какой путь вы уже прошли, куда идете сейчас и что вам препятствует.

Как можно догадаться из названия, каждая буква P означает что-то свое. Аббревиатура PPP расшифровывается как progress, plans, problems (прогресс, планы, проблемы). В разделах должно быть от трех до пяти пунктов, в пункте — от одного до трех предложений. Пример:

02.07.2022

ПРОГРЕСС

- Отладка производительности службы уведомлений.
- Ревью кода для создания шаблонов электронной почты в службе уведомлений.
- Распространение проекта службы обнаружения спама, написана нулевая веха сервиса.

ПЛАНЫ

- Добавить метрики и средства отслеживания в службу обнаружения спама.
- Работа с командой инструментов для поддержки артефактов PyPl в безопасной среде сборки.
- Помощь при принятии на работу новых работников: прогон кода службы обнаружения спама.
- Совместная работа с администратором базы данных над добавлением индекса для превентивного устранения проблем с производительностью службы уведомлений до увеличения нагрузки в связи с праздничными днями.

ПРОБЛЕМЫ

- Затруднения, связанные с привлечением команды к ревью кода моего запроса на включение, — рассмотрения ожидают несколько таких запросов.
- Стабильная работа Redis — это проблема.
- Слишком много собеседований — в среднем четыре в неделю.

Делитесь своим отчетом PPP с менеджером и другими заинтересованными лицами: по электронной почте, через Slack или wiki. Его обновления должны делаться с постоянной периодичностью — еженедельно или ежемесячно в зависимости от принятых в компании правил.

Обновления PPP будут для вас простым делом при условии сохранения прошлых отчетов. Каждый раз, когда потребуется сделать новый PPP, создайте новую запись. Просмотрите проблемы из прошлого PPP и спросите себя: были ли решены какие-либо проблемы из списка? Какие-то проблемы перешли из прошлого PPP в нынешний? Решенные проблемы переносятся в раздел «Прогресс» текущего PPP, а оставшиеся по-прежнему размещаются в разделе «Проблемы». Взгляните на раздел «Планы» из предыдущего отчета: что-то из запланированного вы сделали? Если да, то отнесите эти пункты в раздел «Прогресс». Если нет, то вы собираетесь работать над этим до следующего PPP или возникла проблема, которая помешала выполнить запланированное? Обновите раздел «Планы» и «Проблемы» соответственно. Посмотрите на предстоящую работу и календарь. Обновите раздел «Планы» и укажите любую новую работу, которую вы планируете выполнить до следующего PPP. Весь процесс должен занимать не более пяти минут.

Стратегия OKR

Стратегия OKR — это управленческий метод, с помощью которого компании определяют цели и оценивают их возможный успех. В стратегии OKR компании, команды и отдельные лица определяют цели, а также привязывают ключевые результаты к каждой из этих целей. Отдельная цель может иметь от трех до пяти ключевых результатов, которые являются показателями прогресса в достижении цели.

Инженер, работающий над стабилизацией сервиса заказов, может определить свои цели и ключевые результаты следующим образом.

- **Цель.** Стабилизировать сервис заказов.
- **Ключевой результат.** Согласно проверке работоспособности, время безотказной работы составляет 99,99 %.
- **Ключевой результат.** Задержка 99-го процентиля (P99) меньше 20 мс.

- **Ключевой результат.** Частота возникновения 5XX ошибок ниже 0,01 % всех ответов.
- **Ключевой результат.** Команда поддержки может выполнить местную обработку отказа меньше чем за пять минут.

В идеале содержание списка OKR исходит от высшего руководства и через команды доводится до каждого человека. OKR одного человека способствует достижению цели команды, а OKR команды — достижению цели всей компании. Предложенный выше OKR инженера может использоваться в OKR команды для улучшения стабильности, в то время как OKR команды может быть частью OKR компании по повышению степени удовлетворенности клиентов.

Не превращайте перечисление ключевых результатов в список дел. Ключевые результаты должны объяснять не то, как что-то *сделать*, а то, как вы *узнаете*, когда что-то будет сделано. Существует множество способов достижения цели, и список OKR не должен вас ограничивать. Иногда ситуацию легче всего проиллюстрировать простым примером из жизни: если ваша цель — это успеть на празднование дня рождения бабушки, то ключевым результатом будет «быть в Лос-Анджелесе к 20-му числу», а не «поехать по трассе I-5 19-го числа». Поездка на машине или перелет на самолете — это возможные способы, как добраться до Лос-Анджелеса, но при этом правильно сформулированный список OKR позволяет сделать правильный выбор в момент, когда нужно будет делать выбор, а не во время составления OKR.

Обычно цели и ключевые результаты устанавливаются и оцениваются ежеквартально. Для четкого понимания целей компании и команды работайте совместно с менеджером. Для определения личного OKR лучше всего использовать цели более высокого уровня. Старайтесь не составлять много OKR — это поможет вам сосредоточиться. Оптимальное количество — от одного до трех OKR в квартал. Если у вас набирается больше 5 OKR, вы будете распылять свои силы.

Обычно цели в OKR ставятся чуть выше разумного. Амбициозные цели позволяют стремиться к более высоким достижениям и более широкому их охвату. Данная философия подразумевает, что вы не обязаны полностью выполнять все цели из OKR, иначе это было бы знаком того, что вы мало к чему стремитесь. Большинство OKR выполняются на 60—80 %, что означает достижение только 60—80 % всех целей. Если вы

выполняете более 80 % всех целей, то вы не целеустремленный человек. Выполнение меньше 60 % целей означает, что вы боитесь не оправдать ожиданий или недостаточно реалистичны. Но почему бы не установить OKR на уровне 100 % и не вознаграждать за перевыполнение? При большом количестве амбициозных целей вы вольны выбирать, от каких из них можно отказаться в середине квартала, в то время как при выполнении OKR на 100 % вы обязаны выполнить все. Узнайте, какого принципа придерживаются в вашей компании: рассматривают OKR как обязательные к выполнению цели или как амбициозные с некоторым ожидаемым уровнем неудач.

В каких-то компаниях вместо OKR используют качественные цели, в каких-то опускают букву O и сосредоточиваются только на метриках — *ключевых показателях эффективности* (KPI) — без указания цели. Вне зависимости от структуры работникам и командам нужен способ для постановки целей и оценки прогресса. Убедитесь в том, что вы понимаете, каковы ваши цели и как следует оценивать успех.

Не во всех командах принято определять индивидуальные цели: в некоторых компаниях OKR устанавливается только на уровне команды, отдела или компании. Если ваша организация придерживается такого метода, поговорите со своим менеджером об ожиданиях и о том, как их оценивают.

Оценка производительности

Раз в полгода или год менеджеры проводят формальные оценки производительности коллектива. Корректировка должностей и размеров вознаграждения, как правило, происходит по результатам выполнения оценки производительности. Оценка проводится с помощью шаблона, содержащего список пунктов.

- Что вы сделали в этом году?
- Что в этом году вам хорошо удалось?
- Что можно было сделать лучше в этом году?
- Каковы ваши карьерные ожидания? Кем вы видите себя через пять лет?

Сначала работники проводят самооценку, и после ее результаты оценивают менеджеры. Затем менеджер и работник встречаются и обсуждают оценки. Обычно работникам нужно подписать документ об оценке для подтверждения ее получения.

При написании самооценки не стоит полагаться на свою память. Память несовершенна и сохранит, скорее всего, только самые яркие моменты. Записывайте, что делали на протяжении года, используя список выполненных задач, PPP или ежедневник — так вы сможете поддержать свою память. Посмотрите на выполненные задачи в системе отслеживания ошибок компании. Над какими эпиками или историями вы работали? Не забывайте о проектах, не связанных с кодом. Программы наставничества, ревью кода, проведение интервью, написание постов в блогах, проведение презентаций или ведение документации — вещи, которые не должны остаться незамеченными. Вспомните все, что происходило, для написания честной самооценки.

Прохождение оценки производительности иногда может вызывать стресс. Постарайтесь рассматривать данное мероприятие как возможность оценить то, чего вы достигли, и поговорить о том, что собираетесь делать дальше. Открыто признайте свои недостатки и упущенные возможности и разработайте план роста на следующий год. Не забудьте оставить обратную связь вашему менеджеру. Вас не должна шокировать обратная связь оценки производительности, однако если это случилось, то обратитесь к своему менеджеру и поговорите с ним о непонимании. Успешная оценка производительности должна указать вам на конкретные действия для достижения поставленных вами целей.

Вас могут попросить поучаствовать в оценке по *«Методу 360 градусов»*, когда сотрудники запрашивают обратную связь от всех своих коллег: от верхнего звена (менеджеров), нижнего (подчиненных) и среднего (коллег). Коллеги в отношении вас отвечают на такие вопросы, как: «Что я мог бы сделать лучше?», «Есть ли что-то, что люди боятся мне сказать?» или «Что у меня получается хорошо?». В использовании данной методики поощряются честность и открытость, с ее помощью сотрудники могут рассказать менеджерам о своих делах. Относитесь к оценке по «Методу 360 градусов» со всей серьезностью и старайтесь давать полезные отзывы.

Менеджеры должны давать обратную связь в течение всего года — во время встреч 1:1, отдельно после встреч или в чате. Если вы получаете недостаточно обратной связи, обсудите этот вопрос с менеджером на следующей встрече 1:1. Вы также можете обратиться к своим наставникам или старшим инженерам.

Концепция управления вверх

Как менеджеры «управляют вверх» своими руководителями, так и вы «управляете вверх» своим менеджером, помогая ему и следя за тем, чтобы он помогал вам. Через обратную связь и вы можете помочь менеджеру, и менеджер может помочь вам с достижением ваших целей. Не прекращайте стараться, если у вас что-то не получается. Неэффективное и неадекватное управление может навредить вашему карьерному росту.

Получайте обратную связь

Самооценка, а также оценки производительности и по «Методу 360 градусов» обеспечивают полноценную обратную связь, но проводятся достаточно редко, чтобы вы могли полагаться исключительно на них. Вам нужно регулярно получать обратную связь — так вы быстрее адаптируетесь. Менеджеры не всегда хотят проявлять инициативу и оставлять обратную связь, поэтому вам, возможно, придется просить их это сделать.

Для получения обратной связи воспользуйтесь встречами 1:1. Заранее вышлите заготовленные вопросы, чтобы менеджеру не пришлось импровизировать на ходу. Просите конкретных ответов. Вопрос: «Что я могу сделать, чтобы улучшить процесс тестирования?» — звучит лучше, чем: «Что я могу делать лучше?» При составлении вопросов не ограничивайтесь технической частью. Попросите включить в обратную связь информацию о коммуникации, карьерном росте, лидерстве, возможностях приобретения новых умений и т. д. Воспользуйтесь подсказками из раздела встреч 1:1, если вам требуется определить цели.

Не доверяйте обратной связи полностью. Ваш менеджер предоставит всего лишь одну точку зрения (хотя и важную). Постарайтесь

посмотреть на обратную связь от менеджера своими глазами. Спросите себя, что менеджер упустил при составлении обратной связи, насколько его отзывы совпадают с вашим ощущением, что знает ваш менеджер, чего не знаете вы, и т. д.

В ответ на обратную связь не забывайте также дать обратную связь, чтобы не сложилось впечатления, будто полученная вами обратная связь оказалась бесполезной. Например, скажите своему менеджеру: «Я присоединился к предложенной вами группе инженеров, и мне было очень интересно читать статьи, а также обсуждать их с другими инженерами! Большое спасибо за совет! Я узнал много нового». Положительный отзыв побудит менеджера давать больше обратной связи. Сообщите также, если обратная связь не привела к желаемому результату: «Я присоединился к рекомендованной вами группе инженеров, и, по правде говоря, мне это никак не помогло. Они обсуждают документы и проекты, никак не связанные с моей работой. Не предложите ли вы варианты других команд?»

Когда вы запрашиваете у кого-то отзыв, можете также оставить и свой отзыв. Вопрос, как что-то сделать, часто обнаруживает пробелы в процессах. Ответом на вопрос: «Каким образом я мог предотвратить инцидент на прошлой неделе?» — может быть: «Нам необходима новая тестовая среда». У вашего менеджера должна быть возможность самому прийти к выводу — просто запросите обратную связь, а не предлагайте готовое решение.

Давайте обратную связь

Хорошие менеджеры хотят получать обратную связь от всей команды. Менеджеры должны знать, как идут дела: в чем команда преуспевает, а где имеются проблемы. У каждого члена команды есть своя точка зрения. Обратная связь устраняет «слепые зоны».

Обратная связь может касаться чего угодно: команды, компании, поведения людей, проектов, кадровой политики. Говорите о проблемах, но не сосредоточивайтесь только на них. Положительная обратная связь тоже ценна: иногда менеджерам тяжело понять, какие из предложенных изменений были полезны, ведь в работе менеджеров нет модульных тестов.

При предоставлении обратной связи используйте *модель «Ситуация — Поведение — Влияние»* (SBI). Сначала опишите ситуацию, после чего опишите поведение в конкретной ситуации, которое вы считаете достойным похвалы или проблемным. Затем объясните воздействие, то есть в чем выражается эффект этого поведения и почему оно важно. Пример:

> **Ситуация.** *В январе я закончил изменение кода в новом сервисе разрешений и отдал его команде использования ПО для развертывания. Однако сервис разрешений до сих пор не развернут, а сейчас уже начало марта.*
>
> **Поведение.** *В течение пяти недель ожидаемая дата релиза на панели «Предстоящие релизы» менялась каждую неделю. Обновление базы данных тоже переносилось в течение нескольких недель.*
>
> **Влияние.** *Мы рискуем не уложиться в сроки, из-за чего задержится работа над рядом зависимых проектов. Возможно ли что-то сделать?*

Модель SBI исключает обратную связь, основанную на характере человека, а также на предположениях о его мотивах и намерениях. Данная модель фокусируется на фактах и взаимодействии сторон и помогает избегать конфликтных ситуаций при обсуждениях.

Обратите внимание — в рамках модели SBI вы не предлагаете готового решения. У вас могут быть предложения, но перед тем, как давать рекомендации, стоит узнать о проблеме чуть больше. В процессе работы вы можете, например, обнаружить, что пропустили важную деталь или что на самом деле проблема выглядит не так, как вам представлялось. Возможные решения вы можете обсудить в самом конце разговора, когда будете обладать полной информацией.

Давайте отзывы конфиденциально, в спокойном тоне и регулярно. Встречи 1:1 для этого подходят как нельзя лучше. Получение обратной связи может вызывать сильные эмоции, но не поддавайтесь им: сохраняйте разум ясным и поддерживайте конструктивный разговор. Предоставление отзыва в частном порядке позволяет руководителю вести с вами откровенный разговор и может уберечь обе стороны от чувства уязвимости. Частая обратная связь исключает неожиданности.

Не ждите, пока проблема усугубится и станет слишком поздно. Это не должно быть безнадежной ситуацией.

Модель SBI применяется и для положительной обратной связи.

> **Ситуация.** *На прошлой неделе нам нужно было написать проектный документ для предлагаемых изменений при регистрации, так что я воспользовался возможностью и применил созданный вами шаблон проектного документа.*
>
> **Поведение.** *Раздел, посвященный планам развертывания и коммуникации, помог нам понять, что мы забыли проинформировать команду службы поддержки пользователей о внесенных изменениях.*
>
> **Влияние.** *Как только мы обратились к членам этой команды, мы получили много полезной обратной связи! Написание документа далось легче и быстрее, так как нам не нужно было думать над его структурой. Спасибо за помощь!*

Независимо от того, обмениваетесь ли вы обратной связью с менеджером или коллегами, делаете это в письменной форме или устной, постарайтесь, чтобы обратная связь от вас была такой, какую вы хотели бы получить сами. Спросите себя: какую проблему вы пытаетесь решить? Каков желаемый результат? Что вы подразумеваете под успехом?

Обсуждайте ваши цели

Не думайте, что менеджер знает, чего вы ждете от своей карьеры. Вам необходимо четко формулировать свои цели, чтобы менеджер помог вам их достичь. Формальные обзоры и оценки — отличная возможность обсудить ваши цели.

Если пока у вас нет карьерных целей, это совершенно нормально. Расскажите менеджеру о том, что вам интересно, и он окажет вам помощь. При наличии интересов, выходящих за рамки вашей работы, тоже поделитесь этим с менеджером. Не ограничивайте себя разработкой ПО. Например, вас может заинтересовать продакт-менеджмент или

появится желание создать собственную компанию. Думайте масштабно и с прицелом на далекую перспективу. Вот пример для разговора с менеджером:

> *Можем ли мы сегодня поговорить о карьерном росте? По правде говоря, я не знаю, где я вижу себя через пять лет. Какие карьерные пути вы видите? Чем они отличаются? Мне нравится проект, над которым я сейчас работаю, но меня также интересует безопасность. Будет ли у меня возможность поработать над чем-то связанным с безопасностью?*

Если вы не знаете, что делать, сообщите менеджеру — так вы сможете понять, к какой цели стоит идти. Часть работы менеджера состоит в том, чтобы привести ваши интересы в соответствие с целями компании, и чем больше менеджер будет знать о ваших интересах, тем больше пользы для компании он сможет извлечь.

После обсуждения всех целей наберитесь терпения. У вас много возможностей, и, в конце концов, вы должны максимально ими воспользоваться. Поймите, что существуют различные формы возможностей: участие в новых проектах, решение новых задач, участие в программе наставничества, создание презентаций, написание постов в блогах, обучение или участие в партнерских командах. В определенном смысле все, что вы делаете, — это возможность для роста.

СОЗДАЙТЕ ГРУППУ ПОДДЕРЖКИ

Иногда может быть трудно дать обратную связь, справиться с определенными ситуациями или понять, что правильно, а что нет. Надежные коллеги не только внутри, но и за пределами компании помогут проверить правильность ваших действий. Включение в диалог членов производственных групп, не очень активно участвующих в такого рода обсуждениях, вдвойне необходимо для них самих. Ищите такие организации, как PyLadies, /dev/color и другие сообщества, где люди могут рассмотреть вашу ситуацию и поделиться собственным опытом и историями.

Примите меры, когда что-то идет не так

Каждые отношения между менеджером и сотрудником уникальны, поэтому сложно дать какой-то универсальный совет. Ситуация зависит от компании, команды, менеджера и работника. Единственное, что мы можем вам посоветовать, — это проявлять инициативу в случае, если вы понимаете, что дела идут не так, как вам бы хотелось.

В работе, как и в отношениях, бывают взлеты и падения. Кратковременные ухудшения не требуют радикальных действий, однако если вы постоянно чувствуете разочарование, стресс или иные негативные чувства, вы не должны об этом молчать.

Используйте модель SBI (см. раздел «Давайте обратную связь») для разговора с менеджером. Поговорите с отделом кадров, руководством вашего менеджера или другими наставниками, если вам не хочется разговаривать со своим менеджером. Путь, который вы изберете, зависит от ваших отношений с каждой из сторон. Если у вас нет того, к кому вы можете обратиться, обратитесь в отдел кадров.

Роль отдела кадров заключается в поддержании стабильности, в защите компании от проблем с законом, а также в соблюдении нормативных требований — а это не совсем то же самое, что и делать все правильно или честно. По крайней мере, разговор с отделом кадров гарантирует, что будет сделана запись о вашей проблеме. Как правило, компании реагируют на *тревожные* сообщения.

Если вам говорят, что скоро что-то изменится, дайте своему менеджеру немного времени — от трех до шести месяцев. Менеджеры должны подумать над обратной связью и внести изменения. Вполне вероятно, потребуется немного перестроить процессы и организационные структуры. Возможности продемонстрировать изменения могут быть нечастыми. Обратите внимание на направление изменений. Есть ли улучшения? Видите ли вы стремление к улучшению, которое выражается в конкретных действиях?

Если ничего не меняется, вам стоит задуматься о смене менеджера, команды или даже об увольнении. Перевод из одного отдела в другой внутри компании тоже возможен, если вам нравятся ваши коллеги или компания, в которой вы работаете. Выбор новой компании стоит

рассматривать, если проблемы в месте вашей нынешней работы носят системный характер и вы постоянно сталкиваетесь с такими проблемами, как плохой менеджмент, руководство или токсичная среда. При смене команды действуйте деликатно: сначала поговорите с менеджером, к которому хотите перейти, а затем со своим нынешним менеджером. Переход из одной команды в другую может занять некоторое время, так как вы должны передать свои дела, однако не позволяйте данному процессу затягиваться на срок больше трех месяцев.

ПРОГРАММИСТ В ЗЕРКАЛЕ

Первый опыт работы Дмитрия инженером-разработчиком чуть не заставил его вообще уйти из этой сферы. Технический руководитель Дмитрия был надежным и дружелюбным человеком и хорошим программистом, но ужасным руководителем. Он регулярно оставлял комментарии вроде: «Я думал, что Калифорнийский университет в Беркли (альма-матер Дмитрия) был хорошим заведением» или «Настоящий программист, напротив, будет...» Часто он шутил и об увольнении Дмитрия. Он даже установил на свой монитор зеркало от велосипеда, чтобы в нем отражался монитор Дмитрия. Дело вот в чем: у этого руководителя на самом деле была причина, чтобы использовать зеркало. Через несколько лет работы в данной сфере Дмитрий потерял всякую мотивацию, не верил в свои силы и отлынивал от работы. Он всерьез задумывался о том, чтобы уволиться и стать мануальным терапевтом.

Однако после переоценки ценностей он попытался работать в другой компании. Культура в компании, в которую он попал, была полной противоположностью тому, с чем он имел дело в прошлой компании. Его новые коллеги не выпячивали свое эго и обладали при этом немалыми навыками, а также были уверены в том, что при правильной и хорошей поддержке Дмитрий справится с любой задачей. Уверенность Дмитрия постепенно росла, у него появилась мотивация, он приобретал новые навыки. В результате его карьера восстановилась и все у Дмитрия начало складываться как следует. Однако всего это могло и не быть из-за одного человека, который был хорошим программистом, но плохим менеджером.

Плохое руководство вызывает разочарование, является причиной стресса и может стать причиной застоя вашего карьерного роста. Не каждый менеджер — прекрасный руководитель, и не каждый менеджер подойдет именно вам. Если вы отправили обратную связь и были довольно терпеливы, но ситуация никак не изменилась, то вам нужно двигаться дальше.

Что следует и чего не следует делать

Следует	Не следует
Ожидать, что менеджеры будут для вас доступны	Скрывать проблемы от своего менеджера
Говорить менеджеру о том, что вам нужно	Использовать встречи 1:1 для обновления статуса текущих задач
Определять повестку для встреч 1:1	Писать самооценку по памяти
Делать заметки во время встреч 1:1	Писать поверхностные отзывы
Писать такие отзывы, какие хотели бы получать сами	Ограничивать себя стратегией OKR
Отслеживать свои достижения, чтобы упростить составление самооценки	Воспринимать обратную связь как агрессию
Использовать модель SBI, чтобы обратная связь была менее личной	Терпеть плохое руководство
Думайте о долгосрочных карьерных целях	

Повышение уровня

Хороший способ понять менеджера — читать те книги, которые он сам читает. Книги по менеджменту помогут вам понять, почему менеджеры поступают тем или иным образом. Узнав о проблемах, с которыми сталкиваются менеджеры, вы разовьете свою эмпатию, получите полезные навыки и сможете давать менеджеру более качественную обратную связь.

Советуем начать с книги Камиля Фурнье (Camille Fournier) *The Manager's Path*[1] (O'Reilly Media, 2017). Она познакомит вас с этапами, которые

[1] *Фурнье К.* От разработчика до руководителя. Менеджмент для IT-специалистов.

проходит человек от должности разработчика до главного инженера, а затем и технического директора. В книге рассказывается о каждом уровне работы менеджера и различных процессах управления, например о встречах 1:1. Знакомство с этим изданием поможет вам составить примерный план вашей карьеры.

Из книги *An Elegant Puzzle* (Stripe Press, 2019) Уилла Ларсона (Will Larson) вы получите представление о проблемах, с которыми может столкнуться менеджер, а также о методах, которыми он пользуется для их решения.

Книга *Thanks for the Feedback* (Penguin Books, 2014) Дугласа Стоуна и Шейлы Хин (Douglas Stone, Sheila Heen) поможет вам обработать и принять обратную связь. Обратная связь — это эмоциональная тема. В книге предлагаются инструменты, позволяющие извлечь максимальную выгоду из обратной связи, даже если она «необъективна, несправедлива и вообще вы в плохом настроении». Многие советы из книги можно применить и в других ситуациях.

Мэри Аббеджей (Mary Abbajay) в своем труде *Managing Up* (Wiley, 2018) рассматривает концепцию управления вверх на совершенно ином уровне, нежели предложенный нами. В книге идет речь о разных типах менеджеров, а также о способах работы с ними. Кроме всего прочего, здесь вы найдете советы по работе с грубыми менеджерами и рекомендации, как понять и что делать, когда пришло время двигаться дальше.

Книга *High Output Management* (Vintage, 2015) Эндрю Гроува (Andy Grove) — классическое издание по управлению проектированием. Автор написал книгу в 1983 году и задокументировал в ней философию и методы, которые он создавал и применял в Intel. Данная философия стала основой современного управления проектированием. Прочитайте эту книгу не только из интереса к истории, но в первую очередь потому, что все описанное в книге остается актуальным до сих пор. Скорее всего, ваш менеджер уже хорошо знаком с трудом Энди Гроува, так что у вас будет общий информационный ресурс.

14 Навигация по карьерной лестнице

Карьерная линия инженера-программиста может быть очень длинной. Наша книга «README. Суровые реалии разработчиков» приводит вас к завершению начального этапа вашего путешествия. Вам предстоит всю жизнь учиться чему-то новому, даже став техническим руководителем или менеджером высокого ранга. Какой бы путь вы ни избрали, вы должны постоянно расти.

Все предыдущие главы посвящались конкретной инженерной деятельности, однако в последней главе мы заглянем в будущее, дадим несколько советов относительно карьеры и поделимся некоторыми важными мыслями.

Путь к сеньору и дальше

Карьерная лестница устанавливает иерархию должностей и описывает ожидания от сотрудника на каждом уровне. С прохождением должностей в карьерной лестнице осуществляется карьерный рост в компании. Количество уровней меняется от компании к компании, однако есть два этапа, которые означают значительные изменения в работе: от джуниор-разработчика или инженера ПО к сеньору и далее

к техническому руководителю (staff engineer) или ведущему инженеру (principal engineer).

В главе 1 мы перечислили технические, коммуникативные, лидерские и исполнительские навыки, которыми должен обладать сеньор-разработчик. Важно заметить, что сфера деятельности и фокус сеньора также меняются. Джуниоры внедряют функции и выполняют различные конкретные задания. У сеньоров более неопределенные и сложные для понимания задачи: они помогают команде установить, над чем следует работать, принимают решения на уровне крупных и сложных проектов и не нуждаются в жестком руководстве.

У технических руководителей широкий диапазон обязанностей, выходящих за рамки команды. Они участвуют в разработке инженерных стратегий и архитектуры системы, принимают решения в квартальном планировании, а также управляют процессом инженерной разработки. Технические руководители продолжают заниматься написанием кода, однако, чтобы стать специалистом такого уровня, недостаточно просто быть хорошим программистом: необходимо понимать общую картину и принимать решения, которые будут иметь последствия в будущем.

На уровне персонала карьерная лестница делится на два типа: менеджер и эксперт в своей области. Для продвижения по второй карьерной лестнице *не требуется* управлять другими людьми, так как управление — это иной набор навыков. Если вы задумываетесь о менеджменте, советуем прочитать книгу Камиля Фурнье «От разработчика до руководителя. Менеджмент для IT-специалистов», из которой вы узнаете, что вас ожидает в этой сфере деятельности.

Советы по карьере

Чтобы стать ведущим инженером или техническим руководителем, нужно много времени, настойчивости и терпения. Однако вы можете себе помочь, взяв ответственность за свой карьерный рост на себя.

Развивайте T-shaped-навыки, принимайте участие в инженерных программах и в процессе продвижения по службе, не меняйте места работы слишком часто и следите за собой.

Становитесь T-shaped-специалистами

В разработке ПО имеется множество специализаций, таких как фронтенд, бэкенд, операции, хранилище данных, машинное обучение и т. д. T-shaped-специалисты работают во многих областях и являются экспертами по крайней мере в одной области.

Впервые мы столкнулись с этой концепцией в *Handbook for New Employees* на Valve (https://www.valvesoftware.com/en/publications/). T-shaped-специалисты там описываются так:

> *...люди, которые являются как специалистами широкого профиля (высококвалифицированными во множестве областей, что символизирует верхняя часть буквы T), так и экспертами (одними из лучших в своей области в рамках узкой дисциплины — это вертикальная часть T).*

Как T-shaped-специалист, вы будете принимать решения со знанием дела, сможете вносить изменения, затрагивающие несколько кодовых баз, упростите устранение неполадок. Вы сможете решать сложные проблемы, которые ставят в тупик других, объединив свой опыт со способностью разбираться в разрозненных системах.

Начните с создания собственной базы знаний. В процессе вы познакомитесь с разными подобластями и найдете то, что вам нравится. Ищите проекты, в которых участвуют другие команды, занимающиеся, например, наукой о данных, операциями, фронтендом и т. д. Работайте с кодом других команд и обязательно спрашивайте, можно ли вносить патчи или участвовать в разработке по методу парного программирования в процессе внесения изменений. Если вы сталкиваетесь с проблемами или информацией, которая вызывает у вас интерес, подробно их изучите.

Не забывайте о парадигме широты/глубины. Каждый член команды обладает разным опытом. Если ваш коллега не знает, что такое монада, это не значит, что он не силен в чем-то другом. Не будьте слишком строги к себе, поняв, что вы чего-то не знаете: наверняка вы хорошо разбираетесь в какой-нибудь другой области.

В хорошей команде будет множество T-shaped-специалистов. В команде по разработке продукта разработчики, скорее всего, будут иметь глубокие познания в разных областях, в то время как в команде по обслуживанию инфраструктуры все разработчики будут сильны в одной области.

По мере развития и роста компании руководители начнут набирать специалистов для каждой отдельной области, поэтому всем, кто уже работает в компании, тоже придется выбрать специализацию (так как у универсалов область работы будет становиться все меньше и меньше). Обладая T-shaped-навыками, вы отлично приспособитесь к данной ситуации. Вы столкнетесь с тем, что некоторые специалисты, только приходящие на работу в компанию, не являются T-shaped-специалистами, так что вы сможете помочь им адаптироваться и быть полезными и эффективными.

Участвуйте в программах для инженеров

Во многих компаниях есть программы для инженеров, направленные на обучение и развитие. Прием на работу, собеседование, технические обсуждения, конференции, встречи, группы чтения кода, стажировки и наставнические программы — все это возможности для вас.

Ищите и присоединяйтесь к тем программам, которые интересуют вас больше всего. Вы можете участвовать в них двумя способами: либо как слушатель, либо как ментор. Если вы считаете, что в вашей компании нет хороших программ для инженеров, организуйте их сами! Обсудите свои идеи с руководством, а также найдите в своей команде таких же увлеченных людей, которые захотят вам помочь.

Участие в программах для инженеров поможет вам и вашей карьере. Вы сможете познакомиться со многими людьми, заработаете авторитет в компании, овладеете новыми навыками и сможете влиять на корпоративную культуру.

Направляйте свое продвижение

В идеале именно ваш менеджер должен продвигать вашу кандидатуру в правильное и нужное время. Но мир редко бывает идеальным, поэтому, скорее всего, вам самому придется этим заниматься. Изучите процесс повышения в должности, убедитесь в том, что работа, которую вы делаете, имеет ценность и что ее замечают. Не бойтесь говорить, что вас следует повысить, если вам так кажется

Чтобы получить повышение, вы должны знать то, как к вам относятся, а также как выглядит сам процесс повышения. Узнайте о структуре карьерной лестницы в вашей компании и о том, какими навыками вы должны обладать для перехода на более высокую ступень. Поговорите о повышении со своим менеджером. Повышения происходят ежегодно? Кто выбирает кандидатов на повышение? Вам нужен ментор, поручитель или набор документов подготовки к повышению?

Как только вы поймете критерии и сам процесс продвижения по службе, проведите самооценку и получите отзывы о себе от других. Создайте документ, в котором будут кратко перечислены ваши достижения на каждой из ступенек карьерной лестницы. Ищите области, в которых вам нужно развиваться. Получите обратную связь от менеджера, коллег и менторов, рассказав им, зачем вам это нужно, чтобы они не думали, будто вы просто пытаетесь повысить самооценку за их счет. Дайте больше деталей.

- Если сравнить мою проектную документацию с документацией инженеров третьего уровня, много ли будет различий?
- Вы говорили, что у меня получается писать хорошие тесты. Какие именно из написанных мною тестов кажутся вам хорошими?

> Вы можете привести пример менее удачных тестов? У кого в нашей компании получается писать отличные тесты? Чем мои тесты отличаются от их?

Если вы получаете обратную связь, с которой не согласны, пытайтесь понять, в чем дело. Другие люди могут иметь неполное представление о вашей работе или не воспринимать ее как нечто ценное. Может, из-за изменений в команде вы остались без завершенного проекта, а именно его можно было бы показать в качестве выполненной работы? Или, быть может, вы неверно оцениваете собственную работу? Откровенный разговор с менеджером поможет устранить сомнения в обратной связи.

После того как вы начнете правильно оценивать собственную работу и получите обратную связь, обсудите все с вашим менеджером и разработайте план по восполнению пробелов в знаниях. Ожидайте предложений от программ для инженеров или ждите проектов и стажировок, на которых сможете развить определенные навыки.

Иногда люди ожидают повышения совершенно необоснованно, после чего разочаровываются. Многообещающий, но незаконченный проект не является основанием для повышения: менеджерам нужны результаты. Обладание техническими навыками необходимо, но они тоже не являются причиной для повышения: вы должны иметь множество навыков, чтобы команда достигала целей, а компания развивалась. Повышение не зависит от времени — вы можете работать на одной должности год или пять лет, но все будет зависеть от вашего влияния и вклада в развитие компании. Когда вас повысят, имейте в виду, что до следующего повышения вам нужно будет ждать как минимум от трех до шести месяцев. В течение этого периода вам придется доказать, что вы соответствуете требованиям и справляетесь с работой.

Когда доходит до разговоров о повышении, время начинает играть здесь большую роль. Начните обсуждать свое повышение *задолго до того*, как вам покажется, что вы готовы к повышению, — заводите разговор, находясь еще на половине пути. В таком случае и у вас, и у вашего менеджера будет время на согласование и заполнение пробелов. Если вы ждете момента, когда точно будете считать себя готовыми к повышению,

а ваш менеджер с этим не согласится, то весь разговор сведется к разрешению конфликта, а не к разработке плана.

Наконец, помните, что карьерные лестницы отражают общие шаблоны, которым не все могут соответствовать. Работа технического руководителя требует широкого влияния и выполнения задач разряда glue work (координации, улучшения процессов, создания документации, коммуникации и т. д.). Для сеньора-разработчика или ниже требования чаще всего описываются в терминах исключительно способности программировать. Это является проблемой для джуниоров, берущихся за выполнение важной работы, которая не связана с программированием и к которой не прикреплен коммит в Git. Такие инженеры уделяют программированию очень мало времени, поэтому их редко продвигают по службе или переводят на другую должность, к примеру продакт-менеджера. В выступлении и докладе Тани Рейли (Tanya Reilly) (https://noidea.dog/glue/) говорится о том, что вам стоит прекратить заниматься работой, относящейся к разряду glue work, даже во вред команде в краткосрочной перспективе, если менеджер не рассматривает вашу работу как шанс получить повышение. Это горькая пилюля, и такое отношение может показаться вам несправедливым, однако все решения принимает менеджер, а не вы.

Меняйте место работы обдуманно

Смена работы расширит набор ваших навыков и сеть ваших контактов, однако частая смена работы может затормозить ваш карьерный рост и выглядеть плохо в глазах специалистов по найму. Не меняйте работу без уважительной причины.

Сеньоры-разработчики используют прошлый опыт для принятия подобных решений. Если вы постоянно меняете место работы, то никогда не узнаете, как ваши решения отразятся на будущем команды и компании. В таком случае у вас не будет развита интуиция, которая необходима на должности сеньора. Для будущего работодателя частая смена работы станет знаком того, что вас не стоит нанимать на работу, ведь вы можете уйти сразу после того, как появятся первые проблемы или закончится период испытательного срока.

FOMO, или синдром упущенной выгоды, не является причиной для увольнения. Вам может казаться, что в других компаниях используются передовые технологии, что там нет проблем, имеющихся в вашей компании. Как говорится, хорошо там, где нас нет, но проблемы бывают во всех компаниях. Если вы боитесь что-то упустить, то поделитесь своими идеями на встречах или конференциях. Очень часто работодатели оплачивают карьерный рост или даже обучение. Работа над проектами с открытым исходным кодом или сторонними проектами является хорошим способом оставаться востребованным: конечно, при наличии у вас свободного времени. Вы также можете остаться в своей компании, но начать работать с другой командой.

Разумеется, иногда веские причины для смены работы имеются даже после непродолжительной работы на новом месте. Некоторые компании или команды не подходят друг другу, поэтому лучше сразу разобраться с проблемой. Исключительные возможности не приходят по расписанию, поэтому, когда они появятся, вы должны сразу же ими воспользоваться. Будет полезно ознакомиться с различными комплексами технологий, коллегами и инжиниринговыми организациями. Зарплаты инженеров быстро выросли — однако в вашей компании могут не поднимать зарплату так же быстро, как растет рынок. К сожалению, иногда намного легче догнать рынок путем смены работы. Но если у вас хорошая зарплата на прежнем месте работы, не стоит ее менять — развивайтесь и узнавайте что-то новое. Очень важно видеть то, как компании, команды и ПО со временем развиваются.

И наоборот, не задерживайтесь на одной работе слишком долго. Желание избежать стагнации — законная причина для перемен. Опытные разработчики в компании обучают новых разработчиков тому, как все работает, почему были приняты те или иные решения. Такие знания представляют большую ценность и являются частью работы технического руководителя. Но карьерный рост замедлится, если ваши заслуги будут ограничиваться только прошлыми работами, а на данный момент ваш вклад в текущие проекты минимален. Смена работы и компании, поиск себя в чем-то новом могут послужить вашему карьерному росту.

Правильно распределяйте силы

Работа в области ПО не лишена стрессов. Жесткая конкуренция, быстрое развитие технологий, постоянное изучение новой информации... Вам может показаться, что очень многое происходит слишком быстро. Очень часто новые инженеры много и усердно работают, но это прямой путь к выгоранию. Не переутомляйтесь и делайте перерывы в работе.

Инженеры иногда работают по 14 часов в день, но помните, что марафонские сеансы программирования и недостаток сна скорее навредят вашему коду, здоровью и личной жизни. Исследователи, изучающие влияние сна на производительность разработчиков, обнаружили, что «одна ночь недосыпа приводит к снижению качества реализации на 50 %» (Need for Sleep: The Impact of a Night of Sleep Deprivation on Novice Developers' Performance, IEEE Transactions on Software Engineering, 2020). Да, иногда вы будете перерабатывать, но не делайте так, чтобы это вошло в привычку или, что хуже, стало частью вас и вашего образа жизни. Не думайте, что переработка и плохой сон не отразятся на вас в будущем — когда-нибудь придется платить по счетам.

Даже при удобном графике работа может вас вымотать. Возьмите отпуск, чтобы отдохнуть. Некоторые программисты предпочитают в конце года брать большой отпуск, в то время как другие уходят в отпуск каждый квартал. Найдите тот вариант, который устраивает вас, однако не позволяйте времени в отпуске пройти просто так. В большинстве компаний количество максимальных дней отпуска ограниченно. В некоторых компаниях можно уйти в творческий отпуск, обычно продолжительностью от одного до трех месяцев, во время которого вы отдыхаете и изучаете что-то новое.

Помните, что ваша работа — это марафон, а не спринт: впереди у вас еще достаточно времени. Не спешите и наслаждайтесь этим путешествием!

В заключение

Разработка программного обеспечения — это область для отличной карьеры, наполненной увлекательными задачами. Вы сможете внести свой вклад в любое дело — от науки до сельского хозяйства, здравоохранения, сферы развлечений или даже космонавтики. Ваша работа может изменить миллиарды жизней. Работайте с людьми, которые вам нравятся, решайте проблемы, которыми вы увлечены, — и тогда достигнете больших высот. Мы верим в вас. Удачи!

Крис Риккомини, Дмитрий Рябой

README. Суровые реалии разработчиков

Перевел с английского С.Черников

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Кудрейко*
Художественный редактор	*В. Мостипан*
Корректоры	*Т. Никифорова, Н. Терех*
Верстка	*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: 06.2023. Наименование: книжная продукция.
Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции
ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 18.04.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 0000.

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР»

предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург
м. «Выборгская», Б. Сампсониевский пр., д. 29а;
тел. (812) 703-73-73, доб. 6282; e-mail: dudina@piter.com

Москва
м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж;
тел./факс (495) 234-38-15; e-mail: reception@piter.com

БЕЛАРУСЬ

Минск
ул. Харьковская, д. 90, пом. 18
тел./факс: +37 (517)348-60-01, 374-43-25, 272-76-56
e-mail: dudik@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс (812) 703-73-73, доб. 6282; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг в интернет-магазине: на сайте www.piter.com;
тел. (812) 703-73-74, доб. 6216; e-mail: books@piter.com

Вопросы по продаже электронных книг: тел. (812) 703-73-74, доб. 6217;
e-mail: kuznetsov@piter.com

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ЗАКАЗ ДОСТАВКА КНИГ

ЗАКАЗ И ДОСТАВКА КНИГ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ

- **на нашем сайте:** www.piter.com
- **по электронной почте:** books@piter.com
- **по телефону:** (812) 703-73-74 или 8(800) 500-42-17

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ

Наложенным платежом с оплатой при получении в ближайшем почтовом отделении, пункте выдачи заказов (ПВЗ) или курьеру.

С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.

Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, WebMoney и Qiwi-кошелек.

В любом банке, распечатав квитанцию, которая формируется автоматически после оформления вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ

- курьерская доставка до дома или офиса
- на пункт выдачи заказов выбранной вами транспортной компании
- в отделение «Почты России»

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ

- фамилию, имя, отчество, телефон, e-mail
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру
- название книги, автора, количество заказываемых экземпляров