ИДЕАЛЬНАЯ РАБОТА

ПРОГРАММИРОВАНИЕ БЕЗ ПРИКРАС

РОБЕРТ МАРТИН

Clean Craftsmanship

Disciplines, Standards, and Ethics

Robert C. Martin

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

ИДЕАЛЬНАЯ РАБОТА

ПРОГРАММИРОВАНИЕ БЕЗ ПРИКРАС

Роберт Мартин

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018
УДК 004.3
М29

Мартин Роберт

М29 Идеальная работа. Программирование без прикрас. — СПб.: Питер, 2022. — 384 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1910-3

В книге «Идеальная работа. Программирование без прикрас» легендарный Роберт Мартин (Дядюшка Боб) создал исчерпывающее руководство по хорошей работе для каждого программиста. Роберт Мартин объединяет дисциплины, стандарты и вопросы этики, необходимые для быстрой и продуктивной разработки надежного, эффективного кода, позволяющего испытывать гордость за программное обеспечение, которое вы создаете каждый день.

Роберт Мартин, автор бестселлера «Чистый код», начинает с прагматического руководства по пяти основополагающим дисциплинам создания программного обеспечения: разработка через тестирование, рефакторинг, простой дизайн, совместное программирование и тесты. Затем он переходит к стандартам — обрисовывая ожидания «мира» от разработчиков программного обеспечения, рассказывая, как часто различаются эти подходы, и помогает вам устранить несоответствия. Наконец он обращается к этике программиста, давая десять фундаментальных постулатов, которым должны следовать все разработчики программного обеспечения.

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

ББК 32.973.2-018
УДК 004.3

ISBN 978-0136915713 англ.
ISBN 978-5-4461-1910-3

КРАТКОЕ СОДЕРЖАНИЕ

ЧАСТЬ II
СТАНДАРТЫ

ЧАСТЬ III
ЭТИКА

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Я помню, как познакомилась с Дядей Бобом весной 2003 года, вскоре после того, как нашей команде специалистов по информационным технологиям рассказали о методологии Scrum. Как скептически настроенный Scrum-мастер, я слушала рассказы Боба о TDD и инструменте FitNesse и думала: «Зачем *вообще* писать изначально провальные тесты? Разве тестирование не должно идти *за* написанием кода?» Я часто уходила в недоумении, как и многие члены моей команды. Но стремление Боба к профессионализму в написании кода не могло не поражать. Я помню, как однажды, просматривая наш журнал ошибок, он в лоб спросил, с какой стати мы принимаем настолько неверные решения в отношении программных систем, не являющихся нашей собственностью: «Это *активы компании*, а не ваши *личные активы*». Его энтузиазм вызывал любопытство, и через полтора года мы провели рефакторинг, обеспечив 80-процентное автоматизированное покрытие тестами и чистую кодовую базу, что значительно упростило процедуру изменения бизнес-модели и целей компании, сделав намного счастливее как клиентов, так и сотрудников. После этого, вооружившись определением «сделано надежно, как броня», мы молниеносно выстроили защиту от непрерывно охотящихся на уязвимости черных хакеров; в сущности, мы научились защищаться от самих себя. Со временем мы с теплотой начали относиться к Дяде Бобу, который стал для нас настоящим дядей — добросердечным, решительным и смелым человеком, учившим нас стоять за себя и поступать правильно. В то время как другие дяди учили своих племянников кататься на велосипеде или ловить рыбу, наш Дядя Боб учил нас хранить верность своим принципам. Умение и желание в любой ситуации проявлять смелость и любопытство

были лучшим, чему я научилась за все время моей профессиональной деятельности.

Я следовала советам Боба, когда начала работать Agile-коучем, и быстро заметила, что лучшие команды разработчиков владели умением подбирать под конкретный контекст и конкретных клиентов лучшие из практик, принятых в отрасли. Я вспоминала уроки Боба, обнаружив, что лучшие в мире инструменты разработки хороши ровно настолько, насколько успешно люди умеют найти им оптимальное *применение*. Я поняла, что иногда высокий процент покрытия модульными тестами достигается исключительно для того, чтобы поставить галочку и соответствовать метрике, а на деле большинство этих тестов ненадежно. Достижение заданных метриками значений не приносило никакой пользы. При этом лучшим командам не приходилось заботиться о метриках; у них была цель, они были дисциплинированными, гордыми, ответственными — и показатели в каждом случае говорили сами за себя. Книга «Идеальная работа» сплетает все эти уроки и принципы в практические примеры кода и опыт, иллюстрирующие разницу между работой, преследующей единственную цель — уложиться в срок, и реальным созданием систем, стабильных в долгосрочной перспективе.

Книга напоминает, что никогда не нужно соглашаться на меньшее, что нужно делать свое дело с *бесстрашной компетентностью*. Она, как старый друг, напомнит о том, что имеет значение, что работает, а что нет, что увеличивает риск, а что снижает его. Это уроки на все времена. Возможно, в процессе чтения вы обнаружите, что уже практикуете некоторые из описанных в книге техник. Но готова держать пари: вы найдете и новое или, по крайней мере, то, от чего когда-то отказались, например, поскольку не успевали завершить проект в запланированные сроки или по какой-то другой причине. Для новичков в мире разработки, специализирующихся как в бизнесе, так и в технологиях, эта книга — шанс поучиться у одного из лучших специалистов. И даже самые опытные практики найдут здесь способы что-то для себя улучшить. Возможно, эта книга поможет вам снова обрести энтузиазм, возродить желание совершенствоваться в своей профессии, сколько бы препятствий ни стояло на вашем пути.

Разработчики программного обеспечения правят миром, и Дядя Боб в очередной раз хотел бы напомнить людям, обладающим такой вла-

стью, о профессиональной дисциплине. Он продолжает повествование с того места, на котором закончил книгу «Чистый код». Поскольку разработчики в буквальном смысле слова пишут правила для всех людей, Дядя Боб напоминает о необходимости соблюдать строгий этический кодекс, напоминает, что именно они отвечают за то, что делает написанный ими код, за то, как люди его используют, и за то, где он выходит из строя. Ошибки в программном обеспечении могут лишить как средств к существованию, так и жизни. ПО влияет на наш образ мыслей, на принимаемые нами решения, а благодаря искусственному интеллекту и предсказательной аналитике — еще и на социальное и стадное поведение. Поэтому разработчики должны чувствовать свою ответственность и действовать с большой осторожностью и эмпатией, ведь от их действий зависит здоровье и благополучие людей. Дядя Боб помогает нам выстоять перед лицом этой ответственности и *стать профессионалами, которые нужны обществу*.

Поскольку на момент написания этого предисловия приближается двадцатая годовщина с момента создания Манифеста гибкой разработки программного обеспечения, данную книгу можно считать прекрасной возможностью вернуться к основам: своевременным и скромным напоминанием о постоянно растущей сложности мира ПО, а также о том, что перед человечеством и перед собой мы обязаны практиковать этичную разработку. Не торопитесь быстрее прочитать «Идеальную работу». Позвольте принципам укорениться внутри вас. Практикуйте их. Улучшайте их. Учите им других. Держите эту книгу на своей книжной полке. Пусть она станет вашим верным другом — вашим Дядей Бобом, вашим проводником, — пока вы с любопытством и отвагой прокладываете себе путь в этом мире.

Стася Хаймгартнер Вискарди (Stacia Heimgartner Viscardi),
наставник по CST и Agile

ВСТУПЛЕНИЕ

Прежде чем вы приступите к чтению, нужно обсудить пару моментов, чтобы убедиться, что вы, мой любезный читатель, понимаете, в какой системе отсчета существует эта книга.

О ТЕРМИНЕ «МАСТЕРСТВО»

Начало XXI века отмечено терминологическими спорами. Индустрия программного обеспечения внесла в эти дискуссии свою лепту. Термин, который часто считают недостаточно полно описывающим суть, — *мастер своего дела* (craftsman).

Я довольно много думал над этим вопросом, разговаривал с людьми, придерживающимися различных мнений, и пришел к выводу, что лучшего термина для использования в контексте этой книги нет.

Я рассматривал и такие альтернативы, как «специалист», «умелец», «ремесленник», но ни одна из них не имела нужной исторической весомости. А мне было очень важно подчеркнуть ее.

Словосочетание «мастер своего дела» вызывает в памяти человека, обладающего глубокими знаниями и опытом в профессиональной деятельности. Человека, который свободно оперирует своими инструментами и применяет свои профессиональные навыки. Который гордится результатами своего труда, и поэтому можно быть уверенным в том, что он будет вести себя с достоинством и профессионализмом в соответствии со своим призванием.

Возможно, кто-то из вас не согласится с моим выбором. Я понимаю почему. Но надеюсь, что вы не истолкуете выбранный мной термин как попытку подчеркнуть исключительность в каком-либо смысле. Подобное ни в коем случае не входит в мои намерения.

ЕДИНСТВЕННЫЙ ПРАВИЛЬНЫЙ ПУТЬ

В процессе чтения этой книги может возникнуть ощущение, что здесь описан *единственный возможный путь к мастерству*. Но я всего лишь описал собственный путь, а для вас он может оказаться совсем другим. Выбор только за вами.

Нужен ли вообще *один правильный путь*? Я не знаю. Возможно. Потребность в строгом определении профессии программиста растет. К цели можно идти разными путями, в зависимости от важности создаваемого программного обеспечения. Но как вы скоро убедитесь, отделить критически важное ПО от неважного не так-то просто.

В одном я уверен. Времена «судей»[1] прошли. Сейчас уже недостаточно того, что каждый программист поступает так, как считает правильным. Появляются определенные практики, стандарты и этика. Предстоит решить, будем ли мы, программисты, определять их для себя сами или они будут навязаны нам теми, кто нас не знает.

ВВЕДЕНИЕ В КНИГУ

Эта книга написана для программистов и для их руководителей. И одновременно для всего нашего общества. Ведь именно мы, программисты, невольно оказались в самом его центре.

Для себя

Программистам с многолетним стажем, вероятно, знакомо чувство удовлетворения, наступающее после развертывания системы.

[1] Отсылка к Книге Судей Израилевых.

Вы испытываете определенную гордость за то, что приложили к этому руку. Вы счастливы от осознания успешно завершенной работы.

Но гордитесь ли вы тем, *как* вы достигли такого результата? Чем вызвана ваша гордость? Самим фактом завершения работы или это гордость за ваше мастерство? Вы гордитесь тем, что система была развернута? Или тем, как вы ее построили?

Придя домой после тяжелого рабочего дня, вы смотрите в зеркало и говорите: «Сегодня я отлично поработал»? Или чувствуете желание принять душ?

Слишком многие из нас в конце дня ощущают себя грязными. Слишком многие считают, что им приходится выполнять некачественную работу. Слишком многим кажется, что низкое качество — закономерный результат гонки за высокой скоростью. Слишком многие думают, что производительность обратно пропорциональна качеству.

В данной книге я пытаюсь сломать такие ментальные установки. Это книга о том, как *работать качественно*. Она описывает дисциплины и практики, которые должен знать каждый программист, чтобы работать быстро, продуктивно и каждый день гордиться результатами своего труда.

Для общества

В XXI веке впервые в истории человечества выживание общества стало зависеть от технологии, практически лишенной какого-либо подобия дисциплины или контроля. Программное обеспечение вторглось во все аспекты современной жизни, от заваривания утреннего кофе до вечерних развлечений, от стирки одежды до вождения автомобиля, от соединения людей во Всемирную сеть до социального и политического разделения. В современном мире мало жизненных аспектов, на которые бы не оказывало влияние ПО. При этом те, кто его создает, представляют собой всего лишь сборище разношерстных ремесленников, едва имеющих представление о том, что они делают.

Если бы мы, программисты, лучше понимали, что делаем, может быть, не было бы сбоя голосования в Айове в 2020-м? Не погибли бы 346 че-

ловек в двух авариях 737 Max? Не потеряла бы финансовая фирма Knight Capital Group 460 миллионов долларов за 45 минут? Не погибли бы 89 человек из-за внезапного ускорения автомобилей Toyota?

Каждые пять лет число программистов в мире удваивается. При этом их практически не обучают. Им показывают инструменты, дают разработать несколько несложных проектов, а затем бросают в работу, чтобы удовлетворить экспоненциально растущий спрос на новое программное обеспечение. С каждым днем шаткая конструкция, которую мы называем ПО, все глубже проникает в нашу инфраструктуру, наши институты, наши правительства и нашу жизнь. И с каждым днем растет риск катастрофы.

Какую катастрофу я имею в виду? Это не крах нашей цивилизации и не внезапное исчезновение всех программных систем одновременно. Готовый обрушиться карточный домик состоит не из самих программных систем. Скорее под угрозой находится хрупкая основа общественного доверия.

Слишком много происшествий с самолетами 737 Max, с самопроизвольным ускорением автомобилей Toyota, с претензиями к автомобилям Volkswagen со стороны California EPA или со сбоем голосования, как это получилось в Айове. Еще немного громких случаев сбоев программного обеспечения или злоупотреблений — и отсутствие у разработчиков дисциплины и этики, а также нехватка стандартов прикуют к себе внимание недоверчивой и разгневанной общественности. И тогда начнется регулирование, нежелательное для любого из нас. Регулирование, которое лишит нас возможности свободно исследовать и расширять мастерство разработки ПО; которое наложит серьезные ограничения на рост технологий и экономики.

Эта книга написана не для того, чтобы остановить безудержное стремление ко все большему внедрению программного обеспечения. Не ставлю я целью и снижение темпов его создания. Тем более что это была бы пустая трата сил. Обществу нужно ПО, и оно в любом случае его получит. Попытка подавить эту потребность не остановит надвигающуюся катастрофу общественного доверия.

Скорее своей книгой я пытаюсь убедить разработчиков программного обеспечения и их руководителей в необходимости дисциплины, а так-

же научить практикам, стандартам и этике, эффективно повышающим умение создавать надежные, отказоустойчивые продукты. Только изменив способ работы программистов, повысив их дисциплину, научив их правильным практикам, этике и стандартам, можно укрепить карточный домик и предотвратить его обрушение.

Структура книги

Книга состоит из трех частей, описывающих три уровня: практики, стандарты и этику.

Первая часть, посвященная различным *практикам*, описывает самый низкий уровень и носит прагматичный, технический и предписывающий характер. Ее будет полезно прочитать и понять программистам всех мастей. Я дал несколько ссылок на видеоролики, в реальном времени демонстрирующие ритм разработки через тестирование и рефакторинг. В тексте я тоже попытался дать представление об этом ритме, но ничто не способно сделать это так же хорошо, как видео.

Вторая часть посвящена *стандартам*. Это средний уровень. Здесь я знакомлю вас с ожиданиями, которые окружающий мир возлагает на нашу профессию. Это хороший материал для руководителей, позволяющий им понять, чего ожидать от профессиональных программистов.

Информация высшего уровня — это часть, посвященная *этике*. Здесь в форме клятвы или набора обещаний я описываю этический контекст профессии программиста. В этой части вы встретите множество исторических и философских дискуссий. Ее имеет смысл читать как программистам, так и их руководителям.

Примечание для руководителей

Страницы этой книги содержат много полезной информации. Но они наполнены и множеством технических деталей, которые вам, скорее всего, не нужны. Поэтому я советую читать начало каждой главы и прекращать чтение, когда начинается описание ненужных вам технических подробностей.

Обязательно изучите часть II «Стандарты» и часть III «Этика». И обязательно прочтите введение в каждую из пяти практик.

БЛАГОДАРНОСТИ

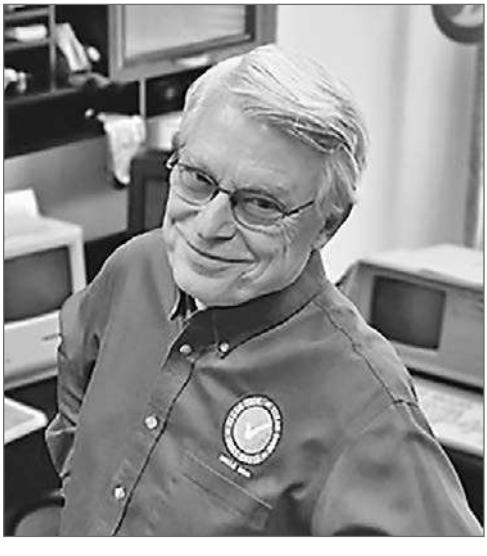
Спасибо моим мужественным рецензентам: Деймону Пулу (Damon Poole), Эрику Кричлоу (Eric Crichlow), Хизер Кансер (Heather Kanser), Тиму Оттингеру (Tim Ottinger), Джеффу Лангру (Jeff Langr) и Стасе Вискарди (Stacia Viscardi). Они спасли меня от множества неверных шагов.

Кроме того, я очень благодарен Джули Файфер (Julie Phifer), Крису Зану (Chris Zahn), Менке Мехте (Menka Mehta), Кэрол Лалльe (Carol Lallier) и всем сотрудникам издательства Pearson, которые неустанно совершенствуют выпускаемые книги.

Как всегда, огромное спасибо моему творчески одаренному и талантливому иллюстратору Дженнифер Конке (Jennifer Kohnke). Ее картинки всегда вызывают у меня улыбку.

И конечно же, спасибо моей прекрасной жене и замечательной семье.

ОБ АВТОРЕ



Роберт С. Мартин, также известный как **Дядя Боб** (Uncle Bob), написал первую строку кода в возрасте 12 лет в 1964 году. Работает программистом с 1970 года. Соосновательь компании cleancoders.com, предлагающей видеоуроки для разработчиков программного обеспечения, и основатель компании Uncle Bob Consulting LLC, оказывающей консультационные услуги и услуги по обучению персонала крупным корпорациям. Был ведущим специалистом в консалтинговой фирме 8th Light, Inc. в городе Чикаго.

Опубликовал десятки статей в специализированных журналах и регулярно выступает на международных конференциях и выставках. Создатель популярной серии обучающих видео на сайте cleancoders.com.

Мартин написал несколько книг, еще для некоторых он выступил редактором:

- *Designing Object-Oriented C++ Applications Using the Booch Method*;
- *Patterns Languages of Program Design 3*;
- *More C++ Gems*;
- *Extreme Programming in Practice*;
- *Agile Software Development: Principles, Patterns, and Practices*[1];
- *UML for Java Programmers*;
- *Clean Code*[2];
- *The Clean Coder*[3];
- *Clean Architecture: A Craftsman's Guide to Software Structure and Design*[4];
- *Clean Agile: Back to Basics*[5].

Как лидер в сфере разработки программного обеспечения, Мартин три года был главным редактором журнала *C++ Report* и первым председателем группы Agile Alliance.

---

[1] *Мартин Р. С.* Быстрая разработка программ: Принципы, примеры, практика.

[2] *Мартин Р. С.* Чистый код: Создание, анализ и рефакторинг. — СПб.: Питер.

[3] *Мартин Р. С.* Идеальный программист: Как стать профессионалом разработки ПО. — СПб.: Питер.

[4] *Мартин Р. С.* Чистая архитектура: Искусство разработки программного обеспечения. — СПб.: Питер.

[5] *Мартин Р. С.* Чистый Agile. Основы гибкости. — СПб.: Питер.

ОТ ИЗДАТЕЛЬСТВА

Обучающие видеоролики, ссылки на которые дает автор, — на английском языке. Срок бесплатного доступа к ним ограничен и составляет десять дней.

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 МАСТЕРСТВО

Мечта летать, наверное, так же стара, как само человечество. Древнегреческий миф о Дедале и Икаре датируется примерно 1550 годом до нашей эры. В последующие тысячелетия в погоне за этой мечтой множество смелых, но безрассудных людей привязывали к себе несуразные приспособления и прыгали со скал и башен навстречу неизбежной смерти.

Ситуация изменилась примерно 500 лет назад. Машины, эскизы которых нарисовал Леонардо да Винчи, не могли летать, но, по крайней мере, подход к их проектированию был логичным. Именно да Винчи понял, что полет возможен, поскольку сопротивление воздуха работает в обе стороны. Сопротивление, возникающее, когда на воздух давят сверху, создает такую же подъемную силу. Этот принцип стал основой создания всех современных самолетов.

Про идеи да Винчи забыли до середины XVIII века. А затем начались лихорадочные изыскания возможности летать. XVIII и XIX века стали временем упорных исследований и экспериментов в сфере воздухоплавания. Строились, тестировались, отбрасывались и совершенствовались безмоторные прототипы. Начала формироваться такая наука, как аэронавтика. Появились определения подъемной силы, сопротивления, тяги и гравитации. Смельчаки стали предпринимать попытки полетов.

Некоторые падали и погибали.

С конца XVIII века в течение почти 50 лет отец современной аэродинамики сэр Джордж Кейли (George Cayley) строил экспериментальные установки, прототипы и полноразмерные модели. Кульминацией его усилий стал первый пилотируемый полет планера.

А смельчаки продолжали падать и погибать.

Затем наступила эпоха паровых машин, которая принесла с собой возможность управляемых полетов. Были построены десятки прототипов и проведены множество экспериментов. Летный потенциал стали исследовать многочисленные ученые и энтузиасты. В 1890 году Клеман Адер (Clément Ader) на двухмоторной паровой машине пролетел 50 метров.

Но все равно оставались те, кто падал и погибал.

Двигатель внутреннего сгорания полностью изменил правила игры. Вероятнее всего, первый контролируемый полет совершил Густав Уайтхед (Gustave Whitehead) в 1901 году. Первый же по-настоящему управляемый полет на оснащенном двигателем аппарате тяжелее воздуха выполнили 17 декабря 1903 года в местечке Килл-Девил-Хиллз штата Северная Каролина братья Райт (Wright Brothers). Но даже тогда хватало и тех, кто падал и погибал.

Тем не менее всего за одну ночь мир изменился. Одиннадцать лет спустя, в 1914 году, над Европой шли воздушные бои на бипланах. И хотя в этих боях многие разбивались и погибали, столько же разбилось и погибло при попытках научиться летать. Принципы полета были более-менее понятными, а вот как полет осуществляется *технически*, люди почти не понимали.

Спустя еще два десятилетия грозные истребители и бомбардировщики несли смерть и разрушения городам Франции и Германии. Эти самолеты летали очень высоко, были оснащены пулеметами и обладали огромной разрушительной силой.

За время Второй мировой было потеряно 65 тысяч американских самолетов. Но из них только 23 тысячи были потеряны в боях. Куда чаще летчики гибли потому, что толком *не умели* летать.

Следующее десятилетие ознаменовалось появлением реактивных самолетов, преодолением звукового барьера и взрывным ростом количества коммерческих авиалиний и гражданских авиаперевозок. Начался век высоких скоростей, и состоятельные люди получили возможность за считаные часы перемещаться между городами и странами.

Количество авиакатастроф при этом ужасало, так как мы еще многого не понимали в самолетостроении и пилотировании. Тем не менее к концу 1950-х пассажирские самолеты Боинг 707 уже летали по всему миру. А к концу 1960-х появился первый широкофюзеляжный реактивный самолет Боинг 747. Воздушные путешествия стали самым безопасным[1] и эффективным средством передвижения в истории. Но это потребовало много времени и человеческих жертв.

---

[1] Если не брать в расчет Боинги 737 Max.

Чесли Салленбергер (Chesley Sullenberger) родился в 1951 году в городе Денисон, штат Техас. Настоящее дитя века высоких скоростей. Он научился летать в шестнадцать и в конце концов начал пилотировать сверхзвуковые истребители F-4 Phantom. В 1980 году перешел на работу в гражданскую авиацию и стал пилотом US Airways.

Пятнадцатого января 2009 года, сразу после вылета из аэропорта Ла-Гуардия пилотируемый Салленбергером Airbus A320, на борту которого находились 155 человек, столкнулся со стаей гусей и потерял оба реактивных двигателя. Благодаря опыту, приобретенному за более чем 20 тысяч часов воздушных полетов, Салленбергеру удалось развернуть выведенный из строя лайнер и приводниться на поверхность реки Гудзон. Сто пятьдесят пять человек были спасены, поскольку командир воздушного судна Салленбергер был настоящим мастером своего дела.

Мечта о быстрых и точных вычислениях и управлении данными тоже, похоже, существует столько же времени, сколько и человечество. Тысячи лет назад люди использовали для счета пальцы, палочки, бусины. Более четырех тысяч лет назад появились счеты. Около двух тысяч лет назад создали механические устройства для предсказания движения звезд и планет. А около 400 лет назад изобрели логарифмическую линейку.

В начале XIX века Чарлз Бэббидж (Charles Babbage) начал строить механические вычислительные аппараты, которые приводились в действие специальными рукоятками. Это были настоящие вычислительные комплексы с памятью и арифметической обработкой. Но их производство затруднял низкий уровень металлообработки. Бэббидж построил несколько прототипов, но коммерческого успеха они не имели.

В середине 1800-х годов у него возникла идея гораздо более мощного программируемого вычислительного устройства, которое в итоге стало прообразом современного цифрового компьютера. Свое творение Бэббидж назвал *аналитической машиной*.

Дочь лорда Байрона Ада, графиня Лавлейс, переводя на английский язык лекцию Бэббиджа, записанную по-французски, пришла к неожиданному выводу, что *со временем такая машина не будет ограничена*

работой с числами, а сможет обрабатывать любые объекты. Поэтому Аду часто называют первым в мире настоящим программистом.

Из-за отсутствия финансирования и низкого уровня технологий того времени аналитическая машина Бэббиджа так и не была построена. На много десятилетий прогресс в области цифровых компьютеров остановился. Впрочем, то была золотая пора механических *аналоговых* счетных машин.

В 1936 году Алан Тьюринг (Alan Turing) показал, что не существует общего способа доказать решаемость произвольного диофантового уравнения[1]. Для этого математик воспользовался моделью в виде простого, хотя и бесконечного цифрового компьютера, и доказал существование невычислимых чисел. В процессе работы над этим доказательством были изобретены конечные автоматы, машинный язык, язык символов, макросы и примитивные подпрограммы. Тьюринг изобрел то, что сегодня мы бы назвали программным обеспечением.

Почти в то же время Алонзо Черч независимо от Тьюринга сформулировал и доказал эту же задачу, попутно разработав лямбда-исчисление — основную концепцию функционального программирования.

В 1941 году Конрад Цузе (Konrad Zuse) построил первый электромеханический программируемый цифровой компьютер Z3. Он состоял из более чем 2000 реле и работал с тактовой частотой от 5 до 10 Гц. Машина использовала двоичную арифметику, длина машинного слова составляла 22 бита.

Во время Второй мировой войны Тьюринга пригласили помочь экспертам из Блетчли-парка (центра британской разведки), которые бились над расшифровкой кодов немецкой «Энигмы». Она представляла собой электромеханическую роторную машину, которая случайным образом меняла символы текстовых сообщений, транслируемых по радиотелеграфу. Тьюринг помог создать устройство для расшифровки кодов «Энигмы».

После войны он сыграл важную роль в создании и программировании одного из первых в мире ламповых компьютеров — Automatic

[1] Уравнение с целыми коэффициентами.

Computing Engine (ACE). Первоначальный прототип содержал 1000 электронных ламп и обрабатывал двоичные числа со скоростью миллион бит в секунду.

В 1947 году, написав несколько программ для этой машины и изучив ее возможности, Тьюринг прочитал лекцию, во время которой прозвучали следующие пророческие заявления:

> *Нам потребуется большое количество способных математиков для преобразования задач в форму, подходящую для машинной обработки.*
>
> *Одной из трудностей станет необходимость придерживаться определенных практик, позволяющих не терять из виду то, что делаем.*

И за одну ночь мир изменился.

За несколько лет была изобретена память на магнитных сердечниках. Появилась возможность за микросекунды получать доступ к сотням тысяч, если не к миллионам битов памяти. А массовое производство электронных ламп привело к появлению более дешевых и надежных компьютеров. Становилось реальностью мелкосерийное массовое производство. К 1960 году фирма IBM продала 140 компьютеров модельного ряда 70х. Это были огромные машины на электронных лампах стоимостью в миллионы долларов.

Тьюринг использовал для написания программ двоичный код, но все понимали, что это непрактично. В 1949 году Грейс Хоппер (Grace Hopper) придумала слово *компилятор*, а к 1952 году создала его первую версию A-0. В конце 1953 года Джон Бэкус (John Backus) представил первую спецификацию языка FORTRAN. К 1958 году появились ALGOL и LISP.

Первый работающий транзистор был создан Джоном Бардином (John Bardeen), Уолтером Браттейном (Walter Brattain) и Уильямом Шокли (William Shockley) в 1947 году. В 1953 году был введен в эксплуатацию первый транзисторный компьютер. Переход с электронных ламп на транзисторы изменил все. Компьютеры стали меньше, быстрее, дешевле и намного надежнее.

К 1965 году IBM выпустила 10 тысяч компьютеров модели 1401. Они сдавались в аренду за 2500 долларов в месяц, что было вполне доступно среднему бизнесу. Предприятия нуждались в программистах, соответственно, спрос на них стал расти.

Кто программировал все эти машины? Университетских курсов не существовало. В 1965 году не было возможности поступить в высшее учебное заведение, чтобы научиться программировать. Поэтому программистов брали из бизнеса. Это были зрелые люди в возрасте от 30 до 50 лет.

К 1966 году IBM ежемесячно производила 1000 компьютеров серии System/360. Бизнесу этого было мало. Это были машины с объемом памяти 64 Кбайт и выше, умеющие выполнять сотни тысяч инструкций в секунду.

В том же году в Норвежском вычислительном центре в процессе работы над операционной системой Univac1107 Оле-Йохан Даль (Ole-Johan Dahl) и Кристен Нюгор (Kristen Nygard) изобрели Simula 67, который можно считать объектным расширением языка ALGOL. Это был первый объектно-ориентированный язык.

И все это всего через два десятка лет после лекции Алана Тьюринга!

Через два года, в марте 1968-го, Эдсгер Дейкстра (Edsger W. Dijkstra) написал в журнал *Communications of the ACM (CACM)* свое знаменитое письмо. Редактор озаглавил его «О вреде оператора goto»[1]. Так родилось структурное программирование.

В 1972 году в лабораториях Белла в штате Нью-Джерси Кен Томпсон (Ken Thompson) и Деннис Ритчи (Dennis Ritchie) в промежутке между работой над собственными проектами выпросили у коллег из соседней группы время на компьютере PDP 7 и изобрели операционную систему UNIX и язык программирования C.

После этого события стали развиваться с почти головокружительной скоростью. Посмотрите на этот перечень ключевых дат. Задайте себе

[1] *Dijkstra E. W.* Go To Statement Considered Harmful // Communications of the ACM. 1968. № 3.

вопрос: сколько в мире компьютеров? А сколько программистов? Откуда они все взялись?

1970 — с 1965 года корпорация Digital Equipment Corporation продала свыше 50 тысяч компьютеров PDP-8.

1970 — Уинстон Ройс (Winston Royce) написал статью «Управление разработкой больших программных систем», в которой описывалась каскадная модель разработки.

1971 — фирма Intel выпустила микропроцессор 4004.

1974 — фирма Intel выпустила микропроцессор 8080.

1977 — фирма Apple выпустила первый серийный персональный компьютер Apple II.

1979 — фирма Motorola выпустила 16-битный микропроцессор 68000.

1980 — Бьёрн Страуструп (Bjarne Stroustrup) разработал язык программирования C, добавив к нему возможность работы *с классами* (чтобы сделать похожим на язык Simula).

1980 — Алан Кей (Alan Kay) изобрел объектно-ориентированный язык Smalltalk.

1981 — фирма IBM выпустила первый массовый персональный компьютер IBM PC.

1983 — фирма Apple выпустила первый персональный компьютер Macintosh, имеющий 128 Кбайт памяти.

1983 — Бьёрн Страуструп переименовал C с классами в C++.

1985 — Министерство обороны США приняло каскадную модель в качестве официального стандарта разработки программного обеспечения (стандарт DOD-STD-2167A).

1986 — издательство Addison-Wesley выпустило книгу Бьёрна Страуструпа «Язык программирования C++».

1991 — издательство Benjamin/Cummings выпустило книгу Гради Буча (Grady Booch) «Объектно-ориентированный анализ и проектирование с примерами приложений».

1991 — Джеймс Гослинг (James Gosling) изобрел язык Java (изначально называвшийся Oak).

1991 — Гвидо ван Россум (Guido Van Rossum) придумал язык Python.

1995 — издательство Addison-Wesley выпустило книгу «Приемы объектно-ориентированного проектирования. Паттерны проектирования», написанную Эрихом Гаммой (Erich Gamma), Ричардом Хелмом (Richard Helm), Джоном Влиссидесом (John Vlissides) и Ральфом Джонсоном (Ralph Johnson).

1995 — Юкихиро Мацумото (Yukihiro Matsumoto) создал язык программирования Ruby.

1995 — Брендан Эйх (Brendan Eich) создал язык JavaScript.

1996 — компания Sun Microsystems выпустила первую официальную версию языка Java.

1999 — компания Microsoft придумала язык C# (сначала называвшийся Cool) и платформу .NET.

2000 — проблема 2000 года.

2001 — написан Agile Manifesto (манифест гибкой разработки программного обеспечения).

За период с 1970 по 2000 год тактовая частота компьютеров увеличилась на три порядка, плотность упаковки данных — на четыре, дисковое пространство, как и объем оперативной памяти, — на шесть-семь порядков. Причем если раньше за доллар можно было купить один бит оперативной памяти, то теперь — гигабит. Немыслимо представить, как изменилось аппаратное обеспечение, но даже если просто суммировать все вышеупомянутые мной вещи, можно сказать, что наши возможности возросли примерно на 30 порядков.

И все это чуть более чем через полвека после лекции Алана Тьюринга.

Сколько сейчас программистов? Сколько строк кода написано? Насколько хорош этот код?

Попытайтесь сравнить это с историей становления авиации. Видите сходство? Видите, как постепенно развивалась теоретическая часть,

как энтузиасты шли на приступ и, бывало, терпели поражение, как понемногу рос профессионализм? Как десятилетиями люди не совсем понимали, что делают?

И теперь, когда от наших навыков зависит само существование нашего общества, есть ли среди нас Салленбергеры, которые сейчас так нужны? Подготовлены ли программисты, столь же квалифицированные, как современные пилоты авиакомпаний? Есть ли у нас профессионалы, в которых наверняка будет нужда?

Мастерство — это доскональное знание того, как получить отличный результат. Оно появляется как следствие хорошего обучения и богатого опыта. До недавнего времени в индустрии программного обеспечения не хватало ни того ни другого. Как правило, программисты недолго занимались написанием программ, поскольку рассматривали это занятие всего лишь как ступеньку на пути к руководящей работе. Соответственно, не многие из этих людей приобретали достаточный опыт, чтобы обучать программированию других. В довершение всего, количество новичков удваивается примерно каждые пять лет, в результате чего доля опытных программистов остается чрезвычайно низкой.

При таком подходе большинство программистов не осваивают принятые практики, стандарты и этику, то есть вообще не знакомятся с вещами, отвечающими за профессионализм. Все относительно короткое время их работы эти люди остаются необученными новичками. Неудивительно, что большая часть написанного ими кода не соответствует стандартам, плохо структурирована, небезопасна, содержит ошибки и в основном находится в ужасном состоянии.

В этой книге я хочу рассказать о стандартах, принятых практиках и этических нормах, то есть о вещах, которые, по моему мнению, должен знать и соблюдать каждый программист, чтобы постепенно приобрести необходимые в его профессии знания и навыки.

I Принятые практики

Что такое принятая практика? Это набор правил, состоящий из обязательной и произвольной частей. Обязательная часть — это то, что делает практику действенной; это причина ее существования. Произвольная часть придает практике форму и содержание. Без произвольной части практика существовать не может.

Например, хирург перед операцией моет руки. Понаблюдав за этим процессом, вы увидите, что мытье рук выполняется особым образом. Хирург не просто намыливает их под проточной водой, как это делает любой из нас. Он следует ритуальной процедуре, которая выглядит примерно так:

- взять соответствующее мыло;
- взять подходящую щетку;
- для каждого пальца сделать:
  - десять движений по верхней стороне;
  - десять движений по левой стороне;
  - десять движений по тыльной стороне;
  - десять движений по правой стороне;
  - десять движений под ногтем;
- и т. д.

Обязательная часть практики должна быть очевидной. Хирург обязан иметь очень чистые руки. Но обратили ли вы внимание на произвольную часть? Почему движений десять, а не восемь или двенадцать? Зачем делить палец на пять областей? Почему не на три или на семь?

В данном случае все эти числа выбраны произвольно. Просто считается, что этого будет достаточно.

В этой книге я расскажу вам о пяти практиках профессиональной разработки программного обеспечения. Одним из них уже полвека. Другим всего пара десятков лет. Но все они показали свою полезность. Без них в сфере создания ПО было бы практически немыслимым само понятие профессионального мастерства.

Каждая из этих практик имеет собственные обязательные и произвольные элементы. В процессе чтения вы можете столкнуться с тем, что некоторые из этих практик покажутся вам необоснованными или ненужными. В этом случае попытайтесь понять, какие элементы — обязательные или только произвольные — вызывают это чувство. Не позволяйте произвольным элементам сбить вас с толку. Как только вы поймете сущность каждой практики, влияние произвольных элементов, скорее всего, уменьшится.

Скажем, в 1861 году Игнац Земмельвейс (Ignaz Semmelweis) опубликовал статью о необходимости мытья рук для врачей. Результаты его исследований ошеломляли. Он смог показать, что в случаях, когда перед осмотром беременных врачи тщательно мыли руки водным раствором хлора, смертность от родильной горячки, от которой в то время умирала каждая десятая женщина, падала практически до нуля.

Но врачи того времени, рассматривая предложенную Земмельвейсом практику, не смогли отделить обязательную часть от произвольной. Водный раствор хлора был произвольной частью. Суть практики заключалась в самом факте мытья рук. Но мыть их хлорной водой было неудобно, поэтому врачи отвергли саму идею.

Прошло много десятилетий, прежде чем они стали это делать.

ЭКСТРЕМАЛЬНОЕ ПРОГРАММИРОВАНИЕ

В 1970 году, после статьи Уинстона Ройса каскадная разработка стала общепринятой практикой. Исправление этой ошибки заняло почти 30 лет.

К 1995 году специалисты в сфере программного обеспечения начали рассматривать другой, более поэтапный подход. Была предложена к рассмотрению методология Scrum, разработка, управляемая функциональностью (feature-driven development, FDD), метод разработки динамических систем (dynamic systems development method, DSDM) и методология Crystal. Но в целом в отрасли мало что изменилось.

В 1999 году издательство Addison-Wesley выпустило книгу Кента Бека *Extreme Programming Explained*[1]. Предложенная Беком концепция базировалась на идеях из вышеперечисленных подходов, добавляя к ним кое-что новое, а именно *практики разработки*.

Следующие два года энтузиазм по отношению к экстремальному программированию рос экспоненциально. Именно это и привело к Agile-революции. Экстремальное программирование по сей день остается наиболее определенным и наиболее полным из всех Agile-методов. Эта глава посвящена принятым в нем практикам разработки.

Жизненный цикл

На рис. 1.1 вы видите *жизненный цикл* Рона Джеффриса, содержащий перечень практик XP. Я расскажу вам о четырех практиках, расположенных в центре, и об одной крайней слева.

Рис. 1.1. Практики экстремального программирования

[1] *Бек К.* Экстремальное программирование. — СПб.: Питер.

В центре находятся такие практики, как разработка через тестирование (test-driven development, TDD), рефакторинг, простота проектирования и парное программирование (которое мы будем называть *совместным программированием*). В крайней левой позиции располагается наиболее технически и инженерно-ориентированная из коммерческих практик XP — пользовательское тестирование.

Это основополагающие практики профессиональной разработки программного обеспечения.

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ

Разработка через тестирование — ключевая дисциплина. Без нее все остальные практики невозможны или бесполезны. Именно поэтому две следующие главы, в которых она описывается, занимают почти половину книги и наполнены техническими деталями. Такой подход может показаться вам несбалансированным. Более того, мне тоже так кажется. Я изо всех сил пытался понять, как с этим быть, но пришел к выводу, что подобный перекос возник как следствие ситуации в нашей отрасли. К сожалению, с этой практикой хорошо знакомо слишком маленькое количество программистов.

Практика разработки через тестирование определяет действия программиста вплоть до секунд. Ее невозможно применить заранее или постфактум. Ее нельзя не заметить, поскольку она пронизывает весь процесс. Ее не получится придерживаться частично. Вы или практикуете разработку через тестирование, или нет.

Суть TDD очень проста. Все начинается с маленьких циклов и тестов. Причем тесты всегда оказываются на первом месте. Они первыми пишутся. Они первыми приводятся в порядок. Они предшествуют любой задаче. При этом все задачи разбиты на мельчайшие циклы.

Время цикла измеряется не в минутах — в секундах. Оно измеряется в символах, а не в строках. Цикл обратной связи замыкается, едва открывшись.

Цель TDD — создать набор тестов, которому можно полностью доверять. Если этот набор тестов пройден, значит, при развертке кода можно чувствовать себя в безопасности.

Разработка через тестирование — самая обременительная и сложная из всех практик. Она обременительна, поскольку влияет на все остальные аспекты. С нее вы начинаете и ею заканчиваете. Она накладывает ограничения на все действия. Она поддерживает темп работы, невзирая на давление окружающей среды.

Сложность разработки через тестирование обусловлена сложностью кода. Для каждого варианта кода существует соответствующий вариант TDD. При этом тесты должны соответствовать коду, но не быть с ним связаны, должны охватывать почти все аспекты кода, но при этом выполняться за секунды. Разработка через тестирование — скрупулезно нарабатываемый и сложный навык, который осваивается с большим трудом, но дает потрясающие результаты.

РЕФАКТОРИНГ

Рефакторинг — это практика, позволяющая писать чистый код. Она трудно реализуема, а порой и невозможна без TDD[1]. Соответственно, получить чистый код без TDD сложно или невозможно.

Рефакторинг превращает плохо структурированный код в код с лучшей структурой, *не меняя его поведения*. Последняя часть здесь — самая важная. Именно неизменность поведения кода гарантирует, что даже после изменения его структуры он останется *безопасным*.

Хотя программные системы и деградируют со временем, в них предпочитают не вмешиваться из страха повлиять на их поведение. Наличие безопасного способа очистки позволяет *привести код в порядок*, остановив деградацию.

---

[1] Существуют и другие практики, так же хорошо, как TDD, способствующие выполнению рефакторинга. Например, предложенная Кентом Беком концепция TCR (test && commit || revert). Впрочем, на момент написания данного текста она не получила широкого распространения и интересна исключительно с академической точки зрения.

Как гарантировать, что улучшения не повлияют на поведение? С помощью тестов.

Практика рефакторинга также очень сложна, поскольку существует множество способов создания плохо структурированного кода, а значит, и множество стратегий его очистки. Более того, каждая из этих стратегий должна плавно и согласованно вписываться в цикл разработки через тестирование. Как видите, эти практики настолько тесно переплетены, что практически неразделимы. Сложно представить рефакторинг без TDD, а TDD без рефакторинга.

ПРОСТОТА ПРОЕКТИРОВАНИЯ

Жизнь на земле можно описывать на разных уровнях. Высший уровень — это экология, изучающая взаимодействие в рамках биосистем. Ниже находится физиология, изучающая внутреннюю механику жизни. Еще ниже — микробиология, рассматривающая клетки, нуклеиновые кислоты, белки и другие макромолекулярные системы. Те, в свою очередь, описываются химией, которая базируется на квантовой механике.

Если таким же способом рассматривать программирование, то TDD окажется аналогом квантовой механики. Рефакторинг будет соответствовать химии, а простота проектирования — микробиологии. На уровне физиологии у нас окажутся принципы SOLID, объектно-ориентированное проектирование и функциональное программирование, а на уровне экологии — архитектура.

Простота проектирования практически невозможна без рефакторинга. Ведь именно она выступает конечной целью рефакторинга, в то время как сам он является единственным практическим средством для достижения этой цели. Именно рефакторинг позволяет получить проект, состоящий из простых атомарных фрагментов, хорошо вписывающихся в более крупные структуры программ, систем и приложений.

Простота проектирования базируется на четырех несложных правилах, так что придерживаться данной практики легко. Однако, в отличие от TDD и рефакторинга, это неточная дисциплина. Здесь

приходится опираться на рассуждения и опыт. Первый признак, по которому можно отличить выучившего правила новичка от понимающего принципы специалиста, — отличный результат. Майкл Физерс (Michael Feathers) назвал это *чувствовать проект*.

СОВМЕСТНОЕ ПРОГРАММИРОВАНИЕ

Практика совместного программирования отражена в искусстве работы в команде. Она включает в себя такие вещи, как парное или групповое программирование, анализ кода и мозговые штурмы. Совместное программирование предполагает участие вообще всех членов команды — как программистов, так и всех остальных. Это основной способ поделиться знаниями, обеспечить согласованность работы и объединить команду в функционирующее целое.

Совместное программирование — наименее техническая и наименее директивная из всех практик. Но порой она оказывается самой важной, поскольку создать эффективную команду очень непросто, и это большая ценность.

ПОЛЬЗОВАТЕЛЬСКОЕ ТЕСТИРОВАНИЕ

Практика пользовательского тестирования связывает команду разработчиков программного обеспечения с его заказчиками. Перед разработчиками стоит цель — обеспечить поведение системы в соответствии с данной заказчиками спецификацией. Для проверки этого соответствия пишутся тесты. Успешное их прохождение означает, что система ведет себя так, как указано в требованиях.

Представители заказчика должны иметь возможность прочитать и понять эти тесты и внести в них коррективы. Наблюдение за процессом тестирования и участие в нем позволяют заказчикам убедиться, что программное обеспечение делает именно то, что от него требуется.

2 РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ

Рассказ о TDD занял у меня две главы. Сначала я подробно, с множеством технических деталей опишу вам основы данной разработки. В текущей главе вы пошагово познакомитесь с этой практикой. Вы найдете здесь множество фрагментов кода и ссылки на видеоролики.

В следующей главе я расскажу о ловушках и непонятных ситуациях, с которыми сталкиваются новички в TDD; я упомяну в том числе написание тестов для низкоуровневых компонентов, таких как базы данных и графические пользовательские интерфейсы. Вы познакомитесь с принципами и шаблонами проектирования тестов и узнаете о некоторых интересных и серьезных теоретических возможностях.

ОБЩИЕ СВЕДЕНИЯ

Ноль — очень важная цифра. Это значение равновесия. Когда оба плеча весов в равновесии, стрелка показывает ноль. Атом, в ядре которого количество электронов совпадает с количеством протонов, имеет нулевой заряд, то есть оказывается электрически нейтральным. Равна нулю и сумма сил, приложенных к находящемуся в состоянии покоя объекту. Ноль — число баланса.

Вы когда-нибудь задумывались, почему состояние расчетного счета в банке называют балансом? Дело в том, что это состояние представляет собой итог всех транзакций, всех снятий денег со счета и внесений их на счет. При этом в транзакции принимают участие две стороны, *между* которыми происходит перемещение денег.

Схематично это можно представить следующим образом: на *ближней* стороне ваш банковский счет, а на *дальней* — счет второго участника транзакции. Если на ближней стороне деньги вносятся на счет, значит, где-то на дальней стороне эта сумма снимается со счета. Каждый раз, когда вы выписываете чек, с вашего счета снимаются деньги и вносятся на какой-то другой. Сумма всех транзакций по счету и есть баланс. Сумма всех ближних и дальних сторон, участвовавших в транзакциях, должна быть равна нулю.

Две тысячи лет назад Гай Плиний Секунд, известный как Плиний Старший, реализовал эту формулу бухгалтерского учета, придумав

принцип двойной записи. Этот принцип веками совершенствовался каирскими банкирами, а затем венецианскими купцами. В 1494 году друг Леонардо да Винчи монах-францисканец Лука Пачоли (Luca Pacioli) впервые подробно описал принцип двойной записи в своей книге. Печатный станок уже был изобретен, и это способствовало распространению информации.

В 1772 году, когда Европа столкнулась с суровой рецессией, Джозайе Веджвуду (Josiah Wedgwood) пришлось изрядно побороться за успех. Продукция его завода керамики перестала пользоваться спросом. Веджвуд ввел на предприятии бухгалтерский учет с двойной записью, чтобы понять, где рождается прибыль и как ее увеличить. Это позволило предотвратить надвигающееся банкротство и построить бизнес, который дожил до наших дней.

Веджвуд был не одинок. Индустриализация радикально изменила экономическую ситуацию в Европе и Америке. Как следствие, все больше и больше фирм начали использовать эту практику для управления денежными потоками.

Вот что писал в 1795 году Иоганн Вольфганг фон Гете в своем романе «Годы учения Вильгельма Мейстера»:

> *— Отбрось ее, швырни в огонь! — возопил Вернер. — Идея ее отнюдь не похвальна; этот опус претил мне с самого начала, а на тебя навлек неодобрение отца. Стихи, может, и складные, но изображение фальшивое. До сих пор помню твою дряхлую, немощную колдунью — олицетворение ремесла. Верно, ты набрел на этот образ в какой-нибудь убогой мелочной лавчонке. О торговом деле ты тогда понятия не имел; я же не знаю человека, чей кругозор был бы шире, должен быть шире, нежели кругозор настоящего коммерсанта. Сколь многому учит нас порядок в ведении дел! Он позволяет нам в любое время обозреть целое, не отвлекаясь на возню с мелочами. Какие преимущества дает купцу двойная бухгалтерия! Это одно из прекраснейших изобретений ума человеческого, и всякому хорошему хозяину следует ввести ее в свой обиход.*

Сегодня двойная запись используется практически во всех странах. Эта практика в значительной степени *лежит в основе* профессии бухгалтера.

Но обратите внимание, какими словами Гете описывает так ненавистные ему средства «коммерции»:

> *До сих пор помню твою дряхлую, немощную колдунью — олицетворение ремесла. Верно, ты набрел на этот образ в какой-нибудь убогой мелочной лавчонке.*

Вы когда-нибудь видели код, подходящий под это описание? Уверен, что да. У меня тоже есть такой опыт. И если вы работаете так же долго, как я, то вам явно доводилось в изобилии наблюдать такой код. И так же как я, вы написали огромное количество такого кода.

Теперь еще раз обратимся к словам Гете:

> *Сколь многому учит нас порядок в ведении дел! Он позволяет нам в любое время обозреть целое, не отвлекаясь на возню с мелочами.*

Примечательно, что этими словами Гете описывает огромное преимущество, которое дает простая практика двойной записи.

Программное обеспечение

Для ведения современного бизнеса жизненно необходимо содержать счета в порядке, а для этого нужна практика двойной записи. Но разве надлежащее обслуживание программного обеспечения менее важно? Ведь в XXI веке программное обеспечение лежит в основе любого бизнеса.

Какая практика позволит разработчикам получить четкое представление о создаваемом ПО и контроль над ним, причем в той же степени, какую практика двойной записи дает бухгалтерам и менеджерам? Возможно, вы считаете, что разработка программного обеспечения и бухгалтерия относятся к настолько разным сферам, что их невозможно, да и незачем сопоставлять друг с другом. Позволю себе не согласиться.

Бухгалтерский учет сродни магическому искусству. Человек, несведущий в его ритуалах, мало что понимает в работе бухгалтера. Что является результатом этой работы? Набор документов, систематизи-

рованных непонятным для неспециалиста образом. Эти документы наполнены символами, смысл в которых зачастую может увидеть только бухгалтер. Но ошибка хотя бы в одном из этих символов может обойтись очень дорого, вплоть до остановки предприятия и ареста его руководства.

Теперь подумайте, сколько общего в бухгалтерском учете и разработке программного обеспечения. Создание ПО — без преувеличения магия. Те, кто не разбирается в его секретах, не могут даже представить, как все устроено. Результат труда, по сути, тоже представляет собой набор документов, причем систематизированных запутанным образом и наполненных понятными только посвященным символами. И снова ошибки могут повлечь за собой крайне неприятные последствия.

Как видите, эти профессии очень похожи. В обоих случаях речь идет об огромной и скрупулезной работе с деталями. И для успеха необходима серьезная подготовка и опыт. Представители обеих профессий создают сложные документы, в которых важна точность на уровне отдельных символов.

Даже если бухгалтеры и программисты не хотят этого признавать, их профессии очень похожи. И более молодой профессии имеет смысл присмотреться к практикам из более старой.

Как вы увидите далее, TDD тоже является практикой двойной записи. Это та же самая методика, преследующая ту же цель и дающая те же результаты. Все повторяется в связанных аккаунтах, за балансом которых необходимо следить с помощью тестов.

Три закона TDD

Прежде чем я начну знакомить вас с тремя законами, хотелось бы сделать некоторые предварительные замечания. Разработка через тестирование сводится к следующим действиям.

1. Создание набора тестов, прохождение которых позволяет перейти к рефакторингу кода и предполагает его последующее развертывание. То есть успешное прохождение этого набора тестов означает, что систему можно безопасно развернуть.

2. Написание окончательной версии кода, достаточно несвязанной, чтобы этот код можно было тестировать и подвергать рефакторингу.
3. Формирование чрезвычайно короткого цикла обратной связи, который позволяет поддерживать стабильный ритм и производительность при написании программы.
4. Создание тестов и производственного кода, которые не связаны в такой степени, что их удобно обслуживать и ничто не препятствует репликации вносимых в них изменений.

Практика TDD сводится к соблюдению трех законов, практически полностью состоящих из произвольной части. Отсутствие обязательной части подтверждает, например, тот факт, что конечная цель может достигаться разными средствами. В частности, с помощью предложенной Кентом Беком практики test && commit || revert (TCR). Несмотря на свою непохожесть на TDD, TCR позволяет получить точно такие же результаты.

Следовать трем законам, составляющим основу TDD, очень тяжело, особенно поначалу. Для этого нужны навыки и знания, получить которые не так-то просто. Но без них попытки следовать законам практически всегда заканчиваются разочарованием и отказом от практики. Постепенно я дам вам всю необходимую информацию, а пока просто будьте готовы к внутреннему сопротивлению при изучении этого материала. Так и должно быть.

Первый закон

Не пишите код, пока не напишете первый тест.

Опытному программисту этот закон может показаться глупым. У вас тоже, скорее всего, возник вопрос, как написать тест без кода, который следует протестировать. Это естественная реакция, ведь считается, что сначала пишется код, а только *потом* тесты для него. Но если вдуматься, то само знание того, как должен работать код, позволяет написать для него тест. Хотя, конечно, эта идея контринтуитивна и кажется совершенно нелогичной.

Второй закон

Пишите минимальный тест, который не будет пройден. Добейтесь его прохождения, написав необходимый код.

Опытный программист сразу поймет, что первая же строка теста вызовет сбой компиляции, ведь она предполагает взаимодействие с кодом, которого пока не существует. Фактически это означает невозможность написать тест, состоящий более чем из одной строки, так как вам сразу придется переключиться на написание производственного кода.

Третий закон

Ограничьтесь минимальным количеством кода, необходимым для прохождения текущего теста. После этого сразу пишите следующий тест.

На этом цикл завершается. Очевидно, что эти три закона вовлекают вас в цикл продолжительностью всего несколько секунд. В утрированном виде процесс выглядит примерно так:

- вы пишете строку тестового кода, и этот тест, конечно же, не проходит;
- вы пишете строку производственного кода, позволяющую пройти тест;
- вы пишете следующую строку тестового кода, и тест снова не проходит;
- вы пишете следующий рабочий код, позволяющий пройти тест;
- вы пишете еще одну-две строки тестового кода и обнаруживаете, что проверка утверждения не пройдена;
- вы пишете еще одну-две строки производственного кода, обеспечивающие положительный результат проверки утверждения.

Вся ваша дальнейшая работа будет выглядеть только так.

С точки зрения опытного программиста, ситуация, скорее всего, выглядит абсурдно. Три закона запирают нас внутри цикла, длящегося

всего несколько секунд. Нам все время приходится переключаться между тестовым и производственным кодом. Невозможно просто добавить оператор `if` или цикл `while` или написать функцию. Мы навсегда попали в ловушку постоянных переключений.

На первый взгляд кажется, что это утомительно, скучно и медленно, что такой подход препятствует прогрессу и прерывает цепочку размышлений. Мало того, можно сказать, что все это выглядит просто глупо. Создается ощущение, что в результате мы можем породить только спагетти-код, практически неструктурированный, представляющий собой беспорядочный конгломерат из тестов и кода, который в состоянии пройти эти тесты.

Но не стоит торопиться с выводами, пока мы не посмотрели на процесс более детально.

Отказ от отладки

Представьте команду разработчиков, следующую трем вышеупомянутым законам. О любом из них в любой момент времени можно сказать, что код, который он пишет, совсем недавно успешно прошел все тесты.

А теперь попробуйте представить, как меняется жизнь разработчика, о коде которого можно сказать, что минуту назад он успешно прошел тестирование. Как вы думаете, насколько большой отладки требует его код? Очевидно, что в какой-то особой отладке такой код не нуждается.

Вы хорошо знакомы с работой отладчика? В любой момент готовы пометить функцию как отлаживаемую? Создали для себя множество горячих клавиш? Привычно расставляете точки останова и точки наблюдения и с головой погружаетесь в процесс отладки?

Это совершенно ненужные навыки!

Дело в том, что отлично освоить отладчик можно только в процессе его интенсивного использования. Но тратить много времени на отладку нерационально. Целесообразнее писать работающий код, а не исправлять неработающий.

Мне бы хотелось, чтобы вы настолько редко прибегали к отладчику, что забыли бы назначение горячих клавиш. Я хочу, чтобы, на-

тыкаясь на значки входа в процедуры и выхода из них, вы с трудом вспоминали, что это такое. Чтобы в ваших руках отладчик работал неуклюже и медленно. Очень надеюсь, что наши желания совпадают. Чем комфортнее вам работать с отладчиком, тем дальше вы уходите с правильного пути.

Разумеется, я не могу вам обещать, что три закона сделают отладчик совсем ненужным. В любом случае время от времени вам придется к нему прибегать. Когда имеешь дело с программным обеспечением, это неизбежно. Но частота и продолжительность сеансов отладки резко сократятся. Большую часть времени вы будете писать работающий код, а не исправлять неработающий.

Документирование

Если вы когда-либо интегрировали в систему пакеты сторонних производителей, то знаете, что в составе каждого такого пакета есть PDF-файл, созданный техническим писателем. Это файл с описанием процедуры интеграции. В конце документа почти всегда можно найти *примеры кода*.

Разумеется, первым делом вы заглядываете именно в конец. Зачем читать, что написал *о коде* технический писатель, если можно посмотреть сам код. Код скажет вам гораздо больше, чем все, что о нем написано. Если повезет, то его даже удастся скопировать в свое приложение и заставить там работать.

Разработчик, который соблюдает три закона, пишет *примеры кода* для всей системы. Ведь создаваемые тесты объясняют каждую мельчайшую деталь функционирования. Если вы хотите узнать, как создать определенный объект, тесты покажут все возможные способы его создания. А чтобы понять, как вызвать определенную функцию API, опять же нужно обратиться к тестам, благо они демонстрируют как саму функцию, так и все ее потенциальные ошибки и исключения. В наборе тестов есть все, что требуется для знакомства с деталями системы.

Тесты из этого набора представляют собой документацию, описывающую всю систему на самом низком уровне. И это документация на хорошо понятном вам языке. Она полностью непротиворечива. Она

создана настолько в соответствии с законами, что работает. Она не может рассинхронизироваться с системой.

Тесты — это почти идеальная документация.

Нет, преувеличивать их достоинства я не хочу. Цель и назначение системы тесты описывают не очень хорошо. Для этого существует документация высокого уровня. Однако на низком уровне они лучше любого документа, который можно было бы написать. Потому что это код. А он показывает, как все обстоит на самом деле.

Если вы боитесь, что тесты так же сложны для понимания, как и система в целом, то это не так. Каждый тест представляет собой небольшой фрагмент кода, сфокусированный на какой-то очень узкой части системы. Сами по себе тесты систему не образуют. Друг о друге тесты не знают, так что вам не придется иметь дело с запутанным нагромождением зависимостей. Каждый тест понятен сам по себе и сообщает вам именно то, что нужно узнать в очень узкой части системы.

Разумеется, некоторые люди пишут непрозрачные и хитросплетенные тесты, в которых не так-то просто разобраться. Поэтому одна из целей моей книги — научить вас писать тесты, представляющие собой четкое и понятное описание системы, лежащей в их основе.

Дыры в покрытии кода

Приходилось ли вам писать тесты постфактум? Мне кажется, такой опыт есть у большинства программистов. Ведь тесты чаще всего пишутся для уже готового кода. Но это не самое увлекательное занятие, не так ли?

К моменту, когда мы приступаем к написанию тестов, уже известно, что система работает. Это проверено вручную. Тесты пишутся только из чувства долга или потому, что руководство требует определенного покрытия кода. Поэтому приходится заниматься рутинной работой, зная, что каждый написанный нами тест будет пройден. Скучно до зубовного скрежета.

Нередко оказывается, что написать тест очень трудно. Ведь мы всеми силами пытались заставить код работать и не думали о том, чтобы сделать его удобным для тестирования. В результате можно даже попасть

в ситуацию, когда написание теста возможно только после внесения изменений в проект.

Но такие шаги требуют огромного приложения сил и отнимают массу времени. Более того, в процессе редактирования в код могут вкрасться ошибки. А мы-то уже знаем, что все работает, так как проверили вручную. Поэтому самым логичным выходом из этой трудной ситуации видится отказ от написания теста. Не говорите мне, будто никогда так не поступали. Я практически уверен, что такой опыт у вас есть.

Понятное дело, что так поступали не только вы, но и другие члены вашей рабочей группы, поэтому готовый набор тестов полон дыр.

Примерно представить количество этих дыр можно по громкости и продолжительности смеха программистов в процессе тестирования. Чем громче смех, тем меньше покрытие кода.

Очевидно, что пользы от такого набора тестов немного. Он может показать места сбоев и дать представление о том, что некоторые вещи работают.

Но хороший набор тестов отличается полным покрытием и позволяет уверенно рекомендовать *развертывание* системы. Если ваш набор тестов не внушает такого же доверия, то какой от него прок?

Приятные эмоции

Соблюдение трех законов вызывает совсем другие эмоции, чем нудное написание тестов. Прежде всего, это весело. Нет, конечно, не так весело, как бывает, когда ты сорвал джекпот в Вегасе. И не так весело, как поход на вечеринку или настольные игры с ребенком. Пожалуй, *весело* не совсем подходящее в данном случае слово.

Вспомните, что вы чувствовали, когда заработала ваша первая программа. Возможно, это произошло в местном компьютерном клубе, где стоял TRS-80 или Commodore 64. Возможно, вы написали легкомысленный бесконечный цикл, который снова и снова выводил на экран ваше имя, и ушли домой с легкой улыбкой на лице, чувствуя себя хозяином Вселенной, которому со временем будут подчиняться все компьютеры.

Отголосок этого чувства появляется всякий раз, когда вы проходите цикл TDD. Каждый тест, провалившийся именно так, как вы ожидали, заставляет вас кивнуть и слегка улыбнуться. Каждый раз, когда вы пишете код, который проходит этот тест, вы вспоминаете, что когда-то были властелином Вселенной и до сих пор сохранили эту *силу*.

Каждый раз при прохождении цикла TDD в вашем мозге вырабатывается небольшое количество эндорфинов, благодаря чему вы чувствуете себя чуть более компетентным, уверенным и готовым к новым свершениям. Это не очень сильное, но тем не менее довольно приятное чувство.

Проектирование

Впрочем, приятные эмоции — не главное. Куда важнее тот факт, что если тесты пишутся первыми, вы просто не сможете написать код, который трудно протестировать. Такая последовательность заставляет разрабатывать код с учетом уже готовых тестов. Поэтому соблюдение трех законов гарантирует легко тестируемый код.

Что затрудняет тестирование? Связи и зависимости. Вы же неизбежно будете писать несвязанный код. Причем отсутствие связанности будет достигаться способами, которые вы не могли и представить в момент начала работы.

И это очень хорошо.

Маленькая вишенка на торте

Итак, соблюдение трех законов TDD дает следующие преимущества:

- тратится больше времени на написание работающего кода и меньше — на отладку неработающего кода;
- попутно генерируется практически идеальная низкоуровневая документация;
- это приносит приятные эмоции или, по крайней мере, вдохновляет;
- создаваемый набор тестов позволяет уверенно выполнить развертывание;
- проекты становятся менее связанными.

Надеюсь, я смог показать вам, что TDD — стоящая вещь и что перечисленных мной причин достаточно, чтобы вы смогли преодолеть первоначальную реакцию, даже если это было отвращение.

Впрочем, есть еще одна причина, по которой практика TDD приобретает первостепенную важность.

Страх

Программировать непросто. Возможно, это самый трудный из всех навыков, в которых люди пытаются достичь мастерства. Сегодня наша цивилизация зависит от сотен тысяч взаимосвязанных приложений, каждое из которых состоит из сотен тысяч, если не десятков миллионов строк кода. Ни один из созданных людьми аппаратов больше не имеет такого количества подвижных деталей.

Каждое из приложений поддерживается командами разработчиков, которые до смерти боятся что-либо менять. Ирония в том, что все это программное обеспечение написано, чтобы позволить нам легко менять поведение машин.

Но разработчики знают, что каждое изменение сопряжено с риском сбоя, причину которого бывает сложно обнаружить и устранить.

Представьте ужасно запутанный код. Скорее всего, вам не придется напрягаться, чтобы вызвать в воображении такой образ. Большинство из нас сталкивается с подобным каждый день.

Готов поклясться: при виде такого кода вы иногда начинаете испытывать желание привести его в порядок. Но я практически уверен, что следом, как молот Тора, обрушивается мысль: «Я к этому не прикоснусь!» Ведь вы прекрасно знаете: если, прикоснувшись к этому коду, вы что-то сломаете, то он станет вашим *навсегда*.

И вы испытываете страх. Вы боитесь кода, поддержкой которого занимаетесь. Вас пугают последствия в виде возможного сбоя.

В результате код начинает деградировать. Никто не будет приводить его в порядок. Никто не хочет его улучшать. Если какие-то изменения неизбежны, то они делаются способом, не наиболее подходящим для системы, а наиболее безопасным для программиста.

Проект приходит в упадок, код портится, продуктивность команды падает, и все это продолжается, пока производительность не начнет стремиться к нулю.

Вы когда-нибудь сталкивались с сильным замедлением работы системы из-за плохого кода? Теперь вы знаете, откуда он берется. Такой код появляется потому, что ни у кого не хватает смелости его улучшить. Никто не решается привести его в порядок.

Мужество

Представьте набор тестов, которому вы настолько доверяете, что готовы рекомендовать развертывание системы, прошедшей эти тесты. Более того, процесс тестирования занимает считаные секунды. В этом случае будет ли вам страшно даже подумать об аккуратной очистке системы?

Снова представьте на экране своего компьютера ужасно запутанный код. Что помешает вам начать приводить его в порядок? У вас же есть тесты, которые сообщат о любой неисправности в момент ее возникновения.

С помощью этого набора тестов вы можете безопасно очистить код. С помощью этого набора тестов вы можете *безопасно* очистить код. *С помощью этого набора тестов вы можете безопасно очистить код.*

Нет, это не опечатка. Просто я очень хотел, чтобы вы прочувствовали этот момент. С помощью этого набора тестов вы можете безопасно очистить код!

А раз это безопасно, вы, без сомнения, *сделаете* это. Аналогичным образом поступят остальные члены команды. Потому что беспорядок никто не любит.

Правило туриста

При наличии набора тестов, которому вы полностью доверяете, вы можете смело следовать простому правилу:

> *Возвращайте в систему контроля версий код в лучшем состоянии, чем он был, когда вы начали с ним работать.*

Представьте, что таким образом поступают все. Прежде чем вернуть исправленный код в репозиторий, люди делают небольшое доброе дело: немного его чистят.

В результате код каждый раз очищается после того, как кто-то с ним поработал. После внесения каждого исправления он становится лучше, чем был.

Подумайте, как могла бы выглядеть поддержка такой системы? Как поменялись бы оценки времени работы и расписание? Насколько длинными остались бы списки ошибок? Потребовалась бы для этих списков автоматизированная база данных?

Основная причина

Сохранение кода в порядке. Его постоянная очистка. Именно поэтому мы практикуем TDD. Потому что мы хотим гордиться своей работой.

Чтобы при взгляде на код сразу было понятно, что он чистый. Чтобы мы знали, что каждое прикосновение к нему делает его лучше, чем он был. И чтобы, глядя в зеркало перед сном, мы могли улыбнуться, осознавая, что сегодня хорошо поработали.

Четвертый закон

Подробно о рефакторинге я расскажу в следующих главах. А пока только скажу, что рефакторинг — это четвертый закон TDD.

Первые три закона помещают нас в цикл из написания небольшого количества тестового кода, затем — производственного кода, который проходит тест. Это напоминает светофор, каждые несколько секунд меняющий цвет с красного на зеленый.

Но такой вариант цикла приведет к быстрой деградации тестового и рабочего кода. Почему? Потому что, как правило, люди не в состоянии хорошо делать два дела одновременно. Если сосредоточиться на написании теста, то скорее всего, пострадает качество производственного кода, и наоборот. Сосредоточившись на реализации желаемого поведения, мы начинаем уделять меньше внимания реализации задуманной структуры.

Не стоит обманывать себя. Невозможно все делать одинаково хорошо. Сама по себе задача реализовать желаемое поведение кода уже достаточно сложна. И в разы труднее сделать так, чтобы этот код *еще и* имел правильную структуру. Здесь нам на помощь приходит совет Кента Бека:

> *Сначала заставьте его работать, затем перепишите его правильно.*

В результате к трем законам TDD добавляется еще один: рефакторинг. То есть сначала вы пишете небольшое количество тестового кода, затем — небольшой код, проходящий этот тест, а после этого приводите весь написанный код в порядок. Наш светофор начинает выглядеть так, как показано на рис. 2.1.

Рис. 2.1. Новый вид рабочего цикла

Скорее всего, вы уже имеете представление о том, что такое рефакторинг. Как я уже говорил, мы подробно разберем его в следующих главах. А пока позвольте развеять несколько мифов и заблуждений.

- Рефакторинг выполняется постоянно. В каждом цикле TDD происходит очистка кода.
- Рефакторинг не меняет поведение. Он проводится только в случаях, когда выполняются все тесты, причем в процессе рефакторинга тесты продолжают успешно выполняться.

- Рефакторинг *никогда* не фигурирует в расписании или плане. На него не нужно отдельно выделять время или спрашивать разрешения. Вы просто *все время* его делаете.

Рефакторинг имеет смысл воспринимать примерно так же, как мытье рук после туалета. Действие, которое принято выполнять из соображений приличия.

ОСНОВЫ

Создать эффективные примеры TDD в тексте очень сложно, поскольку трудно передать ритм данного процесса. Я попытаюсь сделать это с помощью временных меток и выносок. Но чтобы реально ощутить этот ритм, его нужно пронаблюдать своими глазами.

Поэтому я добавил к каждому из приведенных ниже примеров видео. Пожалуйста, первым делом посмотрите его, а только потом читайте текст с объяснениями. Если у вас нет возможности это сделать, то обращайте особое внимание на временные метки в примерах, чтобы хотя бы таким способом почувствовать ритм.

Простые примеры

Скорее всего, встречая подобные примеры, вы не воспринимали их всерьез, ведь они иллюстрируют слишком маленькие и слишком простые проблемы. Вы можете подумать, что TDD работает для таких вот «детских» задач, но вряд ли подойдет для сложных систем. И серьезно ошибетесь.

Основная цель любого хорошего разработчика программного обеспечения — разбить большие и сложные системы на набор небольших простых задач. Программисты вообще разбивают эти системы на отдельные строки кода. Так что приведенные ниже примеры вполне подходят для иллюстрации TDD *вне зависимости от размеров проекта*.

Я могу это подтвердить. Мне приходилось работать над большими системами, которые были построены с помощью разработки через

тестирование, в связи с чем по собственному опыту могу сказать вам, что ритм и методы TDD не зависят от масштабов проекта. Размер в данном случае не имеет значения.

Вернее, он никак не влияет на процедуры и ритм. А вот на скорость и связность тестов — очень сильно. Но об этом мы будем говорить в следующих главах.

Стек

Видео для просмотра: Stack.

Для доступа к видео зарегистрируйтесь на сайте https://learning.oreilly.com/videos/clean-craftsmanship-disciplines/9780137676385/.

Начнем с очень простой задачи: создание стека целых чисел. В процессе решения обратите внимание на тот факт, что тесты дают ответ на любые вопросы о поведении стека. Таким образом становится видна ценность тестов в качестве документации. Обратите также внимание, что мы немного жульничаем, подставляя для прохождения теста абсолютные значения. В TDD это обычная стратегия, смысл которой я объясню немного позже.

Итак, начнем:

```
// T: 00:00 StackTest.java
package stack;

import org.junit.Test;

public class StackTest {
  @Test
  public void nothing() throws Exception {
  }
}
```

Правильным считается начинать с теста, который ничего не делает. Прохождение такого теста просто показывает, что среда выполнения работает.

Затем возникает вопрос, что тестировать, ведь у нас пока нет кода.

Ответить на него очень просто. Предположим, нам уже известен код, который мы собираемся написать: `public class stack`. Здесь мы вспоминаем первый закон и пишем тест, который будет успешно проходить такой код.

> *Правило 1. Напишите тест для проверки кода, который вы собираетесь написать.*

Это первое из многих правил. Все эти правила, по большому счету, эвристические. Они больше напоминают небольшие советы, которые я буду время от времени давать в процессе рассмотрения примеров.

Правило 1 — не высшая математика. Ничего сложного для понимания в нем нет. Если можно написать строку кода, то можно написать и тест, который будет ее проверять. И ничто не мешает написать его первым. Так и сделаем:

```
// T:00:44 StackTest.java
public class StackTest {
  @Test
  public void canCreateStack() throws Exception {
    MyStack stack = new MyStack();
  }
}
```

Полужирным шрифтом я выделяю изменения или добавления, показывая таким способом фрагменты, из-за которых код не компилируется. Я назвал переменную `MyStack`, так как название `Stack` в языке Java уже зарезервировано и используется в качестве ключевого слова.

Обратите внимание, что во фрагменте кода я дал тесту более описательное название. Теперь в соответствии со вторым законом нужно создать стек, поскольку без этого код `MyStack` компилироваться не будет. При этом нужно придерживаться третьего правила: не писать больше, чем требуется для прохождения теста:

```
// T: 00:54 Stack.java
package stack;

public class MyStack {
}
```

Прошло всего десять секунд, а наш тест уже компилируется и проходит. Причем большая часть из этих секунд ушла на перестановку окон на экране, как показано на рис. 2.2. Это было сделано, чтобы можно было видеть оба файла одновременно. В левом окне происходит тестирование, а справа располагается рабочий код.

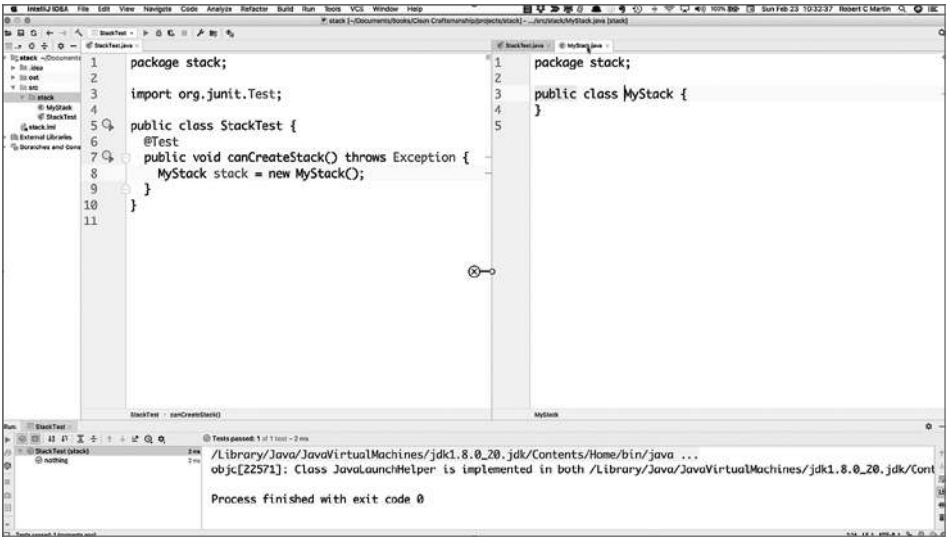


Рис. 2.2. Новый вид экрана

Имя `MyStack` — не самый лучший выбор, но с его помощью мы избежали конфликта имен. Теперь, когда оно объявлено в пакете `stack`, изменим его обратно на `Stack`. У меня это заняло 15 секунд. Тест все еще прекрасно проходит.

```
// T:01:09 StackTest.java
public class StackTest {
  @Test
  public void canCreateStack() throws Exception {
    Stack stack = new Stack();
  }
}

// T: 01:09 Stack.java
```

```
package stack;

public class Stack {
}
```

Здесь мы подошли к следующему правилу: красный → зеленый → рефакторинг. Никогда не упускайте возможность навести порядок.

> *Правило 2. Сделайте так, чтобы тест перестал проходить. Сделайте так, чтобы он снова начал проходить. Очистите код.*

Написать работающий код достаточно сложно. Написать работающий и чистый код еще сложнее. К счастью, выполнение этой задачи можно разбить на два этапа. Сначала пишем работающий код, не обращая внимания на его качество. Затем, благодаря наличию тестов, мы легко можем почистить этот код, сохранив его работоспособность.

То есть на каждом витке цикла TDD мы пользуемся возможностью навести порядок в созданном собственными руками беспорядке.

Вы могли заметить, что наш тест не утверждает никакого поведения. Он компилируется и проходит, но не дает информации о созданном нами стеке. Это можно исправить за 15 секунд:

```
// T: 01:24 StackTest.java
public class StackTest {
  @Test
  public void canCreateStack() throws Exception {
    Stack stack = new Stack();
    assertTrue(stack.isEmpty());
  }
}
```

Здесь вступает в дело второй закон, и нам нужно скомпилировать этот код:

```
// T: 01:49
import static junit.framework.TestCase.assertTrue;

public class StackTest {
  @Test
  public void canCreateStack() throws Exception {
    Stack stack = new Stack();
```

```
    assertTrue(stack.isEmpty());
  }
}

// T: 01:49 Stack.java
public class Stack {
  public boolean isEmpty() {
    return false;
  }
}
```

Двадцать пять секунд спустя тест компилируется, но проваливается. Это сделано преднамеренно. Я специально добавил утверждение `isEmpty`, возвращающее значение `false`, поскольку, согласно первому закону, тестирование должно окончиться неудачей. Зачем это нужно? Чтобы убедиться, что в ситуациях, когда тест не должен проходить, все так и есть. И мы наполовину проверили, как он работает. Проверим вторую половину, изменив утверждение `isEmpty` таким образом, чтобы оно возвращало значение `true`:

```
// T: 01:58 Stack.java
public class Stack {
  public boolean isEmpty() {
    return true;
  }
}
```

Все, теперь тест проходит. Мне потребовалось всего *9 секунд*, чтобы убедиться, что тест функционирует как нужно.

Как правило, когда программисты сначала видят значение `false`, а потом `true`, они смеются, поскольку происходящее напоминает какие-то странные уловки. На самом же деле это всего лишь проверка функционирования теста. Если мы можем убедиться, что там, где он должен, он проходит, а там, где не должен, проваливается, то почему этого *не сделать*?

Что дальше? Мне нужно добавить функцию `push`, поэтому в соответствии с правилом 1 я напишу тест, который будет проверять ее работу:

```
// T 02:24 StackTest.java
@Test
public void canPush() throws Exception {
```

```
  Stack stack = new Stack();
  stack.push(0);
}
```

Тест не компилируется, значит, согласно второму закону нужно написать код, который заставит его компилироваться:

```
// T: 02:31 Stack.java
public void push(int element) {

}
```

Теперь тест компилируется, но в нем отсутствует утверждение. Поэтому нужно добавить предикат, что после однократного применения метода `push` стек перестает быть пустым:

```
// T: 02:54 StackTest.java
@Test
public void canPush() throws Exception {
  Stack stack = new Stack();
  stack.push(0);
  assertFalse(stack.isEmpty());
}
```

Разумеется, такой тест не пройдет, поскольку метод `isEmpty` возвращает значение `true`. Нужна более интеллектуальная проверка, например, добавим для отслеживания пустоты стека логический флаг:

```
// T: 03:46 Stack.java
public class Stack {
  private boolean empty = true;

  public boolean isEmpty() {
    return empty;
  }
  public void push(int element) {
    empty=false;
  }
}
```

Этот тест уже проходит. Отмечу, что с момента прохождения предыдущего теста прошло 2 минуты. В соответствии с правилом 2 пришло время заняться очисткой кода. У нас дублируется код создания стека, поэтому превратим стек в поле класса и инициализируем его:

```
// T: 04:24 StackTest.java
public class StackTest {
  private Stack stack = new Stack();

  @Test
  public void canCreateStack() throws Exception {
    assertTrue(stack.isEmpty());
  }

  @Test
  public void canPush() throws Exception {
    stack.push(0);
    assertFalse(stack.isEmpty());
  }
}
```

Эта операция заняла 30 секунд, и теперь тест благополучно проходит.

Но мне не совсем нравится его имя `canPush`, я предпочитаю его поменять.

```
// T: 04:50 StackTest.java
@Test
public void afterOnePush_isNotEmpty() throws Exception {
  stack.push(0);
  assertFalse(stack.isEmpty());
}
```

Так он выглядит лучше и, конечно же, все еще продолжает проходить.

Теперь в соответствии с первым законом добавим еще одну проверку. Если протолкнуть в стек один элемент и тут же его вытолкнуть, то стек должен опустеть:

```
// T: 05:17 StackTest.java
@Test
public void afterOnePushAndOnePop_isEmpty() throws Exception {
  stack.push(0);
  stack.pop()
}
```

Исправленный код перестал компилироваться, поэтому действуем в соответствии с первым законом:

```
// T: 05:31 Stack.java
public int pop() {
```

```
  return -1;
}
```

Третий закон позволит нам закончить тест:

```
// T: 05:51
@Test
public void afterOnePushAndOnePop_isEmpty() throws Exception {
  stack.push(0);
  stack.pop();
  assertTrue(stack.isEmpty());
}
```

Тест провален, поскольку флаг `empty` так и остается со значением `true`. Исправим эту недоработку:

```
// T: 06:06 Stack.java
public int pop() {
  empty=true;
  return -1;
}
```

Тестирование благополучно завершено. С момента предыдущего теста прошло 76 секунд.

Очистка тут не требуется, поэтому действуем в соответствии со вторым законом. После двух применений метода `push` размер стека должен стать равным 2:

```
// T: 06:48 StackTest.java
@Test
public void afterTwoPushes_sizeIsTwo() throws Exception {
  stack.push(0);
  stack.push(0);
  assertEquals(2, stack.getSize());
}
```

Ошибки компиляции заставляют действовать в соответствии со вторым законом. Но исправить эти ошибки очень легко. Добавим в производственный код инструкцию `import`, а также следующую функцию:

```
// T: 07:23 Stack.java
public int getSize() {
  return 0;
}
```

Теперь все компилируется, но тест не проходит.

Разумеется, для прохождения теста достаточно тривиальной правки:

```
// T: 07:32 Stack.java
public int getSize() {
  return 2;
}
```

Наши действия выглядят несколько нелепо, зато мы убедились, что тест проваливается, когда должен это делать, и проходит, когда все так, как нам нужно, причем процесс проверки занял всего 11 секунд. Так что у нас не было причин от нее отказываться.

Но полученное решение более чем примитивно, поэтому согласно правилу 1 я поищу лучший вариант. Ну да, с первого раза не получилось (можете надо мной посмеяться):

```
// T: 08:06 StackTest.java
@Test
public void afterOnePushAndOnePop_isEmpty() throws Exception {
  stack.push(0);
  stack.pop();
  assertTrue(stack.isEmpty());
  assertEquals(1, stack.getSize());
}
```

Признаю, это было действительно глупо. Но программисты время от времени совершают глупые ошибки, я не исключение. При первом написании примера я не сразу заметил эту ошибку, поскольку ожидал, что тестирование провалится.

Зато сейчас я полностью уверен, что мои тесты хорошо работают, так что можно внести в код изменения, при которых они пройдут успешно:

```
// T: 08:56
public class Stack {
  private boolean empty = true;
  private int size = 0;

  public boolean isEmpty() {
    return size == 0;
  }
```

```
  public void push(int element) {
    size++;
  }

  public int pop() {
    --size;
    return -1;
  }

  public int getSize() {
    return size;
  }
}
```

К моему изумлению, тестирование провалилось и на этот раз. К счастью, я быстро обнаружил ошибку и внес необходимые коррективы:

```
// T: 09:28 StackTest.java
@Test
public void afterOnePushAndOnePop_isEmpty() throws Exception {
  stack.push(0);
  stack.pop();
  assertTrue(stack.isEmpty());
  assertEquals(0, stack.getSize());
}
```

Все тесты проходят благополучно. С момента предыдущего тестирования прошло 3 минуты и 22 секунды.

Для полноты картины я решил добавить еще и проверку размера:

```
// T: 09:51 StackTest.java
@Test
public void afterOnePush_isNotEmpty() throws Exception {
  stack.push(0);
  assertFalse(stack.isEmpty());
  assertEquals(1, stack.getSize());
}
```

Разумеется, тест был пройден.

Вернемся к первому закону. Что должно произойти при опустошении стека? Следует ожидать исключения, информирующего о недостаточном наполнении буфера:

```
// T: 10:27 StackTest.java
@Test(expected = Stack.Underflow.class)
public void poppingEmptyStack_throwsUnderflow() {
}
```

Следуя второму закону, добавим это исключение:

```
// T: 10:36 Stack.java
public class Underflow extends RuntimeException {
}
```

В результате сможем выполнить такой тест:

```
// T: 10:50 StackTest.java
@Test(expected = Stack.Underflow.class)
public void poppingEmptyStack_throwsUnderflow() {
  stack.pop();
}
```

Тест, разумеется, провалится, но это легко исправить:

```
// T: 11:18 Stack.java
public int pop() {
  if (size == 0)
    throw new Underflow();
  --size;
  return -1;
}
```

Тест проходит. С момента предыдущего тестирования прошла 1 минута и 27 секунд.

Снова начнем действовать в соответствии с первым законом. Стек должен помнить, что в него было добавлено. Проверим простейший случай:

```
// T: 11:49 StackTest.java

@Test
public void afterPushingX_willPopX() throws Exception {
  stack.push(99);
  assertEquals(99, stack.pop());
}
```

Тест провален, поскольку метод `pop` в настоящее время возвращает –1. Для прохождения теста сделаем так, чтобы он возвращал 99:

```
// T: 11:57 Stack.java
public int pop() {
  if (size == 0)
    throw new Underflow();
  --size;
  return 99;
}
```

Этого явно недостаточно, поэтому в соответствии с правилом 1 добавим к тесту необходимый минимум кода, который сделает его немного умнее:

```
// T: 12:18 StackTest.java
@Test
public void afterPushingX_willPopX() throws Exception {
  stack.push(99);
  assertEquals(99, stack.pop());
  stack.push(88);
  assertEquals(88, stack.pop());
}
```

Такой тест провалится из-за возвращаемого значения `99`. Чтобы обеспечить его прохождение, добавим поле для записи последнего добавленного в стек значения:

```
// T: 12:50 Stack.java
public class Stack {
  private int size = 0;
  private int element;

  public void push(int element) {
    size++;
    this.element = element;
  }

  public int pop() {
    if (size == 0)
      throw new Underflow();
    --size;
    return element;
  }
}
```

Теперь тест проходит. С момента предыдущего тестирования прошло 92 секунды.

Подозреваю, что к этому моменту я вам изрядно надоел. Возможно, вы мысленно кричите на меня: «Перестань маяться дурью и просто напиши этот проклятый стек!» Но я всего лишь следую правилу 3.

Правило 3. Не гонитесь за золотом.

Любого новичка в TDD посещает огромное искушение первым делом заняться сложными или интересными вещами. Например, в случае написания стека так соблазнительно начать с тестирования поведения FILO (first-in-last-out, «первым пришел — последним вышел»). Именно такой подход называется «погоней за золотом». Я думаю, вы обратили внимание, что я намеренно избегал тестирования чего-либо, напоминающего стек. Я сосредоточился на вспомогательных элементах, таких как пустота и размер.

Почему я не погнался за золотом? Зачем вообще придумали правило 3? Дело в том, что слишком рано погнавшись за золотом, вы, как правило, упускаете множество деталей. Скоро я вам покажу, что вместе с этим вы упускаете и возможность упростить код.

Впрочем, сейчас в соответствии с первым законом нужно написать неработающий тест. И самый очевидный кандидат для тестирования на этом этапе — поведение FILO:

```
// T: 13:36 StackTest.java
@Test
public void afterPushingXandY_willPopYthenX() {
  stack.push(99);
  stack.push(88);
  assertEquals(88, stack.pop());
  assertEquals(99, stack.pop());
}
```

Тест проваливается, так как для его прохождения следует помнить более одного значения. Наверное, в этом случае имеет смысл использовать массив. Поэтому я переименую поле в `elements` и превращу его в массив:

```
// T: 13:51 Stack.java
public class Stack {
  private int size = 0;
  private int[] elements = new int[2];
```

```
  public void push(int element) {
    size++;
    this.elements = element;
  }

  public int pop() {
    if (size == 0)
      throw new Underflow();
    --size;
    return elements;
  }
}
```

По очереди рассмотрим ошибки компиляции этого кода. Переменной `elements` в методе `push` не хватает квадратных скобок:

```
// T: 14:01 Stack.java
public void push(int element) {
  size++;
  this.elements[] = element;
}
```

В эти скобки нужно что-то поместить. Хм-м-м. В предыдущем фрагменте кода есть инкремент переменной `size++`:

```
// T: 14:07 Stack.java
public void push(int element) {
  this.elements[size++] = element;
}
```

Метод `pop` также содержит переменную `elements`, нуждающуюся в квадратных скобках:

```
// T: 14:13
public int pop() {
  if (size == 0)
    throw new Underflow();
  --size;
  return elements[];
}
```

О, смотрите! Сюда можно поместить декремент переменной `--size`:

```
// T: 14:24
public int pop() {
```

```
  if (size == 0)
    throw new Underflow();
  return elements[--size];
}
```

Тесты снова стали проходить. С момента предыдущего тестирования прошло 94 секунды.

На этом все. Разумеется, можно было бы сделать больше. Пока наш стек содержит всего два элемента и не умеет обрабатывать ситуацию переполнения, но я уже продемонстрировал вам все, что хотел. Поэтому оставляю эти усовершенствования вам в качестве упражнения.

Итак, создание с нуля стека целых чисел заняло у меня 14 минут и 24 секунды. Ритм, который вы наблюдали, достаточно типичен. Именно так ощущается разработка через тестирование, независимо от масштаба проекта.

Упражнение

Реализуйте с помощью показанной выше методики очередь целых чисел, обрабатываемых по принципу «первым пришел — первым ушел». Используйте для хранения массив фиксированного размера. Для отслеживания мест добавления и удаления элементов вам, вероятно, понадобятся два указателя. Завершив работу, вы можете обнаружить, что реализовали циклический буфер.

Простые множители

Видео для просмотра: Prime Factors.

Для доступа к видео зарегистрируйтесь на сайте https://learning.oreilly.com/videos/clean-craftsmanship-disciplines/9780137676385/.

Этот пример имеет небольшую предысторию. К 2002 году я уже пару лет использовал TDD и изучал язык Ruby. Мой сын Джастин попросил меня помочь с домашним заданием. Требовалось найти простые множители для набора целых чисел.

Я сказал Джастину, что будет лучше, если он попытается решить задачу самостоятельно. Но пообещал написать программу, которая

проверит его работу. Джастин ушел в свою комнату, а я принялся обдумывать алгоритм нахождения простых множителей.

Самый очевидный подход в данном случае — создать список простых чисел с помощью решета Эратосфена, а затем проверить, подходят ли они в качестве множителей. Я уже собирался писать код, когда мне пришла в голову мысль: а что, если я *просто начну писать тесты и посмотрю, что получится*?

Лучше всего, если вы сможете посмотреть видео, поскольку многие нюансы попросту невозможно описать словами. На этот раз я не буду останавливаться на временных метках и упоминать ошибки компиляции и прочие мелочи. Думаю, вы уже составили мнение об этих аспектах и теперь можно просто показывать постепенный прогресс тестов и кода.

Я начал решение задачи с наиболее очевидного и вырожденного случая. Тем более что именно так предписывало правило 4.

> *Правило 4. Пишите самый простой, самый конкретный, самый вырожденный[1] тест, который не будет пройден.*

В нашем случае — это умножение на 1. И самый вырожденный тест, который не будет пройден, — просто вернуть значение `null`.

```
public class PrimeFactorsTest {
  @Test
  public void factors() throws Exception {
    assertThat(factorsOf(1), is(empty()));
  }

  private List<Integer> factorsOf(int n) {
    return null;
  }
}
```

Обратите внимание, что тестируемую функцию я добавил в тестовый класс. Обычно так не делают, но в данном случае это очень удобно,

[1] Слово «вырожденный» здесь используется для обозначения простейшей отправной точки.

так как мне не придется все время переключаться между двумя исходными файлами.

Мой тест не проходит, но это легко исправить. Достаточно вернуть пустой список:

```
private List<Integer> factorsOf(int n) {
  return new ArrayList<>();
}
```

Все. Теперь тест проходит. Следующий наиболее вырожденный случай: умножение на 2:

```
assertThat(factorsOf(2), contains(2));
```

Тест провален, но ситуацию снова несложно исправить. Именно поэтому на начальном этапе нужно писать тесты для вырожденных случаев: прохождение таких тестов почти всегда легко обеспечить.

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n>1)
    factors.add(2);
  return factors;
}
```

Если вы смотрели видео, то уже знаете, что это делается в два этапа. Сначала я извлек `new ArrayList<>()` в переменную `factors`, а затем добавил оператор `if`.

Я отдельно упоминаю об этом, поскольку первый шаг сделан в соответствии с правилом 5.

Правило 5. По возможности обобщайте.

Исходная константа `new ArrayList<>()` имеет особенности. Ее можно поместить в переменную для последующих манипуляций. Это не очень большое обобщение, но зачастую даже такого вполне достаточно.

Тесты при этом без проблем проходятся. А вот тестирование следующего вырожденного случая дало интересный результат:

```
assertThat(factorsOf(3), contains(3));
```

Тест провалился. Согласно правилу 5, следовало выполнить обобщение. И это несложное обобщение привело к прохождению теста. Скорее всего, вы сейчас удивлены. Присмотритесь к коду, чтобы понять, что произошло.

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n>1)
    factors.add(n);
  return factors;
}
```

Я помню, как изумился факту, что изменение всего одного символа, простое обобщение, привело к тому, что код прошел как новый тест, так и все предыдущие.

В первый момент показалось, что я добился успеха, но следующий шаг меня разочаровал. При всей очевидности следующего теста:

```
assertThat(factorsOf(4), contains(2, 2));
```

я не понимал, как написать его в общей форме. Я смог придумать только проверку делимости `n` на 2, но общим решением это нельзя было назвать. Тем не менее другого у меня не было:

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n>1) {
    if (n%2 == 0) {
      factors.add(2);
      n /= 2;
    }
    factors.add(n);
  }
  return factors;
}
```

Этот код мало того что не универсален, так еще и не проходит предыдущий тест. Он не проходит тест на множитель 2. Надеюсь, вам понятно почему. При уменьшении `n` в 2 раза оно становится равным 1, и это значение помещается в список.

Ситуацию можно исправить с помощью еще менее универсального кода:

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n > 1) {
    if (n % 2 == 0) {
      factors.add(2);
      n /= 2;
    }
    if (n > 1)
      factors.add(n);
  }
  return factors;
}
```

Вы можете справедливо заметить, что я обеспечиваю прохождение тестов, добавляя все новые условия `if`. По большому счету, вы правы. Более того, вы можете обвинить меня еще и в нарушении правила 5, ведь ни один из недавно добавленных фрагментов кода нельзя назвать универсальным. Однако на тот момент я просто не видел других вариантов.

Впрочем, возможность обобщения все-таки есть. Обратите внимание на одинаковые предикаты двух операторов `if`. Как будто перед нами фрагменты развалившегося цикла. Действительно, нет причин, по которым второй оператор `if` должен находиться внутри первого.

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n > 1) {
    if (n % 2 == 0) {
      factors.add(2);
      n /= 2;
    }
  }
  if (n > 1)
    factors.add(n);
  return factors;
}
```

В таком виде тест проходит. Как и следующие три:

```
assertThat(factorsOf(5), contains(5));
assertThat(factorsOf(6), contains(2,3));
assertThat(factorsOf(7), contains(7));
```

Это свидетельствовало, что я на правильном пути, и уменьшило мои переживания из-за уродливых операторов `if`.

Следующий тест для вырожденного случая должен провалиться, поскольку код решения просто не может поместить в список три элемента:

```
assertThat(factorsOf(8), contains(2, 2, 2));
```

Но особым сюрпризом для меня стал способ, которым я добился прохождения теста. В соответствии с правилом 5 я заменил оператор `if` циклом `while`:

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n > 1) {
    while (n % 2 == 0) {
      factors.add(2);
      n /= 2;
    }
  }
  if (n > 1)
    factors.add(n);
  return factors;
}
```

Помню, как сидел перед компьютером и восхищался. Мне казалось, что произошло нечто особенное. В чем дело, я понял позже и сейчас поделюсь с вами. Сработало правило 5. Оказывается, цикл `while` — это общая форма оператора `if`, а оператор `if` — вырожденная форма цикла `while`.

Следующий тест, для множителя 9, также должен завершиться неудачно, поскольку наше решение никак не учитывает тройки:

```
assertThat(factorsOf(9), contains(3, 3));
```

Для прохождения теста следует устранить данную недоработку. Я сделал это следующим образом:

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n > 1) {
```

```
    while (n % 2 == 0) {
      factors.add(2);
      n /= 2;
    }
    while (n % 3 == 0) {
      factors.add(3);
      n /= 3;
    }
  }
  if (n > 1)
    factors.add(n);
  return factors;
}
```

Получилось ужасно. Это не только грубое нарушение правила 5, но и огромное дублирование кода. Даже не знаю, что из этого хуже!

И здесь начинает действовать *принцип обобщения*:

> *По мере того как тесты становятся более конкретными, код становится более универсальным.*

Добавление каждого нового теста делает набор тестов более конкретным. Каждое применение правила 5 делает код решения более общим. Я еще вернусь к этому принципу. Он критически важен для проектирования тестов и для предотвращения их *нестабильности*.

Убрать дублирование и устранить нарушение правила 5 можно, поместив исходный код разложения на множители внутрь цикла:

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  int divisor = 2;
  while (n > 1) {
    while (n % divisor == 0) {
      factors.add(divisor);
      n /= divisor;
    }
    divisor++;
  }
  if (n > 1)
    factors.add(n);
  return factors;
}
```

В видео вы могли заметить, что это делалось в несколько этапов. Первым делом три двойки были извлечены в переменную `divisor`. Следующим шагом стало введение инкремента `divisor++`. Затем я перенес инициализацию переменной `divisor` выше оператора `if`. И наконец, заменил `if` на цикл `while`.

И снова этот переход `if` –> `while`. Заметили, что предикат исходного оператора `if` стал предикатом внешнего цикла `while`? Мне это показалось удивительным. В этом есть что-то от наследования. Как будто существо, которое я попытался создать, постепенно эволюционировало путем череды крошечных мутаций.

Теперь оператор `if` внизу попросту не нужен. Цикл завершается только при `n = 1`. А именно это условие проверял нижний оператор `if` для завершения моего состоящего из двух частей цикла!

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  int divisor = 2;
  while (n > 1) {
    while (n % divisor == 0) {
      factors.add(divisor);
      n /= divisor;
    }
    divisor++;
  }

  return factors;
}
```

После небольшого рефакторинга получим:

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();

  for (int divisor = 2; n > 1; divisor++)
    for (; n % divisor == 0; n /= divisor)
      factors.add(divisor);

  return factors;
}
```

Готово! В видео после этого я добавляю еще один тест, проверяющий достаточность алгоритма.

Помню, как я четко увидел структуру этого алгоритма и задался вопросом: откуда он взялся и как работает?

Очевидно, что породил его я. В конце концов, именно мои пальцы бегали по клавиатуре. Но это был совсем не тот алгоритм, который я хотел использовать изначально. Куда делось решето Эратосфена? Где список простых чисел? Ничего подобного в коде не было!

Хуже того, я не понимал, почему алгоритм работает. Меня поразило, что я могу создать алгоритм, не понимая принципа его действия. Пришлось потратить некоторое время на его изучение, чтобы осознать, что произошло. Меня ставил в тупик инкрементный счетчик `divisor++` внешнего цикла, который гарантировал проверку как множителей всех целых чисел, включая составные! Например, для целого числа 12 проверялось, является ли 4 множителем. Почему в списке отсутствовало значение 4?

Ответ на этот вопрос можно найти, если обратить внимание на порядок выполнения. К моменту, когда счетчик достигал значения 4, из переменной `n` уже удалялись все двойки. И если вдуматься, это все то же решето Эратосфена, просто в другой, необычной форме.

Суть в том, что я вывел этот алгоритм, по очереди тестируя частные случаи. Я не продумывал его заранее. Приступая к работе, я понятия не имел, как он будет выглядеть. Казалось, что он сам собой сгенерировался на моих глазах. Это действительно напоминало эмбрион, шаг за шагом эволюционирующий во все более сложный организм.

Даже сейчас, вглядевшись в код, можно увидеть скромное начало. Это и остатки первого оператора `if`, и фрагменты остальных изменений. Как хлебные крошки, указывающие путь.

Нам осталась волнующая перспектива. Возможно, TDD — универсальный метод постепенного построения алгоритмов. Может быть, правильно упорядоченный набор тестов позволит использовать TDD для пошагового детерминативного написания любой компьютерной программы.

В 1936 году Алан Тьюринг и Алонзо Черч по отдельности доказали, что не существует обобщенной процедуры, определяющей возможность на-

писания программы для произвольной задачи[1]. При этом они изобрели процедурное и функциональное программирование соответственно. У меня сложилось впечатление, что TDD может стать универсальной процедурой получения алгоритмов для проблем, допускающих решение.

Игра в боулинг

В 1999 году мы с Бобом Коссом (Bob Koss) были на конференции по C++. В свободное время нам пришла идея попрактиковать новую на тот момент концепцию TDD. За основу было решено взять простую задачу: подсчет очков при игре в боулинг.

Партия в боулинг состоит из десяти фреймов. В каждом из них игрок может совершить два броска, за каждый из которых начисляются очки по количеству сбитых кеглей. Если игрок сбивает все десять кеглей в первом броске, это называется *страйк*. Если сбивает за два броска, это называется *спэр*. Шар, свалившийся в желоб (рис. 2.3), вообще не приносит очков.

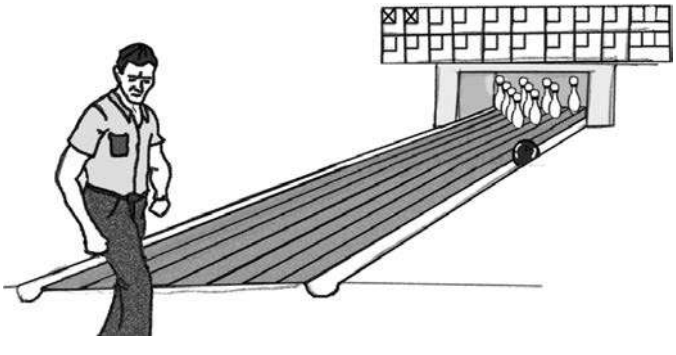


Рис. 2.3. Печально известная ситуация в боулинге

[1] Речь идет о «проблеме разрешимости» Гильберта. Давид Гильберт искал универсальные методы определения разрешимости произвольного диофантова уравнения. Оно представляет собой математическую функцию с целочисленными входами и выходами. Компьютерная программа также представляет собой математическую функцию с целочисленными входами и выходами. Следовательно, проблему Гильберта можно описать в терминах компьютерных программ.

Краткая формулировка правил подсчета очков выглядит так:

- в случае страйка начисляется 10 очков за кегли, сбитые в этом фрейме, плюс количество сбитых кеглей за следующие два броска;
- если сбит спэр, то начисляется 10 очков плюс количество сбитых следующим шаром кеглей;
- в остальных случаях засчитывается количество кеглей, сбитых двумя бросками.

На рис. 2.4 показана типичная таблица подсчета очков.

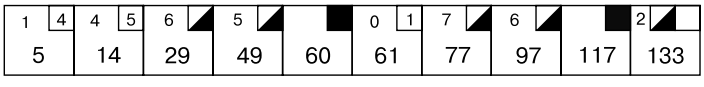


Рис. 2.4. Запись счета типичной игры

С первой попытки игрок сбил одну кеглю, со второй — еще четыре, в сумме набрав 5 очков.

Во втором фрейме он сбил сначала четыре, а потом пять кеглей, что дало ему 9 очков за фрейм и 14 в сумме.

В третьем фрейме он сбил сначала шесть, а затем четыре кегли (спэр). Очки в этом случае нельзя сосчитать, пока игрок не начнет следующий фрейм.

В четвертом фрейме выбито пять кеглей. Теперь можно подсчитать очки для предыдущего фрейма. За него игрок получит 15 очков, а его общий счет станет равен 29.

Выбитый в четвертом фрейме спэр может быть подсчитан только после пятого фрейма, в котором игрок выбивает страйк. В результате за четвертый фрейм он получает 20 очков, а всего 49.

Выбитый в пятом фрейме страйк не может быть засчитан, пока игрок не бросит еще два шара. К сожалению, он выбивает 0 и 1, что приносит ему за пятый фрейм всего 11 очков. Общая сумма при этом достигает 60.

Так продолжается до десятого, последнего фрейма. Здесь выбивается спэр, что дает возможность бросить один дополнительный шар.

Теперь посмотрим на эту информацию с точки зрения объектно-ориентированного программирования. Какие классы и отношения вы бы использовали для вычисления счета игры в боулинг? Сможете нарисовать их средствами UML?[1]

Скорее всего, ваша диаграмма будет похожа на представленную на рис. 2.5.

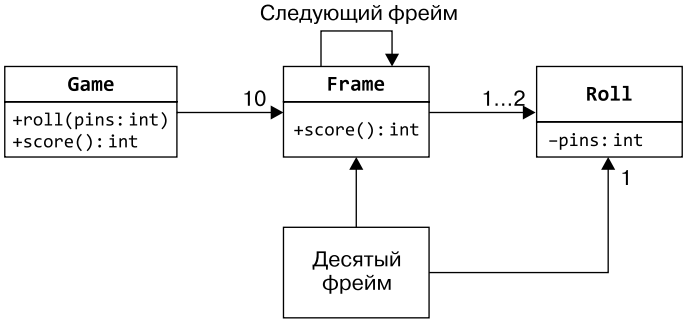


Рис. 2.5. UML-диаграмма подсчета очков в боулинге

Игра (`Game`) состоит из десяти фреймов (`Frames`). В каждом фрейме один или два броска (`Rolls`), за исключением подкласса `TenthFrame`, который наследует `1..2` и добавляет еще один бросок, получая `2..3`. Каждый объект `Frame` указывает на следующий объект `Frame`, так что функция подсчета очков (`score`) в случае спэра или страйка может заглядывать вперед.

В классе `Game` две функции. Функция `roll` вызывается при каждом броске шара, и ей передается количество сбитых игроком кеглей. Функция `score` вызывается после всех бросков и возвращает счет за игру.

---

[1] Унифицированный язык моделирования. Если вы не знакомы с UML, то не волнуйтесь — это просто набор стрелок и прямоугольников.

Хорошая, простая объектно-ориентированная модель, код которой написать несложно. Будь у нас команда из четырех человек, работу можно было бы разделить на четыре класса, а примерно через день встретиться и объединить все эти классы, заставив их работать как целое.

А еще можно воспользоваться TDD. Если вы еще не посмотрели видео, то сделайте это сейчас и переходите к дальнейшему чтению.

Посмотрите видео: Bowling Game.

Для доступа к видео зарегистрируйтесь на сайте https://learning.oreilly.com/videos/clean-craftsmanship-disciplines/9780137676385/.

Начнем, как обычно, с теста, который ничего не делает, чтобы проверить возможность компиляции и выполнения. После этой проверки тест удаляется:

```
public class BowlingTest {
  @Test
  public void nothing() throws Exception {
  }
}
```

Далее добавим утверждение, что можем создать экземпляр класса `Game`:

```
@Test
public void canCreateGame() throws Exception {
  Game g = new Game();

}
```

Заставим этот код компилироваться и дадим нашей IDE указание создать отсутствующий класс, обеспечив прохождение теста:

```
public class Game {
}
```

Теперь попробуем бросить шар:

```
@Test
public void canRoll() throws Exception {
  Game g = new Game();
  g.roll(0);
}
```

Для прохождения этого теста укажем IDE, что нужно создать функцию `roll`, аргумент которой будет носить значимое имя `pins` (кегли):

```
public class Game {
  public void roll(int pins) {
  }

}
```

Подозреваю, вы уже заскучали. Пока что ничего нового не происходит. Но потерпите немного, скоро станет интересно. Тесты уже начали понемногу дублироваться. От дублирующегося кода нужно избавиться, поэтому процедуру создания игры вынесем в функцию `setup`:

```
public class BowlingTest {
  private Game g;

  @Before
  public void setUp() throws Exception {
    g = new Game();
  }
}
```

В результате первый тест перестает работать, поэтому мы его удаляем. Второй тест также становится бесполезным, поскольку в нем отсутствуют утверждения. Его тоже можно удалить. Свою роль эти тесты-ступеньки уже сыграли.

> **Цель некоторых тестов** — *заставить нас создавать классы, функции или другие необходимые структуры. Иногда такие тесты настолько вырождены, что ничего не утверждают или утверждают нечто очень примитивное. Часто они со временем заменяются более полными тестами и могут быть безопасно удалены. Такие тесты называют тестами-ступеньками, так как они напоминают ступеньки лестницы, позволяющие постепенно поднимать сложность до нужного уровня.*

Теперь добавим утверждение, что мы можем подсчитать количество очков. Правда, для этого игра сначала должна быть пройдена полностью. Напомню, что функцию `score` можно вызвать только после броска последнего шара.

В соответствии с правилом 4 запустим самую простую и самую вырожденную игру, какую только можно придумать:

```
@Test
public void gutterGame() throws Exception {
  for (int i=0; i<20; i++)
    g.roll(0);
  assertEquals(0, g.score());
}
```

Обеспечить прохождение такого теста очень легко. Нужно сделать так, чтобы функция `score` вернула значение `0`. Но сначала я заставил ее вернуть `-1` (здесь это не показано), просто чтобы убедиться, что в этом случае тест не проходит:

```
public class Game {
  public void roll(int pins) {
  }

  public int score() {
    return 0;
  }
}
```

Ладно, я пообещал вам, что скоро станет интересно, и этот момент почти настал. Подготовительную работу я практически закончил. Следующий тест — еще один пример следования правилу 4. Еще один самый вырожденный случай, который мне удалось придумать, — по одному очку за каждый бросок. Чтобы его протестировать, просто скопируем код предыдущего теста и заменим 0 на 1:

```
@Test
public void allOnes() throws Exception {
  for (int i=0; i<20; i++)
    g.roll(1);
  assertEquals(20, g.score());
}
```

Результатом стал дублирующийся код. При рефакторинге от него нужно будет избавиться. Но сначала выполним тестирование. Для этого нужно сложить результаты всех бросков:

```
public class Game {
  private int score;
```

```
  public void roll(int pins) {
    score += pins;
  }

  public int score() {
    return score;
  }
}
```

Конечно, это не алгоритм подсчета очков в боулинге. Более того, трудно представить, как превратить то, что я сейчас делаю, в такой алгоритм. Впрочем, время покажет. Пока что займемся рефакторингом.

Дублирование в данном случае можно устранить путем извлечения повторяющегося кода в отдельную функцию `rollMany`. Здесь очень помогает пункт меню IDE *Extract Method*. При этом происходит автоматическое обнаружение и замена экземпляров дублирующегося кода:

```
public class BowlingTest {
  private Game g;

  @Before
  public void setUp() throws Exception {
    g = new Game();
  }

  private void rollMany(int n, int pins) {
    for (int i = 0; i < n; i++) {
      g.roll(pins);
    }
  }
  @Test
  public void gutterGame() throws Exception {
    rollMany(20, 0);
    assertEquals(0, g.score());
  }

  @Test
  public void allOnes() throws Exception {
    rollMany(20, 1);
    assertEquals(20, g.score());
  }
}
```

Перейдем к следующему тесту. Придумать еще следующий вырожденный вариант уже трудно, поэтому я решил рассмотреть ситуацию спэра, сделав ее максимально простой. Всего один спэр, с одним бонусным шаром. Все остальные шары я рассматриваю как попавшие в желоб.

```
@Test
public void oneSpare() throws Exception {
  rollMany(2, 5); // спэр
  g.roll(7);
  rollMany(17, 0);
  assertEquals(24, g.score());
}
```

Проверим мою логику: в каждом фрейме этой игры бросают два шара. Первыми двумя выбивается спэр. Следующий шар — бонусный бросок после спэра, а завершают игру 17 шаров, попавших в желоб.

Счет за первый фрейм равен 17. Это 10 за спэр плюс 7, выпавшие в следующем фрейме. Таким образом, счет за всю игру равен 24, поскольку 7 считается дважды. Проверьте сами и убедитесь.

Разумеется, этот тест провалится. Что нужно сделать для его прохождения? Посмотрим на код:

```
public class Game {
  private int score;

  public void roll(int pins) {
    score += pins;
  }

  public int score() {
    return score;
  }
}
```

Счет вычисляется функцией `roll`, соответственно, нужно ее отредактировать, добавив возможность подсчета очков в случае спэра. В итоге получится очень некрасивый код:

```
public void roll(int pins) {
  if (pins + lastPins == 10) { // ужас!
```

```
    // Бог знает что...
  }
  score += pins;
 }
```

Переменная `lastPins` должна быть полем класса `Game`, запоминающим результат последнего броска. И если два последних броска дают в сумме 10, это спэр. Правильно?

В этот момент вы должны почувствовать, как напрягаются все мышцы, разыгрывается аппетит и начинается головная боль. И от волнения повышается кровяное давление.

Мы попросту идем не туда!

Любой программист бывал в таких ситуациях, не так ли? Вопрос в том, что делать дальше.

Всякий раз, когда у вас возникает чувство, что вы двигаетесь не туда, доверьтесь ему! Первым делом попробуем выяснить, что случилось.

В данном случае речь идет об ошибке проектирования. Вы можете справедливо задаться вопросом: о какой ошибке проектирования может идти речь в случае кода из двух строк? Но она есть, причем вопиющая и очень серьезная. Как только я на нее укажу, вы со мной согласитесь. Но сначала попробуйте найти ее самостоятельно.

Ошибка закралась в самом начале. Если судить *по названиям* наших двух функций класса, то какая из них вычисляет общий счет? Разумеется, функция `score`. А где на самом деле происходит подсчет? В функции `roll`. Это перепутанная ответственность.

> **Перепутанная ответственность:** *ошибка проектирования, при которой название функции указывает на выполнение определенных вычислений, однако на самом деле они выполняются в другом месте.*

Сколько раз вам случалось задействовать функцию, которая, если верить ее названию, выполняла определенную задачу, и обнаружить, что на самом деле это не так? При этом вы понятия не имели, где в системе на самом деле выполняется эта задача. Почему возникают такие ситуации?

Из-за умных программистов. Вернее, программистов, *считающих* себя умными.

Я поступил очень умно, суммировав кегли внутри функции `roll`, не так ли? Мне было известно, что эта функция будет вызываться по разу для каждого броска, а значит, мне оставалось только сложить результаты. Именно поэтому я добавил суммирование прямо в эту функцию. Удивительно умный ход! Зато теперь я естественным образом подошел к правилу 6.

> *Правило 6. Если вам кажется, что с кодом что-то не так, то внесите коррективы в проект и только потом продолжайте работу.*

Как в данном случае исправить ошибку проектирования? Нужно переместить процедуру подсчета очков туда, где она должна находиться. Возможно, в процессе перемещения мы сможем понять, как обеспечить прохождение теста в случае спэра.

Если вычисления из функции `roll` убираются, то нужно сделать так, чтобы она запоминала результаты всех бросков, например, помещая их в массив. Затем функция `score` сможет суммировать его элементы.

```
public class Game {
  private int rolls[] = new int[21];
  private int currentRoll = 0;

  public void roll(int pins) {
    rolls[currentRoll++] = pins;
  }

  public int score() {
    int score = 0;
    for (int i = 0; i < rolls.length; i++) {
      score += rolls[i];
    }
    return score;
  }
}
```

Этот код не проходит тест в ситуации спэра, но проходит в двух других случаях. Более того, причина провала теста осталась той же самой.

Ведь несмотря на то, что мы полностью изменили структуру кода, его поведение осталось прежним. Кстати, именно так звучит *определение* рефакторинга.

> **Рефакторинг** — *изменение структуры кода, никак не затрагивающее поведение этого кода*[1].

Можно ли сейчас пройти тест в случае спэра? Возможно, но выглядеть оно все равно будет не очень:

```
public int score() {
  int score = 0;
  for (int i = 0; i < rolls.length; i++) {
    if (rolls[i] + rolls[i+1] == 10) { // ужасно
      // Что теперь?
    }
    score += rolls[i];
  }
  return score;
}
```

Это правильный вариант? Разумеется, нет. Этот код работает только при четных значениях переменной `i`. Для распознавания спэра оператор `if` должен выглядеть так:

```
if (rolls[i] + rolls[i+1] == 10 && i%2 == 0) { // ужасно
```

Фактически мы вернулись к правилу 6. У нас еще одна ошибка проектирования. Что это может быть?

Вернитесь к UML-диаграмме. В соответствии с ней в классе `Game` должно быть десять экземпляров `Frame`. Целесообразно ли это? Посмотрите на наш цикл. На данный момент он повторяется 21 раз! Имеет ли это хоть какой-то смысл?

Подумаем. Если бы вам впервые показали код для подсчета очков в боулинге, то какое число вы ожидали бы в нем увидеть? 21? Или 10?

---

[1] *Фаулер М.* Рефакторинг: улучшение существующего кода.

Надеюсь, вы ответили 10, поскольку партия в боулинг состоит из десяти фреймов. Где в нашем алгоритме подсчета очков число 10? Его там нет!

Как добавить это значение в алгоритм? Нужно перебирать массив *по одному фрейму за раз*. Как это сделать?

Например, можно в процессе перебора рассматривать по два шара за раз, не так ли? Я имею в виду вот такой код:

```
public int score() {
  int score = 0;
  int i = 0;
  for (int frame = 0; frame<10; frame++) {
    score += rolls[i] + rolls[i+1];
    i += 2;
  }
  return score;
}
```

Этот код тоже проходит первые два теста, но не дополнительный тест. Причина та же, что и раньше. Как видите, поведение не изменилось. Мы провели реальный рефакторинг.

Возможно, вы уже готовы разорвать эту книгу, поскольку знаете, что перебирать массив по два шара просто неправильно. В случае страйка во фрейме будет всего один шар, а в десятом фрейме их может быть три.

Правда, до сих пор ни в одном из тестов не фигурировал ни страйк, ни десятый фрейм. Так что пока можно позволить себе вольность в виде двух шаров на фрейм.

Можно ли сейчас пройти тест для случая спэра? Да. Это очень просто:

```
public int score() {
  int score = 0;
  int i = 0;
  for (int frame = 0; frame < 10; frame++) {
    if (rolls[i] + rolls[i + 1] == 10) { // спэр
      score += 10 + rolls[i + 2];
      i += 2;
    } else {
```

```
      score += rolls[i] + rolls[i + 1];
      i += 2;
    }
  }
  return score;
}
```

Тест пройден. Правда, код выглядит отвратительно. Можно переименовать `i` в `frameIndex` и избавиться от уродливой строки с комментарием, превратив ее в симпатичный маленький метод:

```
public int score() {
  int score = 0;
  int frameIndex = 0;
  for (int frame = 0; frame < 10; frame++) {
    if (isSpare(frameIndex)) {
      score += 10 + rolls[frameIndex + 2];
      frameIndex += 2;
    } else {
      score += rolls[frameIndex] + rolls[frameIndex + 1];
      frameIndex += 2;
    }
  }
  return score;
}

private boolean isSpare(int frameIndex) {
  return rolls[frameIndex] + rolls[frameIndex + 1] == 10;
}
```

Это выглядит лучше. Аналогичным образом можно убрать уродливую строку из тестового кода для ситуации спэра:

```
private void rollSpare() {
  rollMany(2, 5);
}

@Test
public void oneSpare() throws Exception {
  rollSpare();
  g.roll(7);
  rollMany(17, 0);
  assertEquals(24, g.score());
}
```

Замена таких конструкций симпатичными функциями почти всегда является хорошей идеей. Люди, которые позже будут читать ваш код, будут вам благодарны.

Что еще протестировать? Думаю, нужно проверить работу кода в случае страйка:

```
@Test
public void oneStrike() throws Exception {
  g.roll(10); // страйк
  g.roll(2);
  g.roll(3);
  rollMany(16, 0);
  assertEquals(20, g.score());
}
```

Проверим. У нас будет страйк, 2 бонусных шара и 16 шаров, попавших в желоб для заполнения оставшихся восьми фреймов. В первом фрейме счет 15, во втором — 5. В остальных фреймах получаем 0, так что всего будет 20.

Разумеется, тест будет провален. Обеспечим его прохождение:

```
public int score() {
  int score = 0;
  int frameIndex = 0;
  for (int frame = 0; frame < 10; frame++) {
    if (rolls[frameIndex] == 10) { // страйк
      score += 10 + rolls[frameIndex+1] + rolls[frameIndex+2];
      frameIndex++;
    }
    else if (isSpare(frameIndex)) {
      score += 10 + rolls[frameIndex + 2];
      frameIndex += 2;
     } else {
      score += rolls[frameIndex] + rolls[frameIndex + 1];
      frameIndex += 2;
    }
  }
  return score;
}
```

Этот код проходит тест. Обратите внимание, что параметр `frameIndex` увеличивается только на единицу. Ведь в случае страйка во фрейме

бросают только один шар. Помните, вы беспокоились по этому поводу?

Это очень хороший пример того, что происходит при правильном проектировании. Остаток кода начинает как бы сам собой вставать на свои места. Так что обращайте особое внимание на правило 6. Это сэкономит вам огромное количество времени.

Этот код можно немного почистить. Уберем конструкцию с комментарием в метод `isStrike`. Часть математических расчетов также можно поместить в функции со значимыми именами. После этого код примет вот такой вид:

```
public int score() {
  int score = 0;
  int frameIndex = 0;
  for (int frame = 0; frame < 10; frame++) {
    if (isStrike(frameIndex)) {
      score += 10 + strikeBonus(frameIndex);
      frameIndex++;
    } else if (isSpare(frameIndex)) {
      score += 10 + spareBonus(frameIndex);
      frameIndex += 2;
    } else {
      score += twoBallsInFrame(frameIndex);
      frameIndex += 2;
    }
  }
  return score;
}
```

Тестовый код также можно немного привести в порядок, превратив код с комментарием в метод `rollStrike`:

```
@Test
public void oneStrike() throws Exception {
  rollStrike();
  g.roll(2);
  g.roll(3);
  rollMany(16, 0);
  assertEquals(20, g.score());
}
```

Что еще нужно проверить? Мы пока не тестировали десятый фрейм. Но мне уже начинает нравиться этот код. Думаю, можно нарушить правило 3 и *погнаться за золотом*. Проверим идеальную игровую ситуацию!

```
@Test
public void perfectGame() throws Exception {
  rollMany(12, 10);
  assertEquals(300, g.score());
}
```

Представим, что в первых девяти фреймах выбит страйк, а в десятом фрейме — страйк и две десятки. Это принесет нам 300 очков.

Что произойдет при запуске теста, написанного для такой ситуации? Нас ждет неудача, верно? Но нет! Тест пройден! Он пройден, поскольку мы закончили писать код! Функция `score` готова. В этом можно убедиться, прочитав ее. Следите за мной:

```
Для каждого из десяти фреймов
  Если выбит страйк,
    Начислено 10 очков плюс бонусный бросок
      (следующие два шара).
  Если выбит спэр,
    Начислено 10 очков плюс бонусный бросок
      (еще один шар).
  В противном случае
    Считаем кегли, выбитые двумя шарами.
```

Код читается как правила подсчета очков в боулинге. Вернитесь в начало главы и еще раз прочитайте правила. Сравните их с кодом. А затем спросите себя, видели ли вы когда-нибудь, чтобы код настолько соответствовал исходным требованиям?

Скорее всего, вы не можете понять, почему работает код. Ведь в системе учета результатов десятый фрейм не похож на все остальные; но решение не содержит код, обрабатывающий этот особый случай. Как такое могло получиться?

Дело в том, что в десятом фрейме нет ничего особенного. Он по-другому записывается в таблице результатов, но оценивается при

этом так же, как и все остальные. Поэтому для него не требуется отдельный код.

А ведь я собирался сделать для него подкласс!

Еще раз посмотрим на UML-диаграмму. Можно было бы раздать задания трем-четырем программистам и через пару дней объединить результаты их труда. Проблема в том, что такой подход тоже сработал бы. Рабочая группа праздновала бы сдачу 400 строк кода[1], даже не подозревая, что алгоритм состоит из цикла `for` и двух операторов `if` и умещается в 14 строк.

Вы видели такое решение в начале пути? Понимали, что нам хватит цикла `for` и двух операторов `if`? Или вы ожидали, что на одном из этапов тестирования я в конце концов напишу подкласс `Frame` для обработки десятого фрейма, и считали, что именно это представляет основную сложность?

Вы знали о готовности алгоритма до тестирования десятого фрейма? Или думали, что над алгоритмом еще работать и работать? Разве не удивительно, что можно настроиться на долгий процесс, но написав тест, обнаружить, что мы уже закончили?

Мне приходилось слышать, что если бы я следовал исходной UML-диаграмме, то написал бы код, который проще редактировать и поддерживать. Но это же абсурд! Что бы вы предпочли поддерживать: 400 строк кода в четырех классах или 14 строк с одним циклом `for` и двумя операторами `if`?

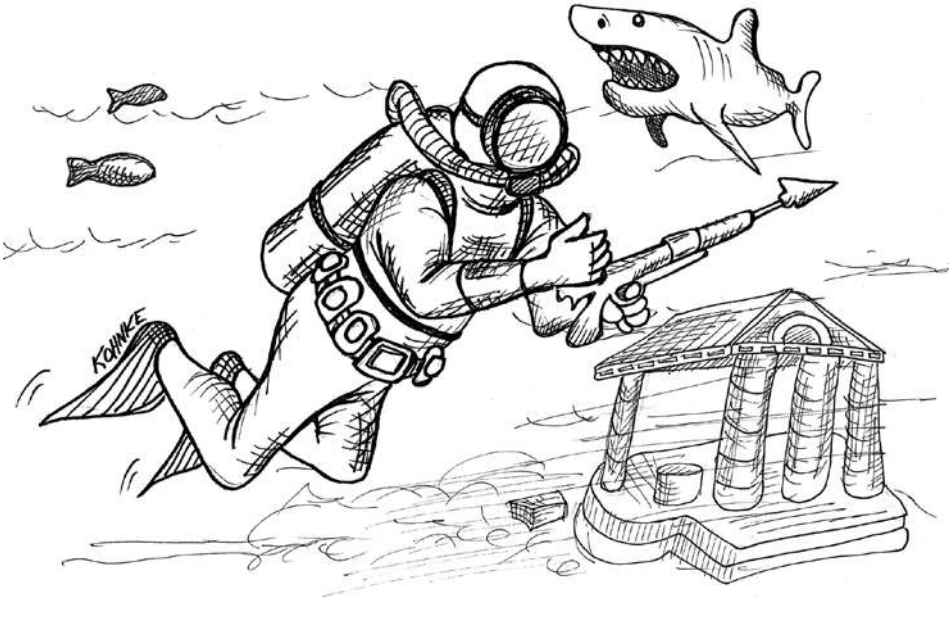
РЕЗЮМЕ

В этой главе я познакомил вас с мотивацией и основами TDD. Возможно, от избытка информации у вас уже голова пошла кругом. Вы узнали очень много. Но недостаточно. Следующая глава значительно углубит ваши знания, так что, прежде чем перевернуть страницу, можно немного отдохнуть.

---

[1] Я знаю об этих 400 строках, поскольку сам писал такую программу.

3 ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ TDD



Пристегните ремни. Вас ждет быстрая поездка по ухабистой дороге. Как говорил доктор Морбиус во время демонстрации устройства креллов: «Подготовьте свой разум к новой шкале научных ценностей».

СОРТИРОВКА 1

Последние два примера предыдущей главы породили интересный вопрос. Откуда берется алгоритм, генерируемый в процессе разработки через тестирование? Понятно, что его порождает наш мозг, но не так, как мы привыкли это делать. Мистическим образом последовательность неудачных тестов извлекает этот алгоритм на свет божий без необходимости продумывать его заранее.

Есть вероятность, что TDD представляет собой пошаговую процедуру поиска алгоритма для решения любой задачи. Это можно сравнить с доказательством математической или геометрической теоремы. Все начинается с базовых постулатов — неудачных тестов для вырожденных случаев. Затем, шаг за шагом, формулируется решение.

С каждым шагом тесты становятся все более ограничивающими и специфичными, а рабочий код, наоборот, — все более универсальным. Процесс продолжается до тех пор, пока производственный код не станет настолько обобщенным, что вы уже не сможете придумать ни одного теста, который будет невозможно пройти. Это означает, что задача решена.

Еще раз посмотрим на этот процесс. На этот раз мы воспользуемся вышеописанным подходом для получения алгоритма сортировки массива целых чисел.

Было бы неплохо, если бы вы сначала посмотрели видео: SORT 1.

Для доступа к видео зарегистрируйтесь на сайте https://learning.oreilly.com/videos/clean-craftsmanship-disciplines/9780137676385/.

Начнем, как всегда, с теста, который ничего не делает:

```
public class SortTest {
  @Test
```

```
  public void nothing() throws Exception {

  }
}
```

А вот и первый тест, который не проходит. Мы тестируем вырожденный случай в виде пустого массива:

```
public class SortTest {

  @Test
  public void sorted() throws Exception {
    assertEquals(asList(), sort(asList()));
  }
  private List<Integer> sort(List<Integer> list) {
    return null;
  }
}
```

Очевидно, что тест не будет пройден, но этот недостаток легко исправить:

```
private List<Integer> sort(List<Integer> list) {
  return new ArrayList<>();
}
```

Поднимемся на уровень выше и рассмотрим тест для списка, содержащего одно целое число:

```
assertEquals(asList(1), sort(asList(1)));
```

Понятно, что тест будет провален. Для его прохождения сделаем рабочий код немного более общим:

```
private List<Integer> sort(List<Integer> list) {
  return list;
}
```

Замечательно, правда? Этот трюк я вам уже показывал в предыдущей главе в примере с простыми множителями. Кажется, это довольно распространенная практика — обеспечивать прохождение первых двух тестов путем возврата наиболее вырожденного ответа, за которым следует входной аргумент.

Следующий случай тривиален: два упорядоченных элемента. Этот код благополучно проходит тест. Вы можете сказать, что не было смысла его писать, но иногда просто приятно посмотреть на удачное тестирование.

```
assertEquals(asList(1, 2), sort(asList(1, 2)));
```

Если же изменить порядок следования элементов, то тест провалится.

```
assertEquals(asList(1, 2), sort(asList(2, 1)));
```

Для прохождения теста достаточно минимального редактирования. Просто поменяем элементы местами:

```
private List<Integer> sort(List<Integer> list) {
  if (list.size() > 1) {
    if (list.get(0) > list.get(1)) {
      int first = list.get(0);
      int second = list.get(1);
      list.set(0, second);
      list.set(1, first);
    }
  }
  return list;

}
```

Возможно, вы уже поняли, к чему я клоню. Если да, то не портите сюрприз всем остальным. Просто запомните этот момент — я вернусь к нему в следующем разделе.

Тесты для следующих двух случаев проходят благополучно. В первом случае входной массив уже упорядочен. Во втором нарушен порядок первых двух элементов, но наше текущее решение меняет их местами.

```
assertEquals(asList(1, 2, 3), sort(asList(1, 2, 3)));
assertEquals(asList(1, 2, 3), sort(asList(2, 1, 3)));
```

Следующий неудачный тест для трех элементов, два из которых не упорядочены:

```
assertEquals(asList(1, 2, 3), sort(asList(2, 3, 1)));
```

Для его прохождения поместим наш алгоритм сравнения и замены в цикл, который идет вниз по списку:

```
private List<Integer> sort(List<Integer> list) {
  if (list.size() > 1) {
    for (int firstIndex=0; firstIndex < list.size()-1; firstIndex++) {
      int secondIndex = firstIndex + 1;
      if (list.get(firstIndex) > list.get(secondIndex)) {
        int first = list.get(firstIndex);
        int second = list.get(secondIndex);
        list.set(firstIndex, second);
        list.set(secondIndex, first);
      }
    }
  }
  return list;
}
```

Можете сказать, к чему все идет? Большинство из вас, вероятно, да. Для следующего провального теста возьмем случай с тремя элементами в обратном порядке:

```
assertEquals(asList(1, 2, 3), sort(asList(3, 2, 1)));
```

Провальный результат говорит сам за себя. Функция `sort` возвращает значение `[2, 1, 3]`. Обратите внимание, что цифра 3 оказалась в конце списка. Это хорошо! Но первые два элемента все равно идут не по порядку. Нетрудно понять, почему так произошло. Сначала тройка поменялась местами с двойкой, а затем — с единицей. Но при этом двойка и единица остались неупорядоченными. Их нужно еще раз поменять местами.

То есть для прохождения этого теста нужно поместить цикл сравнения и перестановки в другой цикл, который постепенно будет уменьшать длину сравниваемого массива. Наверное, это проще понять, посмотрев на код:

```
private List<Integer> sort(List<Integer> list) {
  if (list.size() > 1) {
    for (int limit = list.size() - 1; limit > 0; limit--) {
      for (int firstIndex = 0; firstIndex < limit; firstIndex++) {
        int secondIndex = firstIndex + 1;
        if (list.get(firstIndex) > list.get(secondIndex)) {
          int first = list.get(firstIndex);
```

```
            int second = list.get(secondIndex);
            list.set(firstIndex, second);
            list.set(secondIndex, first);
          }
        }
      }
    }
    return list;
}
```

В качестве завершающего штриха проведем более масштабное тестирование:

```
assertEquals(
            asList(1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9, 9, 9),
            sort(asList(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9,
                        3)));
```

Тест пройден, поэтому наш алгоритм сортировки можно считать готовым.

Откуда он взялся? Мы же не проектировали это заранее. Все произошло само собой путем принятия небольших решений, необходимых для прохождения каждого следующего теста. Вуаля!

Что это за алгоритм? Разумеется, сортировка простыми обменами, или *пузырьковая сортировка*, — один из худших возможных алгоритмов.

Так что, возможно, TDD — хороший способ пошаговой разработки плохих алгоритмов.

СОРТИРОВКА 2

Сделаем еще одну попытку. На этот раз я пойду немного другим путем. Снова рекомендую начать с просмотра видео и после этого продолжить чтение.

Видео: SORT 2.

Для доступа к видео зарегистрируйтесь на сайте https://learning.oreilly.com/videos/clean-craftsmanship-disciplines/9780137676385/.

Начнем, как и раньше, с тестов для самых вырожденных случаев и кода, который их проходит:

```
public class SortTest {
  @Test
  public void testSort() throws Exception {
    assertEquals(asList(), sort(asList()));
    assertEquals(asList(1), sort(asList(1)));
    assertEquals(asList(1, 2), sort(asList(1, 2)));
  }

  private List<Integer> sort(List<Integer> list) {
    return list;
  }
}
```

Опять возьмем два неупорядоченных элемента:

```
assertEquals(asList(1, 2), sort(asList(2, 1)));
```

Но если в первый раз мы сравнивали их и меняли местами непосредственно во входном списке, то теперь результаты сравнения будут записываться в новый список:

```
private List<Integer> sort(List<Integer> list) {
  if (list.size() <= 1)
    return list;
  else {
    int first = list.get(0);
    int second = list.get(1);
    if (first > second)
      return asList(second, first);
    else
      return asList(first, second);
  }
}
```

Здесь желательно остановиться и подумать. Впервые столкнувшись с этим тестом, я беспечно написал решение, считая его единственно возможным. Но ошибся. Как видите, есть и другой путь.

Действительно, время от времени для прохождения теста можно написать более одного варианта кода. Это как развилка на дороге. И нужно понять, каким путем мы пойдем дальше.

Посмотрим, как происходит выбор пути следования.

Очевидно, что дальше нужно протестировать три упорядоченных элемента:

```
assertEquals(asList(1, 2, 3), sort(asList(1, 2, 3)));
```

Но в отличие от предыдущего случая, тест провалится. Причина неудачи в том, что вне зависимости от выбранного пути прохождения код не может вернуть список, содержащий более двух элементов. Но мы очень легко можем обеспечить прохождение вот такого теста:

```
private List<Integer> sort(List<Integer> list) {
  if (list.size() <= 1)
    return list;
  else if (list.size() == 2){
    int first = list.get(0);
    int second = list.get(1);
    if (first > second)
      return asList(second, first);
    else
      return asList(first, second);
  }
  else {
    return list;
  }
}
```

Конечно, все это выглядит просто до примитивности, но следующий тест — для трех элементов, из которых первые два идут не по порядку, — не оставляет от этой простоты камня на камне. Понятно, что этот тест не будет пройден:

```
assertEquals(asList(1, 2, 3), sort(asList(2, 1, 3)));
```

И как же нужно действовать, чтобы его пройти? Для списка из двух элементов существуют два варианта, и в решении они оба были использованы. Но для трех элементов вариантов будет уже *шесть*. Неужели теперь придется рассматривать шесть возможных комбинаций?

Но это противоречит здравому смыслу. Нужен более простой подход. Попробуем воспользоваться законом трихотомии.

Согласно этому закону, между двумя числами A и B возможны только три отношения: A < B, A = B или A > B. Поэтому я произвольно выберу из списка элемент и посмотрю, как он соотносится с другими.

Соответствующий код выглядит так:

```
else {
  int first = list.get(0);
  int middle = list.get(1);
  int last = list.get(2);
  List<Integer> lessers = new ArrayList<>();
  List<Integer> greaters = new ArrayList<>();

  if (first < middle)
    lessers.add(first);
  if (last < middle)
    lessers.add(last);
  if (first > middle)
    greaters.add(first);
  if (last > middle)
    greaters.add(last);

  List<Integer> result = new ArrayList<>();
  result.addAll(lessers);
  result.add(middle);
  result.addAll(greaters);
  return result;
}
```

Не пугайтесь, а просто внимательно посмотрите, что делает этот код.

Сначала три значения извлекаются в три переменные: `first`, `middle` и `last`. Это сделано для удобства, чтобы не засорять код кучей вызовов `list.get(x)`.

Затем создается список для элементов, которые меньше, чем `middle`, и еще один список для элементов, которые больше, чем `middle`. Обратите внимание, что переменная `middle` в нашем списке представлена в единственном экземпляре.

Далее с помощью четырех операторов `if` мы помещаем элементы `first` и `last` в соответствующие списки.

После этого остается создать список `result`, куда мы по очереди поместим значения из списка `lessers`, переменную `middle` и значения из списка `greaters`.

Понимаю, что вам может не нравиться этот код. Мне он тоже не очень нравится. Но он работает. Тест пройден.

Следующие два теста также проходят:

```
assertEquals(asList(1, 2, 3), sort(asList(1, 3, 2)));
assertEquals(asList(1, 2, 3), sort(asList(3, 2, 1)));
```

К этому моменту проверены четыре из шести возможных вариантов для списка из трех уникальных элементов. Но проверка двух оставшихся `[2,3,1]` и `[3,1,2]` ожидаемым образом потерпит неудачу.

А теперь представим, что из-за нетерпения или по недосмотру мы сразу перешли к тестированию списков с четырьмя элементами:

```
assertEquals(asList(1, 2, 3, 4), sort(asList(1, 2, 3, 4)));
```

Разумеется, тест закончится неудачей, поскольку текущее решение предполагает не более трех элементов в списке. Перестанет работать и упрощение в виде переменных `first`, `middle` и `last`. Более того, у вас может появиться вопрос, почему переменная `middle` была выбрана в качестве элемента 1. Почему она не может быть элементом 0?

Что ж, превратим последний тест в комментарий и сделаем переменную `middle` элементом 0:

```
int first = list.get(1);
int middle = list.get(0);
int last = list.get(2);
```

Сюрприз! Тест со списком `[1,3,2]` не проходит. Понимаете почему? Раз переменная `middle` равна 1, то значения 3 и 2 добавляются в список `greaters` в неправильном порядке.

Получается, написанное решение уже умеет сортировать список из двух элементов. А в списке `greaters` именно два элемента; соответственно, чтобы пройти тест, достаточно вызвать для этого списка метод `sort`:

```
List<Integer> result = new ArrayList<>();
result.addAll(lessers);
result.add(middle);
result.addAll(sort(greaters));
return result;
```

Теперь для значений `[1,3,2]` тест проходит, а вот для значений `[3,2,1]` проваливается, поскольку в этом случае неупорядоченным оказывается список `lessers`. Но это легко исправить:

```
List<Integer> result = new ArrayList<>();
result.addAll(sort(lessers));
result.add(middle);
result.addAll(sort(greaters));
return result;
```

Как видите, прежде чем переходить к списку из четырех элементов, стоило протестировать два оставшихся варианта с тремя элементами.

> *Правило 7. Полностью протестируйте текущий, более простой случай и только потом переходите к более сложному.*

Как бы то ни было, теперь нужно пройти тест для списка из четырех элементов. Я в этот момент превратил тест из комментариев обратно в код и увидел, что он провалился (здесь это не показано).

Текущий алгоритм сортировки трехэлементного списка можно обобщить, особенно теперь, когда переменная `middle` стала первым элементом. Для формирования списков `lessers` и `greaters` достаточно применить фильтры:

```
else {
  int middle = list.get(0);
  List<Integer> lessers =
    list.stream().filter(x -> x<middle).collect(toList());
  List<Integer> greaters =
    list.stream().filter(x -> x>middle).collect(toList());
  List<Integer> result = new ArrayList<>();
  result.addAll(sort(lessers));
  result.add(middle);
  result.addAll(sort(greaters));
  return result;
}
```

Неудивительно, что теперь легко проходит и этот тест, и следующие два:

```
assertEquals(asList(1, 2, 3, 4), sort(asList(2, 1, 3, 4)));
assertEquals(asList(1, 2, 3, 4), sort(asList(4, 3, 2, 1)));
```

Теперь неплохо бы подробнее изучить элемент `middle`. Представим, что он не уникален:

```
assertEquals(asList(1, 1, 2, 3), sort(asList(1, 3, 1, 2)));
```

Тест провален. Это означает, что нужно перестать относиться к переменной `middle` как к чему-то особенному:

```
else {
  int middle = list.get(0);
  List<Integer> middles = list.stream().filter(x -> x == middle).
collect(toList());
  List<Integer> lessers = list.stream().filter(x -> x<middle).
collect(toList());
  List<Integer> greaters = list.stream().filter(x -> x>middle).
collect(toList());

  List<Integer> result = new ArrayList<>();
  result.addAll(sort(lessers));
  result.addAll(middles);
  result.addAll(sort(greaters));
  return result;
}
```

Теперь тест проходит. Но вы помните, что располагалось выше `else`? Сейчас я вам покажу:

```
if (list.size() <= 1)
  return list;
else if (list.size() == 2){
  int first = list.get(0);
  int second = list.get(1);
  if (first > second)
    return asList(second, first);
  else
    return asList(first, second);
}
```

А нужно ли нам присваивание `==2`? Нет. После его удаления все тесты по-прежнему проходят.

Хорошо, а как насчет первого оператора `if`? Он все еще нужен? Вообще-то его можно поменять на кое-что получше. Я просто покажу вам окончательный алгоритм:

```
private List<Integer> sort(List<Integer> list) {
  List<Integer> result = new ArrayList<>();

  if (list.size() == 0)
    return result;
  else {
    int middle = list.get(0);
    List<Integer> middles = list.stream().filter(x -> x == middle).
                            collect(toList());
    List<Integer> lessers = list.stream().filter(x -> x < middle).
                            collect(toList());
    List<Integer> greaters = list.stream().filter(x -> x > middle).
                             collect(toList());

    result.addAll(sort(lessers));
    result.addAll(middles);
    result.addAll(sort(greaters));
    return result;
  }
}
```

У этого алгоритма есть название: *быстрая сортировка*. Это один из лучших известных алгоритмов сортировки.

Насколько он лучше? На моем ноутбуке он провел сортировку массива из миллиона случайных целых чисел от нуля до миллиона за 1,5 секунды. Пузырьковая сортировка из предыдущего раздела справится с этой задачей примерно за шесть месяцев.

Этот момент вызывает некоторое беспокойство. Для сортировки списка с двумя неупорядоченными элементами нашлось два решения. Одно привело меня к пузырьковой сортировке, другое — к быстрой сортировке.

Это показывает, насколько важно идентифицировать развилки и выбирать правильный путь. В данном случае один путь дал мне довольно плохой алгоритм, а другой — очень хороший.

Можно ли идентифицировать ветвления и определять, куда лучше идти? Вполне. Но это отдельная тема, которую я раскрою в следующей главе.

МЕРТВАЯ ТОЧКА

Думаю, к этому моменту вы уже просмотрели достаточно видеороликов, чтобы составить представление о ритме TDD. С этого момента видео уже не будет, дальше пойдет только текст.

Новички в TDD часто попадают в затруднительное положение. Они пишут отличный тест, а затем обнаруживают, что его можно пройти, только реализовав алгоритм целиком. Я называю это «достижением мертвой точки».

Для выхода из мертвой точки нужно удалить последний тест и придумать что-то более простое.

> *Правило 8. Если для прохождения текущего теста требуется написать слишком много кода, то удалите этот тест и напишите более простой, который будет легче пройти.*

На занятиях я часто использую упражнение, часто приводящее учеников в мертвую точку. Более половины тех, кто пытается его проделать, застревают и, что интересно, отступать не хотят.

Это старая добрая задача на перенос слов. В сплошной текст нужно вставить знаки переноса строки таким образом, чтобы он поместился в столбец шириной в `N` символов. При малейшей возможности разбивайте на отдельные слова.

Студентам предлагается написать такую функцию:

```
Wrapper.wrap(String s, int w);
```

В качестве входной строки я предлагаю начало Геттисбергской речи Авраама Линкольна:

```
"Four score and seven years ago our fathers brought forth upon this
continent a new nation conceived in liberty and dedicated to the
proposition that all men are created equal"[1]
```

При желаемой ширине 30 символов вывод выглядит вот так:

```
====:====:====:====:====:====:
Four score and seven years ago
Our fathers brought forth upon
This continent a new nation
Conceived in liberty and
Dedicated to the proposition
That all men are created equal
====:====:====:====:====:====:
```

Как написать этот алгоритм путем разработки через тестирование? Попробуем начать с такого теста:

```
public class WrapTest {
  @Test
  public void testWrap() throws Exception {
    assertEquals("Four", wrap("Four", 7));
  }

  private String wrap(String s, int w) {
    return null;
  }
}
```

Сколько законов TDD я нарушил? Можете назвать их? Как бы то ни было, продолжим. Обеспечить прохождение этого теста несложно:

```
private String wrap(String s, int w) {
 return "Four";
}
```

---

[1] Минуло восемьдесят семь лет, как отцы наши основали на этом континенте новую нацию, своим рождением обязанную свободе и посвятившую себя доказательству того, что все люди рождены равными.

Следующий тест кажется довольно очевидным:

```
assertEquals("Four\nscore", wrap("Four score", 7));
```

Код, обеспечивающий его прохождение, тоже довольно очевиден:

```
private String wrap(String s, int w) {
  return s.replace(" ", "\n");
}
```

Я просто заменил все пробелы знаками конца строки. Идеально. Но прежде чем пойдем дальше, мы немного почистим этот код:

```
private void assertWrapped(String s, int width, String expected) {
  assertEquals(expected, wrap(s, width));
}

@Test
public void testWrap() throws Exception {
  assertWrapped("Four", 7, "Four");
  assertWrapped("Four score", 7, "Four\nscore");
}
```

Теперь код выглядит лучше, и можно написать следующий провальный тест. Если просто следовать за текстом Геттисбергской речи, то следующий тест будет выглядеть так:

```
assertWrapped("Four score and seven years ago our", 7,
  "Four\nscore\nand\nseven\nyears\nago our");
```

Он и в самом деле провалится. Его можно даже немного усилить следующим образом:

```
assertWrapped("ago our", 7, "ago our");
```

Осталось понять, как отредактировать код, чтобы он смог пройти этот тест. Похоже, заменять *все* пробелы знаками конца строки *не нужно*. Но в этом случае важно понять, *какие пробелы* подлежат замене. Или может быть, стоит заменить их все, а затем решить, какие превратить обратно в пробелы?

Поразмышляйте на эту тему. Не думаю, что вы найдете легкое решение. А это значит, что мы оказались в мертвой точке. Пройти данный

тест можно только при условии, что мы сразу напишем очень большую часть алгоритма переноса слов.

Нам остается только удалить один или несколько тестов и заменить их более простыми, которые мы сможем проходить постепенно. Попробуем сделать это:

```
@Test
public void testWrap() throws Exception {
  assertWrapped("", 1, "");
}

private String wrap(String s, int w) {
  return "";
}
```

Вот теперь я написал действительно вырожденный тест, не так ли? Я сделал это, следуя правилу, которым пренебрег ранее.

Теперь нужен еще один тест для вырожденного случая. Как насчет вот такого?

```
assertWrapped("x", 1, "x");
```

Это довольно простой тест, прохождение которого можно легко обеспечить:

```
private String wrap(String s, int w) {
  return s;
}
```

Снова тот же шаблон. Я прошел первый тест, возвращая вырожденную константу. А прохождение второго теста я обеспечил, возвращая входные данные. Очень интересно. Какой вырожденный случай протестировать теперь?

```
assertWrapped("xx", 1, "x\nx");
```

Тест провален, поскольку мой код возвращает `"xx"`. Однако обеспечить его прохождение несложно:

```
private String wrap(String s, int w) {
  if (w >= s.length())
```

```
    return s;
  else
    return s.substring(0, w) + "\n" + s.substring(w);

}
```

Все легко получилось. Какой следующий вырожденный случай мы будем тестировать?

```
assertWrapped("xx", 2, "xx");
```

На этот раз тест пройден. Хорошо. Тогда попробуем вот такой тест:

```
assertWrapped("xxx", 1, "x\nx\nx");
```

Тест провален. Здесь напрашивается какой-то цикл. Хотя подождите. Есть более простой способ:

```
private String wrap(String s, int w) {
  if (w >= s.length())
    return s;
  else
    return s.substring(0, w) + "\n" + wrap(s.substring(w), w);
}
```

О рекурсии мы вспоминаем достаточно редко, не так ли? Возможно, имеет смысл делать это чаще.

В тестах уже просматривается некий шаблон, как вы считаете? Там пока нет ни слов, ни даже пробелов. Просто строка из символов `x` и счетчик от 1 до размера этой строки. Следующий тест будет выглядеть так:

```
assertWrapped("xxx", 2, "xx\nx");
```

И мы его пройдем. Как и следующий тест:

```
assertWrapped("xxx", 3, "xxx");
```

Вероятно, дальше рассматривать этот шаблон уже нет смысла. Пришло время добавлять пробелы:

```
assertWrapped("x x", 1, "x\nx");
```

Тест провалится, поскольку код возвращает `"x\n \nx"`. Ситуацию можно исправить, удалив все префиксные пробелы перед рекурсивным вызовом метода `wrap`.

```
return s.substring(0, w) + "\n" + wrap(s.substring(w).trim(), w);
```

Теперь тест проходит. А у нас появился шаблон, в соответствии с которым мы напишем следующий тест:

```
assertWrapped("x x", 2, "x\nx");
```

Тест не проходит из-за пробела, завершающего первую подстроку. От него можно избавиться с помощью еще одного вызова метода `trim`:

```
return s.substring(0, w).trim() + "\n" + wrap(s.substring(w).trim(), w);
```

Тест пройден. Следующий тест, построенный по этому шаблону, также проходит:

```
assertWrapped("x x", 3, "x x");
```

Что дальше? Наверное, можно попробовать вот это:

```
assertWrapped("x x x", 1, "x\nx\nx");
assertWrapped("x x x", 2, "x\nx\nx");
assertWrapped("x x x", 3, "x x\nx");
assertWrapped("x x x", 4, "x x\nx");
assertWrapped("x x x", 5, "x x x");
```

Все тесты без проблем проходят. Вероятно, нет особого смысла смотреть, что получится, если добавить четвертый `x`.

Лучше попробуем сделать вот так:

```
assertWrapped("xx xx", 1, "x\nx\nx\nx");
```

Тест пройден. Как и следующие два теста в последовательности:

```
assertWrapped("xx xx", 2, "xx\nxx");
assertWrapped("xx xx", 3, "xx\nxx");
```

А вот этот тест уже провалится:

```
assertWrapped("xx xx", 4, "xx\nxx");
```

Причина неудачи в том, что код возвращает `"xx x\nx"`. Ведь в нем отсутствует пробел между двумя «словами». Где этот пробел? *Перед* символом `wth`. Для его поиска нам нужно двигаться в обратном от `w` направлении:

```
private String wrap(String s, int w) {
  if (w >= s.length())
    return s;
  else {
    int br = s.lastIndexOf(" ", w);
    if (br == -1)
      br = w;
    return s.substring(0, br).trim() + "\n" +
             wrap(s.substring(br).trim(), w);
  }
}
```

Теперь тест проходит. У меня появилось чувство, что работа закончена. На всякий случай проведем еще несколько тестов:

```
assertWrapped("xx xx", 5, "xx xx");
assertWrapped("xx xx xx", 1, "x\nx\nx\nx\nx");
assertWrapped("xx xx xx", 2, "xx\nxx\nxx");
assertWrapped("xx xx xx", 3, "xx\nxx\nxx");
assertWrapped("xx xx xx", 4, "xx\nxx\nxx");
assertWrapped("xx xx xx", 5, "xx xx\nxx");
assertWrapped("xx xx xx", 6, "xx xx\nxx");
assertWrapped("xx xx xx", 7, "xx xx\nxx");
assertWrapped("xx xx xx", 8, "xx xx xx");
```

Все эти тесты проходят. Думаю, теперь все. Попробуем Геттисбергскую речь с длиной строки 15:

```
Four score and
seven years ago
our fathers
brought forth
upon this
continent a new
nation
conceived in
liberty and
dedicated to
the proposition
that all men
```

```
are created
equal
That looks right.
```

Итак, что показал этот пример? Прежде всего, если вы застряли, то откажитесь от тестов, которые привели вас в мертвую точку, и начните писать более простые тесты. При написании тестов старайтесь применять

> *Правило 9. Пошагово следуйте найденному шаблону, который охватывает тестовое пространство.*

НАСТРОЙ, ДЕЙСТВУЙ, ПРОВЕРЬ

А теперь кардинально сменим тему.

Много лет назад Билл Уэйк (Bill Wake) определил фундаментальную для всех тестов закономерность. Он назвал ее шаблоном 3A, или AAA. Эта аббревиатура расшифровывается как Arrange/Act/Assert (настрой/действуй/проверь).

При написании теста первым делом следует *упорядочить* предназначенные для тестирования данные. Обычно это делается в методе `Setup` или в самом начале тестовой функции. Цель в том, чтобы привести систему в состояние, необходимое для запуска тестирования.

Затем приходит время *действия*. Когда тест делает то, для чего он предназначен.

После этого выполняется *проверка*. Обычно это просмотр результатов действия с целью убедиться, что система оказалась в нужном состоянии.

В качестве простого примера рассмотрим тест для программы, подсчитывающей очки при игре в боулинг, которую мы писали в главе 2:

```
@Test
public void gutterGame() throws Exception {
  rollMany(20, 0);
  assertEquals(0, g.score());
}
```

Процесс настройки в этом тесте заключается в создании класса `Game` в функции `Setup` и функции `rollMany(20, 0)`, которая вычисляет счет при попадании шара в желоб.

Активная часть теста — вызов метода `g.score()`.

Проверяющую часть теста составляет утверждение `assertEquals`.

За два с половиной десятилетия моей практики TDD я ни разу не видел тест, построенный не по этой схеме.

Введение в BDD

В 2003 году практикующий преподаватель TDD Дэн Норт (Dan North) совместно с Крисом Стивенсоном (Chris Stevenson) и Крисом Матцем (Chris Matz) сделали то же открытие, что и Билл Уэйк, но назвали его по-другому: Given-When-Then (GWT).

Это послужило началом новой методологии: *разработки, управляемой поведением* (behavior-driven development, BDD).

Сначала BDD считали улучшенным способом написания тестов. Дэну и другим сторонникам этой методологии новый словарь понравился больше, и его добавили в такие инструменты тестирования, как JBehave и RSpec.

В терминах BDD тест `gutterGame` будет выглядеть так:

```
Если дано (Given), что за игру шар попал в желоб 20 раз,
Когда (When) запрашивается счет игры,
Тогда (Then) этот счет равен нулю.
```

Понятно, что для превращения такой записи в исполняемый тест требуется синтаксический анализ. Его можно выполнить с помощью инструментов JBehave и RSpec. Очевидна и синонимичность тестов TDD и BDD.

Со временем словарь BDD отошел от тестирования в сторону задач спецификации системы. Сторонники этой методологии поняли, что даже когда постулаты GWT не преобразуются в тесты, они сохраняют ценность как спецификации поведения.

В 2013 году Лиз Кио (Liz Keogh) сказала о BDD:

> *Это использование примеров для того, чтобы объяснить, как ведет себя приложение... И разговоры об этих примерах.*

Тем не менее полностью отделить BDD от тестирования очень трудно, хотя бы из-за синонимичности словарей GWT и AAA. Если у вас есть какие-либо сомнения по этому поводу, то смотрите:

- если *дано*, что (Given) тестовые данные были *упорядочены* (Arranged);
- *когда* (When) выполняется протестированное *действие* (Act);
- *затем* (Then) *доказывается* (Asserted) ожидаемый результат.

Конечные автоматы

Причина, по которой я уделил такое внимание синонимичности GWT и AAA, заключается в том, что есть еще одна известная структура из трех элементов, которая часто встречается в программном обеспечении: переход между состояниями конечного автомата.

В качестве примера рассмотрим диаграмму состояний/переходов турникета в метро (рис. 3.1).

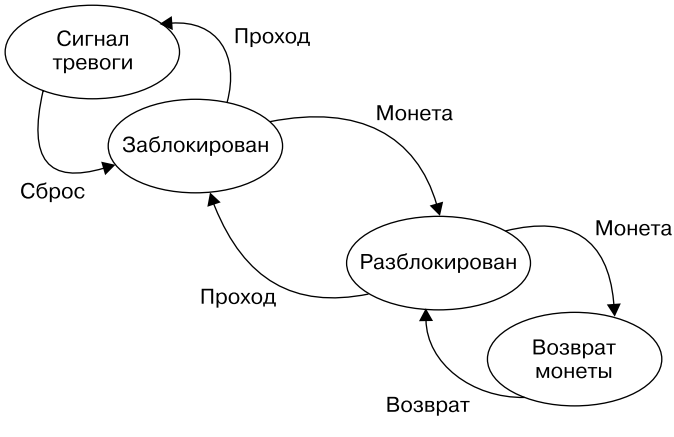


Рис. 3.1. Диаграмма состояний турникета метро

Турникет начинает работу в заблокированном состоянии. Опущенная монета переведет его в разблокированное состояние. После прохода через него турникет возвращается в заблокированное состояние. В случае прохода без оплаты турникет срабатывает. Если кто-то опускает две монеты, то лишняя монета возвращается.

Эту диаграмму можно превратить вот в такую таблицу переходов состояний (табл. 3.1).

Таблица 3.1. Переходы состояний

Текущее состояние	Событие	Следующее состояние
Заблокирован	Монета	Разблокирован
Заблокирован	Проход	Сигнал тревоги
Разблокирован	Монета	Возврат монеты
Разблокирован	Проход	Заблокирован
Возврат монеты	Монета возвращена	Разблокирован
Сигнал тревоги	Сброс	Заблокирован

Каждая строка таблицы представляет собой переход из текущего состояния в следующее, вызванный каким-то событием. Причем это структура из трех элементов, аналогичная GWT или AAA. Сейчас я вам покажу, что у каждой такой структуры есть синоним в соответствующей тройке GWT или AAA:

```
Если дано (Given), что турникет заблокирован (Locked)
Когда (When) происходит событие опускание монеты (Coin)
Тогда (Then) он переходит в состояние "разблокирован" (Unlocked).
```

Получается, что *каждый написанный тест представляет собой переход между состояниями конечного автомата, описывающего поведение системы*.

Повторите это про себя несколько раз. Каждый тест — это переход между состояниями конечного автомата, который вы пытаетесь создать в своей программе.

Знали ли вы, что программа, которую вы пишете, является конечным автоматом? Это действительно так. Каждая программа представляет собой конечный автомат. Поскольку компьютеры — не что иное, как обработчики состояний конечного автомата. Сам компьютер с каждой выполняемой инструкцией переходит из одного конечного состояния в другое.

Соответственно, тесты, которые вы пишете в процессе TDD, и поведения, описываемые средствами BDD, являются всего лишь переходами конечного автомата, который вы пытаетесь создать. А готовый набор тестов и есть этот *конечный автомат*.

Возникает очевидный вопрос: как убедиться, что все переходы, которые по вашему замыслу должен обрабатывать этот конечный автомат, закодированы в виде тестов? Как гарантировать, что описываемый тестами конечный автомат является тем самым, который должна реализовывать ваша программа?

Что может быть лучше, чем сначала сформулировать все переходы в виде тестов, а затем написать реализующий их производственный код?

И снова про BDD

Не кажется ли вам удивительным и даже немного забавным тот факт, что сторонники BDD, возможно, сами того не осознавая, пришли к выводу, что лучший способ описать поведение системы — это описать ее в виде конечного автомата?

ТЕСТОВЫЕ ДВОЙНИКИ

В 2000 году Стив Фриман (Steve Freeman), Тим Маккиннон (Tim McKinnon) и Филип Крейг (Philip Craig) опубликовали статью[1] под

[1] Статья Фримана, Маккиннона и Крейга Endo-Testing: Unit Testing with Mock Objects была представлена на конференции XP2000. Она доступна по адресу http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.3214&rep=rep1&type=pdf.

названием «Эндоскопическое тестирование: модульное тестирование с подставными объектами». О том, как эта статья повлияла на сообщество разработчиков программного обеспечения, свидетельствует распространенность придуманного ими термина: *подставной объект* (mock). Появился даже соответствующий глагол. В настоящее время мы используем *фиктивные* фреймворки для *имитации* (mock) различных вещей.

В те годы идея TDD только начинала проникать в сообщество программистов. Большинство из нас никогда не применяло для тестирования кода такую вещь, как объектно-ориентированное проектирование. Большинство из нас никогда не применяло для тестирования кода какой-либо дизайн. И это вызывало всевозможные проблемы.

Нет, мы умели тестировать простые вещи, такие как примеры из предыдущих глав. Но существовали и задачи, процесс тестирования которых было невозможно представить. Например, как протестировать код, реагирующий на сбой ввода/вывода? Мы же не можем заставить устройство ввода-вывода дать сбой во время модульного теста. Или как протестировать код, взаимодействующий с внешним сервисом? Потребуется ли для этого подключать внешний сервис? И как выполнить тестирование кода, который обрабатывает сбои этого сервиса?

Первые приверженцы TDD писали программы на языке Smalltalk. Для них слово «объект» означало нечто из физического мира. Наверняка они использовали подставные объекты, но скорее всего, совершенно не задумывались на эту тему. Более того, когда в 1999 году я представил идею подставного объекта в языке Java одному эксперту по языку Smalltalker и по TDD, он ответил: «Чрезмерно громоздкий механизм».

Тем не менее метод прижился и стал базисным элементом в TDD.

Но прежде чем я начну подробный рассказ на эту тему, нужно определиться с терминологией. Почти все мы неправильно используем термин *mock object*, по крайней мере в формальном смысле. Современные тестовые двойники сильно отличаются от тех, о которых шла речь в статье 2000 года об эндоскопическом тестировании. Разница

уже настолько значительна, что для отдельных значений был принят другой словарь.

В 2007 году вышла книга Джерарда Мессароша (Gerard Meszaros) *xUnit Test Patterns: Refactoring Test Code*[1]. В ней он определил формальную лексику, которой мы пользуемся в настоящее время. Неофициально мы все еще применяем словосочетание mock objects для обозначения любых тестовых двойников, но когда требуется точность, используем словарь Мессароша.

Мессарош выделил пять типов объектов, подпадающих в категорию mock objects. Это фиктивные объекты: пустышки (dummies), заглушки (stubs), шпионы (spies), подставные объекты (mocks) и имитации (fakes). Мессарош назвал их все *тестовыми двойниками* (test doubles).

И это действительно очень хорошее название. При съемках опасных сцен актера заменяют каскадером, а если нужно снять, например, крупный план рук, выполняющих какие-то действия, которые не умеет выполнять актер, то в кадре окажутся руки дублера. Или дублеры тела, которых снимают в отдельных кадрах со спины или с правильного ракурса. Именно такую функцию выполняют тестовые двойники. Они заменяют другие объекты в процессе тестирования.

Существует своего рода иерархия тестовых двойников (рис. 3.2). Самые простые — пустышки. Заглушки — это пустышки, шпионы — это заглушки, а подставные объекты — это шпионы. Имитации выделяются в отдельный вид.

Механизм, который используют все тестовые двойники (и который мой приятель-эксперт по языку Smalltalker счел «чрезмерным»), — это всего лишь полиморфизм. Например, для тестирования кода управления внешней службой нужно изолировать эту службу с помощью интерфейса, допускающего разные реализации. После чего остается создать реализацию, которая выступит вместо этой службы. Именно она называется тестовым двойником.

Но, пожалуй, проще всего объяснить все это путем демонстрации.

---

[1] *Мессарош Дж.* Шаблоны тестирования xUnit. Рефакторинг кода тестов.

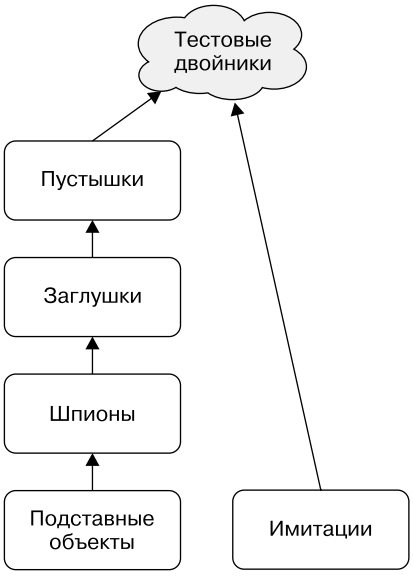


Рис. 3.2. Тестовые двойники

Пустышка

Создание тестового двойника обычно начинается с интерфейса — абстрактного класса без реализованных методов. Например, вот такого:

```
public interface Authenticator {
  public Boolean authenticate(String username, String password);
}
```

Цель этого интерфейса — предоставить приложению способ аутентификации через имена пользователей и пароли. Для подтвержденных пользователей функция `authenticate` возвращает значение `true`, для неподтвержденных — значение `false`.

Предположим, мы хотим проверить, можно ли щелчком на значке закрытия убрать окно для ввода пользовательских данных до того, как пользователь введет туда свое имя и пароль. Тест в этом случае может выглядеть так:

```
@Test
public void whenClosed_loginIsCancelled() throws Exception {
  Authenticator authenticator = new ???;
  LoginDialog dialog = new LoginDialog(authenticator);
  dialog.show();
  boolean success = dialog.sendEvent(Event.CLOSE);
  assertTrue(success);
}
```

Обратите внимание, что класс `LoginDialog` необходимо создавать с помощью интерфейса `Authenticator`. Но в тесте обращений к этому интерфейсу нет, соответственно, непонятно, что мы должны передать в `LoginDialog`.

Создание реального интерфейса `RealAuthenticator` — затратная операция, так как в его конструктор нужно передавать экземпляр класса `DatabaseConnection`. И скорее всего, конструктор этого класса потребует реальных пользовательских данных для полей `databaseUser` и `databaseAuthCode`. (Уверен, что вы уже сталкивались с подобными ситуациями.)

```
public class RealAuthenticator implements Authenticator {
  public RealAuthenticator(DatabaseConnection connection) {
    //...
  }

  //...

}

public class DatabaseConnection {
  public DatabaseConnection(UID databaseUser, UID databaseAuthCode) {
    //...
  }
}
```

Чтобы воспользоваться в тесте интерфейсом `RealAuthenticator`, придется сделать нечто ужасное:

```
@Test
public void whenClosed_loginIsCancelled() throws Exception {
  UID dbUser = SecretCodes.databaseUserUID;
  UID dbAuth = SecretCodes.databaseAuthCode;
  DatabaseConnection connection = new DatabaseConnection(dbUser,
```

```
dbAuth);
  Authenticator authenticator = new RealAuthenticator(connection);
  LoginDialog dialog = new LoginDialog(authenticator);
  dialog.show();
  boolean success = dialog.sendEvent(Event.CLOSE);
  assertTrue(success);
}
```

Этот огромный код добавляется в наш тест только для создания интерфейса `Authenticator`, который нужен лишь на стадии тестирования и потом никогда не будет использоваться. Вдобавок в тесте появляются две ненужные зависимости, которые могут вызвать сбой или на стадии компиляции, или во время загрузки. Ну и зачем нам эта головная боль?

> *Правило 10. Не включайте в тесты вещи, которые там не требуются.*

Вместо этого воспользуемся объектом-пустышкой (рис. 3.3).

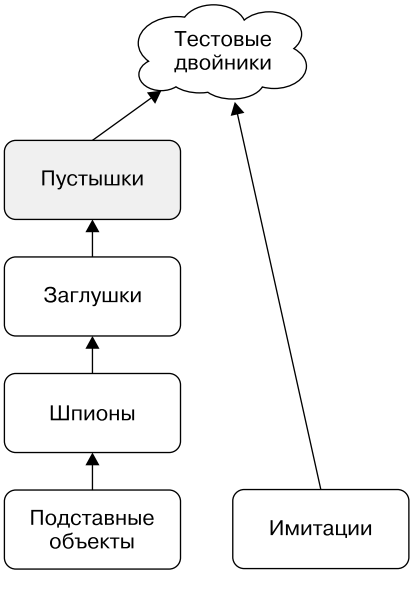


Рис. 3.3. Фиктивный объект

Пустышка — это реализация, которая *ничего* не делает. Каждый метод интерфейса-пустышки реализован так, чтобы не выполнять *никаких действий*. Если метод должен возвращать значение, то значение, возвращаемое пустышкой, должно быть как можно ближе к `null`, или нулю.

В нашем примере интерфейс `AuthenticatorDummy` будет выглядеть так:

```
public class AuthenticatorDummy implements Authenticator {
  public Boolean authenticate(String username, String password) {
    return null;
  }
}
```

Если быть точным, то это именно та реализация, которую создает моя IDE по команде `Implement Interface`.

Теперь тест можно написать без лишнего кода и ненужных зависимостей:

```
@Test
public void whenClosed_loginIsCancelled() throws Exception {
  Authenticator authenticator = new AuthenticatorDummy();
  LoginDialog dialog = new LoginDialog(authenticator);
  dialog.show();
  boolean success = dialog.sendEvent(Event.CLOSE);
  assertTrue(success);
}
```

Итак, пустышка — это тестовый двойник, реализующий интерфейс, не выполняющий никаких действий. Он применяется, когда тестируемая функция принимает в качестве аргумента объект, но логика теста не требует наличия этого объекта.

Я использую пустышки не очень часто по двум причинам. Во-первых, мне не нравится, когда в коде не фигурируют аргументы имеющейся функции. Во-вторых, я не люблю объекты с цепочками зависимостей, таких как `LoginDialog->Authenticator->DatabaseConnection->UID`. В будущем подобные цепочки всегда становятся источниками проблем.

Разумеется, бывают случаи, когда эти проблемы неизбежны. Тогда я предпочитаю использовать пустышку, вместо того чтобы бороться со сложными объектами из приложения.

Заглушка

Как мы видим на рис. 3.4, по сути *заглушка* (stub) — это пустышка; она тоже реализуется таким образом, чтобы не выполнять никаких действий. Но в отличие от пустышки, возвращает не ноль, или `null`, а те значения, которые тестируемая функция должна возвращать при разных сценариях.

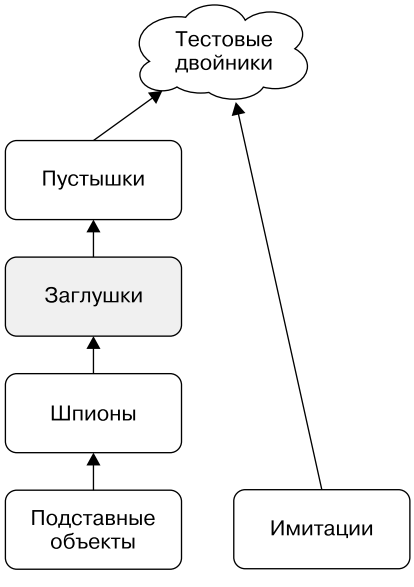


Рис. 3.4. Заглушка

Рассмотрим тест, который проверяет, заканчивается ли неудачей попытка входа, когда интерфейс `Authenticator` отклоняет параметры `username` и `password`:

```
public void whenAuthorizerRejects_loginFails() throws Exception {
  Authenticator authenticator = new ?;
  LoginDialog dialog = new LoginDialog(authenticator);
  dialog.show();
  boolean success = dialog.submit("bad username", "bad password");
  assertFalse(success);
}
```

Если бы здесь использовался интерфейс `RealAuthenticator`, то появилась бы проблема его инициализации со всеми этими неприятными `DatabaseConnection` и `UID`. И это была бы не единственная наша проблема. Непонятно, какое имя пользователя и пароль тут можно указать.

Знай мы содержимое базы данных пользователей, можно было бы выбрать значения переменных `username` и `password`. Другое дело, что так поступать ни в коем случае не следует из-за формирования зависимости между данными тестов и производственными данными. В результате любое изменение производственных данных выводит тест из строя.

Правило 11. Не используйте в тестах производственные данные.

Вместо этого воспользуемся заглушкой. Для этого теста нам понадобится интерфейс `RejectingAuthenticator`, который будет возвращать из метода `authorize` значение `false`:

```
public class RejectingAuthenticator implements Authenticator {
  public Boolean authenticate(String username, String password) {
    return false;
  }
}
```

Просто добавим эту заглушку в наш тест:

```
public void whenAuthorizerRejects_loginFails() throws Exception {
  Authenticator authenticator = new RejectingAuthenticator();
  LoginDialog dialog = new LoginDialog(authenticator);
  dialog.show();
  boolean success = dialog.submit("bad username", "bad password");
  assertFalse(success);
}
```

Мы ожидаем, что метод `submit` объекта `LoginDialog` вызовет функцию `authorize`, которая вернет значение `false`.

Для проверки успешности входа в систему в случае, когда интерфейс принимает имя пользователя и пароль, в эту игру можно сыграть с другой заглушкой:

```
public class PromiscuousAuthenticator implements Authenticator {
  public Boolean authenticate(String username, String password) {
    return true;
  }
}
@Test
public void whenAuthorizerAccepts_loginSucceeds() throws Exception {
  Authenticator authenticator = new PromiscuousAuthenticator();
  LoginDialog dialog = new LoginDialog(authenticator);
  dialog.show();
  boolean success = dialog.submit("good username", "good password");
  assertTrue(success);
}
```

Итак, заглушка — это пустышка, возвращающая определенные значения, необходимые для проверки различных сценариев прохождения кода.

Шпион

Шпион (рис. 3.5) — это заглушка. Он тоже возвращает предопределенные результаты вызовов. При этом он еще и помнит, что с ним было сделано, и позволяет это тестировать.

Проще всего это можно объяснить на примере:

```
public class AuthenticatorSpy implements Authenticator {
  private int count = 0;
  private boolean result = false;
  private String lastUsername = "";
  private String lastPassword = "";

  public Boolean authenticate(String username, String password) {
    count++;
    lastPassword = password;
    lastUsername = username;
    return result;
  }

  public void setResult(boolean result) {this.result = result;}
  public int getCount() {return count;}
  public String getLastUsername() {return lastUsername;}
  public String getLastPassword() {return lastPassword;}
}
```

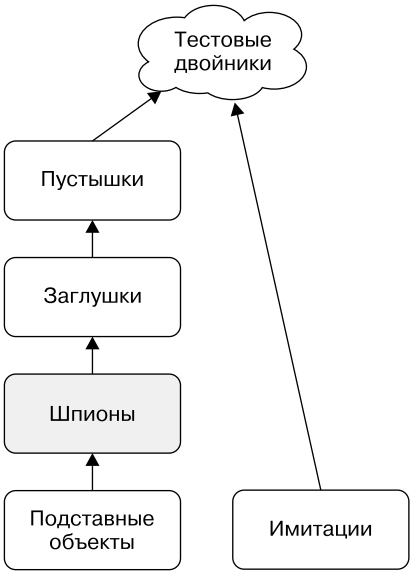


Рис. 3.5. Шпион

Обратите внимание, что метод `authenticate` отслеживает количество своих вызовов, а также последние значения параметров `username` и `password` и предоставляет методы доступа к ним. Именно такое поведение делает этот класс шпионом.

Важно и то, что метод `authenticate` возвращает переменную `result`, значение которой может быть задано методом `setResult`. Фактически наш шпион представляет собой программируемую заглушку.

Ниже представлен тест, в котором мы можем его использовать:

```
@Test
public void loginDialog_correctlyInvokesAuthenticator() throws
Exception {
  AuthenticatorSpy spy = new AuthenticatorSpy();
  LoginDialog dialog = new LoginDialog(spy);
  spy.setResult(true);
  dialog.show();
  boolean success = dialog.submit("user", "pw");
  assertTrue(success);
```

```
  assertEquals(1, spy.getCount());
  assertEquals("user", spy.getLastUsername());
  assertEquals("pw", spy.getLastPassword());
}
```

Название теста говорит о том, что он отслеживает корректность вызова интерфейса `Authenticator` классом `LoginDialog`. Он удостоверяется в том, что метод `authenticate` вызывается всего один раз, и использует аргументы, которые были переданы в метод `submit`.

Шпионом может служить даже простая логическая переменная, значение которой устанавливается при вызове определенного метода. Или же это может быть относительно сложный объект, сохраняющий историю каждого вызова и всех переданных при этом вызове аргументов.

Шпионы отлично позволяют убедиться в корректности работы тестируемого алгоритма. Но они небезопасны, так как связывают тесты с *реализацией* тестируемой функции. Позже я еще расскажу об этом.

Подставной объект

Наконец, очередь дошла до настоящих mock-объектов (рис. 3.6). Именно их Маккиннон, Фриман и Крейг описывали в статье, посвященной эндоскопическому тестированию.

Подставной объект (mock) — это шпион. Он возвращает предопределенные для каждого сценария тестирования значения и запоминает, что происходило в процессе тестирования. И в дополнение к этому подставной объект осведомлен о том, что от него ожидается, и в зависимости от этого обеспечивает прохождение или провал теста.

Другими словами, в него записываются все тестовые утверждения.

Впрочем, лучше один раз увидеть на практике, поэтому создадим класс `AuthenticatorMock`:

```
public class AuthenticatorMock extends AuthenticatorSpy{
  private String expectedUsername;
  private String expectedPassword;
  private int expectedCount;
```

```
  public AuthenticatorMock(String username, String password,
                           int count) {
    expectedUsername = username;
    expectedPassword = password;
    expectedCount = count;
  }

  public boolean validate() {
    return getCount() == expectedCount &&
      getLastPassword().equals(expectedPassword) &&
      getLastPassword().equals(expectedUsername);
  }
}
```

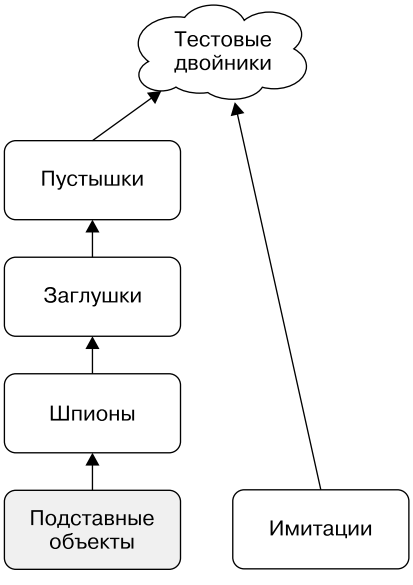


Рис. 3.6. Подставной объект

Как видите, у подставного объекта три ожидаемых поля (их имена начинаются с expected), которые устанавливаются конструктором. Соответственно, это программируемый подставной объект. Обратите также внимание, что класс `AuthenticatorMock` является производным

от класса `AuthenticatorSpy`. В подставном объекте повторно используется код шпиона.

Финальное сравнение выполняет функция `validate`. Если собранные шпионом параметры `count`, `lastPassword` и `lastUsername` соответствуют указанным в подставном объекте ожиданиям, то данная функция возвращает значение `true`.

Теперь вы легко сможете понять смысл теста, в котором используется этот подставной объект:

```
@Test
public void loginDialogCallToAuthenticator_validated() throws
Exception {
  AuthenticatorMock mock = new AuthenticatorMock("Bob", "xyzzy", 1);
  LoginDialog dialog = new LoginDialog(mock);
  mock.setResult(true);
  dialog.show();
  boolean success = dialog.submit("Bob", "xyzzy");
  assertTrue(success);
  assertTrue(mock.validate());
}
```

В подставном объекте мы указали свои ожидания. Это имя пользователя `"Bob"`, пароль `"xyzzy"`, а количество вызовов метода `authenticate` — 1.

Затем создается `LoginDialog` с подставным объектом, который представляет собой интерфейс `Authenticator`. Подставной объект программируется на прохождение теста. После чего мы отображаем окно диалога и отправляем запрос на вход с данными `"Bob"` и `"xyzzy"`. Удостоверившись в успешном входе в систему, мы утверждаем, что заданные в подставном объекте ожидания оправдались.

Вот так все работает. При этом подставные объекты могут быть очень сложными. Например, представьте, что мы ожидаем три вызова функции `f` с тремя разными наборами аргументов и возврат трех разных значений. А функция `g` при этом должна вызываться один раз между первыми двумя вызовами `f`. Осмелитесь написать такой подставной объект без модульных тестов?

Я не очень люблю подставные объекты. Они связывают поведение шпиона с тестовыми утверждениями, и мне это не нравится. Я считаю, что тест должен быть непосредственно связан с утверждениями, не перенося утверждения на какой-то другой, более глубокий механизм. Но это только мое мнение.

Имитация

Пришло время разобраться с последним из тестовых двойников — *имитацией* (fake) (рис. 3.7). Это не пустышка, не заглушка, не шпион и не подставной объект. Это совершенно другая разновидность тестового двойника — симулятор.

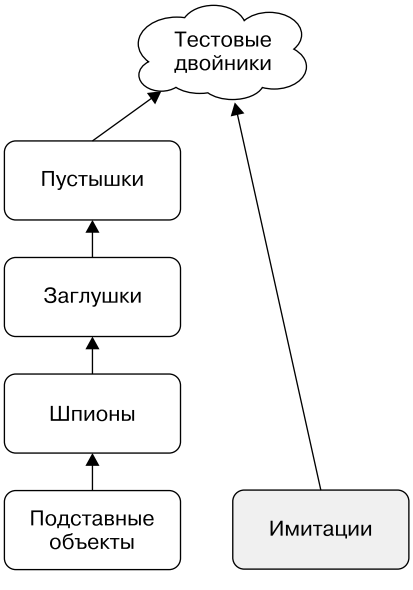


Рис. 3.7. Имитация

В конце 1970-х я работал в компании, которая разработала систему тестирования телефонных линий. Центральный компьютер из сервисного центра (service area computer, SAC) через модемные каналы свя-

зывался с компьютерами, установленными на коммутационных станциях. Последние назывались тестерами абонентской линии (central office line tester, COLT).

Подключенный к коммутационному оборудованию COLT мог создавать электрическое соединение между любой телефонной линией узла и измерительным оборудованием. Результаты измерения электронных характеристик телефонной линии передавались в SAC.

Анализируя эти результаты, SAC определял наличие и местоположение ошибок.

Как мы тестировали такую систему?

Мы построили имитацию в виде COLT, интерфейс переключения которого заменили симулятором. Этот симулятор имитировал звонок по телефонной линии и измерение ее характеристик. Зафиксированные необработанные результаты он возвращал, базируясь на номере телефона, который его попросили протестировать.

Такой подход позволил нам протестировать программное обеспечение SAC для связи, управления и анализа, обойдясь без установки настоящего COLT и даже без установки настоящего коммутационного оборудования и «реальных» телефонных линий.

Сегодня имитация — это тестовый двойник, реализующий какие-то рудиментарные бизнес-правила, причем тесты могут задавать поведение имитаций. Впрочем, проще показать на примере:

```
@Test
public void badPasswordAttempt_loginFails() throws Exception {
  Authenticator authenticator = new FakeAuthenticator();
  LoginDialog dialog = new LoginDialog(authenticator);
  dialog.show();
  boolean success = dialog.submit("user", "bad password");
  assertFalse(success);
}

@Test
public void goodPasswordAttempt_loginSucceeds() throws Exception {
  Authenticator authenticator = new FakeAuthenticator();
  LoginDialog dialog = new LoginDialog(authenticator);
```

```
  dialog.show();
  boolean success = dialog.submit("user", "good password");
  assertTrue(success);
}
```

Оба теста используют объект класса `FakeAuthorizer`, но передают в него разные пароли. Предполагается, что при вводе неверного пароля вход в систему невозможен.

Код класса `FakeAuthenticator` написать легко:

```
public class FakeAuthenticator implements Authenticator {
  public Boolean authenticate(String username, String password)
  {
    return (username.equals("user") && password.equals("good
password"));
  }
}
```

Проблема с использованием имитаций состоит в том, что по мере роста приложения все время появляются новые условия, подлежащие проверке. А каждый новый тест увеличивает размер имитации. В один прекрасный момент ее код может оказаться настолько большим и сложным, что ему потребуются собственные тесты.

Я редко пользуюсь имитациями именно по этой причине.

Принцип неопределенности TDD

Использовать подставные объекты или не использовать, вот в чем вопрос. Хотя, разумеется, в данном случае ответ очевиден, а вопрос состоит в том, *когда* их использовать.

На этот счет есть две точки зрения: лондонская и чикагская. Именно им посвящен конец этой главы. Но прежде чем углубиться в эту тему, важно понять суть проблемы. Проблема возникает из-за *неопределенности*, с которой порой приходится сталкиваться в процессе TDD.

Чтобы как можно более наглядно проиллюстрировать эту неопределенность, мне придется пойти на крайние меры. В качестве примера

я рассмотрю ситуацию, невозможную в реальности, зато очень четко демонстрирующую вещи, которые я пытаюсь до вас донести.

Напишем функцию, вычисляющую синус угла. Угол задается в радианах. Каким должен быть наш первый тест?

Если помните, начинать имеет смысл с самого вырожденного случая, поэтому удостоверимся, что наша функция может вычислить синус нуля:

```
public class SineTest {
  private static final double EPSILON = 0.0001;
  Test
  public void sines() throws Exception {
    assertEquals(0, sin(0), EPSILON);
  }

  double sin(double radians) {
    return 0.0;
  }
}
```

У тех, кто привык думать на перспективу, такой код должен вызвать беспокойство. Ведь этот тест накладывает ограничение только на значения функции — `sin(0)`.

Большинство функций, которые пишутся путем TDD, растущий набор тестов ограничивает настолько, что в какой-то момент функция начинает проходить любые тесты, которые мы добавляем в набор. Вы это видели в примерах с простыми множителями и с игрой в боулинг. Каждый тест сужал пространство возможных решений, пока, наконец, не оставалось всего одно.

Но функция `sin(r)` ведет себя не так. Тест для утверждения `sin(0) == 0` проходит, но похоже, что решение не ограничено одной точкой.

Это станет более очевидным после следующего теста. Каким он должен быть? Почему бы не попробовать вариант `sin(π)`?

```
public class SineTest {
  private static final double EPSILON = 0.0001;
  @Test
  public void sines() throws Exception {
```

```
    assertEquals(0, sin(0), EPSILON);
    assertEquals(0, sin(Math.PI), EPSILON);
  }

  double sin(double radians) {
    return 0.0;
  }
}
```

И снова возникает чувство, что мы ничем не ограничены. Кажется, этот тест ничего не добавляет к решению. Он не содержит даже косвенных намеков на то, как решить задачу. Поэтому попробуем вариант со значением π/2:

```
public class SineTest {
  private static final double EPSILON = 0.0001;
  @Test
  public void sines() throws Exception {
    assertEquals(0, sin(0), EPSILON);
    assertEquals(0, sin(Math.PI), EPSILON);
    assertEquals(1, sin(Math.PI/2), EPSILON);
  }
  double sin(double radians) {
    return 0.0;
  }
}
```

Этот тест провален. Как обеспечить его прохождение? Тест не дает нам никаких подсказок. Возможно, имеет смысл добавить какой-нибудь ужасный оператор `if`, но это приведет только к тому, что таких операторов постепенно станет много.

На данном этапе можно вспомнить, что наш синус раскладывается в ряд Тейлора, и попытаться это реализовать.

Написать такой код не слишком сложно:

```
public class SineTest {
  private static final double EPSILON = 0.0001;
  @Test
  public void sines() throws Exception {
    assertEquals(0, sin(0), EPSILON);
    assertEquals(0, sin(Math.PI), EPSILON);
```

```
    assertEquals(1, sin(Math.PI/2), EPSILON);
  }

  double sin(double radians) {
    double r2 = radians * radians;
    double r3 = r2*radians;
    double r5 = r3 * r2;
    double r7 = r5 * r2;
    double r9 = r7 * r2;
    double r11 = r9 * r2;
    double r13 = r11 * r2;
    return (radians - r3/6 + r5/120 - r7/5040 + r9/362880 -
r11/39916800.0 + r13/6227020800.0);
  }
}
```

Тест пройден, но код выглядит откровенно некрасиво. Тем не менее попробуем вычислить таким способом несколько других синусов:

```
public void sines() throws Exception {
  assertEquals(0, sin(0), EPSILON);
  assertEquals(0, sin(Math.PI), EPSILON);
  assertEquals(1, sin(Math.PI/2), EPSILON);
  assertEquals(0.8660, sin(Math.PI/3), EPSILON);
  assertEquals(0.7071, sin(Math.PI/4), EPSILON);
  assertEquals(0.5877, sin(Math.PI/5), EPSILON);
}
```

Все тесты проходят. Но решение уродливо, так как его точность ограничена. Для получения предельного значения с нужной точностью в ряду Тейлора должно быть достаточное количество членов. (Обратите внимание, как меняется константа `ESPILON`.)

```
public class SineTest {
  private static final double EPSILON = 0.000000001;
  @Test
  public void sines() throws Exception {
    assertEquals(0, sin(0), EPSILON);
    assertEquals(0, sin(Math.PI), EPSILON);
    assertEquals(1, sin(Math.PI/2), EPSILON);
    assertEquals(0.8660254038, sin(Math.PI/3), EPSILON);
    assertEquals(0.7071067812, sin(Math.PI/4), EPSILON);
    assertEquals(0.5877852523, sin(Math.PI/5), EPSILON);
  }
```

```
  double sin(double radians) {
    double result = radians;
    double lastResult = 2;
    double m1 = -1;
    double sign = 1;
    double power = radians;
    double fac = 1;
    double r2 = radians * radians;
    int n = 1;
    while (!close(result, lastResult)) {
      lastResult = result;
      power *= r2;
      fac *= (n+1) * (n+2);
      n += 2;
      sign *= m1;
      double term = sign * power / fac;
      result += term;
    }

    return result;
  }

  boolean close(double a, double b) {
    return Math.abs(a - b) < .0000001;
  }
}
```

Кажется, мы получили ожидаемый результат. Хотя подождите. Что случилось с TDD? И откуда нам знать, что алгоритм работает корректно? Его код получился очень длинным. Как определить, правильный ли этот код?

Наверное, стоит проверить еще несколько значений. Вот только наши тесты уже стали неудобочитаемыми, поэтому выполним небольшой рефакторинг:

```
private void checkSin(double radians, double sin) {
  assertEquals(sin, sin(radians), EPSILON);
}

@Test
public void sines() throws Exception {
  checkSin(0, 0);
```

```
  checkSin(PI, 0);
  checkSin(PI/2, 1);
  checkSin(PI/3, 0.8660254038);
  checkSin(PI/4, 0.7071067812);
  checkSin(PI/5, 0.5877852523);

  checkSin(3* PI/2, -1);
}
```

Тестирование проходит благополучно. Добавим еще парочку проверок:

```
checkSin(2*PI, 0);
checkSin(3*PI, 0);
```

Вариант с 2π работает, а с 3π — нет. Хотя мы подошли достаточно близко: `4.6130E-9`. Вероятно, можно исправить ситуацию, увеличив предел сравнения в функции `close()`, но это уже напоминает подтасовку и для значений 100π или 1000π, скорее всего, не сработает. Лучшим решением было бы уменьшить угол, сохраняя значения в промежутке между 0 и 2π.

```
double sin(double radians) {
  radians %= 2*PI;
  double result = radians;
```

Все работает. А как насчет отрицательных чисел?

```
checkSin(-PI, 0);
checkSin(-PI/2, -1);
checkSin(-3*PI/2, 1);
checkSin(-1000*PI, 0);
```

В данном случае тоже все хорошо. Проверим большие значения, которые не являются кратными 2π:

```
checkSin(1000*PI + PI/3, sin(PI/3));
```

Этот тест тоже прошел. Какие еще варианты можно попробовать? Существуют ли значения, для которых тесты проходить не будут?

Увы! Я не знаю.

Суть принципа неопределенности

Итак, мы столкнулись с ситуацией неопределенности. Сколько бы значений ни проверяли, мы не будем уверены в том, что не упустили некий момент. Будет казаться, что существует какое-то входное значение, которое не даст правильного выходного значения.

К счастью, большинства функций эта ситуация не касается. После того как написан последний тест, вы точно *знаете*, что все работает должным образом. Но бывают и менее приятные функции, заставляющие задуматься о том, что, возможно, существуют значения, при которых тестирование проваливается.

Решить эту проблему с помощью тестов можно только одним способом: проверить все возможные значения. А поскольку числа двойной точности занимают в памяти 64 бита, то получается, нам нужно написать чуть меньше чем 2×10^{19} тестов. Я к таким подвигам не готов.

Итак, какой информации о рассматриваемой функции мы можем доверять? Верно ли то, что с помощью ряда Тейлора можно вычислить синус угла, заданного в радианах? Да, мы видели математическое доказательство этого факта и можем быть уверены, что ряд Тейлора сойдется к правильному значению.

Но как написать набор тестов, которые докажут, что мы корректно производим все расчеты?

Наверное, можно проанализировать каждый член ряда Тейлора. Например, для $\sin(\pi)$ члены ряда Тейлора равны 3,141592653589793, −2,0261201264601763, 0,5240439134171688, −0,07522061590362306, 0,006925270707505149, −4,4516023820919976E-4, 2,114256755841263E-5, −7,727858894175775E-7, 2,2419510729973346E-8.

Впрочем, этот тест ничем не лучше уже написанного. Вышеприведенные значения применимы только к конкретному тесту, и мы не знаем, можно ли их использовать для расчета синуса другого угла.

Нет, нужно что-то другое. Что-то однозначное, *доказывающее*, что наш алгоритм действительно корректно считает ряд Тейлора.

Но что собой представляет этот ряд? Это бесконечная знакопеременная сумма нечетных степеней x, деленная на нечетные факториалы:

$$\sum_{n=1}^{\infty}(-1)^{(n-1)}\frac{x^{2n-1}}{(2n-1)!}.$$

Или, другими словами,

$$x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\frac{x^9}{9!}-\ldots$$

Как это нам поможет? При наличии шпиона, информирующего о том, как рассчитываются члены ряда Тейлора, можно написать вот такой тест:

```
@Test
public void taylorTerms() throws Exception {
  SineTaylorCalculatorSpy c = new SineTaylorCalculatorSpy();
  double r = Math.random() * PI;
  for (int n = 1; n <= 10; n++) {
    c.calculateTerm(r, n);
    assertEquals(n - 1, c.getSignPower());
    assertEquals(r, c.getR(), EPSILON);
    assertEquals(2 * n - 1, c.getRPower());
    assertEquals(2 * n - 1, c.getFac());
  }
}
```

Теперь мы присваиваем переменной `r` случайные значения, а переменной `n` — значения от 0 до 10, то есть начали рассматривать происходящее в обобщенном виде. Прохождение этого теста покажет нам, что на выход калькуляторам `sign`, `power` и `factorial` поступают корректные данные.

Пройти тест позволит вот такой простой код:

```
public class SineTaylorCalculator {
  public double calculateTerm(double r, int n) {
    int sign = calcSign(n-1);
    double power = calcPower(r, 2*n-1);
```

```
    double factorial = calcFactorial(2*n-1);
    return sign*power/factorial;
  }

  protected double calcFactorial(int n) {
    double fac = 1;
    for (int i=1; i<=n; i++)
      fac *= i;
    return fac;
  }

  protected double calcPower(double r, int n) {
    double power = 1;
    for (int i=0; i<n; i++)
      power *= r;
    return power;
  }

  protected int calcSign(int n) {
    int sign = 1;
    for (int i=0; i<n; i++)
      sign *= -1;
    return sign;
  }
}
```

Обратите внимание, что мы не тестируем функции, выполняющие расчеты. Они довольно просты и в тестировании, скорее всего, не нуждаются. Это особенно видно на фоне остальных тестов, написанием которых я сейчас займусь.

Код шпиона выглядит так:

```
package London_sine;

public class SineTaylorCalculatorSpy extends SineTaylorCalculator {
  private int fac_n;
  private double power_r;
  private int power_n;
  private int sign_n;
  public double getR() {
    return power_r;
  }

  public int getRPower() {
```

```
    return power_n;
  }

  public int getFac() {
    return fac_n;
  }

  public int getSignPower() {
    return sign_n;
  }

  protected double calcFactorial(int n) {
    fac_n = n;
    return super.calcFactorial(n);
  }

  protected double calcPower(double r, int n) {
    power_r = r;
    power_n = n;
    return super.calcPower(r, n);
  }

  protected int calcSign(int n) {
    sign_n = n;
    return super.calcSign(n);
  }

  public double calculateTerm(double r, int n) {
    return super.calculateTerm(r, n);
  }
}
```

Если учесть, что наш код проходит этот тест, то насколько сложно будет написать алгоритм суммирования?

```
public double sin(double r) {
  double sin=0;
  for (int n=1; n<10; n++)
    sin += calculateTerm(r, n);
  return sin;
}
```

Возможно, вас не устраивает эффективность всего того, что я проделал выше, но верите ли вы, что это работает? Правильно ли функция `calculateTerm` вычисляет члены ряда Тейлора? Корректно ли

функция `sin` производит их суммирование? Достаточно ли десяти итераций? Как проверить все это, не прибегая к исходным тестам с конкретными значениями?

Ниже представлен интересный тест. Все значения `sin(r)` должны быть между −1 и 1 (как и полагается).

```
@Test
public void testSineInRange() throws Exception {
  SineTaylorCalculator c = new SineTaylorCalculator();
  for (int i=0; i<100; i++) {
    double r = (Math.random() * 4 * PI) - (2 * PI) ;
    double sinr = c.sin(r);
    assertTrue(sinr < 1 && sinr > -1);
  }
}
```

Тест пройден. Теперь с учетом вот этого тождества

```
public double cos(double r) {
  return (sin(r+PI/2));
}
```

проверим теорему Пифагора: $\sin^2 + \cos^2 = 1$.

```
@Test
public void PythagoreanIdentity() throws Exception {
  SineTaylorCalculator c = new SineTaylorCalculator();
  for (int i=0; i<100; i++) {
    double r = (Math.random() * 4 * PI) - (2 * PI) ;
    double sinr = c.sin(r);
    double cosr = c.cos(r);
    assertEquals(1.0, sinr * sinr + cosr * cosr, 0.00001);
  }
}
```

Хм. Для прохождения теста потребовалось увеличить количество членов ряда до 20, что, конечно, абсурдно много. Но как я уже говорил, в этом упражнении я рассматриваю утрированную ситуацию.

Насколько после всех этих тестов мы можем быть уверены в корректном вычислении синуса? Не знаю, как вы, но мне вполне достаточно. Я знаю, что в члены ряда Тейлора передаются корректные значения.

Я могу на глаз оценить простые калькуляторы, да и функция `sin` проявляет свойства синуса.

Ладно, просто на всякий случай протестируем несколько значений:

```
@Test
public void sineValues() throws Exception {
  checkSin(0, 0);
  checkSin(PI, 0);
  checkSin(PI/2, 1);
  checkSin(PI/3, 0.8660254038);
  checkSin(PI/4, 0.7071067812);
  checkSin(PI/5, 0.5877852523);
}
```

Да, все работает. Прекрасно. Я разрешил свои сомнения и теперь могу быть уверенным в корректности вычисления синусов. Спасибо шпиону!

И снова о принципе неопределенности TDD

Хотя подождите. Знаете ли вы, что есть для вычисления синусов алгоритм получше? Он называется CORDIC. Я не буду в него углубляться, так как это выходит за рамки темы текущей главы. Просто предположим, что мы решили отредактировать нашу функцию, заменив разложения в ряд Тейлора этим алгоритмом.

После этого наш тест со шпионом перестанет работать!

Это легко объяснимо. Достаточно посмотреть, сколько кода написано для реализации алгоритма на базе разложения в ряд Тейлора. Я создал целых два класса, `SineTaylorCalculator` и `SineTaylorCalculatorSpy`. Теперь этот код попросту не нужен, зато придется разрабатывать новую стратегию тестирования.

Тесты, в которых используются шпионы, очень *хрупкие*. Любое изменение алгоритма выводит их из строя, и их приходится редактировать или полностью переписывать.

При этом первоначальные варианты тестов, которые я писал под конкретные значения, продолжат работать с новым алгоритмом CORDIC.

Их переписывать не придется. Здесь мы подошли ко второму аспекту принципа неопределенности TDD.

> **Принцип неопределенности TDD:** *насколько большой определенности вы требуете, настолько негибкими будут ваши тесты. Чем выше гибкость теста, тем меньшую уверенность он дает.*

Лондон против Чикаго

Принцип неопределенности TDD может создать впечатление, что заниматься тестированием — безнадежное дело, но это совсем не так. Он всего лишь накладывает некоторые ограничения на степень полезности тестов.

С одной стороны, жестко ограниченные, хрупкие тесты нам не нужны. С другой стороны, нам бы хотелось быть максимально уверенными. Как инженеры, мы должны найти компромисс между этими двумя желаниями.

Проблема хрупкого теста

Новички в TDD зачастую подходят к разработке тестов не слишком тщательно. К тестам они относятся как к сущностям второго сорта, нарушая все правила связности и согласованности. В результате многие тесты перестают работать даже после небольших изменений производственного кода, а то и после незначительного рефакторинга.

Необходимость переписывать код таких тестов становится причиной разочарования в TDD. Многие новички прекратили практиковать разработку через тестирование исключительно из-за непонимания того факта, что тесты должны проектироваться так же тщательно, как и производственный код.

Чем больше вы связываете тесты с производственным кодом, тем более хрупкими они становятся. При этом мало что обеспечивает такую тесную связь, как шпионы. Они заглядывают в самое сердце алгоритмов, неразрывно связывая их с тестами. А так как подставные объекты — это шпионы, все написанное выше касается и их.

Выбор между гибкостью и определенностью

Если вы избегаете написания шпионов, как это делаю я, вам остается тестировать значения и свойства. Тестированием значений мы занимались при построении алгоритма вычисления синуса. Это обычное сопоставление входных и выходных значений.

Свойства мы проверяли с помощью тестов `testSineInRange` и `PythagoreanIdentity`. Такие тесты перебирают множество подходящих входных значений, проверяя инварианты. При всей своей убедительности они все равно не устраняют всех сомнений.

С другой стороны, такие тесты настолько не связаны с используемым алгоритмом, что даже полная замена алгоритма никак на них не влияет.

Если вы ставите уверенность превыше гибкости, скорее всего, вы будете использовать множество *шпионов* и мириться с неизбежной хрупкостью тестов.

Если же, как и я, превыше всего вы цените гибкость, то предпочтете писать тесты для конкретных значений, смирившись с мучительной неопределенностью.

Эти подходы привели к появлению двух школ TDD и оказали глубокое влияние на отрасль в целом. Дело в том, что выбор в пользу гибкости или определенности кардинальным образом влияет на процесс проектирования производственного кода, а порой и на его архитектуру.

Лондонская школа

Лондонская школа TDD получила такое название, поскольку в Лондоне проживают Стив Фримен и Нат Прайс, написавшие книгу по этой теме[1]. В этой школе определенность предпочитают гибкости.

---

[1] *Freeman S., Pryce N.* Growing Object-Oriented Software, Guided by Tests. — Addison-Wesley, 2010.

Разумеется, сторонники этого подхода не отказываются от гибкости полностью. Более того, высоко ее ценят. Просто они готовы мириться с определенным отсутствием гибкости в обмен на бóльшую уверенность.

Соответственно, тесты, написанные приверженцами лондонской школы, содержат последовательное и относительно неограниченное количество подставных объектов и шпионов.

При этом внимание в основном фокусируется на алгоритме, а не на результатах. Результаты, конечно, тоже важны, но *куда важнее то, как они получены*. Такой подход приводит к интересной методологии TDD по схеме *снаружи внутрь*.

Программисты, которые следуют такой методологии, начинают с пользовательского интерфейса и постепенно, *по одному варианту за раз*, спускаются к бизнес-правилам. На каждой границе с помощью подставных объектов и шпионов программисты проверяют работоспособность алгоритма, который используют для передачи данных внутрь. В конце концов они добираются до бизнес-правила, реализуют его, подключают к базе данных, а затем с помощью подставных объектов и шпионов проверяют путь обратно к пользовательскому интерфейсу.

Таким способом программист *по очереди* проходит по графу классов, пока не реализует их все.

Это чрезвычайно строгий и упорядоченный подход, который и в самом деле может давать очень хорошие результаты.

Классическая школа, или Школа Чикаго

Чикагская школа TDD получила свое название по месту дислокации компании ThoughtWorks, в которой работал (и работает по сей день) научным сотрудником Мартин Фаулер. Хотя с названием все далеко не однозначно: например, одно время эта школа называлась детройтской.

Чикагская школа ориентирована на гибкость, а не на уверенность. Опять же, это не означает отказ от одного в пользу другого. Представители данной школы знают цену уверенности, но когда у них

есть возможность выбора, предпочитают делать тесты более гибкими. Соответственно, они сильнее фокусируются на результатах, чем на взаимодействиях и алгоритмах.

Это привело к другой философии проектирования. Представители этой школы, как правило, начинают с бизнес-правил, а затем переходят к пользовательскому интерфейсу, то есть работают по схеме *изнутри наружу*.

Этот процесс проектирования так же строго регламентирован, как и лондонский, просто решение задач происходит в противоположном порядке. Представитель школы Чикаго не будет проходить по графу от начала до конца и только потом переходить к рассмотрению следующего случая. Скорее он, несмотря на отсутствие пользовательского интерфейса, займется написанием тестов свойств и значений для реализации бизнес-правил. Пользовательский интерфейс и сшивка его с бизнес-правилами реализуются по мере необходимости.

Бизнес-правила далеко не сразу переносятся в базу данных. По правилам чикагской школы, вместо того чтобы рассматривать по одному варианту использования за раз, предпочитают искать согласованность и одинаковый код в разных слоях. Вместо того чтобы прокладывать тропинку от входных данных конкретного сценария до их вывода, в чикагской школе работают более широкими мазками внутри слоев, начиная с реализации бизнес-правил и постепенно переходя к пользовательскому интерфейсу и базе данных. По мере построения каждого уровня идет поиск шаблонов проектирования и возможностей для повышения абстракции и обобщения.

Это менее упорядоченный, но более целостный подход, чем в лондонской школе. По моему скромному мнению, он дает более четкое представление об общей картине.

Синтез

Несмотря на существование двух школ и на наличие у каждой своих приверженцев, друг с другом они не воюют. Более того, между ними даже нет больших разногласий. Они просто по-разному расставляют акценты.

Здесь уместно обратить внимание на тот факт, что все практикующие TDD, как чикагцы, так и лондонцы, используют в работе оба подхода. Просто кто-то больше склоняется в одну сторону, а кто-то — в другую.

Какой вариант правильный? Разумеется, ни тот ни другой. Мне ближе подход чикагской школы, а вы вполне можете посмотреть на лондонскую школу и решить, что удобнее работать именно так. И это не повод ссориться. Более того, мы с радостью соединим плоды наших усилий, чтобы получить прекрасный общий результат.

Такой синтез становится очень важным, когда речь заходит об архитектуре.

АРХИТЕКТУРА

Компромисс между лондонской и чикагской стратегией связан с архитектурой. Те, кто читал «Чистую архитектуру», знают, что мне нравится разбивать системы на компоненты. Стыки между компонентами я называю *границами* (boundaries). При этом зависимости исходного кода всегда должны пересекать эти границы в сторону политик более высокого уровня.

Это означает, что низкоуровневые компоненты, такие как графические пользовательские интерфейсы (GUI) и базы данных, зависят от компонентов более высокого уровня, таких как бизнес-правила. Обратной зависимости при этом нет. Это буква D в аббревиатуре SOLID — принцип инверсии зависимостей.

При написании тестов для самого низкого уровня я буду пользоваться *шпионами* (а иногда и *подставными объектами*) для всего, что находится за пределами архитектурных границ. Другими словами, при тестировании компонента я использую *шпионов* для имитации всех зависимостей, чтобы убедиться, что тестируемый компонент взаимодействует с ними корректно. То есть в ситуациях, когда мой тест пересекает границы, я предпочитаю лондонский подход.

Но если границы при тестировании не пересекаются, то я предпочитаю действовать, как принято в чикагской школе. Внутри компо-

нентов я больше полагаюсь на тестирование их состояния и свойств, чтобы свести связанность, а значит и хрупкость, моих тестов к минимуму.

Рассмотрим пример. Диаграмма UML на рис. 3.8 показывает набор классов и четыре компонента, которые их содержат.

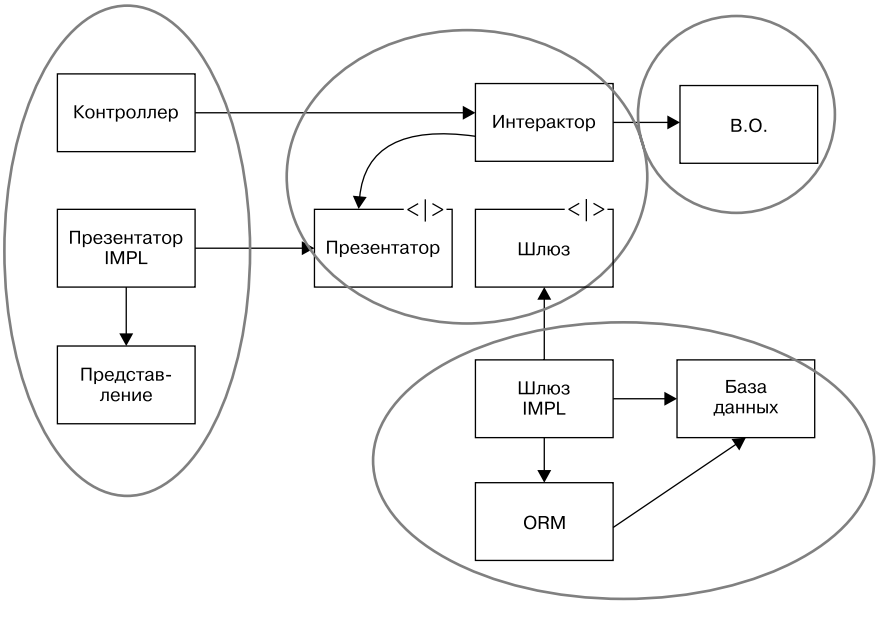


Рис. 3.8. Набор классов и четыре включающих их компонента

Обратите внимание, что все стрелки ведут от низкоуровневых компонентов к компонентам более высокого уровня. Это *правило зависимостей* (Dependency Rule), которое обсуждалось в «Чистой архитектуре». На самом высоком уровне содержатся бизнес-объекты. Ниже располагаются интеракторы, или посредники, и коммуникационные интерфейсы. На самом нижнем уровне — GUI и база данных.

При тестировании бизнес-объектов можно использовать заглушки, но шпионы и подставные объекты не понадобятся, поскольку бизнес-объекты о существовании остальных компонентов попросту не знают.

А вот интеракторы манипулируют бизнес-объектами, базой данных и графическим интерфейсом, поэтому при их тестировании имеет смысл прибегнуть к шпионам, чтобы убедиться в корректности работы базы данных и GUI. Но вряд ли нам потребуется множество шпионов или заглушек между интеракторами и бизнес-объектами, поскольку последние не имеют особо сложных функций.

При тестировании контроллера интерактор практически наверняка будет представлен с помощью шпиона, чтобы вызовы не распространялись на базу данных или на презентатор.

С презентатором складывается очень интересная ситуация. Мы воспринимаем его как часть графического интерфейса, однако на самом деле для его тестирования понадобится шпион. Его не стоит тестировать с реальным представлением (view), поэтому, скорее всего, для его хранения обособленно от контроллера и презентатора понадобится выделить отдельный компонент.

Это небольшое осложнение возникает довольно часто. И нам приходится менять границы областей, потому что этого требуют тесты.

РЕЗЮМЕ

Итак, мы рассмотрели более сложные аспекты TDD, такие как поэтапная разработка алгоритмов, действия при попадании в мертвую точку, связь тестов с конечными автоматами, тестовые двойники и принцип неопределенности. Но это пока не конец. Еще кое-что осталось. Поэтому выпейте чашечку горячего чая и приготовьтесь к новым невероятным знаниям.

4 РАЗРАБОТКА ТЕСТОВ

При чтении трех законов разработки через тестирование, изложенных в главе 2, может сложиться впечатление, что TDD — это просто. Достаточно следовать трем законам, чтобы получить гарантированный результат. Но это далеко не так. Стать мастером в TDD очень сложно. Разработка через тестирование — многослойный процесс, и на освоение различных слоев уходят месяцы, если не годы.

В этой главе мы углубимся в некоторые из этих слоев. Для начала рассмотрим такие непростые ситуации, как тестирование баз данных и графических интерфейсов, а затем поговорим о принципах разработки тестов, шаблонах тестирования и некоторых интересных теоретических возможностях.

ТЕСТИРОВАНИЕ БАЗ ДАННЫХ

Первое правило тестирования баз данных: *не тестировать базы данных*. Вам это не нужно. Можно по умолчанию считать, что база данных работает. Если это не так, то вы узнаете об этом достаточно быстро.

А вот что стоит протестировать — это запросы. Точнее, нам важно проверить, правильно ли формулируются отправляемые в базу данных команды. Если вы пишете код SQL, то захотите проверить, корректно ли работают написанные операторы. Используя фреймворки ORM, такие как Hibernate, вы захотите удостовериться, что Hibernate работает с базой данных так, как вы рассчитываете. Если вы пользуетесь базой данных NoSQL, то захотите проверить, что ваши обращения к ней достигают поставленной цели.

Ни один из этих тестов не требует проверки бизнес-правил; все они связаны только с самими запросами. Поэтому второе правило тестирования баз данных звучит так: *изолируйте базу данных от бизнес-правил*.

Это делается с помощью интерфейса, который на диаграмме рис. 4.1 называется `Gateway`[1]. Внутри этого интерфейса для каждого типа

[1] *Фаулер М.* Шаблоны корпоративных приложений.

запроса, который мы хотим выполнить, создается по одному методу. Эти методы могут принимать аргументы, изменяющие запрос. Например, чтобы получить из базы данных всех сотрудников, принятых на работу после 2001 года, можно вызвать метод интерфейса `getEmployeesHiredAfter(2001)`.

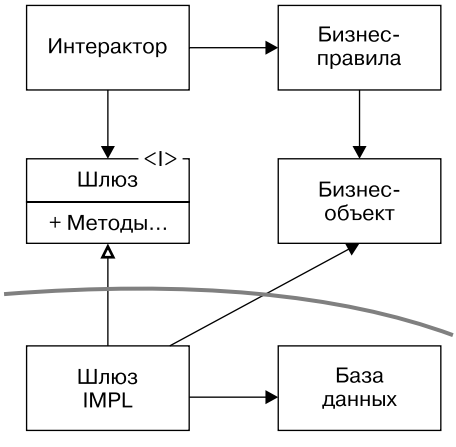


Рис. 4.1. Тестирование базы данных

Интерфейс `Gateway` будет содержать методы для каждого запроса, обновления, удаления или добавления в базу данных. Количество этих интерфейсов зависит от того, как мы разделим базу данных (БД).

Класс `GatewayImpl` реализует шлюз и заставляет БД выполнять то, что от нее требуется. Все SQL-запросы создаются внутри класса `GatewayImpl`. Фреймворком ORM управляет класс `GatewayImpl`. Ни о SQL, ни о фреймворке ORM, ни об API базы данных не известно выше архитектурной границы, разделяющей классы `Gateway` и `GatewayImpl`.

Более того, мы и не хотим, чтобы схема БД была известна выше этой границы. Класс `GatewayImpl` должен распаковывать строки или элементы данных, извлеченные из базы, и создавать на их основе бизнес-объекты, которые и будут передаваться через границу в бизнес-правила.

После этого тестирование БД осуществляется тривиально. Создается достаточно простая тестовая база, а затем вызывается каждая функция-запрос из класса `GatewayImpl` и проверяется, что в тестовой базе происходят нужные изменения. Важно убедиться, что в ответ на каждый запрос возвращается корректный набор бизнес-объектов. Проследите за тем, чтобы каждое обновление, добавление и удаление сопровождалось соответствующими изменениями в базе.

Использовать для тестирования рабочую БД нельзя. Именно поэтому создается тестовая база с достаточным количеством строк и сразу же делается ее резервная копия. Перед каждым следующим запуском тестов базу нужно приводить в исходное состояние, чтобы тестирование *всегда* выполнялось на одних и тех же данных.

При тестировании бизнес-правил замещайте классы `GatewayImpl` заглушками и шпионами. Не тестируйте бизнес-правила на реальной базе данных. Во-первых, такое тестирование будет происходить очень медленно, во-вторых, вы рискуете внести в базу ошибки. Вместо этого проверяйте, корректно ли бизнес-правила и интеракторы манипулируют интерфейсами `Gateway`.

ТЕСТИРОВАНИЕ ГРАФИЧЕСКИХ ИНТЕРФЕЙСОВ

Ниже даны правила тестирования графических интерфейсов.

1. Не тестируйте GUI.
2. Тестируйте все, кроме GUI.
3. Графический интерфейс меньше, чем вам кажется.

Первым делом мы посмотрим на третье правило. Графический интерфейс — очень маленький элемент программного обеспечения, который отображает информацию на экране. Наверное, это самая маленькая часть ПО. Она отправляет команды механизму, который фактически рисует на экране пикселы.

В веб-системах именно GUI создает HTML. В системах с рабочим столом GUI вызывает API программного обеспечения, управляющего

графикой. Задача разработчика состоит в том, чтобы максимально уменьшить это ПО.

Например, должно ли это программное обеспечение уметь форматировать даты, валюту или числа? Нет. Это может сделать другой модуль. Графическому интерфейсу нужны только соответствующие строки, представляющие отформатированные даты, валюты или числа.

Мы называем этот модуль *презентатором*. Именно он отвечает за форматирование и размещение данных, которые должны отображаться на экране или в окне. И он делает для этого все возможное, позволяя создавать абсурдно маленькие графические интерфейсы.

К примеру, презентатор определяет состояние каждой кнопки и пункта меню. В нем указаны их имена и сообщается о необходимости выделения неактивных кнопок серым цветом. Если название кнопки зависит от состояния окна, то именно презентатор получает информацию о состоянии и соответствующим образом меняет это название. Если на экране должна появиться таблица чисел, то именно презентатор создает таблицу нужным образом отформатированных и упорядоченных строк. Если какие-то поля должны быть выделены цветом или другим шрифтом, то именно презентатор устанавливает цвета и шрифты.

Презентатор заботится о форматировании и расположении, создавая простую структуру данных, заполненную строками и флагами, которую графический интерфейс использует для построения отправляемых им на экран команд. Это также позволяет сделать графический интерфейс очень маленьким.

Созданную презентатором структуру данных часто называют *моделью представления* (view model).

На диаграмме, показанной на рис. 4.2, интерактор сообщает презентатору, какие данные должны быть представлены на экране. Это сообщение выглядит как одна или несколько структур данных, передаваемых с помощью набора функций. При этом презентатор экранирован от интерактора интерфейсом. Это сделано, чтобы высокоуровневый интерактор не зависел от реализации презентатора, расположенного на более низком уровне.

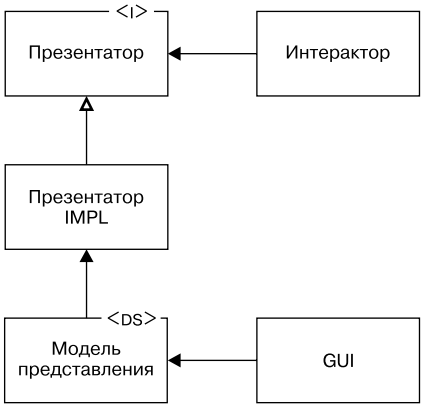


Рис. 4.2. Интерактор отвечает за передачу презентатору информации о том, какие данные нужно вывести на экран

Презентатор строит структуру данных модели представления, а графический интерфейс переводит ее в команды, управляющие экраном.

Понятно, что протестировать интерактор можно, заменив презентатор шпионом. Понятно и то, что презентатор можно протестировать, отправляя ему команды и проверяя результат в модели представления.

Единственное, что нельзя легко протестировать (с помощью автоматических модульных тестов), — это сам графический интерфейс, поэтому мы делаем его очень маленьким.

Впрочем, графический интерфейс можно протестировать визуально. Достаточно передать в него готовый набор моделей представления и собственными глазами убедиться, что они отображаются надлежащим образом.

Существуют инструменты, позволяющие это автоматизировать, но обычно я не советую к ним прибегать. Они, как правило, медленные и хрупкие. Кроме того, графический интерфейс зачастую оказывается очень изменчивым модулем. Каждый раз, когда кто-то хочет изменить внешний вид чего-либо на экране, это влияет на код GUI. Поэтому

написание для него автоматических тестов — зачастую пустая трата времени. Его код меняется так часто и по таким незначительным поводам, что тесты редко остаются действительными в течение длительного времени.

Графический ввод

Тестирование ввода GUI следует тем же правилам: мы делаем GUI как можно более незначительным. На диаграмме, показанной на рис. 4.3, фреймворк GUI — это код, который находится на границе системы. Это может быть веб-контейнер или что-то наподобие библиотеки Swing[1] или языка Processing[2] для управления рабочим столом.

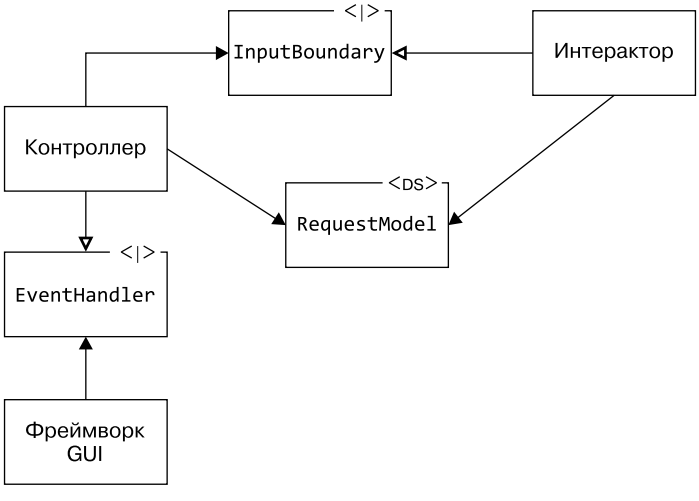


Рис. 4.3. Тестирование GUI

Фреймворк GUI взаимодействует с контроллером через интерфейс `EventHandler`. Это гарантирует, что в исходном коде контроллера не

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/swing/.

[2] https://processing.org/.

будет транзитивной зависимости от фреймворка GUI. Задача контроллера сводится к сбору необходимых событий из этого фреймворка в структуру данных, которую я назвал `RequestModel`.

После заполнения этой структуры контроллер передает ее интерактору через интерфейс `InputBoundary`. Этот интерфейс нужен для обеспечения правильного с архитектурной точки зрения направления зависимостей.

Тестирование интерактора представляет собой тривиальную процедуру. Тесты просто создают соответствующие модели запросов и передают их интерактору. Результаты можно проверить как напрямую, так и с помощью шпионов. Тестирование контроллера также осуществляется очень просто — тесты вызывают события через интерфейс обработчика событий, а затем проверяют, корректную ли модель запроса создал контроллер.

ШАБЛОНЫ ТЕСТИРОВАНИЯ

Для тестов существует множество различных шаблонов проектирования. Они описаны в различных книгах. Могу рекомендовать вам *XUnit Test Patterns* Джерарда Месароша и *JUnit Recipes*[1] Дж. Б. Рейнсбергера и Скотта Стирлинга.

Я не собираюсь описывать здесь все возможные шаблоны и рецепты. Расскажу о трех из них, оказавшихся для меня наиболее полезными.

Связанный с тестом подкласс

Этот шаблон в основном используется как механизм безопасности. Представим, что мы хотим протестировать метод `align` класса `XRay`. Но метод `align`, в свою очередь, вызывает метод `TurnOn`, отвечающий за включение рентгеновского аппарата. Понятно, что никому не нужно, чтобы аппарат включался при каждом тестировании.

[1] *Rainsberger J. B., Stirling S.* JUnit Recipes: Practical Methods for Programmer Testing. — Manning, 2006.

Решение проблемы показано на рис. 4.4. Это создание *связанного с тестом подкласса* (test-specific subclass) класса `XRay`, который переопределит метод `turnOn`. В результате тест создает экземпляр класса `SafeXRay`, а затем вызывает метод `assign`, и данный процесс больше не сопровождается включением рентгеновского аппарата.

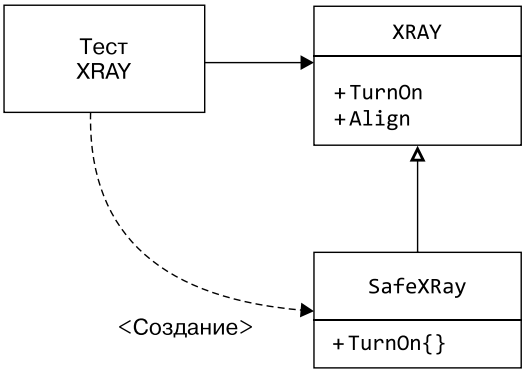


Рис. 4.4. Шаблон Test-Specific-Subclass

Часто бывает полезно сделать этот связанный с тестом подкласс шпионом, получив возможность опрашивать безопасный объект на предмет вызовов небезопасного метода.

Если бы в рассматриваемом нами примере подкласс `SafeXRay` был шпионом, то метод `TurnOn` зафиксировал бы вызов с его стороны, а тестовый метод класса `XRayTest` мог бы узнать из этой записи, действительно ли вызывался метод `TurnOn`.

Иногда шаблон *Test-Specific Subclass* используется не для безопасности, а для удобства и увеличения пропускной способности. Например, когда мы не хотим, чтобы тестируемый метод запускал новый процесс или выполнял дорогостоящие вычисления.

Нередко опасные, неудобные или медленные операции выделяются в отдельные методы с явной целью переопределить их внутри связанного с тестом подкласса. Это лишь один из вариантов влияния тестов на структуру кода.

Самошунтирование

Разновидность этой темы — шаблон *самошунтирования* (Self-Shunt). Дело в том, что часто бывает очень удобно превратить тестовый класс в подкласс, связанный с тестом, как показано на рис. 4.5.

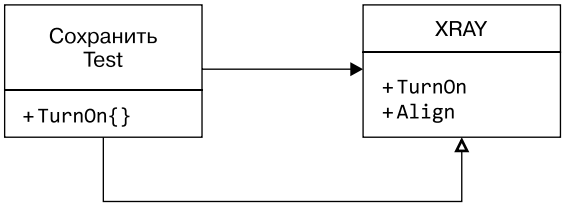


Рис. 4.5. Шаблон Self-Shunt

В этом случае метод `turnOn` переопределяет именно класс `XRayTest`, который может также действовать как шпион, получающий информацию из этого метода.

Я считаю шаблон Self-Shunt очень удобным в ситуациях, когда требуется простой шпион или обеспечение безопасности. Но отсутствие отдельного класса с информативным именем, отвечающего за обеспечение безопасности или за слежку, не лучшим образом сказывается на читабельности кода, поэтому я использую этот шаблон с осторожностью.

Работая с ним, важно помнить, что разные фреймворки тестирования создают тестовые классы в разное время. Если, к примеру, JUnit создает новый экземпляр тестового класса для каждого вызова тестового метода, то NUnit выполняет все тестовые методы в одном экземпляре тестового класса. Поэтому необходимо позаботиться о том, чтобы все переменные внутри шпиона возвращались в исходное состояние.

Скромный объект

Было бы отлично, если бы каждый бит кода в системе можно было протестировать, придерживаясь трех законов TDD, но, к сожалению,

это не так. Особенно трудно выполнять тестирование фрагментов кода, взаимодействующих через аппаратные границы.

Например, как проверить, что отображается на экране, что было отправлено через сетевой интерфейс или через параллельный или последовательный порт ввода-вывода? Без специально разработанных аппаратных механизмов, с которыми могут взаимодействовать тесты, тестирование таких вещей попросту невозможно.

К сожалению, такие аппаратные механизмы часто оказываются медленными и/или ненадежными. Представьте, что мы направили на экран видеокамеру и тест отчаянно пытается определить, является ли изображение с камеры тем, которое было отправлено для отображения на экране. Или представьте сетевой кабель с контуром обратной связи, соединяющий выходной порт сетевого адаптера с входным портом. Тестам придется считывать поток поступающих на входной порт данных и искать определенные данные, которые были отправлены на выходной порт.

В большинстве случаев такое специализированное оборудование неудобно, а то и вовсе непрактично.

Компромиссным решением становится шаблон *Скромный объект* (Humble Object). Его цель — до предела *упростить* код, который с трудом поддается тестированию. Пример такого подхода я уже демонстрировал выше в разделе «Тестирование графических интерфейсов», теперь же рассмотрим его более подробно.

Общая стратегия показана на рис. 4.6. Код, который взаимодействует через границу, разделен на два элемента: презентатор и скромный объект (`HumbleView`). Связь между ними осуществляется с помощью структуры данных `Presentation`.

Предположим, наше приложение (на диаграмме оно не показано) хочет что-то отобразить на экране. Оно отправляет соответствующие данные презентатору, который распаковывает их в максимально простую форму и загружает в структуру данных `Presentation`. Цель распаковки — исключить из `HumbleView` все этапы обработки, кроме простейших. Скромный объект просто должен транспортировать структуру данных `Presentation` через границу.

Рис. 4.6. Общая стратегия

Предположим, приложение хочет создать диалоговое окно с кнопками Post и Cancel, меню выбора идентификаторов заказов, а также таблицу дат и элементов валюты. Данные, которые оно отправляет презентатору, состоят из этой таблицы, представленной в виде объектов `Date` и `Money`, а также списка объектов `Order` для меню.

Презентатор должен превратить все это в строки и флаги и загрузить в структуру данных `Presentation`. Объекты `Money` и `Date` преобразуются в строки, зависящие от местоположения. Объекты `Order` преобразуются в строки идентификаторов. Имена двух кнопок загрузятся в виде строк. Если одна или несколько кнопок должны быть неактивными, то будет установлен соответствующий флаг.

В результате нашему скромному объекту `HumbleView` останется только передать эти строки через границу вместе с метаданными, заданными с помощью флагов. Еще раз напомню, что объект `HumbleView` специально делается чрезмерно простым, чтобы его не нужно было тестировать.

Понятно, что такая стратегия будет работать для любого пересечения границы, а не только для данных, выводимых на экран.

В качестве примера попробуем написать управляющее программное обеспечение для беспилотного автомобиля. Предположим, что руль

управляется шаговым электродвигателем, который за один шаг перемещает его на один градус. Манипуляция шаговым двигателем осуществляется с помощью следующей команды:

```
out(0x3ff9, d);
```

где `0x3ff9` — адрес ввода-вывода контроллера шагового двигателя, а `d` равно `1` для поворота вправо и `0` для поворота влево.

На высоком уровне ИИ нашего беспилотного автомобиля отдает презентатору `SteeringPresenter` команды следующего вида:

```
turn(RIGHT, 30, 2300);
```

Это означает, что автомобиль (не руль!) должен за следующие 2300 миллисекунд повернуться на 30 градусов вправо. Для этого нужно повернуть руль вправо на определенное количество шагов с определенной скоростью, а затем с определенной скоростью вернуть его влево.

Как проверить, что ИИ правильно управляет рулем? Нужно упростить низкоуровневое программное обеспечение, отвечающее за эту операцию. Мы можем сделать это, передав ему вот такой массив структур данных:

```
struct SteeringPresentationElement{
  int steps;
  bool direction;
  int stepTime;
  int delay;
};
```

Низкоуровневый контроллер перебирает элементы этого массива и указывает шаговому электродвигателю, сколько шагов (`steps`) следует сделать в заданном направлении (`direction`), со временем задержки между шагами `stepTime` и временем задержки перед переходом к следующему элементу массива `delay`.

Задача презентатора `SteeringPresenter` состоит в переводе команд ИИ в массив `SteeringPresentationElements`. Для этого презентатору нужно знать скорость автомобиля и отношение угла поворота рулевого колеса к углу поворота колес.

Очевидно, что презентатор `SteeringPresenter` легко тестировать. Тест просто отправляет в него соответствующие команды поворота, а затем проверяет результаты в массиве `SteeringPresentationElements`.

Наконец, обратите внимание на элемент `ViewInterface` на диаграмме. Если связать его, презентатор и структуру данных `Presentation` в единый компонент, то представление `HumbleView` будет зависеть от этого компонента. Это архитектурная стратегия, позволяющая высокоуровневому презентатору не зависеть от подробной реализации представления `HumbleView`.

ПРОЕКТИРОВАНИЕ ТЕСТОВ

Мы все знаем, насколько важно тщательно проектировать производственный код. Но задумывались ли вы когда-нибудь о структуре своих тестов? Многие программисты не уделяют этому должного внимания. Они просто запускают тесты, не вникая в детали их реализации. Такой подход всегда приводит к проблемам.

Проблема хрупких тестов

Одна из проблем, с которыми сталкиваются плохо владеющие TDD программисты, — это проблема хрупких тестов. Набор тестов называют хрупким, если небольшие изменения производственного кода выводят из строя множество тестов. Чем меньшие изменения вызывают такой эффект, тем сильнее это выводит программиста из равновесия. Многие по этой причине быстро отказываются от TDD.

Хрупкость всегда обусловлена архитектурными проблемами. Если небольшое изменение одного модуля вызывает множество изменений в других модулях, то очевидно, что система спроектирована очень плохо. Более того, нарушение работы множества компонентов из-за небольшого изменения — это *определение* плохого дизайна.

Тесты необходимо разрабатывать так же тщательно, как и любую другую часть системы. Все правила проектирования производственного кода применимы и к тестам. В этом отношении тесты не являются

чем-то особенным. Корректное проектирование позволяет ограничить их хрупкость.

Ранние руководства по TDD зачастую игнорировали эти аспекты. Более того, в них порой рекомендовались структуры, противоречащие хорошему дизайну и, соответственно, порождавшие тесты, тесно связанные с производственным кодом, то есть очень хрупкие.

Однозначное соответствие

Одна из распространенных и особенно вредных практик — создание и поддержание однозначного соответствия между модулями производственного кода и тестовыми модулями. Новичков в TDD часто неправильно учат, что каждому модулю или классу с именем χ должен соответствовать тестовый модуль или класс с именем `χTest`.

Но такой подход создает мощную структурную связь между производственным кодом и набором тестов, делая этот набор хрупким. Любое редактирование модульной структуры производственного кода неизбежно сопровождается внесением изменений в модульную структуру тестового кода.

Пример такого структурного связывания показан на рис. 4.7.

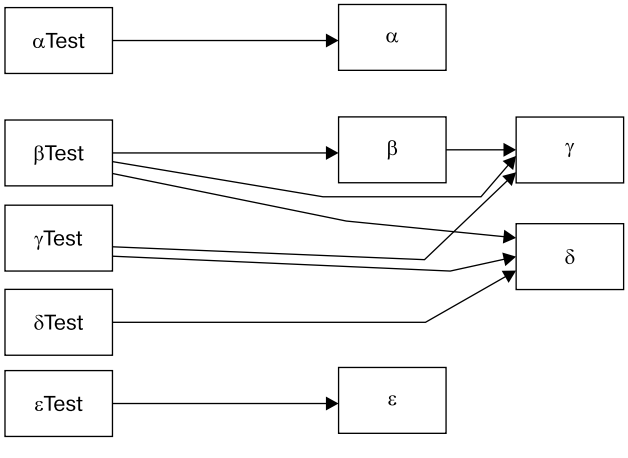


Рис. 4.7. Структурное связывание

В правой части диаграммы — пять модулей производственного кода: α, β, γ, δ и ε. Модули α и ε стоят особняком. А вот модуль β связан с модулем γ, который, в свою очередь, связан с модулем δ. Слева располагаются тестовые модули. Обратите внимание, что каждый из них связан с соответствующим модулем производственного кода. Но из-за связи β с γ и δ модуль βTest также может быть связан с γ и δ.

Связь при этом далеко не всегда очевидна. Модуль βTest, например, может быть связан с модулями γ и δ, поскольку они используются при конструировании модуля β. Или, скажем, методы модуля β могут принимать модули γ и δ в качестве аргументов.

Мощная связь модуля βTest с большей частью производственного кода означает, что незначительное изменение в модуле δ может повлиять на модули βTest, γTest и δTest. Однозначное соответствие между тестами и производственным кодом приводит к очень тесной связанности и хрупкости.

Правило 12. Отделяйте структуру тестов от структуры производственного кода.

Разрыв соответствия

Чтобы убрать соответствие между тестами и производственным кодом или не допустить его появления, тестовые модули следует воспринимать так же, как и все прочие модули в программной системе: как независимые и несвязанные.

Поначалу может показаться, что тут есть логическое противоречие. Раз тесты *проверяют* рабочий код, то просто должны быть с ним связаны. Это действительно так, но дело в том, что проверка кода не предполагает сильной связи. Хорошие проектировщики, разрабатывая способы взаимодействия модулей, всеми силами стараются избегать их связанности.

Как это достигается? С помощью промежуточных слоев.

На диаграмме, показанной на рис. 4.8, мы видим, что модуль `αTest` связан с модулем `α`. После `α` идет целое семейство модулей, которые поддерживают работу `α`, но о которых модуль `αTest` не знает. Модуль `α` является интерфейсом сопряжения с этим семейством. Хороший программист тщательно следит за тем, чтобы ни одна из деталей реализации модулей семейства не вышла за этот промежуточный слой.

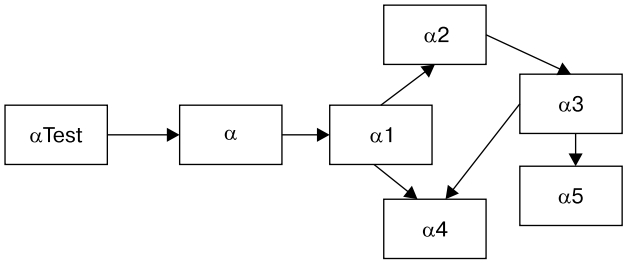


Рис. 4.8. Промежуточный слой

Как показано на диаграмме, отображенной на рис. 4.9, экранировать модуль `αTest` от деталей реализации семейства `α` можно путем вставки полиморфного интерфейса. Это разрушает любые транзитивные зависимости между тестовым модулем и модулями производственного кода.

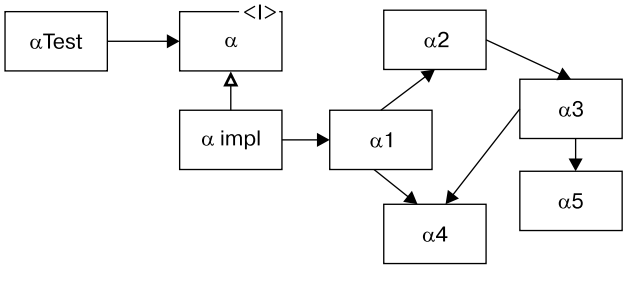


Рис. 4.9. Полиморфный интерфейс между тестом и семейством α

Еще раз повторю, что новичку в TDD такая ситуация может показаться абсурдной. У него сразу возникает вопрос: как можно писать тесты для модуля `a5`, если к нему нельзя получить доступ из модуля `aTest`? Ответ очень прост: вам не нужен доступ к модулю `a5`, чтобы протестировать его функциональность.

Если модуль `a5` выполняет важную функцию для интерфейса `a`, то ее необходимо тестировать через этот интерфейс. Это не произвольное правило, а математически доказанное утверждение. Важные виды поведения также должны быть видимы через интерфейс. Эта видимость может быть прямой или косвенной, но она должна существовать.

Наверное, самым наглядным будет объяснение на примере.

Магазин видеопроката

Традиционный пример, хорошо демонстрирующий концепцию отделения тестов от производственного кода, — расчет суммы чека в магазине видеопроката. Как ни странно, этот пример родился случайно. Впервые подобная задача была использована для демонстрации рефакторинга в первом издании книги Мартина Фаулера «Рефакторинг». Мартин взял довольно уродливое решение на языке Java, а затем показал, как его можно улучшить с помощью рефакторинга.

Мы же напишем программу с нуля с помощью TDD. Все требования будут излагаться по ходу дела.

Требование 1: первый день проката обычного фильма стоит 1,5 доллара, кроме того, за каждый день проката начисляется 1 бонусный балл.

Красный этап: напишем тестовый класс `CustomerTest` для клиента магазина и добавим туда первый тестовый метод.

```
public class CustomerTest {
  @Test
  public void RegularMovie_OneDay() throws Exception {
    Customer c = new Customer();
```

```
    c.addRental("RegularMovie", 1);
    assertEquals(1.5, c.getRentalFee(), 0.001);
    assertEquals(1, c.getRenterPoints());
  }
}
```

Зеленый этап: обеспечить прохождение такого теста очень легко.

```
public class Customer {
  public void addRental(String title, int days) {
  }

  public double getRentalFee() {
    return 1.5;
  }

  public int getRenterPoints() {
    return 1;
  }
}
```

Рефакторинг: теперь можно немного почистить код.

```
public class CustomerTest {
  private Customer customer;

  @Before
  public void setUp() throws Exception {
    customer = new Customer();
  }

  private void assertFeeAndPoints(double fee, int points) {
    assertEquals(fee, customer.getRentalFee(), 0.001);
    assertEquals(points, customer.getRenterPoints());
  }

  @Test
  public void RegularMovie_OneDay() throws Exception {
    customer.addRental("RegularMovie", 1);
    assertFeeAndPoints(1.5, 1);
  }
}
```

Требование 2: второй и третий дни проката обычных фильмов бесплатны, бонусные баллы за них не начисляются.

Зеленый этап: производственный код остается без изменений.

```
@Test
public void RegularMovie_SecondAndThirdDayFree() throws Exception {
  customer.addRental("RegularMovie", 2);
  assertFeeAndPoints(1.5, 1);
  customer.addRental("RegularMovie", 3);
  assertFeeAndPoints(1.5, 1);
}
```

Требование 3: все последующие дни плата за прокат составляет 1,5 доллара и начисляется 1 бонусный балл.

Красный этап: пишем простой тест.

```
@Test
public void RegularMovie_FourDays() throws Exception {
  customer.addRental("RegularMovie", 4);
  assertFeeAndPoints(3.0, 2);
}
```

Зеленый этап: для прохождения теста вносим несложные дополнения в код.

```
public class Customer {
  private int days;

  public void addRental(String title, int days) {
    this.days = days;
  }

  public double getRentalFee() {
    double fee = 1.5;
    if (days > 3)
      fee += 1.5 * (days - 3);
    return fee;
  }

  public int getRenterPoints() {
    int points = 1;
    if (days > 3)
      points += (days - 3);
    return points;
  }
}
```

Рефакторинг: есть небольшое дублирование, которое можно устранить, но это вызовет кое-какие проблемы.

```
public class Customer {
  private int days;

  public void addRental(String title, int days) {
    this.days = days;
  }

  public int getRentalFee() {
    return applyGracePeriod(150, 3);
  }

  public int getRenterPoints() {
    return applyGracePeriod(1, 3);
  }

  private int applyGracePeriod(int amount, int grace) {
    if (days > grace)
      return amount + amount * (days - grace);
    return amount;
  }
}
```

Красный этап: я хочу использовать метод `applyGracePeriod` как для бонусных баллов клиента, так и для стоимости проката, при этом переменная `fee` имеет тип `double`, а переменная `points` — тип `int`. Но для стоимости проката не может использоваться тип `double`! Поэтому мы меняем тип этой переменной на `int`, что приводит к сбою всех тестов. Нужно вернуться назад и отредактировать их.

```
public class CustomerTest {
  private Customer customer;

  @Before
  public void setUp() throws Exception {
    customer = new Customer();
  }

  private void assertFeeAndPoints(int fee, int points) {
    assertEquals(fee, customer.getRentalFee());
    assertEquals(points, customer.getRenterPoints());
```

```
  }

  @Test
  public void RegularMovie_OneDay() throws Exception {
    customer.addRental("RegularMovie", 1);
    assertFeeAndPoints(150, 1);
  }

  @Test
  public void RegularMovie_SecondAndThirdDayFree() throws Exception {
    customer.addRental("RegularMovie", 2);
    assertFeeAndPoints(150, 1);
    customer.addRental("RegularMovie", 3);
    assertFeeAndPoints(150, 1);
  }

  @Test
  public void RegularMovie_FourDays() throws Exception {
    customer.addRental("RegularMovie", 4);
    assertFeeAndPoints(300, 2);
  }
}
```

Требование 4: прокат детских фильмов стоит 1 доллар в день и приносит 1 бонусный балл.

Красный этап: бизнес-правило первого дня тестируется очень просто:

```
@Test
public void ChildrensMovie_OneDay() throws Exception {
  customer.addRental("ChildrensMovie", 1);
  assertFeeAndPoints(100, 1);
}
```

Зеленый этап: прохождение этого теста нам легко обеспечит вот такой некрасивый код:

```
public int getRentalFee() {
  if (title.equals("RegularMovie"))
    return applyGracePeriod(150, 3);
  else
    return 100;
}
```

Рефакторинг: теперь это безобразие нужно привести в пристойный вид. Тип видео никак не должен быть связан с заголовком, поэтому создадим реестр.

```
public class Customer {
  private String title;
  private int days;
  private Map<String, VideoType> movieRegistry = new HashMap<>();

  enum VideoType {REGULAR, CHILDRENS};

  public Customer() {
    movieRegistry.put("RegularMovie", REGULAR);
    movieRegistry.put("ChildrensMovie", CHILDRENS);
  }

  public void addRental(String title, int days) {
    this.title = title;
    this.days = days;
  }

  public int getRentalFee() {
    if (getType(title) == REGULAR)
      return applyGracePeriod(150, 3);
    else
      return 100;
  }

  private VideoType getType(String title) {
    return movieRegistry.get(title);
  }

  public int getRenterPoints() {
    return applyGracePeriod(1, 3);
  }

  private int applyGracePeriod(int amount, int grace) {
    if (days > grace)
      return amount + amount * (days - grace);
    return amount;
  }
}
```

Теперь код выглядит лучше, но нарушает принцип единственной ответственности[1], поскольку класс `Customer` не должен отвечать за инициализацию реестра. Реестр следует инициализировать на ранних этапах настройки системы. Отделим его от класса `Customer`:

```
public class VideoRegistry {
  public enum VideoType {REGULAR, CHILDRENS}

  private static Map<String, VideoType> videoRegistry = new
HashMap<>();

  public static VideoType getType(String title) {
    return videoRegistry.get(title);
  }

  public static void addMovie(String title, VideoType type) {
    videoRegistry.put(title, type);
  }
}
```

`VideoRegistry` — это класс, имеющий всего одно состояние[2], то есть для него возможен лишь один экземпляр. Он статически инициализируется тестом:

```
@BeforeClass
public static void loadRegistry() {
  VideoRegistry.addMovie("RegularMovie", REGULAR);
  VideoRegistry.addMovie("ChildrensMovie", CHILDRENS);
}
```

И это сильно очищает класс `Customer`:

```
public class Customer {
  private String title;
  private int days;

  public void addRental(String title, int days) {
    this.title = title;
    this.days = days;
```

[1] *Мартин Р. С.* Чистая архитектура, искусство разработки программного обеспечения.

[2] *Мартин Р. С.* Быстрая разработка программ. Принципы, примеры, практика.

```
  }

  public int getRentalFee() {
    if (VideoRegistry.getType(title) == REGULAR)
      return applyGracePeriod(150, 3);
    else
      return 100;
  }

  public int getRenterPoints() {
    return applyGracePeriod(1, 3);
  }
  private int applyGracePeriod(int amount, int grace) {
    if (days > grace)
      return amount + amount * (days - grace);
    return amount;
  }
}
```

Красный этап: обратите внимание, что согласно требованию 4 клиент получает за прокат детского фильма всего 1 балл, а не 1 балл в день. Поэтому следующий тест будет выглядеть так:

```
@Test
public void ChildrensMovie_FourDays() throws Exception {
  customer.addRental("ChildrensMovie", 4);
  assertFeeAndPoints(400, 1);
}
```

Я выбрал период в четыре дня, так как `3` сейчас выступает вторым аргументом в функции `applyGracePeriod` внутри метода `getRenterPoints` класса `Customer`. (В TDD мы порой делаем нарочито наивные вещи, но на самом деле прекрасно понимаем, что происходит.)

Зеленый этап: благодаря реестру, мы легко обеспечим прохождение этого теста.

```
public int getRenterPoints() {
  if (VideoRegistry.getType(title) == REGULAR)
    return applyGracePeriod(1, 3);
  else
    return 1;
}
```

Хочу обратить ваше внимание на тот факт, что у нас нет тестов для класса `VideoRegistry`. Точнее, нет прямых тестов. Но косвенно он постоянно тестируется, ведь ни один из наших тестов не прошел бы, если бы класс `VideoRegistry` не функционировал должным образом.

Красный этап: пока что класс `Customer` умеет обрабатывать только один фильм. Напишем тест, проверяющий процесс обработки нескольких фильмов:

```
@Test
public void OneRegularOneChildrens_FourDays() throws Exception {
  customer.addRental("RegularMovie", 4); //$3+2p
  customer.addRental("ChildrensMovie", 4); //$4+1p

  assertFeeAndPoints(700, 3);
}
```

Зеленый этап: код для прохождения этого теста потребует всего лишь небольшого списка и пары циклов. Заодно неплохо переместить содержимое реестра в новый класс `Rental`:

```
public class Customer {
  private List<Rental> rentals = new ArrayList<>();

  public void addRental(String title, int days) {
    rentals.add(new Rental(title, days));
  }

  public int getRentalFee() {
    int fee = 0;
    for (Rental rental : rentals) {
      if (rental.type == REGULAR)
        fee += applyGracePeriod(150, rental.days, 3);
      else
        fee += rental.days * 100;
    }
    return fee;
  }

  public int getRenterPoints() {
    int points = 0;
    for (Rental rental : rentals) {
      if (rental.type == REGULAR)
```

```
        points += applyGracePeriod(1, rental.days, 3);
      else
        points++;
    }
    return points;
  }

  private int applyGracePeriod(int amount, int days, int grace) {
    if (days > grace)
      return amount + amount * (days - grace);
    return amount;
  }
}

public class Rental {
  public String title;
  public int days;
  public VideoType type;

  public Rental(String title, int days) {
    this.title = title;
    this.days = days;
    type = VideoRegistry.getType(title);
  }
}
```

Этот код не сможет пройти старый тест, поскольку в классе `Customer` теперь суммируются два платежа:

```
@Test
public void RegularMovie_SecondAndThirdDayFree() throws Exception {
  customer.addRental("RegularMovie", 2);
  assertFeeAndPoints(150, 1);
  customer.addRental("RegularMovie", 3);
  assertFeeAndPoints(150, 1);
}
```

Придется поделить этот тест на две части. Пожалуй, вот так будет лучше:

```
@Test
public void RegularMovie_SecondDayFree() throws Exception {
  customer.addRental("RegularMovie", 2);
  assertFeeAndPoints(150, 1);
}
```

```
@Test
public void RegularMovie_ThirdDayFree() throws Exception {
  customer.addRental("RegularMovie", 3);
  assertFeeAndPoints(150, 1);
}
```

Рефакторинг: теперь мне многое не нравится в классе `Customer`. Выделим из этих двух уродливых циклов со странными операторами `if` несколько более удобных методов.

```
public int getRentalFee() {
  int fee = 0;
  for (Rental rental : rentals)
    fee += feeFor(rental);
  return fee;
}

private int feeFor(Rental rental) {
  int fee = 0;
  if (rental.getType() == REGULAR)
    fee += applyGracePeriod(150, rental.getDays(), 3);
  else
    fee += rental.getDays() * 100;
  return fee;
}

public int getRenterPoints() {
  int points = 0;
  for (Rental rental : rentals)
    points += pointsFor(rental);
  return points;
}

private int pointsFor(Rental rental) {
  int points = 0;
  if (rental.getType() == REGULAR)
    points += applyGracePeriod(1, rental.getDays(), 3);
  else
    points++;
  return points;
}
```

Кажется, эти две закрытые функции больше работают с классом `Rental`, чем с классом `Customer`. Переместим их в этот класс вместе со

вспомогательной функцией `applyGracePeriod`, чтобы немного очистить класс `Customer`.

```
public class Customer {
  private List<Rental> rentals = new ArrayList<>();

  public void addRental(String title, int days) {
    rentals.add(new Rental(title, days));
  }

  public int getRentalFee() {
    int fee = 0;
    for (Rental rental : rentals)
      fee += rental.getFee();
    return fee;
  }

  public int getRenterPoints() {
    int points = 0;
    for (Rental rental : rentals)
      points += rental.getPoints();
    return points;
  }
}
```

Увеличившийся код класса `Rental` теперь выглядит совсем некрасиво:

```
public class Rental {
  private String title;
  private int days;
  private VideoType type;

  public Rental(String title, int days) {
    this.title = title;
    this.days = days;
    type = VideoRegistry.getType(title);
  }

  public String getTitle() {
    return title;
  }

  public VideoType getType() {
    return type;
  }
```

```
  public int getFee() {
    int fee = 0;
    if (getType() == REGULAR)
      fee += applyGracePeriod(150, days, 3);
    else
      fee += getDays() * 100;
    return fee;
  }

  public int getPoints() {
    int points = 0;
    if (getType() == REGULAR)
      points += applyGracePeriod(1, days, 3);
    else
      points++;
    return points;
  }

  private static int applyGracePeriod(int amount, int days, int
grace) {
    if (days > grace)
      return amount + amount * (days - grace);
    return amount;
  }
}
```

Надо бы избавиться от некрасивых операторов `if`. Каждый новый тип видео будет означать появление нового условия. Очистку мы начнем с некоторых подклассов и полиморфизма.

Перво-наперво, в абстрактном классе `Movie` находится вспомогательная функция `applyGracePeriod` и две абстрактные функции для получения суммы к оплате и количества бонусных баллов.

```
public abstract class Movie {
  private String title;

  public Movie(String title) {
    this.title = title;
  }

  public String getTitle() {
    return title;
```

```
  }

  public abstract int getFee(int days, Rental rental);
  public abstract int getPoints(int days, Rental rental);
  protected static int applyGracePeriod(int amount, int days,
int grace) {

    if (days > grace)
      return amount + amount * (days - grace);
    return amount;
  }
}
```

Класс `RegularMovie` очень простой:

```
public class RegularMovie extends Movie {
  public RegularMovie(String title) {
    super(title);
  }

  public int getFee(int days, Rental rental) {
    return applyGracePeriod(150, days, 3);
  }

  public int getPoints(int days, Rental rental) {
    return applyGracePeriod(1, days, 3);
  }
}
```

А класс `ChildrensMovie` еще проще:

```
public class ChildrensMovie extends Movie {
  public ChildrensMovie(String title) {
    super(title);
  }

  public int getFee(int days, Rental rental) {
    return days * 100;
  }

  public int getPoints(int days, Rental rental) {
    return 1;
  }
}
```

От класса `Rental` осталось немного — всего пара функций-делегатов:

```
public class Rental {
  private int days;
  private Movie movie;

  public Rental(String title, int days) {
    this.days = days;
    movie = VideoRegistry.getMovie(title);
  }

  public String getTitle() {
    return movie.getTitle();
  }

  public int getFee() {
    return movie.getFee(days, this);
  }

  public int getPoints() {
    return movie.getPoints(days, this);
  }
}
```

Класс `VideoRegistry` превратился в фабрику для класса `Movie`.

```
public class VideoRegistry {
  public enum VideoType {REGULAR, CHILDRENS;}

  private static Map<String, VideoType> videoRegistry =
                     new HashMap<>();

  public static Movie getMovie(String title) {
    switch (videoRegistry.get(title)) {
      case REGULAR:
        return new RegularMovie(title);
      case CHILDRENS:
        return new ChildrensMovie(title);
    }
    return null;
  }

  public static void addMovie(String title, VideoType type) {
    videoRegistry.put(title, type);
  }
}
```

А что с классом `Customer`? У него просто все это время было неправильное имя. А на самом деле это класс `RentalCalculator`, который экранирует наши тесты от семейства обслуживающих его классов.

```
public class RentalCalculator {
  private List<Rental> rentals = new ArrayList<>();

  public void addRental(String title, int days) {
    rentals.add(new Rental(title, days));
  }

  public int getRentalFee() {
    int fee = 0;
    for (Rental rental : rentals)
      fee += rental.getFee();
    return fee;
  }

  public int getRenterPoints() {
    int points = 0;
    for (Rental rental : rentals)
      points += rental.getPoints();
    return points;
  }
}
```

Полученный результат схематично представлен на рис. 4.10.

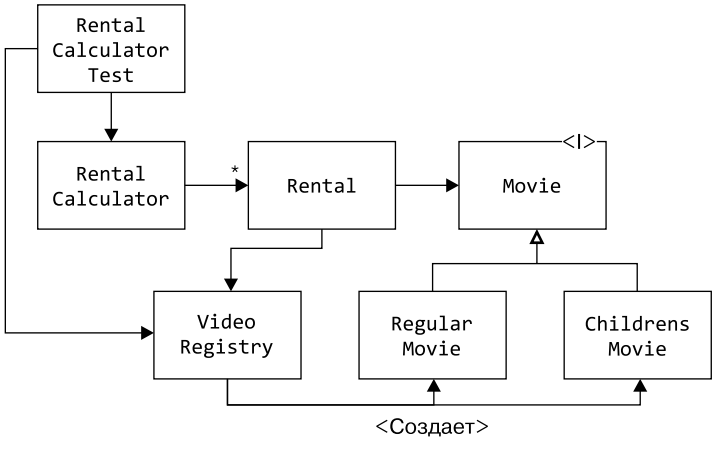


Рис. 4.10. Результат

Все классы справа от `RentalCalculator` создавались в процессе рефакторинга. Но класс `RentalCalculatorTest` ничего о них не знает. Ему известен только класс `VideoRegistry`, который он должен инициализировать тестовыми данными. Более того, эти классы не использует ни один другой тестовый модуль. Модуль `RentalCalculatorTest` косвенно тестирует все остальные классы. Мы убрали однозначное соответствие.

Именно так хорошие программисты экранируют структуру производственного кода от структуры тестов, тем самым избегая проблемы хрупких тестов.

Разумеется, в больших системах этот шаблон будет часто повторяться. Будут многочисленные семейства модулей, защищенных от проверяющих их тестовых модулей собственными фасадными методами или интерфейсными модулями.

Некоторые из вас могут предположить, что тесты, управляющие семейством модулей через фасадные методы, являются интеграционными. Но о них мы поговорим позже. Пока скажу только, что у интеграционных тестов совсем другая цель. Это *тесты для программистов*, написанные другими программистами в целях определения поведения.

Конкретика против общности

Существует еще один параметр разделения тестов и производственного кода. Я рассказывал об этом в главе 2 в примере с простыми множителями. Там я дал вам пару рекомендаций, которые сейчас сформулирую в виде правила.

> *Правило 13. По мере того как тесты становятся более конкретными, код становится более общим.*

Количество модулей производственного кода растет по мере увеличения количества тестов. Но развиваются тесты и производственный код в разных направлениях.

С добавлением каждого нового теста набор становится все более конкретным. А вот семейство тестируемых модулей должно развиваться в противоположном направлении — становиться все более обобщенным (рис. 4.11).

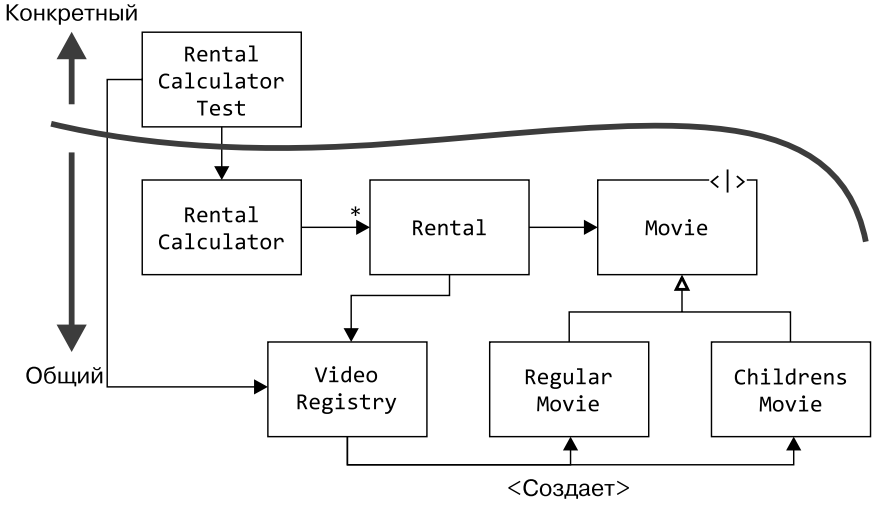


Рис. 4.11. Набор тестов становится более конкретным, а семейство тестируемых модулей — более общим

Это одна из целей рефакторинга. Как это реализуется, вы могли наблюдать в примере с магазином видеопроката. Сначала я написал тест. Затем — некрасивый код для прохождения этого теста. Данный код не был универсальным. Более того, он был глубоко конкретным. Однако на этапе рефакторинга я преобразовал его в более общую форму.

Такое дивергентное развитие тестов и производственного кода означает, что в итоге они приобретут разную форму. Тесты превратятся в линейный список ограничений и спецификаций. Производственный код вырастет в богатый набор логических конструкций разных видов поведения, организованных для взаимодействия с управляющей приложением базовой абстракцией.

Дивергентное развитие еще больше разделяет тесты и производственный код, экранируя каждую из частей от вносимых в другую часть изменений.

Разумеется, полностью разорвать связь не получится. И какие-то изменения в одних модулях всегда будут вызывать перемены в других. Наша цель — не в полном устранении таких изменений, а в их минимизации. И описанные выше методы эффективно справляются с этой задачей.

ОПРЕДЕЛЕНИЕ ОЧЕРЕДНОСТИ ПРЕОБРАЗОВАНИЙ

Итак, многочисленные примеры показали, что в процессе разработки через тестирование мы постепенно конкретизируем тесты, одновременно стремясь придать производственному коду как можно более общий вид. Но как совершаются эти преобразования?

Конкретизация тестов сводится к добавлению нового утверждения в существующий набор или к добавлению нового метода тестирования для упорядочивания, действия и последующего утверждения нового ограничения. Это аддитивные операции. Существующий тестовый код при этом не меняется. Мы просто добавляем новый код.

А вот когда после добавления нового ограничения мы редактируем производственный код, обеспечивая прохождение теста, — это уже не аддитивный процесс. Производственный код необходимо преобразовать, заставив его вести себя по-другому. То есть эти преобразования меняют поведение кода.

После этого производственный код подвергается рефакторингу в целях его очистки. Рефакторинг — это тоже изменение производственного кода, но с сохранением его поведения.

Надеюсь, вы уже обнаружили корреляцию с циклом «красный → зеленый → рефакторинг». Красный этап — аддитивный. Во время зеленого происходит редактирование. Последний этап — восстановительный.

О рефакторинге мы подробно поговорим в следующей главе, а пока подробно рассмотрим преобразования.

Это небольшое редактирование кода, меняющее его поведение и одновременно обобщающее решение. Их суть лучше всего объяснять на примере.

Вспомним упражнения на поиск простых множителей из главы 2. Мы начинали с проваленного теста и реализации для вырожденного случая.

```
public class PrimeFactorsTest {
  @Test
  public void factors() throws Exception {
    assertThat(factorsOf(1), is(empty()));
  }

  private List<Integer> factorsOf(int n) {
    return null;
  }
}
```

Чтобы обеспечить прохождение теста, я преобразовал значение `null` в константу `new ArrayList<>()`:

```
private List<Integer> factorsOf(int n) {
  return new ArrayList<>();
}
```

Это преобразование изменило поведение решения, попутно придав ему более общий вид. Значение `null` представляло собой чрезвычайно конкретный случай. Я же заменил его более универсальной константой.

Следующий проваленный тест также привел к обобщающим преобразованиям:

```
assertThat(factorsOf(2), contains(2));
```

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n>1)
    factors.add(2);
  return factors;
}
```

Для начала я выделил `ArrayList` в переменную `factor`, добавив оператор `if`. Оба этих преобразования являются обобщающими. Переменные всегда более универсальны, чем константы. Правда, условный оператор `if` выполняет лишь частичное обобщение. С одной стороны, в связи с тестом он рассматривает частный случай для множителей 1 и 2, но с другой стороны, эта конкретность смягчается неравенством `n>1`. Причем это неравенство фигурирует и в конечном варианте нашего обобщенного решения.

Вооружившись этими знаниями, посмотрим на другие преобразования.

{} → ничто

Обычно это самое первое преобразование, фигурирующее в начале сеанса TDD. Изначально код попросту отсутствует. Мы пишем тест для самого вырожденного случая, какой только можем придумать. Чтобы данный тест начал компилироваться, но при этом не проходил, мы заставляем тестируемую функцию возвращать значение `null`[1], как это было сделано в упражнении с простыми множителями.

```
private List<Integer> factorsOf(int n) {
  return null;
}
```

Этот код преобразует ничто в функцию, которая ничего не возвращает. Такое преобразование редко приводит к прохождению теста, поэтому обычно за ним сразу же следует другое преобразование.

Ничто → константа

Пример такого преобразования вы тоже видели в упражнении с простыми множителями. Возвращаемое значение `null` преобразовывалось в пустой список целых чисел.

---

[1] Или самое вырожденное допустимое значение.

```
private List<Integer> factorsOf(int n) {
  return new ArrayList<>();
}
```

Видели вы такое преобразование и в упражнении с подсчетом очков в боулинге в главе 2, хотя в этом случае стадия «{} → ничто» отсутствовала, так как мы сразу перешли к константе.

```
public int score() {
  return 0;
}
```

Константа → переменная

Следующий этап — это превращение константы в переменную. Мы видели такое преобразование в упражнении по созданию стека целых чисел (глава 2). Помните, я создал для значения `true`, возвращаемого утверждением `isEmpty`, переменную `empty`?

```
public class Stack {
  private boolean empty = true;

  public boolean isEmpty() {
    return empty;
  }
    . . .
}
```

Фигурировало это преобразование и в упражнении на поиск простых множителей, когда, чтобы пройти тест для значения `3`, я заменил константу `2` аргументом `n`.

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n>1)
    factors.add(n);
  return factors;
}
```

Думаю, вы уже поняли, что все рассмотренные на данный момент преобразования переводят код из очень конкретного состояния

в несколько более общее. Каждый раз это обобщение выступает способом сделать так, чтобы код стал обрабатывать более широкий набор ограничений.

Если хорошенько подумать, то становится понятным, что каждое из этих преобразований расширяет возможности гораздо больше, чем ограничение, которое накладывает на код текущий неудачный тест. И по мере применения таких преобразований гонка между ограничениями тестов и обобщениями кода должна закончиться в пользу обобщений. В итоге производственный код станет настолько универсальным, что сможет соответствовать всем будущим ограничениям в рамках текущих требований.

Но я отвлекся.

Отсутствие условий → выбор

Это преобразование добавляет оператор `if` или его эквивалент. Далеко не всегда это обобщение. Сделать так, чтобы условное выражение, которое должно обеспечить прохождение теста, не стало слишком конкретным, — задача программиста.

Это преобразование вы видели в упражнении с простыми множителями при разложении значения 2. Обратите внимание, что выбранное мной условие не `(n==2)`, так как это было бы слишком конкретно. Я попытался сделать этот оператор более универсальным, взяв в качестве условия неравенство `(n>1)`.

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n>1)
    factors.add(2);
  return factors;
}
```

Значение → список

Это обобщающее преобразование превращает переменную в список значений. Роль списка может играть массив или более сложный кон-

тейнер. Вы видели это преобразование в упражнении со стеком, когда я превратил переменную `element` в массив `elements`.

```
public class Stack {
  private int size = 0;
  private int[] elements = new int[2];

  public void push(int element) {
    this.elements[size++] = element;
  }

  public int pop() {
    if (size == 0)
      throw new Underflow();
    return elements[--size];
  }
}
```

Оператор → рекурсия

Это обобщающее преобразование делает оператор рекурсивным. Такого рода преобразования очень распространены в языках, поддерживающих рекурсию, особенно в Lisp и Logo, где циклы реализуются *исключительно* таким способом. Преобразование изменяет однократно вычисляемое выражение на выражение, вычисляемое на основе самого себя. Его пример вы видели в упражнении с разбиением на строки в главе 3.

```
private String wrap(String s, int w) {
  if (w >= s.length())
    return s;
  else
    return s.substring(0, w) + "\n" + wrap(s.substring(w), w);
}
```

Выбор → итерация

В упражнении на поиск простых множителей я несколько раз преобразовывал операторы `if` в цикл `while`. Это явное обобщение, поскольку итерация — обобщенный вариант выбора, а выбор — просто вырожденная итерация.

```
private List<Integer> factorsOf(int n) {
  ArrayList<Integer> factors = new ArrayList<>();
  if (n > 1) {
    while (n % 2 == 0) {
      factors.add(2);
    n /= 2;
    }
  }
  if (n > 1)
    factors.add(n);
  return factors;
}
```

Значение → измененное значение

Это преобразование изменяет значение переменной, обычно с целью накопления частичных значений в цикле или в инкрементных вычислениях. Примеры вы видели в нескольких упражнениях, и самое показательное из них, наверное, — упражнение на сортировку в главе 3.

Обратите внимание, что первые два присвоения — это обычная инициализация переменных `first` и `second`. А вот операции `list.set(...)` фактически изменяют элементы в списке.

```
private List<Integer> sort(List<Integer> list) {
  if (list.size() > 1) {
    if (list.get(0) > list.get(1)) {
      int first = list.get(0);
      int second = list.get(1);
      list.set(0, second);
      list.set(1, first);
    }
  }
  return list;
}
```

Пример: числа Фибоначчи

Попробуем выполнить простое упражнение и проследим за трансформациями. Напомню формулу расчета чисел Фибоначчи: `fib(0) = 1`, `fib(1) = 1` и `fib(n) = fib(n-1) + fib(n-2)`.

Начнем, как обычно, с написания теста. Почему я использую тип `BigInteger`? Дело в том, что числа Фибоначчи очень быстро становятся большими.

```
public class FibTest {
  @Test
  public void testFibs() throws Exception {
    assertThat(fib(0), equalTo(BigInteger.ONE));
  }

  private BigInteger fib(int n) {
    return null;
  }
}
```

Чтобы обеспечить прохождение теста, воспользуемся преобразованием «ничто → константа».

```
private BigInteger fib(int n) {
  return new BigInteger("1");
}
```

Мне тоже показался странным аргумент типа `String`, но в библиотеке Java это вот так.

Для прохождения следующего теста ничего не придется делать:

```
@Test
public void testFibs() throws Exception {
  assertThat(fib(0), equalTo(BigInteger.ONE));
  assertThat(fib(1), equalTo(BigInteger.ONE));
}
```

А вот этот тест уже не проходит:

```
@Test
public void testFibs() throws Exception {
  assertThat(fib(0), equalTo(BigInteger.ONE));
  assertThat(fib(1), equalTo(BigInteger.ONE));
  assertThat(fib(2), equalTo(new BigInteger("2")));
}
```

Для его прохождения выполним преобразование «отсутствие условия → выбор»:

```
private BigInteger fib(int n) {
  if (n > 1)
    return new BigInteger("2");
  else
    return new BigInteger("1");
}
```

Решение получается скорее конкретным, зато содержит потенциал для использования в функции `fib` отрицательных аргументов.

Следующий тест заставляет меня отправиться за золотом:

```
assertThat(fib(3), equalTo(new BigInteger("3")));
```

Для его прохождения используем преобразование «оператор → рекурсия»:

```
private BigInteger fib(int n) {
  if (n > 1)
    return fib(n-1).add(fib(n-2));
  else
    return new BigInteger("1");
}
```

Это очень элегантное решение требует чрезмерно больших затрат времени[1] и памяти. Слишком ранняя погоня за золотом часто дорого обходится. Можно ли выполнить то же самое другим способом?

Конечно да:

```
private BigInteger fib(int n) {
  return fib(BigInteger.ONE, BigInteger.ONE, n);
}

private BigInteger fib(BigInteger fm2, BigInteger fm1, int n) {
  if (n>1)
    return fib(fm1, fm1.add(fm2), n-1);
  else
    return fm1;

}
```

[1] На моем MacBook Pro с процессором 2,3 ГГц расчет fib(40)==165580141 занимает 9 секунд.

Это достаточно быстрый[1] алгоритм хвостовой рекурсии.

Может показаться, что последнее преобразование представляет собой вариант «оператор → рекурсия», но это не так. Фактически это преобразование «выбор → итерация». На самом деле, если бы компилятор Java соизволил предложить нам оптимизацию хвостовой рекурсии[2], то мы бы практически гарантированно получили следующий код. Обратите внимание на подразумеваемое преобразование `if` → `while`.

```
private BigInteger fib(int n) {
  BigInteger fm2 = BigInteger.ONE;
  BigInteger fm1 = BigInteger.ONE;
  while (n>1) {
    BigInteger f = fm1.add(fm2);
    fm2 = fm1;
    fm1 = f;
    n--;
  }
  return fm1;
}
```

Это небольшое отступление появилось, чтобы сделать важное замечание:

> *Правило 14. Если одно преобразование приводит к неоптимальному решению, то попробуйте другое преобразование.*

Мы уже во второй раз сталкиваемся с ситуацией, когда одно преобразование приводит к неоптимальному решению, в то время как другое дает гораздо лучшие результаты. Первый раз вы наблюдали это в упражнении на сортировку, где именно преобразование «значение → измененное значение» привело к пузырьковой сортировке. Его замена на преобразование «отсутствие условий → выбор» позволила реализовать быструю сортировку. Это был переломный шаг:

```
private List<Integer> sort(List<Integer> list) {
  if (list.size() <= 1)
    return list;
```

---

[1] Вычисление fib(100)==573147844013817084101 занимает 10 миллисекунд.

[2] Java, Java, почему же в тебе этого нет?

```
  else {
    int first = list.get(0);
    int second = list.get(1);
    if (first > second)
      return asList(second, first);
    else
      return asList(first, second);
  }
}
```

Определение очередности преобразований

Итак, что же делать с ситуацией, когда в процессе разработки через тестирование мы оказались на развилке? Направление дальнейшего движения зависит от того, каким преобразованием мы воспользуемся, чтобы обеспечить прохождение текущего теста. Есть ли способ выбрать лучшее преобразование? Как оценить, какое из преобразований лучше? Может быть, у них существует *приоритет*?

Я считаю, что приоритет преобразований существует. И сейчас о нем расскажу. Но сначала хотелось бы пояснить, что мое мнение в данном случае — всего лишь *постулат*. У меня нет математического доказательства, более того, я не уверен, что указанная последовательность работает во всех случаях. С относительной уверенностью могу утверждать только то, что вы, скорее всего, получите лучшую реализацию, если будете выбирать преобразования примерно в следующем порядке:

- {} → ничто;
- ничто → константа;
- константа → переменная;
- отсутствие условия → выбор;
- значение → список;
- выбор → итерация;
- оператор → рекурсия;
- значение → измененное значение.

Не стоит думать, что это закономерный порядок, который нельзя нарушать (например, прибегать к преобразованию «константа → переменная» до завершения преобразования «ничто → константа»). Иногда тест можно пройти, скажем, преобразовав ничто в *выбор* из двух констант и вообще опустив шаг «ничто → константа».

Другими словами, если вы испытываете искушение пройти тест, комбинируя два или более преобразования, то можно пропустить один или несколько тестов. Попробуйте найти тест, который можно пройти, используя только одно из этих преобразований, а затем, оказавшись на развилке, выберите путь, на который ведет преобразование, расположенное *выше* по списку.

Всегда ли работает этот механизм? Наверное, нет, но мне обычно с ним очень везло. Как вы могли убедиться, описанный подход позволил получить лучший результат как для алгоритмов сортировки, так и для алгоритмов Фибоначчи.

Проницательный читатель уже понял, что указанный порядок преобразований приводит нас к реализации решений в стиле функционального программирования.

РЕЗЮМЕ

На этом мы завершаем обсуждение TDD. В последних трех главах мы обсудили множество вопросов. Например, эта глава была посвящена проблемам проектирования тестов и различным шаблонам проектирования.

От графических интерфейсов до баз данных, от конкретики до обобщений и от преобразований до их приоритета. Разумеется, это далеко не все. Еще есть четвертый закон, который нужно соблюдать. Он и будет темой следующей главы.

5 РЕФАКТОРИНГ

В 1999 году я прочитал «Рефакторинг» Мартина Фаулера. Эта книга уже стала классикой, и я призываю вас обязательно ознакомиться с ней. Недавно он опубликовал второе издание, значительно дополненное и модернизированное. Первое издание содержало примеры на языке Java, а второе — на языке JavaScript.

Я читал первое издание, когда мой двенадцатилетний сын Джастин играл в хоккейной команде. Для тех из вас, у кого нет ребенка-хоккеиста, расскажу, что собственно игра занимает пять минут, а потом дети проводят десять-пятнадцать минут вне льда, чтобы остыть.

И вот в этих перерывах между играми я читал замечательную книгу Мартина. Это была первая книга, в которой код представлялся чем-то *податливым*. Большинство изданий того периода давали код в окончательной форме. В книге же Мартина демонстрировалось, как взять плохой код и очистить его.

В процессе чтения я слышал, как родители болели за детей на льду. Я тоже болел, но не за игру. Меня полностью захватила книга у меня в руках. Во многом именно она стала предпосылкой для написания «Чистого кода».

Никто не сказал лучше Мартина:

> *Написать код, понятный компьютеру, может кто угодно. Хорошие программисты пишут код, понятный людям.*

В этой главе я познакомлю вас с моей точкой зрения на искусство рефакторинга. Но она не заменит вам знакомства с книгой Мартина.

ЧТО ТАКОЕ РЕФАКТОРИНГ

На этот раз я перефразирую Мартина:

> *Рефакторинг — это последовательность небольших изменений, улучшающих структуру программного обеспечения без изменения его поведения, что подтверждается прохождением комплексного набора тестов после каждого изменения в последовательности.*

В этом определении есть два важных момента.

Во-первых, *сохранение* поведения. После одного или нескольких рефакторингов поведение программного обеспечения остается неизменным. Удостовериться в этом можно единственным способом: постоянно проводя *всеобъемлющее* тестирование.

Во-вторых, каждый отдельный рефакторинг имеет *маленький* размер. Насколько маленький? У меня есть принцип: *достаточно маленький, чтобы не пришлось прибегать к отладке.*

Существует множество методов рефакторинга, и некоторые из них я опишу на следующих страницах. Бывают и изменения кода, которые не считаются частью канонического рефакторинга, но представляют собой структурные изменения, сохраняющие поведение. Есть настолько шаблонные методы рефакторинга, что за вас их может сделать IDE. Некоторые из методов настолько просты, что их можно без опасений проводить вручную. Более сложные требуют значительного внимания. В этом случае я вспоминаю следующий принцип: если опасаюсь, что мне придется использовать отладчик, то разбиваю планируемое изменение на более мелкие и безопасные части. Если даже в этом случае не удается избежать отладки, то я провожу дальнейшее разбиение.

Правило 15. Избегайте использования отладчиков.

Цель рефакторинга — очистить код. Это часть цикла «красный → зеленый → рефакторинг». Это постоянная, а не запланированная и выполняемая по расписанию деятельность. Вы поддерживаете чистоту кода, выполняя рефакторинг на каждом витке цикла TDD.

Иногда возникают ситуации, когда требуется более масштабный рефакторинг. Рано или поздно вы неизбежно обнаружите, что структура системы нуждается в обновлении, и захотите отредактировать весь код. Это незапланированный процесс. Для проведения рефакторинга не нужно останавливать добавление функционала и исправление ошибок. Достаточно приложить немного дополнительных усилий по рефакторингу к циклу «красный → зеленый → рефакторинг»

и постепенно внести нужные изменения, не прекращая остальной деятельности.

ОСНОВНОЙ ИНСТРУМЕНТАРИЙ

К некоторым методам рефакторинга я прибегаю чаще, чем к остальным. Для их автоматизации я использовал свою IDE. Рекомендую вам выучить эти приемы наизусть и понять тонкости их автоматизации вашей IDE.

Переименование

В моей книге «Чистый код» есть глава, посвященная правильному именованию. Об этом писали и многие другие[1]. Это очень важный аспект.

Дело в том, что правильно выбрать имя непросто. Поиск подходящего имени часто представляет собой процесс последовательных постепенных улучшений. Не бойтесь искать правильные имена. На ранней стадии существования проекта улучшайте их как можно чаще.

С течением времени менять имена становится все труднее. Все больше программистов их запоминают и плохо реагируют на внезапные переименования. Рано или поздно наступает момент, когда переименование важных классов и функций становится возможным только после обсуждения и поиска консенсуса.

Поэтому пока новый код не получил слишком широкой известности, экспериментируйте с именами. Часто переименовывайте классы и методы. В процессе вы обнаружите, что их нужно группировать по-другому. Придется перемещать методы из одного класса в другой, чтобы они соответствовали новым именам. Придется менять

[1] Другой хороший источник информации книга: *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем.

секционирование функций и классов в соответствии с новой схемой именования.

Словом, поиск лучших имен, скорее всего, окажет глубоко положительное влияние на способ разбиения кода на классы и модули.

Поэтому научитесь использовать рефакторинг **Rename** и практикуйте его как можно чаще.

Выделение методов

Метод рефакторинга **Extract Method**, возможно, — самый важный из всех. Мне кажется, именно этот механизм отвечает за поддержание чистоты и хорошую систематизацию кода.

Мой вам совет: *выделяйте, пока есть что выделять*.

Такой подход преследует достижение двух целей. Во-первых, каждая функция должна делать только одну вещь[1]. Во-вторых, код должен читаться как хорошо написанная проза[2].

Если функция выполняет *одно действие*, то никакую другую функцию выделить из нее уже не получится. Чтобы добиться такого эффекта, нужно выделять, выделять и выделять до тех пор, пока это возможно.

Разумеется, это породит множество крошечных функций. Может показаться, что среди этого изобилия непросто понять назначение кода. Да и в принципе в огромном рое функций легко заблудиться.

Однако на самом деле происходит обратная вещь. Назначение кода становится намного более очевидным. Появляются четкие уровни абстракции с понятными границами между ними.

Современные языки изобилуют модулями, классами и пространствами имен. Это позволяет строить иерархию имен, в которой разме-

[1] *Мартин Р. С.* Чистый код. — С. 30.

[2] Там же.

щаются функции. Классы находятся в пространствах имен и, в свою очередь, содержат функции. Закрытые функции используются внутри открытых. Классы содержат внутренние и вложенные классы. И так далее. Используйте эту иерархию для создания структуры, позволяющей другим программистам легко находить написанные вами функции.

И обязательно выбирайте для них хорошие имена. Напомню, что длина имени функции должна быть обратно пропорциональна ее области видимости. Имена открытых функций должны быть относительно короткими. Закрытым функциям можно присваивать более длинные имена.

По мере выделения все новых функций их имена будут становиться все длиннее, поскольку цель каждой следующей выделенной функции — становиться все менее общей. Большинство этих функций будут вызываться только из одного места, соответственно, их назначение будет чрезвычайно конкретным. Имена таких специализированных функций должны быть длинными. Скорее всего, это будут целые выражения или даже предложения.

Вызываться они будут внутри циклов `while` и операторов `if`. Возможны и вызовы из тел этих операторов, порождающие вот такой код:

```
if (employeeShouldHaveFullBenefits())
  AddFullBenefitsToEmployee();
```

В результате ваш код будет читаться *как хорошо написанная проза*.

Выделение методов позволяет следовать *правилу понижения* (step-down rule)[1]. Нужно сделать так, чтобы по мере чтения списка функций мы последовательно опускались по уровням абстракции. Для этого мы выделяем все фрагменты кода из функции, расположенной ниже желаемого уровня.

---

[1] Там же. — С. 61.

Выделение переменной

Если метод **Extract Method** считается самым важным из вариантов рефакторинга, то метод **Extract Variable** — его помощник. Оказывается, процесс выделения методов часто приходится начинать с выделения переменных.

В качестве примера рассмотрим рефакторинг из упражнения по созданию алгоритма подсчета очков в боулинге, которое мы выполняли в главе 2. Вначале был вот такой код:

```
@Test
public void allOnes() throws Exception {
  for (int i=0; i<20; i++)
    g.roll(1);
  assertEquals(20, g.score());
}
```

Затем он стал вот таким:

```
private void rollMany(int n, int pins) {
  for (int i = 0; i < n; i++) {
    g.roll(pins);
  }
}

@Test
public void allOnes() throws Exception {
  rollMany(20, 1);
  assertEquals(20, g.score());
}
```

В процессе рефакторинга были выполнены следующие действия.

1. **Extract Variable:** значение `1` из метода `g.roll(1)` было выделено в переменную `pins`.
2. **Extract Variable:** значение `20` из утверждения `assertEquals(20, g.score())` было выделено в переменную `n`.
3. Обе переменные после перемещения оказались над циклом `for`.
4. **Extract Method:** цикл `for` был выделен в функцию `rollMany`. Имена переменных стали именами аргументов.

5. Встраивание: обе переменные были встроены. Они выполнили свою задачу и больше не нужны.

Еще метод **Extract Variable** часто используется для создания независимой, или *объясняющей, переменной* (explanatory variable)[1]. Например, рассмотрим вот такой оператор `if`:

```
if (employee.age > 60 && employee.salary > 150000)
  ScheduleForEarlyRetirement(employee);
```

С помощью объясняющей переменной его можно сделать более читабельным:

```
boolean isEligibleForEarlyRetirement = employee.age > 60 && employee.
salary > 150000
if (isEligibleForEarlyRetirement)
  ScheduleForEarlyRetirement(employee);
```

Выделение поля

Этот метод рефакторинга может оказывать глубоко положительный эффект. Я использую его нечасто, но каждый раз, когда я к нему прибегаю, код существенно улучшается.

Все начинается с неудачного выделения метода. Рассмотрим класс, преобразующий в отчет CSV-файл с данными. Его код несколько не упорядочен.

```
public class NewCasesReporter {
  public String makeReport(String countyCsv) {
    int totalCases = 0;
    Map<String, Integer> stateCounts = new HashMap<>();
    List<County> counties = new ArrayList<>();

    String[] lines = countyCsv.split("\n");
    for (String line : lines) {
      String[] tokens = line.split(",");
      County county = new County();
```

[1] *Beck K.* Smalltalk Best Practice Patterns. — Addison-Wesley, 1997. P. 108.

```
        county.county = tokens[0].trim();
        county.state = tokens[1].trim();
        // вычисление скользящего среднего
        int lastDay = tokens.length - 1;
        int firstDay = lastDay - 7 + 1;
        if (firstDay < 2)
          firstDay = 2;
        double n = lastDay - firstDay + 1;
        int sum = 0;
        for (int day = firstDay; day <= lastDay; day++)
          sum += Integer.parseInt(tokens[day].trim());
        county.rollingAverage = (sum / n);

        // вычисление суммы случаев
        int cases = 0;
        for (int i = 2; i < tokens.length; i++)
          cases += (Integer.parseInt(tokens[i].trim()));
        totalCases += cases;
        int stateCount = stateCounts.getOrDefault(county.state, 0);
        stateCounts.put(county.state, stateCount + cases);
        counties.add(county);
      }
      StringBuilder report = new StringBuilder("" +
        "County State Avg New Cases\n" +
        "====== ===== =============\n");
      for (County county : counties) {
        report.append(String.format("%-11s%-10s%.2f\n",
          county.county,
          county.state,
          county.rollingAverage));
      }
      report.append("\n");
      TreeSet<String> states = new TreeSet<>(stateCounts.keySet());
      for (String state : states)
        report.append(String.format("%s cases: %d\n",
          state, stateCounts.get(state)));
      report.append(String.format("Total Cases: %d\n", totalCases));
      return report.toString();
    }

    public static class County {
      public String county = null;
      public String state = null;
      public double rollingAverage = Double.NaN;
    }
  }
```

К счастью для нас, автор написал и несколько тестов. Они не очень хороши, но все равно пригодятся.

```
public class NewCasesReporterTest {
  private final double DELTA = 0.0001;
  private NewCasesReporter reporter;

  @Before
  public void setUp() throws Exception {
    reporter = new NewCasesReporter();
  }

  @Test
  public void countyReport() throws Exception {
    String report = reporter.makeReport("" +
      "c1, s1, 1, 1, 1, 1, 1, 1, 1, 7\n" +
      "c2, s2, 2, 2, 2, 2, 2, 2, 2, 7");
    assertEquals("" +
      "County     State     Avg New Cases\n" +
      "======     =====     =============\n" +
      "c1         s1        1.86\n" +
      "c2         s2        2.71\n\n" +
      "s1 cases: 14\n" +
      "s2 cases: 21\n" +
      "Total Cases: 35\n",
    report);
  }

  @Test
  public void stateWithTwoCounties() throws Exception {
    String report = reporter.makeReport("" +
      "c1, s1, 1, 1, 1, 1, 1, 1, 1, 7\n" +
      "c2, s1, 2, 2, 2, 2, 2, 2, 2, 7");
    assertEquals("" +
      "County       State      Avg New Cases\n" +
      "======       =====      =============\n" +
      "c1           s1         1.86\n" +
      "c2           s1         2.71\n\n" +
      "s1 cases: 35\n" +
      "Total Cases: 35\n",
    report);
  }

  @Test
```

```
  public void statesWithShortLines() throws Exception {
    String report = reporter.makeReport("" +
      "c1, s1, 1, 1, 1, 1, 7\n" +
      "c2, s2, 7\n");
    assertEquals("" +
      "County       State       Avg New Cases\n" +
      "======       =====       =============\n" +
      "c1           s1          2.20\n" +
      "c2           s2          7.00\n\n" +
      "s1 cases: 11\n" +
      "s2 cases: 7\n" +
      "Total Cases: 18\n",
    report);
  }
}
```

Тесты дают хорошее представление о том, что делает программа. На ввод подается строка CSV, в которой указан округ и список новых случаев COVID в день. На выходе формируется отчет, демонстрирующий скользящее среднее новых случаев в округе за неделю и предоставляющий итоговые цифры для каждого штата, а также общий итог.

Мне сразу захотелось приступить к выделению методов из этой большой ужасной функции. Начнем с верхнего цикла. Он выполняет все расчеты по округам, поэтому, вероятно, имеет смысл назвать его `calculateCounties`.

Но стоило мне выделить этот цикл и попытаться выделить из него метод, как появилось окно диалога, показанное на рис. 5.1.

Моя IDE предложила назвать функцию `getTotalCases`. Надо отдать должное авторам IDE. Они приложили множество усилий, чтобы добавить такой функционал. Такое имя IDE выбрала потому, что коду после цикла требуется количество новых случаев, а получить это значение он может, только если эта новая функция его вернет.

Но я не хочу вызывать функцию `getTotalCases`. Я предназначил ее для другой цели, и она должна называться `calculateCounties`. Более того, я не хочу передавать эти четыре аргумента. Я хочу передать выделенной функции массив `lines`.

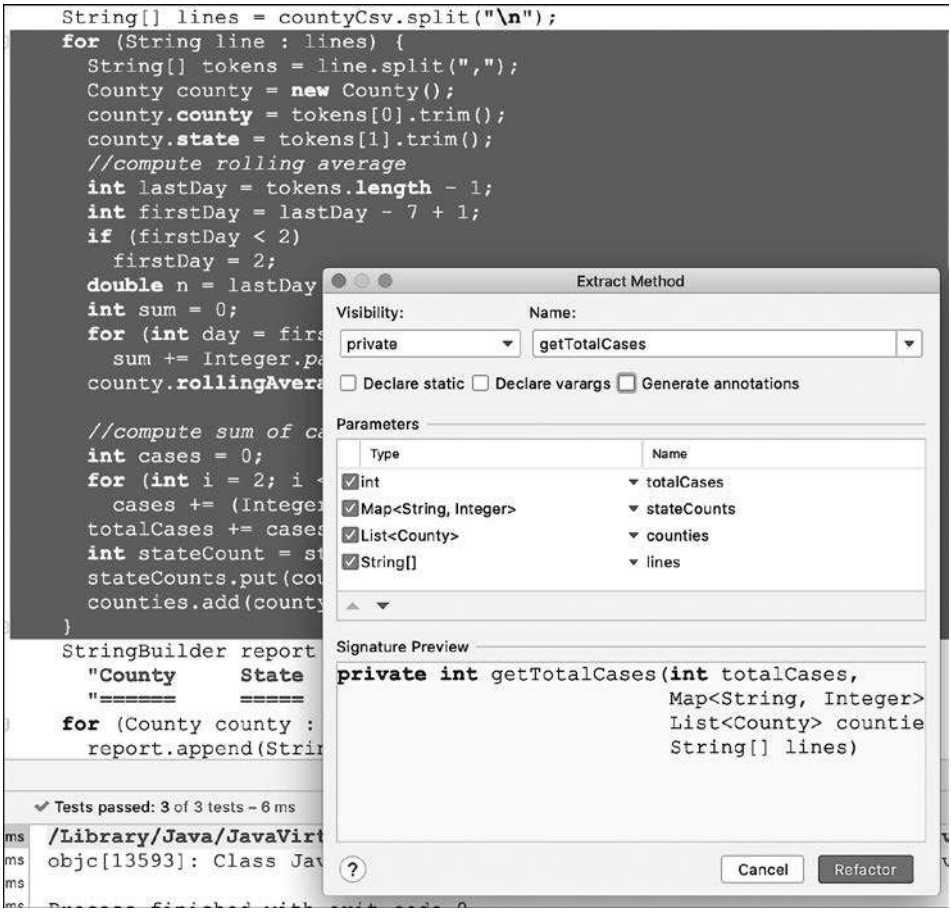


Рис. 5.1. Окно диалога Extract Method

Поэтому я нажимаю кнопку Cancel и еще раз смотрю на код.

Для корректного проведения рефакторинга нужно выделить некоторые локальные переменные цикла в поля класса. Для этого я прибегну к методу **Extract Field**:

```
public class NewCasesReporter {
  private int totalCases;
  private final Map<String, Integer> stateCounts = new HashMap<>();
```

```
  private final List<County> counties = new ArrayList<>();

  public String makeReport(String countyCsv) {
    totalCases = 0;
    stateCounts.clear();
    counties.clear();

    String[] lines = countyCsv.split("\n");
    for (String line : lines) {
      String[] tokens = line.split(",");
      County county = new County();
```

Обратите внимание, что начальные значения этим переменным присваиваются в верхней части функции `makeReport`. Это сохраняет изначальное поведение.

Теперь выделить цикл можно без передачи большего, чем мне нужно, количества переменных и без возвращения функции `totalCases`:

```
public class NewCasesReporter {
  private int totalCases;
  private final Map<String, Integer> stateCounts = new HashMap<>();
  private final List<County> counties = new ArrayList<>();

  public String makeReport(String countyCsv) {
    String[] countyLines = countyCsv.split("\n");
    calculateCounties(countyLines);

  StringBuilder report = new StringBuilder("" +
    "County     State     Avg New Cases\n" +
    "======     =====     =============\n");
  for (County county : counties) {
    report.append(String.format("%-11s%-10s%.2f\n",
      county.county,
      county.state,
      county.rollingAverage));
  }
  report.append("\n");
  TreeSet<String> states = new TreeSet<>(stateCounts.keySet());
  for (String state : states)
    report.append(String.format("%s cases: %d\n",
      state, stateCounts.get(state)));
  report.append(String.format("Total Cases: %d\n", totalCases));
  return report.toString();
}
```

```
private void calculateCounties(String[] lines) {
  totalCases = 0;
  stateCounts.clear();
  counties.clear();

  for (String line : lines) {
    String[] tokens = line.split(",");
    County county = new County();
    county.county = tokens[0].trim();
    county.state = tokens[1].trim();
    // вычисление скользящего среднего
    int lastDay = tokens.length - 1;
    int firstDay = lastDay - 7 + 1;
    if (firstDay < 2)
      firstDay = 2;
    double n = lastDay - firstDay + 1;
    int sum = 0;
    for (int day = firstDay; day <= lastDay; day++)
      sum += Integer.parseInt(tokens[day].trim());
    county.rollingAverage = (sum / n);

    // вычисление суммы случаев.
    int cases = 0;
    for (int i = 2; i < tokens.length; i++)
      cases += (Integer.parseInt(tokens[i].trim()));
      totalCases += cases;
      int stateCount = stateCounts.getOrDefault(county.state, 0);
      stateCounts.put(county.state, stateCount + cases);
      counties.add(county);
    }
  }

  public static class County {
    public String county = null;
    public String state = null;
    public double rollingAverage = Double.NaN;
  }
}
```

Теперь, когда эти переменные стали полями, можно продолжить их выделение и переименование.

```
public class NewCasesReporter {
  private int totalCases;
  private final Map<String, Integer> stateCounts = new HashMap<>();
```

```
private final List<County> counties = new ArrayList<>();

public String makeReport(String countyCsv) {
  String[] countyLines = countyCsv.split("\n");
  calculateCounties(countyLines);

  StringBuilder report = makeHeader();
  report.append(makeCountyDetails());
  report.append("\n");
  report.append(makeStateTotals());
  report.append(String.format("Total Cases: %d\n", totalCases));
  return report.toString();
}

private void calculateCounties(String[] countyLines) {
  totalCases = 0;
  stateCounts.clear();
  counties.clear();
  for (String countyLine : countyLines)
    counties.add(calcluateCounty(countyLine));
}

private County calcluateCounty(String line) {
  County county = new County();
  String[] tokens = line.split(",");
  county.county = tokens[0].trim();
  county.state = tokens[1].trim();

  county.rollingAverage = calculateRollingAverage(tokens);

  int cases = calculateSumOfCases(tokens);
  totalCases += cases;
  incrementStateCounter(county.state, cases);

  return county;
}

private double calculateRollingAverage(String[] tokens) {
  int lastDay = tokens.length - 1;
  int firstDay = lastDay - 7 + 1;
  if (firstDay < 2)
    firstDay = 2;
  double n = lastDay - firstDay + 1;
  int sum = 0;
  for (int day = firstDay; day <= lastDay; day++)
    sum += Integer.parseInt(tokens[day].trim());
```

```
    return (sum / n);
  }

  private int calculateSumOfCases(String[] tokens) {
    int cases = 0;
    for (int i = 2; i < tokens.length; i++)
      cases += (Integer.parseInt(tokens[i].trim()));
    return cases;
  }

  private void incrementStateCounter(String state, int cases) {
    int stateCount = stateCounts.getOrDefault(state, 0);
    stateCounts.put(state, stateCount + cases);
  }

  private StringBuilder makeHeader() {
    return new StringBuilder("" +
      "County     State     Avg New Cases\n" +
      "======     =====     =============\n");
  }

  private StringBuilder makeCountyDetails() {
    StringBuilder countyDetails = new StringBuilder();
    for (County county : counties) {
      countyDetails.append(String.format("%-11s%-10s%.2f\n",
        county.county,
        county.state,
        county.rollingAverage));
    }
    return countyDetails;
  }

  private StringBuilder makeStateTotals() {
    StringBuilder stateTotals = new StringBuilder();
    TreeSet<String> states = new TreeSet<>(stateCounts.keySet());
    for (String state : states)
      stateTotals.append(String.format("%s cases: %d\n",
        state, stateCounts.get(state)));
    return stateTotals;
  }

  public static class County {
    public String county = null;
    public String state = null;
    public double rollingAverage = Double.NaN;
  }
}
```

Теперь класс выглядит намного лучше, но мне не нравится то, что код, форматирующий отчет, находится в одном классе с кодом, вычисляющим данные. Это нарушение принципа единственной ответственности, поскольку формат отчета и расчеты, скорее всего, будут меняться по разным причинам.

Чтобы вытащить отвечающую за вычисления часть кода в новый класс, воспользуемся методом **Extract Superclass**. Он выделит вычисления в суперкласс `NewCasesCalculator`, а класс `NewCasesReporter` будет от него производным.

```
public class NewCasesCalculator {
  protected final Map<String, Integer> stateCounts = new HashMap<>();
  protected final List<County> counties = new ArrayList<>();
  protected int totalCases;

  protected void calculateCounties(String[] countyLines) {
    totalCases = 0;
    stateCounts.clear();
    counties.clear();

    for (String countyLine : countyLines)
      counties.add(calcluateCounty(countyLine));
  }

  private County calcluateCounty(String line) {
    County county = new County();
    String[] tokens = line.split(",");
    county.county = tokens[0].trim();
    county.state = tokens[1].trim();

    county.rollingAverage = calculateRollingAverage(tokens);

    int cases = calculateSumOfCases(tokens);
    totalCases += cases;
    incrementStateCounter(county.state, cases);

    return county;
  }

  private double calculateRollingAverage(String[] tokens) {
    int lastDay = tokens.length - 1;
    int firstDay = lastDay - 7 + 1;
    if (firstDay < 2)
```

```
      firstDay = 2;
    double n = lastDay - firstDay + 1;
    int sum = 0;
    for (int day = firstDay; day <= lastDay; day++)
      sum += Integer.parseInt(tokens[day].trim());
    return (sum / n);
  }

  private int calculateSumOfCases(String[] tokens) {
    int cases = 0;
    for (int i = 2; i < tokens.length; i++)
      cases += (Integer.parseInt(tokens[i].trim()));
    return cases;
  }

  private void incrementStateCounter(String state, int cases) {
    int stateCount = stateCounts.getOrDefault(state, 0);
    stateCounts.put(state, stateCount + cases);
  }

  public static class County {
    public String county = null;
    public String state = null;
    public double rollingAverage = Double.NaN;
  }
}

=======

public class NewCasesReporter extends NewCasesCalculator {
  public String makeReport(String countyCsv) {
    String[] countyLines = countyCsv.split("\n");
    calculateCounties(countyLines);

    StringBuilder report = makeHeader();
    report.append(makeCountyDetails());
    report.append("\n");
    report.append(makeStateTotals());
    report.append(String.format("Total Cases: %d\n", totalCases));
    return report.toString();
  }

  private StringBuilder makeHeader() {
    return new StringBuilder("" +
      "County     State     Avg New Cases\n" +
```

```
        "======     =====     =============\n");
    }

    private StringBuilder makeCountyDetails() {
      StringBuilder countyDetails = new StringBuilder();
      for (County county : counties) {
        countyDetails.append(String.format("%-11s%-10s%.2f\n",
          county.county,
          county.state,
          county.rollingAverage));
      }
      return countyDetails;
    }

    private StringBuilder makeStateTotals() {
      StringBuilder stateTotals = new StringBuilder();
      TreeSet<String> states = new TreeSet<>(stateCounts.keySet());
      for (String state : states)
        stateTotals.append(String.format("%s cases: %d\n",
          state, stateCounts.get(state)));
      return stateTotals;
    }
}
```

Такое секционирование замечательно разделяет код. Теперь формирование отчета и расчеты выполняются в отдельных модулях. И все благодаря тому, что я начал рефакторинг с метода **Extract Field**.

Кубик Рубика

До этого момента я пытался показать вам, насколько эффективным может быть небольшой набор методов рефакторинга. Обычно я редко пользуюсь методами, отличными от представленных выше. Хитрость в том, чтобы досконально изучить выбранный инструментарий и понять все детали его реализации в IDE и приемы его использования.

Я часто сравниваю рефакторинг с кубиком Рубика. Если вы никогда его не собирали, то имеет смысл ознакомиться с инструкциями по сборке. После этого вы относительно легко справитесь с этой задачей.

Оказывается, существует набор «операций», выполнение которых сохраняет большую часть позиций на гранях кубика, одновременно меняя определенные позиции предсказуемым образом. Освоив три-четыре таких операции, вы сможете пошагово собирать кубик.

Чем больше операций вы знаете и чем лучше их выполняете, тем быстрее и точнее будет процесс сборки. Но их желательно очень хорошо изучить. Один пропущенный шаг — и все придется начинать сначала.

Рефакторинг кода очень похож на сборку кубика. Чем больше методов рефакторинга вы знаете и чем более искусно их используете, тем легче вам будет менять код в желаемом направлении.

И да, лучше не забывать про тесты. Без них у вас гарантированно ничего не получится.

ПРАКТИКИ

Если рефакторинг выполнять регулярно и дисциплинированно, то он безопасен, прост и эффективен. Но если вы относитесь к рефакторингу как к разовой, временной и бессистемной деятельности, то от его безопасности и эффективности быстро не останется и следа.

Тесты

Первая из практик — это, конечно же, тесты. Тесты, тесты, тесты, тесты и еще раз тесты. Безопасный и надежный рефакторинг кода невозможен без набора тестов, которому вы полностью доверяете.

Быстрые тесты

Тесты должны выполняться быстро. Рефакторинг просто не получится, если ваши тесты занимают часы (или даже минуты).

Как бы вы ни старались сократить время тестирования большой системы, очень трудно добиться того, чтобы оно занимало несколько

минут. Поэтому я предпочитаю создавать наборы тестов, из которых можно быстро и легко собрать *подмножество для проверки реорганизуемой в текущий момент части кода*. Обычно это позволяет сократить время тестирования с минут до долей секунды. Я запускаю весь набор тестов примерно один раз в час, чтобы проверить, не просочились ли какие-либо ошибки.

Устранение взаимно однозначных соответствий

Создание структуры, позволяющей выбирать необходимые подмножества тестов, означает, что на уровне модулей и компонентов архитектура тестов копирует архитектуру кода. Скорее всего, между вашими высокоуровневыми тестовыми модулями и модулями производственного кода будет взаимно однозначное соответствие.

Но в предыдущем разделе я показал, что глубокие взаимно однозначные соответствия между тестами и кодом порождают хрупкие тесты.

На этом уровне преимущество в скорости, которое дает тестирование с помощью подмножества тестов, перевешивает проблемы, возникающие из-за взаимно однозначного соответствия. Но чтобы не допустить появления хрупких тестов, это соответствие обязательно нужно нарушить. Что и делается ниже уровня модулей и компонентов.

Непрерывный рефакторинг

Я считаю, что когда готовишь еду, мыть посуду нужно сразу же.[1] Я не даю посуде скапливаться в раковине. Ведь до момента, когда блюдо будет готово, достаточно времени, чтобы вымыть все, чем ты пользовался.

К рефакторингу имеет смысл подходить так же. Не надо ждать подходящего момента. Проводите рефакторинг прямо по ходу работы.

[1] Моя жена с этим не согласна.

Помните о цикле «красный → зеленый → рефакторинг» и прокручивайте его каждые несколько минут. Это позволит избежать ситуации, когда хаос разрастается настолько, что к приведению кода в порядок становится страшно подступиться.

Безжалостный рефакторинг

Хлесткое выражение «безжалостный рефакторинг» произнес Кент Бек в речи об экстремальном программировании. И это было очень выразительно. Смысл этой практики в том, чтобы смело приступать к рефакторингу. Не бойтесь пробовать. Не отказывайтесь от изменений. Манипулируйте кодом, как будто это глина, а вы скульптор. Страх перед кодом ведет на темный путь. Стоит туда ступить — и дальше он будет определять вашу судьбу. И попросту вас поглотит.

Поддержка проходимости тестов!

Иногда оказывается, что допущена структурная ошибка, и необходимо редактировать большую часть кода. Подобное может произойти, например, при появлении нового требования, которое невозможно реализовать в рамках существующей архитектуры. Или неожиданно вы понимаете, что структуру проекта можно сильно улучшить.

Действовать в таких случаях следует безжалостно, но рационально. Никогда не ломайте тесты! Точнее, никогда не оставляйте их сломанными более чем на несколько минут. Если реструктуризация занимает несколько часов или дней, то проводите ее небольшими партиями, следя за тем, чтобы тесты продолжали проходить.

Представьте, что вам нужно изменить фундаментальную структуру данных, которой в системе пользуются большие фрагменты кода. Это неизбежно приведет к прекращению работы этих фрагментов, причем одновременно перестанут проходить многие тесты.

Поэтому нужно создать новую структуру данных, воспроизводящую содержимое старой. И постепенно перемещать каждый фрагмент кода

из старой структуры в новую, поддерживая код в состоянии, когда он благополучно проходит тесты.

Все это время можно добавлять новый функционал и исправлять ошибки в соответствии с обычным рабочим графиком. Не нужно просить отдельное время на реструктуризацию. Делайте свою работу, одновременно манипулируя кодом. А когда старая структура данных перестанет использоваться, ее можно будет удалить.

Это может занять недели или даже месяцы, в зависимости от того, насколько значительна реструктуризация. Но и в этом случае система в любой момент будет готова к развертыванию. Ведь даже при частично завершенной реструктуризации прохождение всех тестов означает, что систему можно вводить в эксплуатацию.

Оставляйте себе выход

Пилотов учат, что при полете в районы с плохой погодой нужно все время следить за наличием пути отступления. Аналогичным образом обстоят дела с рефакторингом. Иногда его выполнение может завести в тупик. Например, идея, с которой все началось, оказывается не очень рабочей.

В подобных ситуациях на помощь приходит команда `git reset --hard`.

Поэтому, приступая к рефакторингу, не забудьте пометить репозиторий с исходным кодом, чтобы при необходимости всегда можно было вернуться к начальной версии.

РЕЗЮМЕ

Я намеренно сделал эту главу короткой, поскольку хотел добавить к «Рефакторингу» Мартина Фаулера лишь несколько идей. И снова призываю вас прочитать его книгу, чтобы более глубоко понимать тему.

Наилучший подход к рефакторингу — отобрать наиболее удобные вам методы, которыми вы будете часто пользоваться, и как следует на

практике изучить остальные, не вошедшие в ваш личный инструментарий. Если вы используете IDE для автоматизации рефакторинга, то убедитесь, что в деталях понимаете суть ее работы.

Из-за слишком большой вероятности появления ошибок рефакторинг не имеет смысла без тестов. Даже автоматизированный рефакторинг, проводимый с помощью IDE, порой может допускать ошибки. Поэтому *всегда* подкрепляйте свои достижения на ниве рефакторинга всеобъемлющим набором тестов.

Наконец, будьте дисциплинированны. Проводите рефакторинг часто, безжалостно и без лишних расшаркиваний. Никогда, слышите, никогда не спрашивайте на него разрешения.

6 ПРОСТОЙ ДИЗАЙН

Дизайн. Святой Грааль и конечная цель программного обеспечения. Любой из нас старается сделать его настолько совершенным, чтобы функционал можно было добавлять без усилий и суеты, и настолько надежным, чтобы даже через месяцы и годы постоянного обслуживания система оставалась простой и гибкой. Все эти вещи в итоге упираются в дизайн.

На эту тему я уже много писал. Из-под моего пера вышли книги о принципах проектирования, шаблонах проектирования и архитектуре. И я далеко не единственный; существует огромное количество литературы, посвященной проектированию программного обеспечения.

Но здесь я планирую рассказать не об этом. Рекомендую вам почитать различные издания на тему дизайна, чтобы познакомиться с принципами и шаблонами проектирования и составить представление о том, что же это такое — архитектура программного обеспечения.

Существует один аспект, который наделяет дизайн всеми желаемыми характеристиками. Его можно считать ключом ко всему остальному. Это *простота*. Как однажды сказал Чет Хендриксон (Chet Hendrickson)[1]: «Дядя Боб написал тысячи страниц чистого кода. Кент Бек написал четыре строчки». Именно на этих четырех строчках мы сосредоточимся в этой главе.

На первый взгляд, очевидно, что лучшим дизайном системы будет обладать самый простой проект, который поддерживает весь необходимый функционал, одновременно обеспечивая наибольшую гибкость для изменений. Но это заставляет задуматься о значении термина «простота»[2]. Просто не значит легко. «Простое» означает «без запутанных взаимосвязей», а распутывать взаимосвязи *сложно*.

---

[1] Эти слова Чета Хендриксона, произнесенные на конференции AATC2017, процитировал в своем твите Мартин Фаулер. Я присутствовал на этой конференции и полностью согласился с Четом.

[2] В 2011 году Рич Хики (Rich Hickey) сделал прекрасный доклад Simple Made Easy. Очень советую вам послушать его. (По адресу https://www.youtube.com/watch?v=LKtk3HCgTa8 можно найти ссылку на ролик с субтитрами на русском языке. — *Примеч. пер.*)

Что запутывается в программных системах? Самая дорогостоящая и существенная путаница возникает при соединении политик высокого уровня с низкоуровневыми деталями. Например, при создании связей между SQL и HTML, фреймворков с ключевыми значениями, механизма форматирования отчета с бизнес-правилами, вычисляющими вставляемые в отчет значения. Такой код легко создается, но затрудняет добавление нового функционала, исправление ошибок, а также улучшение и очистку дизайна.

Простым называется дизайн, в котором политики высокого уровня ничего не знают о деталях низкого уровня. Эти высокоуровневые политики обособлены и изолированы от низкоуровневых деталей, соответственно, любое изменение этих деталей никак на них не влияет[1].

Основное средство разделения и изоляции — *абстракция*. Оно позволяет усилить существенное и устранить ненужное. Политики высокого уровня необходимы, поэтому усиливаются. Низкоуровневые детали не играют особой роли, поэтому они обособлены и изолированы.

Реализуется такое абстрагирование с помощью *полиморфизма*. Политики высокого уровня управляют низкоуровневыми деталями через полиморфные интерфейсы. Низкоуровневые детали при этом строятся как реализации этих полиморфных интерфейсов.

В результате получается, что все зависимости исходного кода направлены от низкоуровневых деталей к высокоуровневым политикам. Соответственно, политики ничего не знают о том, как реализованы низкоуровневые детали. А значит, редактирование этих деталей никак их не затрагивает (рис. 6.1).

Если наилучшим считается простейший дизайн, поддерживающий необходимый функционал, то можно предположить, что абстракций, изолирующих политики от деталей реализации, в таком дизайне должно быть как можно меньше.

---

[1] Я много писал об этом в книге «Чистая архитектура. Искусство разработки программного обеспечения».

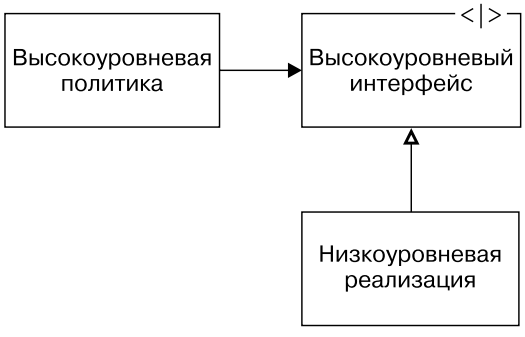


Рис. 6.1. Полиморфизм

Но все это полностью противоположно стратегии, которая активно практиковалась на протяжении 1980-х и 1990-х. В те дни мы были одержимы идеей *работы на перспективу*, добавляя *зацепки для изменений*, которые предполагались в будущем.

Мы шли по этому пути, поскольку в то время программное обеспечение было трудно редактировать, даже в случае простого дизайна.

Почему это было трудно? Из-за долгого времени сборки и еще большего времени тестирования.

В 1980-х годах на сборку небольшой системы мог уйти час или больше, а на тестирование — много часов. Тесты, конечно, делались вручную и поэтому были крайне неадекватными. По мере разрастания и усложнения системы программисты все сильнее боялись ее трогать. Это формировало образ мышления, заточенный на конструкционную избыточность, и подталкивало к созданию намного более сложных систем, чем этого требовал заложенный в них функционал.

Ситуация начала меняться в конце 1990-х с появлением экстремального программирования, а затем гибкой методологии разработки. К тому времени компьютеры стали настолько мощными, что время сборки сократилось до минут, а иногда даже до секунд, а мы обнаружили, что имеем возможность автоматизировать тесты, что сильно ускоряло их выполнение.

Технологический скачок дал возможность на практике применять принцип YAGNI и четыре принципа простого дизайна, описанные Кентом Беком.

YAGNI

А что, если он вам не понадобится?

В 1999 году я вел курс экстремального программирования с Мартином Фаулером, Кентом Беком, Роном Джеффрисом и многими другими. Когда речь зашла об опасности конструкционной избыточности и преждевременных обобщений, кто-то написал на доске YAGNI и произнес: «Вам это не понадобится» (You aren't gonna need it). Вмешался Бек и сказал, что, конечно, возможно, это понадобится, но нужно спрашивать себя: «А что, если нет?»

Именно этот вопрос лежит в основе принципа YAGNI. Каждый раз, когда у вас возникает мысль, что вот тут *хорошо бы добавить крючок на будущее*, спрашивайте себя, что произойдет, не сделай вы это. Если многолетнее присутствие в системе этого крючка будет дорого стоить, а вероятность того, что он в конечном итоге понадобится, низка, то вряд ли имеет смысл его добавлять.

Трудно представить, какое негодование в конце 1990-х вызвала эта новая тенденция. Ведь проектировщики привыкли повсюду добавлять крючки. Это считалось общепринятой передовой практикой.

Так что принцип YAGNI из экстремального программирования резко раскритиковали и назвали ересью и вздором.

По иронии судьбы сейчас это одна из самых важных практик в проектировании программного обеспечения. При наличии хорошего набора тестов и умения качественно проводить рефакторинг стоимость добавления нового функционала и последующего обновления дизайна почти наверняка окажется меньше стоимости реализации и поддержки всех крючков, которые, возможно, когда-либо вам потребуются.

Подход с оставлением зацепок на будущее в принципе спорный. Это очень редко можно реализовать корректно, поскольку невозможно однозначно предсказать, в какую сторону повернут запросы клиентов. В результате мы склонны расставлять куда больше крючков, чем требуется, основываясь на прогнозах, которые сбываются не так уж часто.

Просто мы не были готовы к тому, как на процесс проектирования и архитектуру программного обеспечения повлияет измеряемая в гигагерцах тактовая частота и измеряемая в терабайтах память. До конца 1990-х мы не осознавали, что технологические достижения позволяют значительно упростить дизайн.

Один из величайших парадоксов нашей индустрии заключается в том, что экспоненциальный рост, называемый законами Мура, который подталкивал нас к созданию все более сложных программных систем, одновременно позволил *упростить* дизайн этих систем.

Оказывается, принцип YAGNI стал случайным следствием практически безграничных компьютерных мощностей, оказавшихся в нашем распоряжении. Время сборки сократилось до секунд, кроме того, появилась возможность писать комплексные наборы тестов, которые, опять же, выполняются за секунды, так что теперь мы можем себе позволить не вставлять крючки, а при изменении требований просто проводить рефакторинг проектов.

Означает ли это, что о крючках можно забыть? Разве мы всегда разрабатываем системы только под необходимый в текущий момент функционал? Неужели мы больше не думаем наперед и не планируем будущее?

Нет, принцип YAGNI не имеет в виду ничего подобного. Бывают случаи, когда добавление крючка — хорошая идея. Думать на перспективу и оставлять задел на будущее всегда разумно.

Просто за последние десятилетия соотношение преимуществ и недостатков поменялось настолько резко, что теперь, как правило, большинство крючков лучше не использовать. И поэтому мы задаемся вопросом:

А что, если он вам не понадобится?

ТЕСТОВОЕ ПОКРЫТИЕ

Впервые я столкнулся с правилами простого проектирования Бека в книге *Extreme Programming Explained*. В то время четыре правила выглядели так, как показано ниже.

1. Система (код и тесты) должна сообщать все, что вы хотите сообщить.
2. В системе не должно быть повторяющегося кода.
3. Система должна содержать как можно меньше классов.
4. В системе должно быть как можно меньше методов.

К 2011 году эти правила эволюционировали и стали выглядеть следующим образом.

1. Тесты проходят.
2. Назначение понятно.
3. Нет дублирования.
4. Маленький размер.

К 2014 году Кори Хейнц (Corey Haines) написал о них книгу[1].

В 2015 году Мартин Фаулер перефразировал их у себя в блоге[2].

1. Проходит тесты.
2. Демонстрирует назначение.
3. Отсутствует дублирование.
4. Максимально маленькие элементы.

[1] *Haines C.* Understanding the Four Rules of Simple Design. — Leanpub, 2014.

[2] *Fowler M.* BeckDesignRules, March 2, 2015. https://martinfowler.com/bliki/BeckDesignRules.html.

В этой книге я формулирую первое правило следующим образом.

1. *Покрыт тестами.*

Обратите внимание, как с годами менялись акценты. Первое правило превратилось в два, а последние два объединились. Обратите также внимание, что с течением времени возрастало значение тестов, начиная от взаимодействия и заканчивая охватом.

Степень покрытия

Концепция тестового покрытия возникла давно. Первое упоминание, которое мне удалось найти, относится к 1963 году[1]. Статья начинается с двух абзацев, которые, я думаю, вы найдете интересными, а то и вызывающими воспоминания.

> *Для любой сложной компьютерной программы обязательна успешная проверка. Только после выполнения одного или нескольких сценариев тестирования программа считается готовой для применения к реальным задачам. Каждый сценарий проверяет ту часть программы, которая используется в его вычислениях. Однако слишком часто ошибки проявляются только через несколько месяцев (или даже лет) после начала работы программы. Это сигнализирует о том, что части программы, вызываемые только редко применяемыми входными данными, на этапе проверки не были протестированы должным образом.*
>
> *Чтобы с уверенностью полагаться на программу, недостаточно знать, что она работает большую часть времени или что до сих пор в ней ни разу не возникали ошибки. Главный вопрос тут: можем ли мы рассчитывать на то, что каждый раз программа будет успешно выполнять функциональные требования? Это означает, что после прохождения тестирования не должно оставаться возможности того, что необычное сочетание входных данных или условий вызовет неожиданную ошибку в программе. Процесс тестирования должен задействовать каждую часть программы, чтобы убедиться в ее корректности.*

[1] *Miller J., Maloney C. J.* Systematic Mistake Analysis of Digital Computer Programs // Communications of the ACM 6, no. 2 (1963): 58–63.

Эта статья появилась всего через 17 лет после того, как самая первая программа была запущена на самом первом электронном компьютере[1]. К этому моменту уже было известно, что единственный способ эффективно уменьшить угрозу программных ошибок — протестировать каждую строку кода.

Инструменты покрытия кода существуют уже несколько десятилетий. Я не помню, когда впервые столкнулся с ними. Думаю, это был конец 1980-х или начало 1990-х. В то время я работал на рабочих станциях Sparc от Sun Microsystems, для которых существовал инструмент `tcov`.

Не помню, когда я впервые услышал вопрос: «Какое у вас покрытие кода?» Вероятно, это случилось в самом начале 2000-х. Но после этого представление о том, что покрытие кода выражается числом, стало практически универсальным.

С тех пор для разработчиков программного обеспечения стало обычным делом запускать инструмент покрытия кода как часть процесса непрерывной сборки и для каждой сборки публиковать показатель покрытия кода.

Какой показатель можно считать хорошим? Восемьдесят процентов? Девяносто? Многие команды рады указать в отчетах такие цифры. Но за шесть десятилетий до публикации этой книги Миллер и Мэлоуни ответили на вопрос совсем по-другому. Это 100 процентов.

Имеет ли смысл какой-нибудь другой вариант? Если вас устраивает 80-процентное покрытие, это означает, что вы не знаете, как работают 20 процентов вашего кода. Вас эта ситуация устраивает? А ваших клиентов?

Так что слово *покрыт* в первом правиле простого дизайна означает стопроцентное покрытие. Стопроцентное покрытие строк (line coverage) и стопроцентное покрытие веток (branch coverage).

---

[1] Предполагается, что первым компьютером стала автоматическая вычислительная машина, а первая программа была выполнена в 1946 году.

Асимптотическая цель

Вы можете возмутиться, что 100 процентов — это недостижимая цель. Я даже соглашусь с этим утверждением. Обеспечение стопроцентного покрытия строк и веток — настоящий подвиг. Более того, в некоторых ситуациях это может оказаться непрактичным. Что, впрочем, не означает, что покрытие не допускает улучшений.

Воспринимайте показатель «100 процентов» как асимптотическую цель. Возможно, вы до нее никогда не доберетесь, но это не повод отказаться от попыток к ней приблизиться.

Я участвовал в проектах, объем которых разросся до десятков тысяч строк, постоянно поддерживая показатель покрытия кода близким к 100 процентам.

Дизайн?

Но как высокое покрытие кода связано с простотой дизайна? Почему первое правило касается именно этого аспекта?

Пригодный для тестирования код — это несвязанный код.

Чтобы получить высокий показатель покрытия строк и веток во всех частях программы, нужно сделать каждую часть доступной для тестов. То есть каждая часть должна быть настолько несвязанной с остальным кодом, чтобы ее можно было изолировать и вызывать из отдельного теста. Фактически тесты проверяют не только поведение, но и степень несвязанности. Написание изолированных тестов — это процесс проектирования, поскольку нужно заранее предусмотреть возможность тестирования фрагментов кода.

В главе 4 я рассказал, что тестовый и производственный код развиваются в разных направлениях, чтобы предотвратить слишком сильное связывание. Так решается проблема хрупких тестов. Но существует аналогичная ей проблема хрупких модулей, которая решается тем же способом. Если дизайн системы защищает тесты от хрупкости, защищает он и другие элементы системы.

Но это еще не все

Тесты не только побуждают создавать несвязанные и надежные проекты, но и позволяют улучшать эти проекты с течением времени. Как я уже не раз упоминал, надежный набор тестов значительно снижает страх перед изменениями. Если у вас есть такой набор, который к тому же работает быстро, ничто не мешает вам улучшать дизайн кода каждый раз, когда обнаруживается лучший подход. Или если выясняется, что новые требования невозможно реализовать при текущем дизайне, тесты позволят безбоязненно внести необходимые изменения.

И именно поэтому первое и самое важное правило простого дизайна касается именно покрытия тестами. Без набора тестов, покрывающих всю систему, остальные три правила бесполезны, поскольку их лучше всего применять *постфактум*. Они имеют отношение к рефакторингу, который практически невозможно осуществить без хорошего исчерпывающего набора тестов.

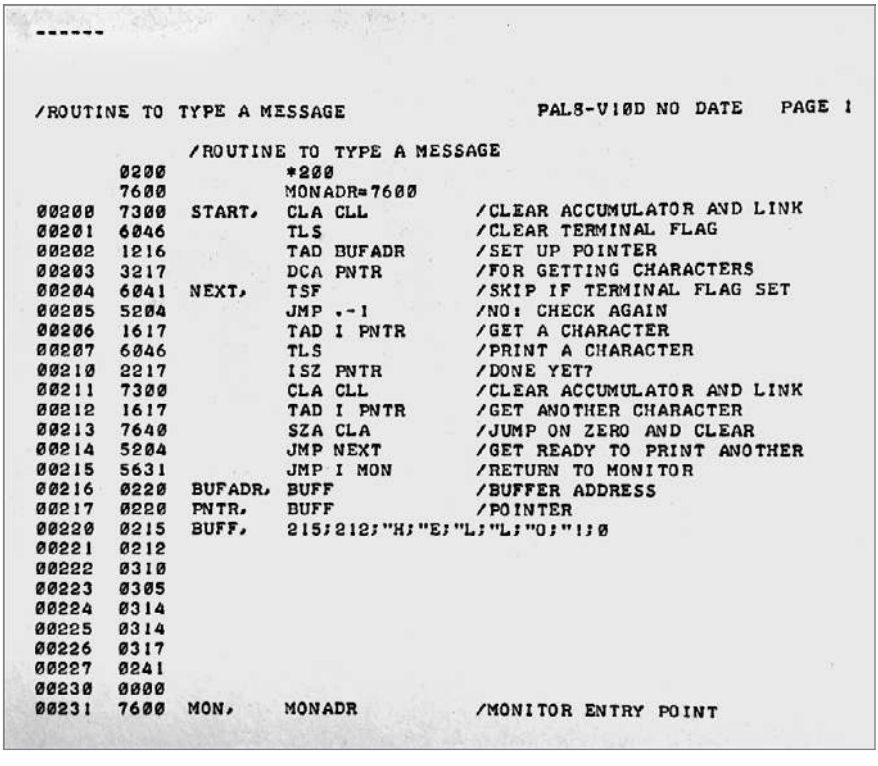
МАКСИМАЛЬНОЕ РАСКРЫТИЕ ПРЕДНАЗНАЧЕНИЯ

На заре эры программирования код не давал представления о том, зачем он нужен. Более того, само слово «код» предполагало нечто неявное и скрытое. Пример кода, который писался в те дни, показан на рис. 6.2.

Обратите внимание на множество комментариев. Без них было не обойтись, поскольку сам код вообще ничего не говорил о предназначении программы.

Впрочем, 1970-е остались далеко позади. Языки, которые мы сейчас используем, *чрезвычайно* выразительны. При должной практике можно научиться создавать код, который читается как «хорошо написанная проза», «не затемняет намерения проектировщика»[1].

[1] *Мартин Р.* Чистый код. — С. 30.



```
------


/ROUTINE TO TYPE A MESSAGE                PAL8-V10D NO DATE   PAGE 1

            /ROUTINE TO TYPE A MESSAGE
       0200           *200
       7600           MONADR=7600
00200  7300  START,    CLA CLL          /CLEAR ACCUMULATOR AND LINK
00201  6046           TLS              /CLEAR TERMINAL FLAG
00202  1216           TAD BUFADR       /SET UP POINTER
00203  3217           DCA PNTR         /FOR GETTING CHARACTERS
00204  6041  NEXT,    TSF              /SKIP IF TERMINAL FLAG SET
00205  5204           JMP .-1          /NO: CHECK AGAIN
00206  1617           TAD I PNTR       /GET A CHARACTER
00207  6046           TLS              /PRINT A CHARACTER
00210  2217           ISZ PNTR         /DONE YET?
00211  7300           CLA CLL          /CLEAR ACCUMULATOR AND LINK
00212  1617           TAD I PNTR       /GET ANOTHER CHARACTER
00213  7640           SZA CLA          /JUMP ON ZERO AND CLEAR
00214  5204           JMP NEXT         /GET READY TO PRINT ANOTHER
00215  5631           JMP I MON        /RETURN TO MONITOR
00216  0220  BUFADR,  BUFF             /BUFFER ADDRESS
00217  0220  PNTR,    BUFF             /POINTER
00220  0215  BUFF,    215;212;"H;"E;"L;"L;"O;"!;0
00221  0212
00222  0310
00223  0305
00224  0314
00225  0314
00226  0317
00227  0241
00230  0000
00231  7600  MON,     MONADR           /MONITOR ENTRY POINT
```

Рис. 6.2. Пример ранней программы

В качестве примера рассмотрим фрагмент кода на языке Java из упражнения с магазином видеопроката, которое вы выполняли в главе 4:

```
public class RentalCalculator {
  private List<Rental> rentals = new ArrayList<>();

  public void addRental(String title, int days) {
    rentals.add(new Rental(title, days));
  }

  public int getRentalFee() {
    int fee = 0;
    for (Rental rental : rentals)
```

```
      fee += rental.getFee();
    return fee;
  }

  public int getRenterPoints() {
    int points = 0;
    for (Rental rental : rentals)
      points += rental.getPoints();
    return points;
  }
}
```

Человек, который не работает над этим проектом, очевидно, не сможет понять всего, что происходит внутри данного кода. Однако даже при самом беглом взгляде несложно определить основной замысел проектировщика. Назначение переменных, функций и типов понятно по их именам. Легко увидеть структуру алгоритма. Это очень красноречивый код. И очень простой.

Базовая абстракция

Чтобы вы не думали, что выразительность сводится исключительно к выбору описательных имен для функций и переменных, должен сказать, что существует еще одна проблема. Это разделение уровней и представление лежащей в основе абстракции.

Программную систему можно назвать выразительной, если каждая строка кода, каждая функция и каждый модуль находятся в однозначно определенном разделе, который четко отображает уровень кода и его место в общей абстракции.

Последнее предложение может оказаться несколько сложным для понимания, поэтому попробую пояснить его на практике.

Представим приложение со сложным набором требований. Здесь я люблю использовать в качестве примера систему начисления заработной платы.

- Сотрудники с почасовой оплатой получают деньги каждую пятницу на основании представленных табелей учета рабочего времени.

При работе более 40 часов в неделю каждый дополнительный час оплачивается по полуторной ставке.

- Сотрудникам, работающим за проценты, зарплата выплачивается в первую и третью пятницу каждого месяца. Она состоит из базового оклада плюс комиссионные, рассчитываемые по предоставленным квитанциям о продажах.
- Сотрудникам с фиксированным окладом зарплата начисляется в последний день месяца.

Без труда можно представить набор функций со сложным оператором `switch` или цепочкой операторов `if/else`, которые описывают вышеперечисленные условия. Но такой набор функций, скорее всего, скроет лежащую в основе кода абстракцию. Что это за абстракция?

```
public List<Paycheck> run(Database db) {
  Calendar now = SystemTime.getCurrentDate();
  List<Paycheck> paychecks = new ArrayList<>();
  for (Employee e : db.getAllEmployees()) {
    if (e.isPayDay(now))
      paychecks.add(e.calculatePay());
  }
  return paychecks;
}
```

Обратите внимание: здесь не упоминаются многочисленные детали, которыми наполнены требования. Основная цель создания приложения заключается в том, что все сотрудники должны получать деньги в свои дни зарплаты. Фундаментальная основа создания простого и выразительного дизайна состоит в отделении высокоуровневой политики от низкоуровневой реализации деталей.

Тесты: вторая половина проблемы

Вспомним первоначальную формулировку первого правила Бека.

> *Система (код и тесты) должна сообщать все, что вы хотите сообщить.*

Он сформулировал это именно так по определенной причине, и в некотором смысле очень жаль, что формулировка была изменена.

Насколько бы говорящим вы ни сделали производственный код, он не сможет передать контекст своего использования. Это задача тестов.

Каждый написанный тест, особенно в ситуации, когда все тесты изолированы и не связаны, демонстрирует, как именно предполагается использовать производственный код. Хорошо написанные тесты — это примеры использования тех частей кода, работу которых они проверяют.

Таким образом, код *в совокупности* с тестами показывает функцию каждого элемента системы и способ его применения.

Как это связано с дизайном? Целиком и полностью. При создании каждого проекта наша основная цель состоит в том, чтобы упростить другим программистам процессы понимания, улучшения и обновления наших систем. И нет лучшего способа достичь этой цели, чем заставить систему сообщать свое предназначение и предполагаемые варианты использования.

МИНИМИЗАЦИЯ ДУБЛИРОВАНИЯ

На заре существования программного обеспечения в принципе не существовало редакторов исходного кода. Код писали карандашом на специальных бланках. Соответственно, лучшим инструментом редактирования был ластик. Возможность скопировать и вставить фрагмент кода попросту отсутствовала.

Поэтому код не дублировался. Куда проще было создать экземпляр фрагмента кода и поместить его в подпрограмму.

Затем появились редакторы исходного кода, а вместе с ними и возможность копирования/вставки. Внезапно стало намного проще скопировать фрагмент кода, вставить его в другое место и редактировать, пока он не заработает.

В результате с годами накапливались системы, в коде которых присутствовало дублирование.

Оно обычно порождает проблемы. Необходимость отредактировать одновременно два или более одинаковых фрагмента возникает достаточно часто. Искать такие фрагменты сложно. Правильно отредактировать их еще сложнее, поскольку они существуют в разных контекстах. Фактически дублирование порождает хрупкость.

В общем, похожие фрагменты кода лучше сводить к одному экземпляру, абстрагируя этот код в новую функцию и предоставляя ей соответствующие аргументы, сообщающие о различиях в контексте.

Такая стратегия работает не всегда. Бывает, например, так, что дублирование происходит в коде, проходящем через сложную структуру данных. И разные части системы будут использовать один и тот же цикл и код обхода только для того, чтобы поработать с этой структурой.

Но так как со временем любая структура данных меняется, программистам придется искать все дубликаты кода обхода, чтобы соответствующим образом обновить их. Чем больше дублируется код, тем выше риск хрупкости.

Дублирование кода обхода можно устранить, инкапсулировав его в одном месте и воспользовавшись для передачи в него необходимых операций лямбда-выражением, объектом `Command`, паттерном «Стратегия» (Strategy) или даже паттерном «Шаблонный метод» (Template Method)[1].

Непреднамеренное дублирование

Убирать дублирующийся код нужно далеко не во всех случаях. Иногда фрагменты кода могут быть очень похожими, даже идентичными,

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

но изменяться по разным причинам[1]. Я называю такую ситуацию *непреднамеренным дублированием* (accidental duplication). Здесь не требуется ничего делать. По мере изменения требований дубликаты будут развиваться по отдельности, и непреднамеренное дублирование исчезнет.

Как видите, работать с дублирующимся кодом не так-то просто. Чтобы определить, какой код продублирован сознательно, а где дублирование было непреднамеренным, а затем инкапсулировать и изолировать дубликаты, требуется взвешенный и обстоятельный подход.

Легкость определения того, какие фрагменты были продублированы сознательно, а какие непреднамеренно, сильно зависит от возможности по виду кода понять его предназначение. Непреднамеренное дублирование имеет разное предназначение, в то время как предназначение сознательно созданных дубликатов пересекается.

Инкапсуляция и изоляция последних с помощью абстрагирования, лямбда-выражений и шаблонов проектирования требует значительного рефакторинга. А рефакторинг невозможен без надежного набора тестов.

Именно поэтому устранение дублирующегося кода стоит в списке правил простого дизайна на третьем месте. После возможностей тестирования и понимания предназначения кода.

МИНИМИЗАЦИЯ РАЗМЕРА

Простой дизайн состоит из простых элементов. А простые элементы имеют небольшой размер. Последнее правило создания простого дизайна гласит: после того, как вы прошли все тесты, максимально проявили предназначение кода и минимизировали дублирование, приходит пора поработать над уменьшением размеров кода. Это касается каждой написанной вами функции. И естественно, это нужно делать без нарушения трех других принципов.

---

[1] См. принцип единственной ответственности в книге: *Мартин Р.* Быстрая разработка программ: Принципы, примеры, практика.

Как этого добиться? В основном путем выделения как можно большего количества функций. В предыдущей главе мы говорили о том, что выделять функции следует до тех пор, пока остается такая возможность.

В результате мы получим прекрасный набор маленьких функций с красивыми длинными именами, позволяющими понять их предназначение.

Простой дизайн

Много лет назад мы с Кентом Беком обсуждали принципы дизайна, и он сказал слова, которые я запомнил навсегда: если как можно точнее следовать четырем изложенным им принципам, то все остальные принципы дизайна будут соблюдены автоматически.

Не знаю, правда ли это. Не знаю, обязательно ли идеально покрытая тестами, выразительная, не имеющая дубликатов кода и имеющая минимальный размер программа будет соответствовать принципу открытости-закрытости или принципу единственной ответственности. Но я совершенно уверен в том, что знание принципов хорошего дизайна и хорошей архитектуры (например, принципов SOLID) значительно облегчает создание четко разделенных и простых проектов.

В этой книге речь не об этих принципах. О них я уже много раз писал[1], как и другие авторы. И очень рекомендую вам прочитать эти книги и изучать эти принципы для дальнейшего совершенствования вашего мастерства.

---

[1] См. мои книги «Чистый код»; «Чистая архитектура»; «Быстрая разработка программ: Принципы, примеры, практика».

7 СОВМЕСТНОЕ ПРОГРАММИРОВАНИЕ

Что значит быть частью команды? Представьте футбольную команду, которой нужно провести мяч по полю, обыграв соперников. Один из игроков спотыкается и падает, но игра продолжается. Что делают члены его команды?

Они адаптируются к новой реальности, меняя свое положение на поле таким образом, чтобы *обеспечить перемещение мяча вперед*.

Так ведет себя команда. Если кто-то из ее членов падает, то остальные прикрывают его, пока он не встанет на ноги.

Как сделать такую же команду из программистов? Как команда может прикрыть сотрудника, который на неделю ушел на больничный или у него просто плохой день и работа не клеится? Мы взаимодействуем! Мы работаем в тесном сотрудничестве, стараясь добиться того, чтобы систему целиком знала вся команда.

Если Боб упал, кто-то из тех, кто в последнее время работал вместе с ним, закрывает образовавшуюся дыру до тех пор, пока он не встанет на ноги.

Старая поговорка о двух головах, которые лучше, чем одна, стала основным посылом совместного программирования. Сотрудничество двух программистов часто называют парным программированием[1]. Если же сотрудничают три или более программиста, это называют командным программированием (mob programming)[2].

Эта практика включает двух или более человек, работающих в одно и то же время над одним и тем же кодом. В настоящее время это обычно делается с помощью инструментов совместного использования экрана. Оба программиста видят на своих экранах один и тот же код. Оба могут управлять им с помощью мыши и клавиатуры. Их рабочие станции синхронизируются друг с другом локально или удаленно.

Подобному взаимодействию не стоит отдавать 100 процентов рабочего времени. Как правило, сеансы совместной работы короткие и неформальные. Общее время совместной работы зависит от опытности, на-

---

[1] *Williams L., Kessler R.* Pair Programming Illuminated. — Addison-Wesley, 2002.

[2] *Pearl M.* Code with the Wisdom of the Crowd. — Pragmatic Bookshelf, 2018.

выков, географического положения и демографических характеристик команды и должно составлять от 20 до 70 процентов[1].

Один сеанс может длиться от 10 минут до часа-двух. Более короткие или более длинные сессии, скорее всего, будут не так полезны. Моя любимая стратегия распределения времени при совместной работе — техника Помидора.[2] Этот метод планирования делит время на «помидоры» длиной примерно по 20 минут с короткими перерывами между ними. Сеанс совместной работы должен длиться от одного до трех помидоров.

Сеансы совместной работы происходят намного быстрее, чем решаются отдельные задачи. Ответственность за конкретную задачу берет на себя кто-то из команды и время от времени приглашает коллег помочь с выполнением обязательств.

А вот за сеанс совместной работы или за код, редактируемый в рамках этого сеанса, не отвечает никто. Точнее, все участники на равных могут считаться авторами кода. При возникновении споров роль арбитра берет на себя программист, ответственный за решаемую задачу.

В процессе работы глаза устремлены на экран, а мысли полностью заняты поиском решения. За клавиатурой могут сидеть один или два человека, причем на протяжении сеанса они могут меняться. Сеанс можно считать гибридом упражнения по программированию на лету и рецензирования кода.

В силу своей насыщенности совместные сеансы требуют много умственной и эмоциональной энергии. Средний программист может выдержать один, максимум два часа такой напряженной работы, а потом ему желательно переключиться на что-то менее трудоемкое.

Вы можете счесть такое сотрудничество неэффективным использованием рабочей силы. Многие уверены, что независимо друг от друга можно сделать больше, чем совместно. Но это далеко не всегда так.

---

[1] Есть команды, практикующие парное программирование 100 процентов времени. Кажется, им это нравится, и это повышает производительность работы.

[2] *Cirillo F.* The Pomodoro Technique. — Currency Publishing, 2018.

Исследования[1] показали, что во время сеанса парной работы производительность падает примерно на 15 процентов, а не вполовину, как опасаются скептики. Зато созданный парой код содержит примерно на 15 процентов меньше дефектов, и (что куда важнее) для каждого функционала пишется примерно на 15 процентов меньше кода.

Два последних показателя подразумевают, что код, написанный во время парного программирования, имеет гораздо лучшую структуру, чем код, созданный программистами-одиночками.

Исследования эффективности коллективного программирования мне пока не попадались, но неофициальные данные[2] обнадеживают.

Опытные сотрудники могут программировать в паре с новичками. В таком случае первые замедляются на время сеанса, зато вторые учатся работать в ускоренном режиме, так что это хороший взаимообмен.

Опытные сотрудники могут работать в паре друг с другом, главное — предварительно убедиться, что в комнате нет оружия.

Новички тоже могут работать парами, хотя в этом случае за ними должен кто-то приглядывать. Более того, новичок скорее *предпочтет* поработать в паре с таким же неопытным коллегой. Если такое случается слишком часто, то одного из пары должен заменить кто-то из старших.

Бывают люди, которым просто не нравится участвовать в подобных вещах. Более того, они лучше работают в одиночку. Не стоит их принуждать или осуждать их предпочтения. Часто их больше устраивает групповая, а не парная работа.

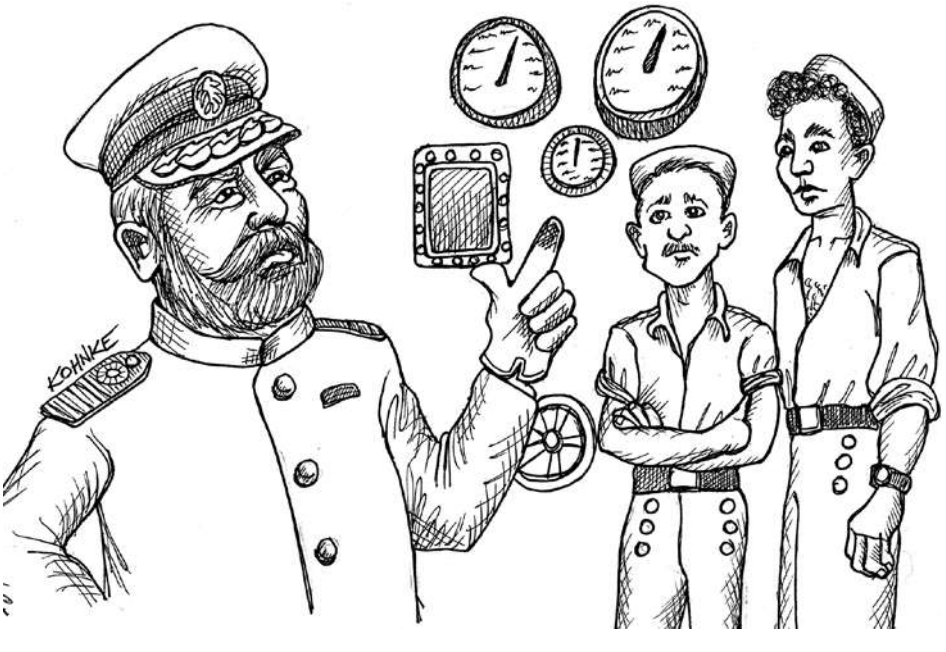
Сотрудничество — это навык, для приобретения которого требуется время и терпение. Не ожидайте, что вы сможете хорошо работать в паре без многочасовой практики. Однако это умение очень полезно как для команды в целом, так и для каждого ее члена.

---

[1] Вот два примера таких исследований: *Williams L., Kessler R. R., Cunningham W., Jeffries R.* Strengthening the Case for Pair Programming // IEEE Software 17, no. 4 (2000), 19–25; и *Nosek J. T.* The Case for Collaborative Programming // Communications of the ACM 41, no. 3 (1998), 105–108.

[2] Agile Alliance, Mob Programming: A Whole Team Approach, AATC2017, https://www.agilealliance.org/resources/sessions/mob-programming-aatc2017/.

8 ПРИЕМОЧНОЕ ТЕСТИРОВАНИЕ



Из всех дисциплин чистого мастерства меньше всего программисты контролируют приемочное тестирование, так как в этом случае предполагается участие бизнеса. К сожалению, многие заказчики до сих пор демонстрировали нежелание взаимодействовать должным образом.

Как узнать, что система готова к развертыванию? Зачастую организации принимают это решение на основании заключения отдела тестирования программного обеспечения. Как правило, это означает, что специалисты по контролю качества вручную запускают множество тестов, которые проверяют различные варианты поведения системы. С точки зрения специалистов, прохождение этих тестов означает, что система ведет себя как нужно и готова к развертыванию.

Но при таком подходе получается, что истинное требование к системе всего одно. Это *прохождение ею тестов*. Неважно, какие требования перечислены в документации; важны только результаты тестирования. Если отдел контроля качества ставит свою подпись, то система развертывается.

Практика приемочного тестирования признает этот простой факт и рекомендует указывать все требования *в виде тестов*. Написанием этих тестов должны заниматься команды, отвечающие за анализ запросов клиентов (business analysis, BA) и за контроль качества (quality assurance, QA), причем тесты по очереди пишутся для каждого функционала незадолго до его реализации. Отдел контроля качества не несет ответственности за выполнение этих тестов. В основном это задача программистов. Соответственно, программисты, скорее всего, автоматизируют эти тесты.

Ни один находящийся в здравом уме программист не захочет раз за разом вручную тестировать систему. Программисты предпочитают автоматизировать как можно больше задач. Особенно если это тесты, за прохождение которых они несут ответственность.

Но так как созданием тестов занимаются команды BA и QA, программистам приходится доказывать, что после автоматизации тесты продолжают выполнять свою задачу. Поэтому для автоматизации нужно использовать язык, понятный даже неспециалистам. Более

того, команды BA и QA должны иметь возможность писать тесты на этом языке.

За прошедшие годы для решения этой задачи было придумано несколько инструментов: FitNesse[1], JBehave, SpecFlow, Cucumber и др. Но дело даже не в них. С технической точки зрения поведение программного обеспечения всегда представляет собой простую функцию, состоящую из входных данных, выполняемого действия и ожидаемых выходных данных. Это хорошо известный шаблон AAA: Arrange/Act/Assert[2].

Все тесты начинаются с подготовки входных данных. Затем происходит отработка тестируемого функционала. В конце тест сверяет полученные значения с ожидаемыми.

Эти три элемента можно задать различными способами, но наиболее доступным является простой табличный формат.

К примеру, на рис. 8.1 показана часть одного из приемочных тестов в инструменте FitNesse. Этот инструмент представляет собой систему на базе «Вики», и показанный на рисунке тест проверяет правильность преобразования различных элементов разметки в HTML. Выполняемое действие: отображение страницы виджетом. Входные данные: «вики»-текст. Выходные данные: текст в формате html.

Другой распространенный шаблон Given-When-Then:

```
Если дана (Given) страница с "вики"-текстом: !1 header
Когда (When) эта страница отображается.
Тогда (Then) она будет содержать: <h1>header</h1>
```

Очевидно, что такие представления потоков данных относительно легко автоматизируются вне зависимости от того, где они написаны: в инструменте приемочного тестирования, в простой электронной таблице или в текстовом редакторе.

---

[1] fitnesse.org.

[2] Авторство этого шаблона приписывают Биллу Уэйку, который описал его в 2001 году (https://xp123.com/articles/3a-arrange-act-assert).

widget should render		
wiki text	html text	
normal text	normal text	
this is ''italic'' text	this is italic text	italic widget
this is '''bold''' text	this is bold text	bold widget
!c This is centered text	This is centered text	
!1 header	header	
!2 header	header	
!3 header	header	
!4 header	header	
!5 header	header	
!6 header	header	
http://files/x	http://files/x	file link
http://fitnesse.org	http://fitnesse.org	http link
SomePage	SomePage\[\?\]	missing wiki word

Рис. 8.1. Фрагмент результата одного из приемочных тестов, выполненного с помощью FitNesse

ПОРЯДОК ДЕЙСТВИЙ

В идеале приемочные тесты должны писаться командами BA и QA. Первые сосредоточены на успешных сценариях, а вторые рассматривают множество причин, по которым система может выйти из строя.

Написание тестов происходит одновременно с разработкой тестируемого функционала или непосредственно предшествует ей. В Agile-проекте тесты пишутся в первые дни спринта, а к его концу все они должны успешно выполняться.

После чего тесты предоставляются программистам, которые автоматизируют их, привлекая к процессу команды BA и QA.

Эти тесты становятся *критерием готовности*. До прохождения всех приемочных тестов функционал считается незавершенным. Конечно, это накладывает на команды BA и QA огромную ответственность. Ведь создаваемые ими тесты должны представлять собой полную спецификацию тестируемого функционала. Набор приемочных тестов является документом, содержащим требования ко всей системе. Этими тестами команды BA и QA удостоверяют, что функционал, который их прошел, полностью готов и работает.

У некоторых команд BA и QA нет привычки писать такую формальную и подробную документацию. В этом случае приемочные тесты могут под их руководством писать программисты. Промежуточная цель — создание тестов, которые *могут читать* и одобрять команды BA и QA. Конечная цель — сделать так, чтобы команды BA и QA могли сами писать тесты.

НЕПРЕРЫВНАЯ СБОРКА

После того как приемочный тест пройден, он переходит в набор тестов, выполняемых во время непрерывной сборки.

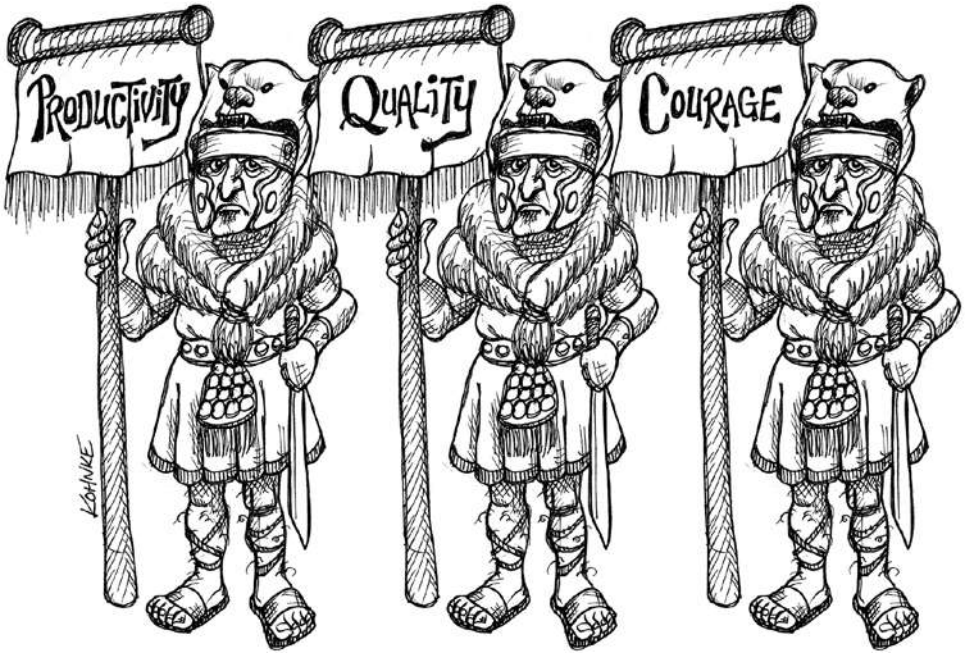
Непрерывная сборка — это автоматизированная процедура, которая запускается при каждой[1] проверке кода в системе управления исходным кодом. Она строит из исходного кода систему, а затем запускает наборы автоматических модульных тестов и автоматизированных приемочных тестов. Отчет о результатах этой процедуры выдается рабочей группе, часто в виде письма на электронную почту. Группа просто обязана следить за тем, чтобы сборка всегда проходила успешно.

Постоянное выполнение таких тестов гарантирует, что вносимые в систему изменения не нарушат работу ее функций. Если успешно пройденный приемочный тест дает сбой во время непрерывной сборки, то команда должна немедленно отреагировать и исправить дефект до того, как в систему будут внесены какие-либо изменения. Позволить сбоям, возникающим при непрерывной сборке, накапливаться, — самоубийство.

---

[1] В течение нескольких минут.

II СТАНДАРТЫ



Стандарты — это базовые *ожидания*. Это нарисованные на песке линии, которые мы договорились не пересекать. Это параметры, которые мы установили как допустимый минимум. Можно сделать больше, чем предполагают стандарты, но нельзя позволять себе недотягивать до них.

ВАШ НОВЫЙ ТЕХНИЧЕСКИЙ ДИРЕКТОР

Представьте, что я ваш новый технический директор. Я собираюсь рассказать вам, чего ожидаю от вас. А потом мы рассмотрим эти ожидания с двух противоположных позиций.

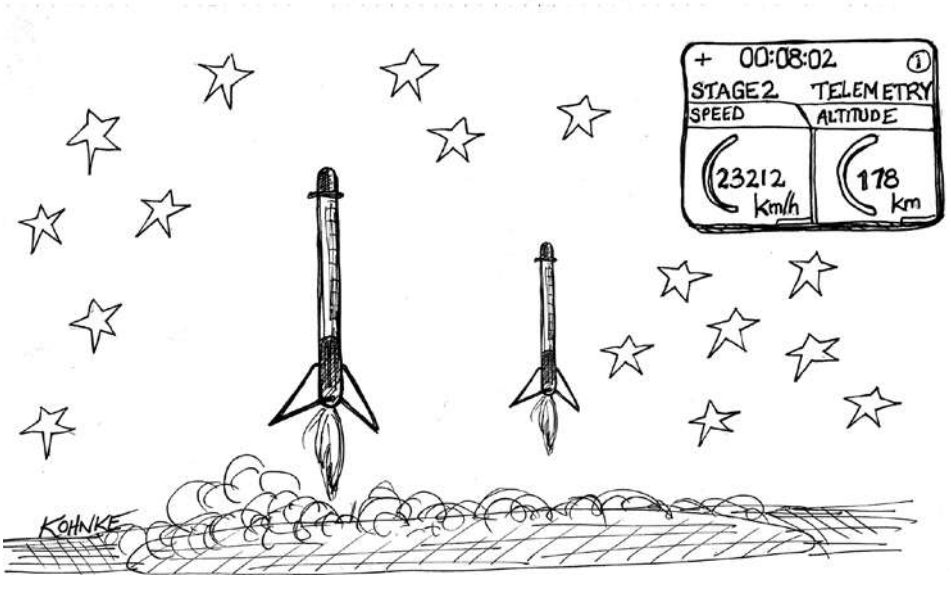
Сначала я познакомлю вас с точкой зрения руководства и пользователей. Для них эти ожидания будут естественными и очевидными. Ни ваше начальство, ни пользователи никогда не будет ожидать меньшего.

Вы, скорее всего, больше знакомы со второй точкой зрения. Это взгляд на ситуацию программистов, архитекторов и технических руководителей. Все они могут счесть эти ожидания чрезмерными, невозможными и даже безумными.

И вот эта разница в точках зрения, несовпадение ожиданий — главный недостаток индустрии программного обеспечения, который желательно исправить как можно быстрее.

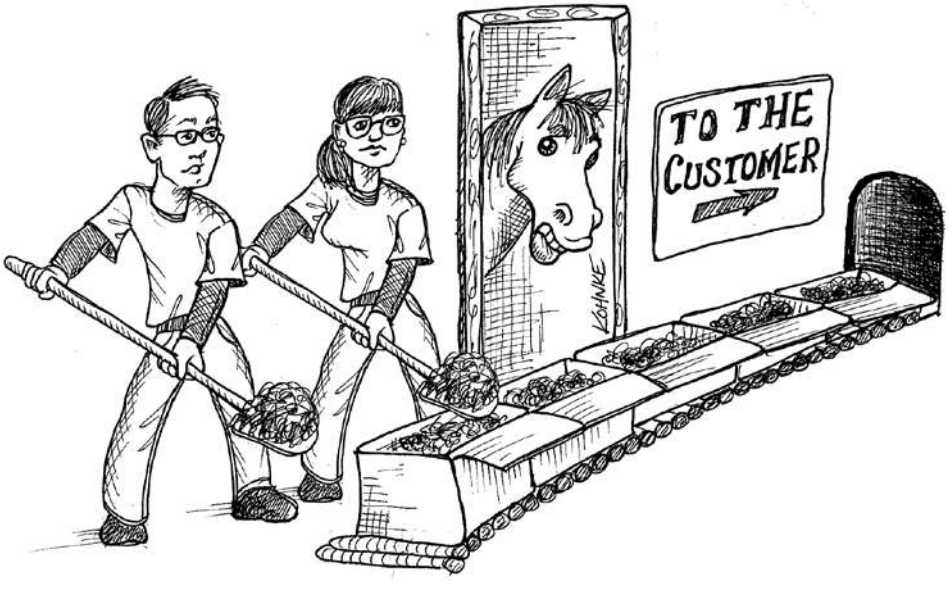
Как ваш новый технический директор, я ожидаю...

9 ПРОИЗВОДИТЕЛЬНОСТЬ



Сейчас я познакомлю вас со своими ожиданиями по поводу производительности.

МЫ НИКОГДА НЕ БУДЕМ ДЕЛАТЬ ДРЯНЬ



Как ваш новый технический директор, я ожидаю, что мы никогда не будем выпускать дрянь.

Я уверен, что вы знаете, что означает это слово. Как ваш технический директор, я ожидаю, что *мы не будем выпускать дрянь*.

Вы когда-нибудь делали дрянную продукцию? У большинства из нас есть такой опыт. В том числе и у меня. Удовольствия от этого я не получал. Мне не нравилось то, что я сделал. Пользователям тоже не нравилось. И моему начальству. Словом, это не нравилось *никому*.

Так почему мы это делаем? Почему мы выпускаем дрянь?

Потому что по каким-то причинам мы решили, что выбора у нас нет. Возможно, из-за необходимости во что бы то ни стало уложиться в срок. Или из-за прогнозов, которые мы постеснялись пропустить мимо ушей. А может, дело просто в небрежности или невнимательности. Или в давлении со стороны руководства. Или в нашей самооценке.

Но какой бы ни была причина, она *не является оправданием*. Не делать дрянь — это минимальный стандарт.

Что такое дрянь? Уверен, вы это знаете и без меня, но все же немного поговорим об этом.

- Любая ошибка в отправленной заказчику программе — это дрянь.
- Любая функция, которая не была протестирована, — это дрянь.
- Любая плохо написанная функция — это дрянь.
- Любая зависимость от деталей — это дрянь.
- Любое ненужное связывание — это дрянь.
- SQL в GUI — это дрянь.
- Схема базы данных в бизнес-правилах — это дрянь.

Список можно продолжить. Но я этого делать не буду. Каждая оплошность в любой практике, которые я описывал в предыдущих главах, может породить дрянь.

Это не означает, что все время нужно досконально соблюдать все пункты этих практик.

Мы инженеры. Инженеры идут на компромиссы. Но инженерный компромисс не имеет ничего общего с небрежностью или халатностью. Если вы отступаете от принятого порядка действий, то на это должна быть веская причина.

Еще важнее иметь хороший план смягчения последствий.

Представьте, что вы пишете код на языке CSS. Заранее писать для него автоматизированные тесты почти всегда бессмысленно, поскольку понять, как будет отображаться CSS на экране, можно только после того, как вы это увидите.

В итоге порядок действий, принятый при разработке через тестирование, будет нарушен. И что нам с этим делать?

Придется протестировать CSS-код вручную, проверяя результат визуально. И сделать это нужно во всех браузерах, которые могут использовать клиенты. Поэтому имеет смысл создать стандартное описание того, что мы хотим увидеть, с вариантами допустимых отклонений. Еще важнее придумать техническое решение, упрощающее процесс ручного тестирования CSS-кода, поскольку заниматься тестированием будем *мы*, а не команда QA.

Все это можно сформулировать одной фразой: *делайте свою работу хорошо!*

Именно этого все на самом деле ждут. Все наши руководители, все пользователи, все, кто когда-либо прикасался к нашему программному обеспечению, ожидают, что мы сделаем свою работу хорошо. И мы не должны их подводить.

Я надеюсь, что мы никогда не будем выпускать дрянь.

ЛЕГКАЯ АДАПТИВНОСТЬ

Словосочетание *программное обеспечение* (software) означает «гибкий продукт». Оно существует для того, чтобы быстро и легко менять поведение наших машин. И создавая с трудом поддающееся модификации ПО, мы разрушаем саму причину его существования.

Тем не менее негибкость программного обеспечения до сих пор является огромной проблемой в нашей отрасли. Дизайну и архитектуре уделяется так много внимания как раз в попытках повысить гибкость и ремонтопригодность наших систем.

Почему программное обеспечение становится негибким и хрупким? Например, потому, что команды разработчиков не хотят заниматься тестированием и рефакторингом, а именно эти практики обеспечивают гибкость и ремонтопригодность. Иногда это связано с усилиями на стадии проектирования.

Впрочем, сколько бы микросервисов мы ни создавали, насколько хорошо бы ни структурировали первоначальный проект и представления об архитектуре, без тестирования и рефакторинга код быстро деградирует и поддерживать систему становится все труднее и труднее.

А это не то, что мне хотелось бы получить. Я ожидаю, что в ответ на просьбу клиента о внесении изменений разработчики смогут предоставить стратегию, в которой *затраты пропорциональны масштабу работы*.

Клиенты могут не понимать внутреннего устройства системы, но четко представляют объем запрашиваемых изменений. И осознают, что изменение может повлиять на многие функции. Они ожидают, что стоимость изменения будет зависеть от его масштаба.

К сожалению, многие системы со временем приобретают такую негибкость, что затраты на внесение изменений становятся слишком большими. Что еще хуже, против определенных вмешательств нередко протестуют сами разработчики на том основании, что запрашиваемые изменения идут вразрез с архитектурой системы.

Архитектура, не допускающая изменений по запросу, противоречит смыслу и назначению программного обеспечения. Ее нужно менять. И ничто не упрощает такие изменения лучше, чем качественно и регулярно выполняемый рефакторинг и набор надежных тестов.

Я ожидаю, что дизайн и архитектура создаваемых вами систем будут развиваться в соответствии с выдвигаемыми к ним требованиями. Я ожидаю, что когда клиенты потребуют изменений, этому не будет мешать существующая архитектура или жесткость и хрупкость системы.

Я ожидаю легко реализуемой адаптивности.

ПОСТОЯННАЯ ГОТОВНОСТЬ

Как ваш новый технический директор, я ожидаю от вас постоянной готовности.

Задолго до того, как набрала популярность гибкая методология разработки, было понятно, что в грамотно управляемом проекте развертывание и выпуск происходят регулярно. Изначально это делалось в быстром ритме: еженедельно или даже ежедневно. Но следование модели каскадной разработки, начавшееся в1970-х, сильно замедлило этот ритм. Между новыми выпусками проходили месяцы, а иногда и годы.

Появившаяся на рубеже тысячелетий гибкая методология разработки заново выдвинула на передний план необходимость более быстрых ритмов. Методология Scrum рекомендовала спринты продолжительностью 30 дней. В экстремальном программировании рекомендуемая длина итерации составляла три недели. Это время быстро сократилось до двух недель. В настоящее время команды разработчиков нередко выполняют развертывание несколько раз в день, эффективно сокращая период разработки практически до нуля.

Мне кажется оптимальным быстрый ритм. Максимум одна-две недели. И я ожидаю, что в конце каждого спринта программное обеспечение будет технически готово к выпуску.

Техническая готовность к выпуску не означает, что бизнес *захочет* его выпустить. В этом программном обеспечении может отсутствовать набор функционала, который бизнес считает полным или необходимым клиентам и пользователям. Техническая готовность всего лишь означает, что если заказчик решит выпустить это ПО, то команда разработчиков и отдел контроля качества не будут против этого возражать. Ведь это программное обеспечение работает, протестировано, для него написана документация, словом, все готово к развертыванию.

Именно это я подразумеваю под *постоянной готовностью*. Я не хочу, чтобы команда разработчиков просила заказчиков подождать. Я против так называемых *стабилизирующих спринтов*. Альфа- и бета-тестирование уместно применять для определения совместимости функционала с пользователями, но не для устранения дефектов кода.

Давным-давно моя компания консультировала группу, занимавшуюся разработкой текстовых процессоров для юристов. Мы учили

их экстремальному программированию. Со временем они дошли до еженедельной записи нового компакт-диска[1], который занимал свое место на стопке еженедельных релизов. Продавцу, который хотел продемонстрировать программное обеспечение потенциальному клиенту, оставалось только зайти в лабораторию и взять из стопки верхний компакт-диск. Вот насколько *подготовленной* была команда разработчиков. Такого уровня готовности я ожидаю и от вас.

Для выхода на такой уровень требуется высокая дисциплина планирования, тестирования, коммуникации и распределения рабочего времени. Все это, конечно же, практикуется в гибкой методологии разработки. Заинтересованные стороны должны совместно выбирать наиболее ценные стратегии создания программного обеспечения. Отдел контроля качества должен быть заинтересован в предоставлении автоматических приемочных тестов, определяющих «готовность». Только тесное сотрудничество разработчиков и поддержание строгой дисциплины в вопросах тестирования, обзоров кода и рефакторинга позволит за короткие периоды разработки добиться значительного прогресса.

Но *всегда быть готовым* — это больше, чем просто следовать правилам и ритуалам гибкой методологии разработки. Это отношение к работе, образ жизни. Это обязательство постоянно обеспечивать прирост ценности.

Я ожидаю, что мы всегда будем готовы.

СТАБИЛЬНАЯ ПРОИЗВОДИТЕЛЬНОСТЬ

Со временем производительность работы над программными проектами часто снижается. Это симптом серьезного кризиса. Подобная ситуация возникает как следствие пренебрежения практиками тестирования и рефакторинга. Именно из-за этого пренебрежения возникает постоянно увеличивающееся препятствие в виде запутанного и хрупкого кода.

---

[1] Да, было время, когда программное обеспечение распространялось на компакт-дисках.

Причем такая ситуация чем дальше, тем больше выходит из-под контроля. Чем более хрупким и жестким становится код, тем труднее его чистить. По мере увеличения хрупкости кода растет и страх перед изменениями. Разработчики все с большей неохотой берутся за очистку кода, поскольку опасаются, что их усилия приведут к появлению еще большего количества дефектов.

Всего за несколько месяцев это приводит к ускоряющейся потере производительности. Постепенно продуктивность команды асимптотически приближается к нулю.

Руководители часто пытаются бороться с этим, добавляя в проект рабочую силу. Но зачастую это не дает эффекта, поскольку новички боятся вмешиваться в систему не меньше программистов, работавших с ней с самого начала. Они быстро начинают вести себя так же, как и остальные члены команды, в результате за решение проблемы так никто и не берется.

В ответ на упреки в потере производительности разработчики часто начинают жаловаться на ужасное состояние кода. А иногда даже ратуют за полную перестройку системы. Причем если такая ситуация возникает, то недовольство, как правило, высказывается до тех пор, пока руководство не будет вынуждено как-то отреагировать.

Разработчики утверждают, что смогут повысить производительность, полностью перепроектировав систему. Они настаивают на том, что знают о допущенных ошибках и не повторят их. Разумеется, руководство им не верит. Но вместе с тем оно отчаянно нуждается в повышении производительности, поэтому в конечном итоге многие руководители соглашаются с требованиями программистов, несмотря на затраты и риски.

Но мне такой вариант развития событий не нравится. Я ожидаю, что команды разработчиков будут постоянно поддерживать высокую производительность. Что они будут тщательно придерживаться практик, которые предохраняют структуру программного обеспечения от деградации.

Я ожидаю стабильной производительности.

10 КАЧЕСТВО

Как у вашего нового технического директора, у меня есть ряд ожиданий по поводу качества.

ПОСТОЯННОЕ УЛУЧШЕНИЕ

Я ожидаю постоянного улучшения.

Со временем люди начинают совершенствовать мир вокруг себя. Наводят порядок в хаосе. Улучшают вещи.

Современные компьютеры лучше тех, которые были раньше. Как и современные автомобили, самолеты, дороги, телефоны, телевидение, услуги связи. Стали более совершенными медицинские технологии. Развились космические технологии. Наша цивилизация значительно улучшилась.

Почему же тогда программное обеспечение деградирует с течением времени? Это совсем не то, чего я ожидаю.

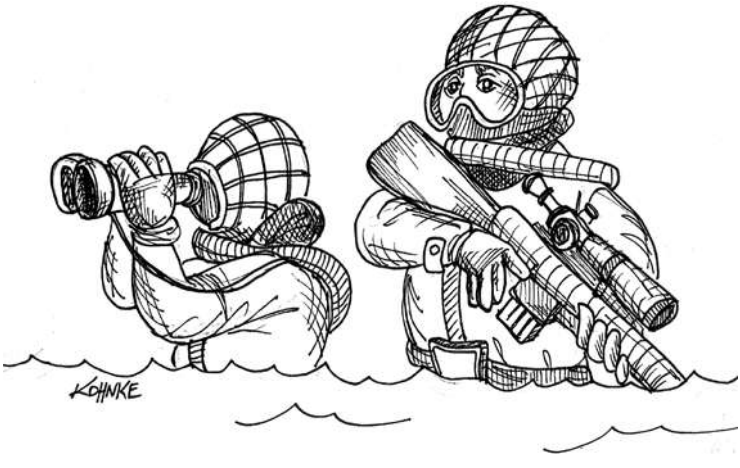
Я ожидаю улучшения дизайна и архитектуры наших систем. Я ожидаю, что с каждой неделей программное обеспечение будет становиться более чистым и гибким. Что чем дольше оно существует, тем *меньших* затрат будет требовать процедура внесения изменений. Я ожидаю, что со временем все будет становиться лучше.

Что для этого нужно? Желание. Определенное отношение к делу. Приверженность практикам, которые, как *известно*, дают нужные результаты.

Я ожидаю, что любой возвращаемый в репозиторий код будет чище, чем он был до того, как с ним начали работать. Я ожидаю, что каждый программист будет *улучшать* код, к которому прикасается. Исправляя ошибку, нужно одновременно улучшать код. Добавляя функционал, нужно одновременно улучшать код. Я ожидаю, что каждая манипуляция с кодом будет давать на выходе лучшую версию этого кода, лучший дизайн и лучшую архитектуру.

Я ожидаю постоянного улучшения.

БЕССТРАШНАЯ КОМПЕТЕНТНОСТЬ



Я ожидаю от вас бесстрашной компетентности.

По мере деградации внутренней структуры системы хаос внутри нее может стать неконтролируемым. Чем больше все запутывается, тем больше разработчики боятся что-либо трогать. Ведь даже простые операции сопряжены с риском. Нежелание вносить изменения и улучшения резко снижает способность программистов обслуживать систему.

Здесь я имею в виду вовсе не потерю компетентности. Программисты не начинают меньше знать и уметь. Скорее речь о том, что растущая сложность системы в какой-то момент превышает уровень компетентности программистов.

Чем труднее становится справляться с системой, тем больше программисты боятся с ней работать. Этот страх усугубляет проблему, поскольку из-за него в систему будут вноситься только те изменения, которые программисты считают наиболее безопасными. А такие изменения редко улучшают систему. Более того, часто так называемые максимально безопасные изменения делают только хуже.

Если потакать тревоге и трусости, то затраты на приведение системы в порядок будут естественным образом расти, увеличится процент дефектов, а соблюдать сроки станет все труднее и труднее. Резко упадут производительность и моральный дух.

Выход тут только один — избавиться от страха. А этого можно достичь с помощью наборов надежных тестов.

Имея на руках такие тесты, обладая навыками рефакторинга и стремясь к простоте дизайна, программисты смогут безбоязненно очищать деградирующую систему. Они будут уверены и компетентны, что позволит быстро устранять деградацию и поддерживать программное обеспечение в состоянии постоянного совершенствования.

Я ожидаю, что команда всегда будет демонстрировать бесстрашную компетентность.

ИСКЛЮЧИТЕЛЬНОЕ КАЧЕСТВО

Я ожидаю от вас исключительного качества.

Когда мы начали считать ошибки естественной частью программного обеспечения? Когда стало приемлемым поставлять ПО с определенным уровнем дефектов? Когда мы решили, что результат бета-тестирования подходит для распространения?

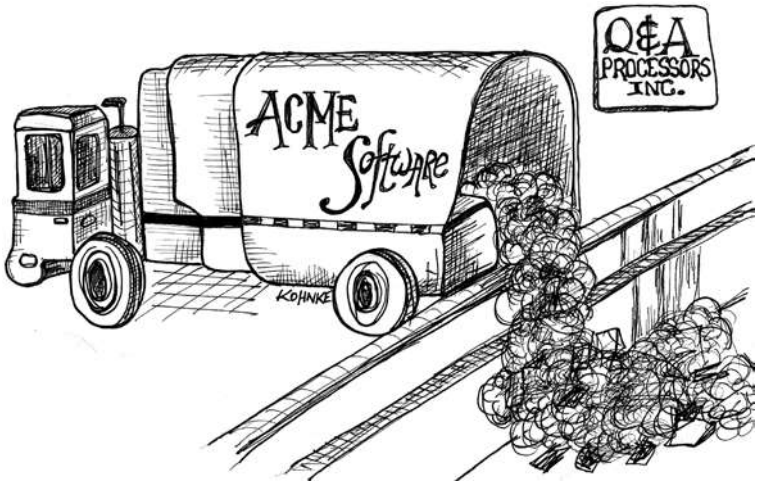
Я не согласен с утверждением о неизбежности ошибок. Я не приемлю отношения к неполадкам как к чему-то ожидаемому. Я ожидаю, что каждый программист будет создавать программное обеспечение, *не имеющее дефектов*.

И я имею в виду не просто нарушения поведения. Мне хотелось бы получать ПО, свободное от дефектов поведения и от дефектов *структуры*.

Достижима ли такая цель? Возможно ли получить то, чего я ожидаю? Достижимо это или нет, но я ожидаю, что каждый программист примет это мое ожидание как стандарт и будет постоянно стремиться к нему.

Я ожидаю исключительного качества от команды программистов.

МЫ НЕ БУДЕМ ЗАВАЛИВАТЬ РАБОТОЙ ОТДЕЛ КОНТРОЛЯ КАЧЕСТВА



Я ожидаю, что мы не будем создавать стрессовые ситуации для отдела контроля качества.

Зачем такие отделы вообще существуют? По какой причине компании оплачивают работу людей, проверяющих результаты труда программистов? Ответ очевиден и неутешителен. Решение создавать отделы контроля качества программного обеспечения обусловлено тем, что программисты плохо делали свою работу.

Когда нам пришла в голову идея, что контроль качества должен выполняться в конце процесса? Во многих организациях команда QA сидит и ждет, пока программисты выпустят программное обеспечение. Разумеется, это далеко не всегда происходит по расписанию, в результате отделу контроля качества порой приходится сокращать тестирование, чтобы успеть к дате выпуска.

Представляете, какое давление испытывают члены команды QA? Тестирование — очень напряженная и утомительная работа, которую приходится сокращать, если поджимает время до срока сдачи проекта. Понятно, что качество при этом не гарантировано.

Болезнь отдела тестирования

Как понять, хорошо ли работает отдел тестирования? Каковы основания для повышения заработной платы его сотрудников? Наверное, это обнаружение ошибок? Можно ли считать лучшим того тестировщика, который находит больше всего ошибок?

Если это так, то для отдела контроля качества ошибки — позитивное явление. Чем их больше, тем лучше! Понятно, что такая ситуация ненормальна.

Впрочем, отдел контроля качества — не единственные, кто так относится к дефектам. Среди программистов есть старая поговорка[1]: «Я могу уложиться в любой график, если от меня не требуют, чтобы программа работала».

Как бы забавно это ни звучало, но именно к такой стратегии порой прибегают разработчики для соблюдения индивидуальных сроков. Если работа тестировщиков все равно заключается в поиске ошибок, то почему бы не передать им программу в срок?

Что тут можно сказать? Подобных вещей попросту не должно существовать. И все же все знают, что разработчики и отдел контроля качества могут долго перекидывать друг другу множество ошибок. И это серьезная болезнь.

Я ожидаю, что мы не будем сбрасывать в отдел контроля качества недоделанные программы.

ОТДЕЛ КОНТРОЛЯ КАЧЕСТВА НИЧЕГО НЕ НАЙДЕТ

Я ожидаю, что если отдел контроля качества возьмется за работу в самом конце процесса, то ничего не найдет. Именно это должно стать целью команды разработчиков. Каждый раз, когда команда QA находит ошибку, разработчикам нужно выяснить, почему та появилась, исправить ее и убедиться, что она никогда не появится снова.

---

[1] Впервые я услышал ее от Кента Бека.

Но при таком подходе у отдела контроля качества неминуемо возникнет вопрос, зачем они нужны в конце процесса. Ведь они никогда ничего не находят.

На самом деле контроль качества должен осуществляться в начале рабочего процесса. Задача тестировщиков заключается вовсе не в обнаружении всех ошибок; это работа программистов. Отдел контроля качества должен с помощью достаточной детализации определять поведение системы с точки зрения тестов, чтобы исключить из окончательной версии системы все дефекты. Эти тесты должны выполняться программистами.

Я ожидаю, что команда QA ничего не найдет.

АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ

В большинстве случаев ручное тестирование — огромная трата денег и времени. Почти любой тест, допускающий автоматизацию, *должен быть* автоматизирован. Это касается модульных, приемочных, интеграционных и системных тестов.

Такую дорогостоящую операцию, как ручное тестирование, следует оставить для ситуаций, в которых необходимо человеческое суждение. Например, если нужно проверить эстетику графического интерфейса, провести исследовательское тестирование или субъективно оценить простоту взаимодействия.

Исследовательское тестирование заслуживает отдельного упоминания. Этот вид тестирования полностью зависит от человеческой изобретательности, интуиции и проницательности. Цель состоит в эмпирической идентификации поведения системы путем обширного наблюдения за ее работой. Тестировщики должны выявлять граничные случаи и придумывать методы их проверки. Это непростая задача, требующая значительного опыта.

С другой стороны, многие тесты без проблем поддаются автоматизации. Подавляющее большинство из них — простые конструкции, описываемые шаблоном Arrange/Act/Assert, которые выполняются путем предоставления входных данных и изучения ожидаемых выход-

ных данных. Разработчики отвечают за предоставление вызываемого функциями API, который позволяет проводить эти тесты быстро и без значительной настройки среды выполнения.

При проектировании системы нужно абстрагироваться от любых медленных операций или операций, требующих тщательной настройки. Например, если приходится активно работать с системой управления реляционными базами данных (RDBMS), то разработчикам следует создать уровень абстракции, инкапсулирующий из нее бизнес-правила. Такая практика позволяет автоматизированным тестам заменять данные из RDBMS стандартными входными данными, значительно увеличивая как скорость, так и надежность тестирования.

Медленные и неудобные периферийные устройства, интерфейсы и фреймворки также должны быть абстрагированы, чтобы отдельные тесты могли выполняться за микросекунды, причем изолированно от любой среды[1] и независимо от любых флуктуаций во времени подключения к сокету, в содержимом базы данных или в поведении фреймворка.

АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ И ПОЛЬЗОВАТЕЛЬСКИЕ ИНТЕРФЕЙСЫ

Автоматизированные тесты *не должны* осуществлять проверку бизнес-правил через пользовательский интерфейс.

Пользовательские интерфейсы меняются по причинам, которые больше связаны с модой, удобствами и маркетинговыми соображениями, чем с бизнес-правилами. И эти изменения начинают затрагивать автоматизированные тесты, если запускать их через пользовательский интерфейс, как показано на рис. 10.1. В результате тесты становятся чрезмерно хрупкими и часто попросту игнорируются, поскольку их слишком сложно сопровождать.

[1] Например, на вашем ноутбуке, когда вы летите над Атлантикой на высоте 10 километров.

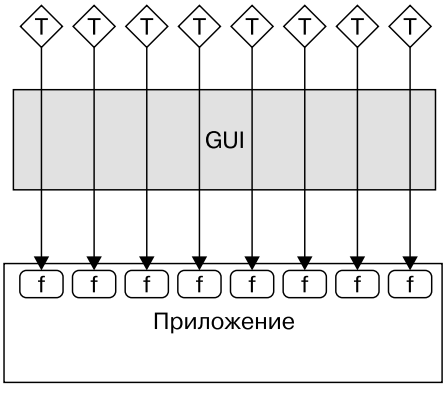


Рис. 10.1. Тесты, управляемые через пользовательский интерфейс

Избежать этой ситуации позволяет изоляция бизнес-правил от пользовательского интерфейса с помощью API вызова функции, как показано на рис. 10.2. Тесты, использующие этот API, полностью независимы от пользовательского интерфейса, и любые вносимые в него изменения никак их не затрагивают.

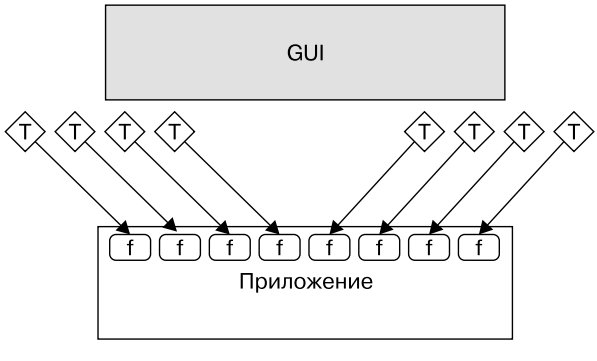


Рис. 10.2. Тесты через API не зависят от пользовательского интерфейса

ТЕСТИРОВАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Автоматическая проверка бизнес-правил через API вызова функции значительно сокращает объем тестирования, необходимый для проверки поведения пользовательского интерфейса. Чтобы сохранить изоляцию от бизнес-правил, эти правила заменяются заглушкой, как показано на рис. 10.3. Заглушка предоставляет пользовательскому интерфейсу какие-то стандартные значения.

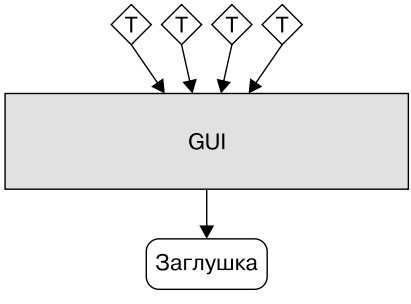


Рис. 10.3. Заглушка предоставляет пользовательскому интерфейсу стандартные значения

Это гарантирует быстроту и однозначность тестов пользовательского интерфейса. Для больших и сложных пользовательских интерфейсов лучше использовать фреймворк автоматизированного тестирования. Использование заглушки вместо бизнес-правил сделает эти тесты намного более надежными.

Небольшие и простые пользовательские интерфейсы часто целесообразнее тестировать вручную, особенно если требуется оценить их эстетическую составляющую. И в этом случае заглушка на месте бизнес-правил значительно упростит тестирование.

Я ожидаю, что каждый допускающий автоматизацию тест *будет* автоматизирован, что тесты будут выполняться быстро и не будут хрупкими.

11 СМЕЛОСТЬ

Как ваш технический директор, я имею несколько ожиданий, связанных со смелостью.

ПРИКРЫВАЕМ ДРУГ ДРУГУ СПИНУ

Для обозначения группы разработчиков проекта мы используем слово *команда*. Но понимаем ли мы, что это такое?

Команда — это группа сотрудников, которые настолько хорошо понимают цель своей работы и настолько хорошо умеют взаимодействовать, что даже выход кого-то из членов команды из строя *не останавливает их продвижение к цели*. Например, на борту корабля у каждого члена экипажа есть своя работа. При этом по очевидным причинам каждый умеет выполнять и чужую работу. Корабль должен продолжать плавание, даже если один из членов экипажа погибнет.

Я ожидаю, что члены команды программистов будут прикрывать друг другу спину, как экипаж корабля. Если некто по каким-то причинам не сможет выполнять свои обязанности, то я ожидаю, что его работу возьмут на себя остальные и будут ее делать, пока он не вернется на свое место.

Человек может временно выпасть из команды по многим причинам. Он может заболеть. Его могут отвлечь домашние неприятности. Он может уйти в отпуск. Но все это не должно останавливать работу над проектом. Образовавшуюся пустоту должны заполнить другие.

Если работавший с базой данных Боб внезапно заболевает, то кто-то должен взять на себя работу с базой данных. Если занимавшийся графическим интерфейсом Джим ушел в отпуск, то кто-то должен взять на себя работу с графическим интерфейсом.

Фактически это означает, что компетентность каждого члена команды не должна ограничиваться только его непосредственными обязанностями. Нужно иметь представление о работе других, чтобы при необходимости на время заменить их.

Эта ситуация имеет и обратную сторону. Обеспечение прикрытия — это *ваша* ответственность. Вы должны позаботиться о том, чтобы не стать единственным незаменимым игроком в команде. Вы обязаны найти и обучить кого-то еще, чтобы в крайнем случае он смог взять на себя ваши обязанности.

Как обучить другого человека вашей работе? Вероятно, лучше всего путем совместного написания кода в течение часа или около того. И если вы достаточно рациональны, то будете практиковать это с более чем одним членом команды. Ведь чем больше людей знает вашу работу, тем больше вероятность, что вас смогут прикрыть в случае какого-либо форс-мажора.

Разумеется, однократного сеанса совместной работы недостаточно. Проект движется вперед, соответственно, нужно постоянно держать других в курсе своих задач.

В этом отношении очень помогает практика совместного программирования.

Я ожидаю, что члены команды программистов смогут прикрывать друг другу спину.

ЧЕСТНАЯ ОЦЕНКА

Я ожидаю честных оценок.

Самая честная оценка, которую может дать программист, — это «я не знаю». Потому что на самом деле неизвестно, сколько времени займет выполнение задачи.

С другой стороны, вы понимаете, что явно справитесь с задачей быстрее, чем за миллиард лет. Итак, честная оценка — это комбинация того, что вы знаете и чего не знаете.

Честная оценка выглядит примерно так:

- вероятность 5 процентов, что я закончу эту задачу до пятницы;
- вероятность 50 процентов, что я закончу до следующей пятницы;

- вероятность 95 процентов, что я закончу до пятницы, которая будет через две недели.

Подобная оценка обеспечивает распределение вероятностей, описывающее вашу неопределенность. Честной ее делает именно признание вашей неуверенности.

Именно в такой форме нужно отвечать, когда руководство просит вас оценить большой проект, например, чтобы попытаться прикинуть его стоимость, прежде чем утвердить его. Именно тогда наиболее ценно честное признание в неопределенности.

Для небольших задач лучше всего использовать практику из гибкой методологии разработки, которая называется Story Points. Это достаточно честная оценка, поскольку она не привязана к временным рамкам. Скорее это описание усилий, затрачиваемых на выполнение задачи, по сравнению с какой-то другой задачей. Используемые числа произвольны, но взаимосвязаны.

Оценка в этом случае выглядит примерно так.

Пользовательская история Deposit стоит 5.

Что такое 5? Это произвольное количество очков относительно некоторой задачи с уже известной оценкой. Например, предположим, что история Login была оценена в 3 очка. Оценивая историю Deposit, вы решаете, что она почти в два раза сложнее истории Login, и ставите ей 5. Больше тут ничего делать не нужно.

В эти оценки распределение вероятностей уже встроено. Во-первых, очки — это не даты и не время. Во-вторых, это не обещания, а предположения. В конце каждой итерации (обычно через неделю или две) набранные очки суммируются. И полученное значение используется для оценки того, сколько очков мы сможем выполнить на следующей итерации.

Я ожидаю честных оценок, дающих представление о степени вашей неуверенности. Я не жду, что вы назовете мне точную дату.

УМЕНИЕ ГОВОРИТЬ «НЕТ»

Я ожидаю, что вы скажете «нет», когда ситуация этого потребует.

Одна из самых важных вещей, которую может сказать программист, — «Нет!» Произнесенный в нужное время и в правильном контексте отказ может сэкономить вашему работодателю огромные суммы денег и предотвратить ужасные неудачи и затруднения.

Это не разрешение на отказ от любых задач. Работа инженера в том, чтобы найти способ сказать «да». Но бывают ситуации, когда соглашаться недопустимо. Мы единственные, кто может определить это. Мы единственные, кто точно это знает. Следовательно, когда ответ действительно «нет», нужно говорить «нет».

Допустим, начальник просит вас сделать что-то к пятнице. Вы понимаете, что выполнить его просьбу невозможно. Значит, начальнику нужно отказать. Было бы разумно объяснить, что вы можете сделать это, например, к следующему вторнику, настаивая на том, что о пятнице не может быть и речи.

Руководители часто не любят слышать «нет». Они могут выразить свое несогласие. Они могут начать спорить. Они могут даже накричать на вас. Эмоциональное противостояние — один из инструментов, которым пользуются некоторые начальники.

Ни в коем случае нельзя идти на уступки. Если ответ отрицательный, то на этом следует настаивать, какое бы давление на вас ни оказывали.

И не попадайтесь на уловку «Может быть, вы хотя бы попытаетесь?». Подобный вопрос кажется вполне обоснованным, не так ли? Однако на самом деле он совсем не обоснован, ведь *вы уже понимаете*, что попытки ни к чему не приведут. Нет ничего, что вы могли бы сделать, чтобы «нет» превратилось в «да». Поэтому соглашаться и говорить, что вы попытаетесь, — это значит просто лгать.

Я ожидаю, что в ситуациях, когда нужно отказать, вы скажете «нет».

НЕПРЕРЫВНОЕ ИНТЕНСИВНОЕ ОБУЧЕНИЕ

Индустрия программного обеспечения очень динамична. Можно спорить, *должно* ли так быть, но нельзя спорить с тем, что *это так*. И поэтому мы все должны постоянно интенсивно обучаться.

Скорее всего, лет через пять вы будете писать программы совсем не на том языке, который используете для этого сегодня. Примерно через год изменится и фреймворк, в котором вы работаете. Нужно быть готовым к этим изменениям и понимать, что вокруг нас все меняется.

Программистам часто советуют[1] каждый год изучать новый язык. Это хороший совет. Причем лучше всего выбирать язык с незнакомым вам стилем. Если вы никогда не писали код на языке с динамической типизацией, то изучите его. Если вы никогда не писали код на декларативном языке, то изучите его. Если вы никогда не писали на языках Lisp, Prolog или Forth, то изучите их.

---

[1] *Хант Э., Томас Д.* Программист-прагматик. Путь от подмастерья к мастеру.

Как и когда этим заниматься? Если работодатель предоставляет время и пространство для такого обучения, то используйте его как можно лучше. В противном случае придется учиться в свободное время. Будьте готовы тратить на это несколько часов в месяц. Постарайтесь найти возможность выделить время на учебу.

Да, я знаю, что у вас есть семейные обязательства, вам нужно оплачивать счета и успевать на самолеты, словом, у вас есть своя *жизнь*. Но у вас есть и *профессия*. А профессиональные навыки нуждаются в поддержании и совершенствовании.

Я ожидаю, что мы все будем постоянно активно учиться.

НАСТАВНИЧЕСТВО

Кажется, существует бесконечная потребность в возрастающем количестве программистов. И оно растет бешеными темпами. В универси-

тетах преподают некоторый минимум, а во многих из них, к сожалению, не могут научить вообще ничему.

Поэтому работа по обучению ложится на нас. Программисты, работающие несколько лет, должны взять на себя бремя обучения начинающих.

Возможно, вам кажется, что это сложно. Так и есть. Но это приносит огромную пользу. Лучший способ чему-то научиться — учить других. С этим не сравнится ни один другой метод. Поэтому если хотите чему-то научиться, то учите этому.

Если вы проработали программистом пять, десять или пятнадцать лет, то у вас есть огромный опыт и жизненные уроки, которыми вы можете поделиться с начинающими. Возьмите под свое крыло одного или двух и помогайте им двигаться вперед первые шесть месяцев.

Садитесь с ними рядом и учите писать код. Рассказывайте истории о своих прошлых неудачах и успехах. Давайте информацию о практиках, стандартах и этике. Словом, учите их ремеслу.

Я ожидаю, что все программисты станут наставниками. Я ожидаю, что вы будете помогать другим постигать ремесло.

III ЭТИКА

САМЫЙ ПЕРВЫЙ ПРОГРАММИСТ

Профессия программиста зародилась неблагоприятным летом 1935 года, когда Алан Тьюринг начал работу над своей статьей. Она была посвящена сложной математической задаче, которая более десяти лет ставила в тупик математиков, — проблеме разрешения.

Тьюринг написал отличную статью, но на тот момент понятия не имел, что его работа породит глобальную индустрию, от которой будем зависеть мы все и которая составит основу нашей цивилизации.

Многие не без оснований считают первым программистом дочь лорда Байрона Аду, графиню Лавлейс. Она первой поняла, что вычислительная машина может манипулировать не только числами, но и нечисловыми понятиями. Символами вместо цифр. Более того, Ада написала несколько алгоритмов для аналитической машины Чарльза Бэббиджа, которая, к сожалению, так и не была построена.

Но первые программы[1] для электронного компьютера написал именно Алан Тьюринг. И именно он первым *определил* профессию программиста.

В 1945 году Тьюринг написал код для автоматизированной вычислительной машины (ACE). Это был код на двоичном машинном языке, использующий числа с основанием 32. Раньше никто не видел ничего подобного, поэтому Тьюрингу пришлось изобретать такие концепции, как подпрограммы, стеки и числа с плавающей запятой.

Несколько месяцев поработав над основами и решив с их помощью ряд математических задач, Тьюринг написал отчет, в котором фигурировал следующий вывод:

> *Нам понадобится огромное количество способных математиков, поскольку, скорее всего, предстоит проделать много работы такого рода.*

[1] Некоторые указывают, что Конрад Цузе написал алгоритмы для своего электромеханического компьютера до того, как Тьюринг запрограммировал ACE.

«Огромное количество». Как Тьюринг узнал? На самом деле он понятия не имел, насколько пророческим было его заявление. Теперь у нас их великое множество.

Что еще он сказал? «Способные математики». Считаете ли вы себя способным математиком?

Еще в этом отчете были слова:

> *Одной из трудностей станет необходимость придерживаться определенных практик, позволяющих не терять из виду то, что мы делаем.*

«Определенные практики!» Как он узнал об этом? Как он смог заглянуть на 70 лет вперед и увидеть, что нашей проблемой станет отсутствие дисциплинированного следования утвержденным практикам?

Семьдесят лет назад Алан Тьюринг заложил первый камень в фундамент профессионального программирования. Он сказал, что мы должны быть способными математиками, придерживающимися соответствующих практик.

Можем ли мы сказать это о себе? Можете ли вы сказать это о себе?

75 ЛЕТ

Время жизни одного человека. Вот сколько лет нашей профессии на момент написания этой статьи. Всего семьдесят пять. Что же произошло за эти шестьдесят и еще пятнадцать лет? Более внимательно посмотрим на историю, которую я рассказывал в главе 1.

В 1945 году в мире существовал один компьютер и один программист. Алан Тьюринг. Но количество программистов быстро увеличивалось. Будем использовать этот факт как отправную точку.

> *1945. Компьютеры: O(1). Программисты: O(1). 1945.*

За последующее десятилетие резко возросла надежность, стабильность и энергопотребление электронных ламп, что дало возможность создавать более мощные компьютеры.

К 1960 году компания IBM продала 140 компьютеров серии 700. Это были огромные и дорогие машины. Их могли позволить себе только военные, правительство и очень крупные корпорации. Кроме того, компьютеры были медленными, ограниченными в ресурсах и хрупкими.

Именно в этот период Грейс Хоппер придумала концепцию языка более высокого уровня и ввела термин *компилятор*. К 1960 году ее работа привела к появлению языка COBOL.

В 1952 году Джон Бэкус представил спецификацию языка FORTRAN. За этим быстро последовало развитие языка ALGOL. К 1958 году Джон Маккарти разработал язык LISP. Постепенно языков становилось все больше.

В те времена не было ни операционных систем, ни фреймворков, ни библиотек подпрограмм. Если компьютер выполнял какие-то операции, то лишь потому, что его на них запрограммировали. Соответственно, для поддержания работы одного компьютера в те дни требовался штат из дюжины или более программистов.

К 1960 году, через 15 лет после появления первой программы, в мире насчитывалось O(100) компьютеров. Программистов было на порядок больше: O(1000).

Кем были эти программисты? Это были такие люди, как Грейс Хоппер, Эдсгер Дейкстра, Джон фон Нейман, Джон Бэкус и Джин Дженнингз. Ученые, математики и инженеры. Большинство из них уже сделали карьеру и уже разбирались в бизнесе и в принятых практиках. Многим, если не большинству, было 30, 40 и 50 лет.

1960-е стали десятилетием транзисторов. Постепенно эти маленькие, простые, недорогие и надежные устройства заменили электронные лампы. Влияние, которое транзисторы оказали на компьютеры, изменило правила игры.

К 1965 году выпущено более 10 тысяч компьютеров IBM 1401 на базе транзисторов. Их можно было взять в аренду примерно за 2500 долларов в месяц, что было вполне доступно для тысяч предприятий среднего размера.

Программировались эти машины на таких языках, как Assembler, Fortran, COBOL и RPG. И всем компаниям-арендаторам требовался штат программистов для написания приложений.

В то время IBM была не единственной компанией, производившей компьютеры, поэтому количество компьютеров, которое появилось в мире к 1965 году, можно оценить как O(10 000). И если предположить, что для поддержания работы каждого компьютера требовалось десять программистов, то количество программистов в мире составляло O(100 000).

Через 20 лет после написания первой программы миру уже требовалось несколько сотен тысяч программистов. Откуда они брались? Математиков, ученых и инженеров, которые могли бы удовлетворить эту потребность, не хватало. Из университетов не выходили подготовленные специалисты, поскольку нужных учебных программ попросту не существовало.

В результате компании отбирали лучших и умнейших из бухгалтеров, клерков, планировщиков и любых других сотрудников, обладавших техническими способностями. И таких нашлось достаточно много.

Но это снова были люди, которые уже состоялись как профессионалы в другой области. Им было от тридцати до сорока. Они уже знали, что такое сроки и обязательства, что нужно оставить, а от чего отказаться.[1] Хотя эти люди не были математиками, они были дисциплинированными профессионалами.

Но колесо истории продолжало вращаться. К 1966 году компания IBM ежемесячно производила тысячу компьютеров семейства 360. Они

[1] Приношу извинения Бобу Сигеру. (Фраза what to leave in, and what to leave out взята из песни Сигера Against the Wind).

появлялись повсюду. И для того времени были чрезвычайно мощными. Модель 30 имела 64 Кбайт памяти и могла выполнять 35 тысяч инструкций в секунду.

Именно в этот период, в середине 1960-х, Оле-Йохан Даль и Кристен Нюгор изобрели первый объектно-ориентированный язык Simula-67. В этот же период Эдсгер Дейкстра изобрел структурное программирование, а Кен Томпсон и Деннис Ритчи — язык C и операционную систему UNIX.

А колесо истории катилось дальше. В начале 1970-х стали широко использоваться интегральные схемы. Эти малютки могли содержать десятки, сотни и даже тысячи транзисторов, что позволило сильно уменьшить размер электронных схем.

Так родился мини-компьютер.

В конце 1960-х и начале 1970-х годов компания Digital Equipment Corporation продала 50 тысяч компьютеров PDP-8 и сотни тысяч компьютеров PDP-11.

И они были не единственными! Продажи мини-компьютеров начали активно расти. К середине 1970-х годов существовали десятки торгующих ими компаний, так что через 30 лет после написания первой программы, к 1975 году, в мире насчитывалось около 1 миллиона компьютеров. А что с количеством программистов? К тому моменту соотношение начало меняться, и количество компьютеров приближалось к количеству программистов. Соответственно, к 1975 году программистов было O(1E6).

Откуда все они взялись? Кто они?

Это был я. Я и мои приятели. Я и мое поколение молодых, энергичных, мозговитых мальчиков.

Десятки тысяч новых выпускников в области электронной инженерии и информатики. Мы все были молоды. Мы все были умны. Мы все в Соединенных Штатах были обеспокоены призывом в армию. И почти все мы были мужчинами.

Дело не в том, что большинство женщин не выдерживало конкуренции в этой области. Это началось только в середине 1980-х годов. Просто на тот момент в программирование приходило гораздо больше мальчиков.

На моей первой работе в 1969 году коллектив состоял из нескольких десятков программистов в возрасте от 30 до 40 лет. И от трети до половины из них составляли женщины.

Десять лет спустя я работал в компании, где насчитывалось около 50 программистов, и женщинами из них было только трое.

Итак, через 30 лет после написания Тьюрингом первой программы демографический состав программистов резко сместился в сторону очень молодых мужчин. Сотни тысяч мужчин двадцати с чем-то лет. Как правило, это были не те, кого Тьюринг назвал бы дисциплинированными математиками.

Но бизнес нуждался в программистах. Спрос был огромным. И недостаток дисциплинированности молодых людей компенсировался их энергией.

Мы были дешевой рабочей силой. Сегодня программисты имеют высокие стартовые зарплаты, а тогда компании могли нанимать их за относительно небольшие деньги. В 1969-м моя зарплата составляла 7200 долларов в год.

С тех пор это стало тенденцией. Каждый год появляются выпускники с образованием в области computer science, и кажется, что у промышленности в этом отношении просто ненасытный аппетит.

За 30 лет, отделяющих 1945-й от 1975-го, количество программистов выросло как минимум в миллион раз. За следующие 40 лет темпы роста немного замедлились, но по-прежнему остаются очень высокими.

Как вы думаете, сколько программистов было в мире к 2020 году? Если учесть еще и программистов на языке VBA[1], думаю, что се-

[1] Visual Basic for Applications.

годня в мире должны быть сотни миллионов представителей этой профессии.

Здесь явно прослеживается экспоненциальный рост. Экспоненциальная функция по мере роста удваивается. Можно легко сделать расчет. Эй, Альберт, какая скорость роста обеспечивает увеличение количества программистов с 1 до 100 миллионов за 75 лет?

Логарифм по основанию 2, который даст нам 100 миллионов, составляет примерно 27. Разделив 75 на 27, получим примерно 2,8. Фактически это означает, что количество программистов удваивалось где-то каждые два с половиной года.

На самом деле, как вы видели ранее, в первые десятилетия скорость была выше, а потом немного замедлилась. По моим оценкам, сейчас время удвоения составляет около пяти лет. Каждые пять лет количество программистов в мире удваивается.

Из этого факта следуют ошеломительные выводы. Если количество программистов в мире удваивается каждые пять лет, то получается, что половина из них имеет опыт работы менее пяти лет. И так будет до тех пор, пока продолжается удвоение. В результате индустрия программирования оказывается в шатком положении — в состоянии вечной неопытности.

БОТАНИКИ И СПАСИТЕЛИ

Вечная неопытность. О, не волнуйтесь, это не о вашей квалификации. Это означает, что к моменту, когда вы проработаете пять лет, количество программистов удвоится. А когда ваш опыт достигнет десяти лет, их станет больше в четыре раза.

Если посмотреть на количество молодых людей в программировании, напрашивается вывод, что это профессия для молодых. И возникает вопрос: «А где же люди в возрасте?»

Мы здесь! Мы никуда не ушли. Просто изначально нас было не очень много.

Проблема в том, что нас недостаточно, чтобы обучать всех менее опытных коллег. На каждого программиста с 30-летним стажем приходится 63 программиста, которым требуется обучение, причем тридцать два из них — полные новички.

Отсюда состояние вечной неопытности, для выхода из которого попросту нет достаточного количества наставников. Одни и те же старые ошибки повторяются снова и снова.

Но за последние 70 лет произошло кое-что еще. Программисты добились того, чего, я уверен, Алан Тьюринг никогда не ожидал: дурной славы.

Еще в 1960-х годах мало кто знал, кто такой программист. Их было недостаточно для какого-либо влияния на социум. Не было такого, чтобы программисты жили по соседству со многими людьми.

В 1970-х ситуация начала меняться. Отцы уже советовали сыновьям (а иногда и дочерям) получать образование в области computer science. В мире стало достаточно программистов, чтобы каждый мог знать кого-то из них, хотя бы через общих знакомых. Так родился образ занудного фанатика, поедающего кексы twinkie.

Компьютеры мало кто видел, но практически все о них слышали. Тем более что компьютеры появлялись в таких телешоу, как «Звездный путь», и таких фильмах, как «2001: Космическая одиссея» и «Колосс: проект Форбина». И там компьютеры слишком часто изображались как злодеи. Хотя в книге Роберта Хайнлайна 1966 года *The Moon Is a Harsh Mistress*[1] компьютер был самоотверженным героем.

Но обратите внимание, что в каждом из этих случаев программист не был значимым персонажем. Тогда общество не знало, как к ним относиться. Они были призрачными, скрытыми и какими-то незначительными по сравнению с самими компьютерами.

У меня остались теплые воспоминания об одной телевизионной рекламе той эпохи. Жена и муж, маленький занудный парень в очках,

[1] *Хайнлайн Р.* Луна — суровая хозяйка.

с карманным протектором и с калькулятором, сравнивали цены в продуктовом магазине. Миссис Ольсен описала мужа как «компьютерного гения» и продолжила рассказ о преимуществах кофе определенной марки.

Программист в этой рекламе был наивен, начитан и малозначителен. Умный человек, но лишенный мудрости и здравого смысла. Таких обычно не приглашают на свои вечеринки. Программисты считались людьми, которых часто били в школе.

К 1983 году начали появляться персональные компьютеры, и стало понятно, что подростки интересуются ими по многим причинам. В то время уже у многих был хотя бы один знакомый программист. Нас считали профессионалами, но по-прежнему совершенно непостижимыми.

В том же году в фильме «Военные игры» молодой Мэттью Бродерик играл разбирающегося в компьютерах подростка и хакера. Он взламывает систему военного командования США, думая, что это видеоигра, и начинает обратный отсчет до Третьей мировой войны. В конце фильма он спасает мир, убеждая компьютер, что единственный выигрышный ход — не играть.

Компьютер и программист поменялись ролями. Теперь именно компьютер стал доверчивым наивным персонажем, а программист превратился в проводника, если не в источник мудрости.

Нечто подобное мы видели в фильме «Короткое замыкание» 1986 года, в котором компьютеризированный робот, известный как Номер 5, доверчив и невинен, но учится мудрости с помощью своего создателя/программиста и его девушки.

К 1993 году все резко изменилось. В фильме «Парк Юрского периода» злодеем стал программист, а такой персонаж, как компьютер, вообще отсутствовал. Компьютер был просто инструментом.

Общество начало понимать, кто мы такие и какую роль играем. Всего за 20 лет программисты прошли путь от ботаника до учителя и злодея.

Но видение снова изменилось. В фильме 1999 года «Матрица» главные герои были и программистами, и спасителями. Ведь их божественная сила проистекала из способности читать и понимать «код».

Наши роли быстро менялись. Из злодея в спасители всего за несколько лет. Общество в целом начало понимать нашу способность творить как благо, так и зло.

ОБРАЗЦЫ ДЛЯ ПОДРАЖАНИЯ И ЗЛОДЕИ

Пятнадцать лет спустя, в 2014 году, я приехал в офис компании Mojang в Стокгольме, чтобы прочитать несколько лекций о чистом коде и разработке через тестирование. Если вы не знали, Mojang — это компания, которая выпустила игру Minecraft.

После этого, поскольку погода была отличной, мы с программистами из Mojang сели на террасе кафе поболтать. Вдруг к забору подбежал мальчик лет двенадцати и крикнул одному из программистов: «Ты Jeb?»

Он обращался к Йенсу Бергенстену, одному из ведущих программистов Mojang.

Парень попросил у Йенса автограф и засыпал его вопросами. Он не смотрел ни на кого другого.

А я просто сидел рядом...

Так или иначе, программисты стали образцами для подражания и кумирами для наших детей. Они мечтают вырасти такими, как Jeb, Dinnerbone или Notch.

Программисты, настоящие программисты стали героями.

Но там, где есть настоящие герои, должны быть и настоящие злодеи.

В октябре 2015 года генеральный директор компании Volkswagen North America Майкл Хорн давал показания в Конгрессе США относительно программного обеспечения автомобилей, которое

позволяло занижать реальный уровень выбросов вредных веществ в атмосферу, обманывая тестовые устройства Агентства по охране окружающей среды. Когда Хорна спросили, почему компания так поступила, он обвинил во всем программистов. Он сказал: «Пара инженеров-программистов поставила это по неизвестным причинам».

Разумеется, о «неизвестных причинах» Хорн лгал. Он знал истинные причины, как знала это и компания Volkswagen. Слабая попытка переложить вину на программистов была шита белыми нитками.

Но с формальной точки зрения он был совершенно прав. Этот мошеннический код написали какие-то программисты.

И из-за них — кем бы они ни были — мы снискали дурную славу. Будь у нас настоящая профессиональная организация, их лишили бы права работать в этой сфере. И это было бы правильно. Они предали всех нас. Они запятнали честь нашей профессии.

Вот к чему мы пришли. Прошло 75 лет. За это время мы прошли путь от ботаников до образцов для подражания и злодеев.

Общество только начало понимать, кто мы такие, а также какие угрозы и какие перспективы мы несем.

МЫ ПРАВИМ МИРОМ

Но общество еще не все понимает. Впрочем, как и мы сами. Дело в том, что мы, программисты, правим миром.

Это может показаться преувеличением, но давайте подумаем. Сейчас в мире больше компьютеров, чем людей. И эти компьютеры решают за нас множество важных задач. Они посылают нам напоминания. Они управляют нашими календарями. Они доставляют наши сообщения в Facebook и хранят наши фотоальбомы. Они выполняют телефонные звонки и доставляют текстовые сообщения. Они управляют двигателями наших автомобилей, а также тормозами, педалью газа, а иногда даже рулем.

Без них мы не можем готовить еду. Не можем стирать одежду. Они сохраняют тепло в наших домах. Развлекают, когда нам скучно.

Отслеживают нашу банковскую информацию и наши кредитные карты. Они помогают оплачивать наши счета.

На самом деле большинство людей в современном мире каждую минуту бодрствования взаимодействуют с какой-либо программной системой. Некоторые продолжают это взаимодействие даже во сне.

Дело в том, что без программного обеспечения в нашем обществе не происходит *ничего*. Не покупается и не продается ни один продукт. Не принимается и не применяется ни один закон. Не ездят автомобили. Не доставляются товары Amazon. Не работают телефоны. Не генерируется электричество. Не подвозят еду в магазины. Из кранов не течет вода. Ни одна из этих вещей не происходит без программного мониторинга и координации.

И это программное обеспечение пишем *мы*. Это делает нас повелителями мира.

Другие люди думают, что пишут правила. Но эти правила передаются нам. *Мы* же пишем правила, которые выполняются машинами, контролирующими и координирующими все аспекты нашей жизни.

Общество еще не совсем это понимает. Пока еще не понимает. Но в один прекрасный день это станет слишком очевидным.

Мы, программисты, тоже пока не до конца это понимаем. Пока. Но опять же близится день, когда мы окажемся лицом к лицу с неприятной правдой.

КАТАСТРОФЫ

За прошедшие годы мы видели множество катастроф, связанных с программным обеспечением. Некоторые из них были очень показательными.

Например, в 2016 году мы потеряли спускаемый аппарат «Скиапарелли», предназначавшийся для испытания технологии посадки на поверхность Марса. Из-за проблемы с программным обеспечением

была неверно рассчитана высота, и «Скиапарелли» совершил свободное падение с высоты около 4 километров.

В 1999 году мы потеряли аппарат для исследования марсианского климата. Это случилось из-за того, что команды по тяге двигателя в ПО аппарата использовали международную систему единиц (СИ), в то время как ПО на Земле, которое создавало эти команды, использовало британскую единицу измерений. Из-за этой ошибки аппарат прошел над поверхностью Марса на высоте 57 километров вместо расчетных 110 километров и разрушился в атмосфере.

В 1996 году окончился неудачей испытательный полет ракеты-носителя Ariane 5. Ракета была уничтожена через 37 секунд после запуска из-за того, что конвертация данных из 64-разрядного числа с плавающей запятой в 16-разрядное привела к зависанию компьютера. А процедура, обрабатывающая такую ситуацию, была исключена для увеличения производительности системы.

Нужно ли упоминать об аппарате лучевой терапии Therac-25, который из-за ошибок в программном обеспечении давал пациентам слишком большую дозу облучения, что привело к смерти трех человек и ухудшению состояния еще троих?

Или, может быть, следует поговорить о банкротстве компании Knight Capital, которая за 45 минут потеряла 460 миллионов долларов из-за установки флага, активировавшего давно не использовавшийся, но оставленный в системе код?

Или стоит вспомнить об ошибке переполнения стека, вызывавшей неконтролируемое ускорение автомобилей Toyota, из-за которого погибли 89 человек?

Или скандал вокруг проекта HealthCare.gov, когда программный сбой чуть не стал причиной отмены нового спорного американского закона в сфере здравоохранения?

Эти чрезвычайные происшествия обошлись в миллиарды долларов и множество человеческих жизней. И все они были вызваны программистами.

Мы, программисты, через написанный нами код убиваем людей.

Уверен, что конкретно вы пришли в эту профессию не для того, чтобы убивать. Вероятно, вы стали программистом из-за того, что в один прекрасный день испытали восторг, видя, как написанный вами бесконечный цикл выводит на экран ваше имя.

Но факты есть факты. К сожалению, сейчас наши действия могут разрушать судьбы, лишая людей средств к существованию, а то и жизни.

Возможно, недалек тот момент, когда какой-нибудь бедолага-программист сделает глупую ошибку, из-за которой погибнут десятки тысяч людей.

Это не досужие домыслы. Это просто вопрос времени.

И когда это произойдет, политики всего мира потребуют отчета. Потребуют, чтобы мы показали, как собираемся избегать подобных ошибок в будущем.

И если мы придем на этот суд без заявления об этических принципах, без перечня стандартов или определенных практик, если мы просто придем и начнем жаловаться на своих начальников, которые устанавливают необоснованные графики и сроки, — то будем признаны *виновными*.

КЛЯТВА

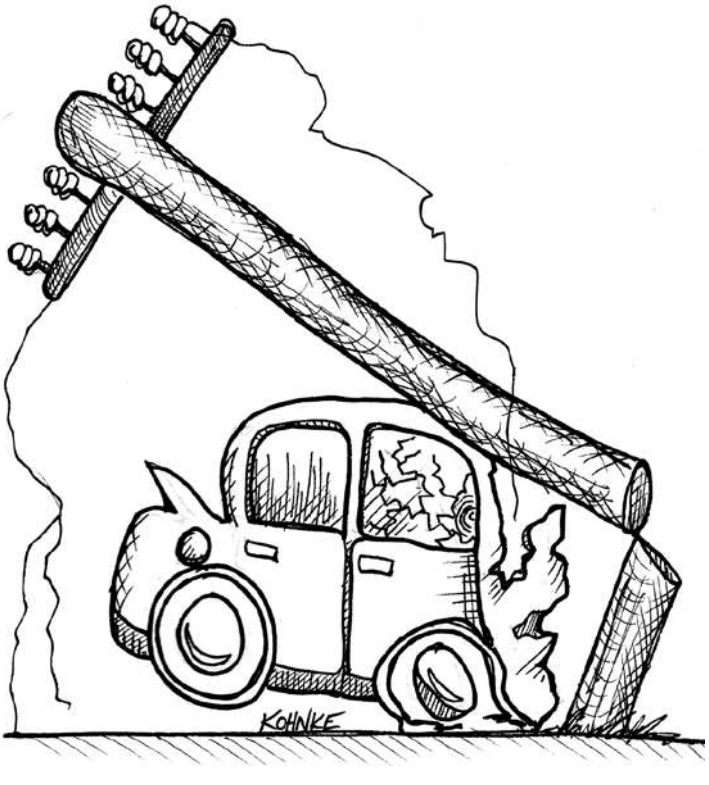
Чтобы начать обсуждение этических принципов разработчиков программного обеспечения, я предлагаю следующую клятву.

Для защиты и сохранения чести профессии программиста обещаю, что в меру своих возможностей и суждений:

1) *я не буду создавать вредоносный код;*
2) *я буду писать самый лучший код, на который способен. Я не позволю себе намеренно увеличивать количество кода с дефектами в поведении или структуре;*
3) *для каждой версии я буду предоставлять быстрое, надежное и воспроизводимое доказательство того, что все элементы кода работают должным образом;*

4) *я буду делать частые небольшие релизы, чтобы не мешать работе других;*
5) *при каждой возможности я буду бесстрашно и неустанно улучшать свои творения. Я никогда не позволю своему коду деградировать;*
6) *я приложу все усилия для поддержания своей и чужой продуктивности на как можно более высоком уровне. Я не буду делать ничего из того, что снижает эту продуктивность;*
7) *я буду постоянно заботиться о том, чтобы мои коллеги могли подменить меня, а я их;*
8) *я всегда буду давать честную и точную оценку. Я не буду давать обещания, которые не могу гарантированно выполнить;*
9) *я буду уважать своих коллег-программистов за их этические принципы, стандарты, дисциплину и навыки. Никакие другие качества или характеристики не будут влиять на мое отношение к коллегам;*
10) *я никогда не перестану учиться и совершенствовать свои профессиональные навыки.*

12 ВРЕД



Клятва содержит пункты, в которых упоминается причинение вреда.

ПРЕЖДЕ ВСЕГО — НЕ НАВРЕДИ

Обещание 1. Я не буду создавать вредоносный код.

Клятва профессионального программиста начинается с заповеди — *не навреди*! Она означает, что ваш код не должен наносить вред ни пользователям, ни работодателям, ни руководству, ни коллегам-программистам.

Вы должны знать, что делает ваш код. Вы должны быть уверены в том, что он работает и что он чистый.

Я уже упоминал случай в компании Volkswagen, когда программисты написали код, намеренно искажающий результаты тестирования выбросов. Это и есть пример вредоносного кода. С его помощью удалось обмануть Агентство по охране окружающей среды и получить разрешение на продажу автомобилей, в выхлопах которых было в 20 раз больше оксидов азота, чем считалось безопасным. То есть этот код потенциально наносил вред здоровью всех, кто оказывался рядом с этими автомобилями.

Как следовало поступить с этими программистами? Понимали ли они назначение этого кода?

Я бы их уволил и привлек к ответственности, поскольку они *должны* были понимать, что делают. Приказы руководства — не оправдание. Код писали именно программисты.

Это сложный вопрос, не так ли? Мы пишем код, заставляющий работать машины. И эти машины зачастую могут причинить огромный вред. Поскольку именно нас привлекают к ответственности за любой вред, нанесенный нашим кодом, мы должны заранее думать о том, как он будет использоваться.

Обязательства каждого программиста должны определяться в зависимости от уровня его опыта и ответственности. По мере роста опыта

и продвижения по карьерной лестнице возрастает и ответственность как за свои действия, так и за действия подчиненных.

Понятно, что нерационально возлагать на младших программистов такую же ответственность, как на начальников отделов. Как нерационально считать начальников отделов ответственными в той же мере, что и старших разработчиков.

Но лица, занимающие руководящие должности, должны соответствовать очень высоким стандартам и нести полную ответственность за своих подчиненных.

Это не означает, что в случае чего вся вина ляжет на старших разработчиков или руководителей. Каждый программист лично несет ответственность за то, что делает создаваемый им код, и за вред, который этот код может принести.

Не навреди обществу

Прежде всего, нельзя причинять вред обществу, в котором вы живете.

Это правило нарушили программисты из компании Volkswagen. Написанное ими программное обеспечение приносило пользу их работодателю, но нанесло ущерб обществу в целом. Мы, программисты, никогда не должны так поступать.

Но как понять, не навредишь ли ты обществу? Например, не вредно ли для общества программное обеспечение, управляющее системами вооружения? А ПО для азартных игр? Жестокие или сексистские видеоигры? Порнография?

Если программное обеспечение не выходит за рамки закона, то может ли оно все равно оказаться вредным для общества?

Откровенно говоря, здесь судить только вам. Вам просто нужно постараться сделать лучший выбор из возможных. Слушайте, что говорит вам ваша совесть.

Еще одним примером причиненного обществу вреда может служить неудачный запуск сервиса HealthCare.gov, хотя в данном случае вред

был непреднамеренным. Принятый Конгрессом США Закон о защите пациентов и доступном здравоохранении был подписан президентом в 2010 году.

Среди многочисленных директив этого закона фигурировало требование создать и активировать сайт 1 октября 2013 года.

Прописывать в законе дату активации новой массивной программной системы — безумие. Но настоящая проблема заключалась в том, что 1 октября 2013 года сайт действительно заработал.

Как вы думаете, сильно ли в тот день были напуганы программисты?

Боже мой, я думаю, они включили его.

Да, этого действительно не следовало делать.

Моя бедная мама. Что она теперь будет делать?

В этом случае техническая ошибка поставила под удар новую государственную политику. Из-за неудачного запуска сайта закон почти отменили. И что бы вы ни думали о политической подоплеке ситуации, обществу был нанесен вред.

Кто виноват в произошедшем? Все программисты, начальники отделов и директора, которые знали, что система не готова к запуску, и тем не менее молчали.

Этот вред обществу был причинен каждым разработчиком программного обеспечения, пассивно-агрессивно настроенным по отношению к своему руководству, каждым, кто говорил: «Я просто делаю свою работу, все остальное — не мои проблемы». Часть вины несет на себе каждый, кто знал, что что-то не так, но ничего не сделал, чтобы остановить развертывание системы.

Ведь одна из причин, по которой вас наняли как программиста, заключалась в том, что вы видите, когда что-то идет не так. У вас есть знания, позволяющие заранее идентифицировать проблему. Поэтому вы обязаны заявить о сложившейся ситуации до того, как произойдет нечто ужасное.

Нарушение функционирования

Вы должны *знать*, что ваш код работает. Вы должны *знать*, что он не нанесет вреда ни вашей компании, ни вашим пользователям, ни вашим коллегам-программистам.

Первого августа 2012 года технические специалисты компании Knight Capital Group загрузили на серверы новое программное обеспечение. К сожалению, обновлению подверглись только семь серверов из восьми.

Никто не знает, почему так получилось. Возможно, причиной стала чья-то небрежность.

Система компании Knight Capital Group занималась торговлей акциями на Нью-Йоркской фондовой бирже. В частности, она разбивала крупные сделки на множество более мелких, чтобы другие участники торгов не могли посмотреть размер исходной сделки и соответствующим образом скорректировать цены.

Простая версия этого алгоритма, которая называлась Power Peg, была отключена восемь лет назад и заменена инструментом SMARS (Smart Market Access Routing System). Как ни странно, старый код Power Peg из системы не удалили. Его просто отключили с помощью флага.

Этот флаг использовался для регулирования взаимоотношений между родительским и дочерним процессами. Установка этого флага активировала дочерние сделки. Как только родительский заказ выполнялся, он давал сигнал прекратить размещение дочерних сделок, и флаг снимался.

Программисты деактивировали код Power Peg, просто сняв флаг.

К сожалению, в новой версии программного обеспечения, установленной на семь серверов из восьми, назначение этого флага изменилось. После его установки восьмой сервер начал совершать дочерние сделки в очень быстром бесконечном цикле.

Программисты увидели, что что-то идет не так, но не поняли, что происходит. Для отключения неисправного сервера потребовалось

45 минут, в течение которых в бесконечном цикле заключались невыгодные сделки.

В результате за 45 минут торгов компания Knight Capital Group приобрела ненужные ей акции на сумму более 7 миллиардов долларов и была вынуждена продать их с убытком в 460 миллионов долларов. Хуже того, у компании было всего 360 миллионов долларов наличными, и она обанкротилась.

Сорок пять минут. Одна глупая ошибка. Четыреста шестьдесят миллионов долларов.

В чем состояла ошибка? Дело в том, что программисты *не знали*, что собирается делать их система.

Допускаю, что вы обеспокоены моим требованием досконально знать, как работает код. Я понимаю, что это недостижимая цель; всегда останутся какие-то пробелы.

Вопрос не в идеальной осведомленности обо всех нюансах. Но вы должны точно ЗНАТЬ, что код не причинит вреда.

Бедолагам из Knight Capital Group не хватило знаний, чтобы предотвратить ущерб. Учитывая, что было поставлено на карту, они должны были сделать все для недопущения такой ситуации.

Другим вопиющим примером стал случай с системой программного обеспечения автомобилей Toyota, которая заставляла машину неконтролируемо ускоряться.

Это программное обеспечение убило 89 человек. Еще больше получили травмы.

Представьте, что вы едете по оживленному деловому району, а ваша машина вдруг начинает разгоняться. Тормоза перестают работать. Считаные секунды, и вы мчитесь, минуя светофоры и пешеходные переходы, просто не имея возможности остановиться.

Расследование показало, что причиной могло быть ПО. И вполне возможно, было.

Программное обеспечение убивало людей.

У написавших этот код программистов *не было информации* о том, что их код никого не убьет. То есть они не могли гарантировать пользователям безопасность. Хотя должны были. Они просто обязаны были много раз все проверить и убедиться в том, что их код никого НЕ убьет.

Все опять сводится к степени риска. Когда ставки высоки, нужно максимально приблизить свою компетентность к совершенству. Если на карту поставлены жизни, то нужно быть совершенно *уверенным*, что ваш код никого не убьет. Если на карту поставлены состояния, то нужно быть совершенно *уверенным*, что ваш код никого не разорит.

С другой стороны, когда мы пишем чат-приложение или простой интернет-магазин, кажется, что речь уже не идет о таких серьезных вещах.

... или идет?

Представьте, что человеку, который пользуется вашим чатом, вдруг потребовалась неотложная медицинская помощь и он набрал «Помогите мне. Позвоните 911». Но из-за сбоя в приложении это сообщение было стерто.

Или представьте кражу личных данных и банковской информации клиентов вашего интернет-магазина через дыру в безопасности сайта.

А что, если из-за неправильного функционирования вашего кода клиенты фирмы, на которую вы работаете, уйдут к конкуренту?

Недооценить потенциальный вред от программного обеспечения очень легко. Утешительно думать, что ваше ПО не может никому навредить, поскольку оно недостаточно важное. Однако не забывайте, что написание программного обеспечения обходится недешево. Так что на карту поставлены по крайней мере деньги, потраченные на его разработку. Не говоря уже о доверии пользователей.

Одним словом, почти всегда на карту поставлено больше, чем вы думаете.

Нарушение структуры

Не следует нарушать структуру кода. Вы должны содержать код в чистоте и хорошо его структурировать.

Как вы думаете, почему программисты Knight Capital Group *не подозревали* о том, что их код может причинить вред?

Думаю, ответ очевиден. Они забыли, что в системе все еще присутствует алгоритм Power Peg. Они забыли, что он запускается с помощью флага, который они переназначили. Кроме того, все были уверены, что на всех серверах будет работать одно и то же программное обеспечение.

Программисты не знали, что у системы появилось вредоносное поведение из-за оставленного в ней мертвого кода. Именно поэтому так важна структура кода и его чистота. Чем запутаннее структура, тем труднее понять, как будет работать код. Чем больше беспорядка, тем выше неопределенность.

Возьмем, к примеру, случай с автомобилями Toyota. Почему программисты даже не подозревали, что их программы могут убивать людей? Как вы думаете, имеет ли значение тот факт, что в коде этой программы присутствовало более 10 тысяч глобальных переменных?

Путаница в коде не дает шанса понять, что делает программное обеспечение, и, следовательно, снижает вашу способность предотвращать вред.

Запутанное программное обеспечение — это вредоносное программное обеспечение.

Вы можете возразить, что иногда приходится идти на неаккуратные вставки в программу, чтобы как можно быстрее исправить неприятные ошибки.

Конечно. Разумеется. Если оперативная вставка запутанного кода позволяет предотвратить какие-то неприятные последствия, то ее нужно сделать.

Если глупая идея работает, значит, это не такая уж глупая идея.

Но оставив эту неаккуратную вставку, вы причините вред. И чем дольше она остается в коде, тем больше вреда может причинить.

Я напомню, что компания Knight Capital Group не обанкротилась бы, если бы алгоритм Power Peg удалили из кодовой базы. Именно этот старый, давно не использовавшийся код и стал причиной катастрофы.

Что подразумевается под «нарушением структуры»? Очевидно, что тысячи глобальных переменных — это структурный дефект. Как и мертвый код, оставленный в кодовой базе.

Нарушения структуры — это дефекты архитектуры и содержания исходного кода. Это все, что делает код трудным для чтения, понимания, изменения или повторного использования.

Каждый профессиональный разработчик программного обеспечения обязан знать способы создания и стандарты хорошей структуры. Уметь проводить рефакторинг, писать тесты, распознавать плохой код, разделять программные элементы и создавать соответствующие архитектурные границы. Он должен знать и применять принципы проектирования низко- и высокоуровневого дизайна. Любой старший разработчик отвечает за то, чтобы его менее опытные коллеги изучили все эти вещи и использовали при написании кода.

Программное обеспечение

Слово software (программное обеспечение) начинается с прилагательного *soft* (мягкое). Программное обеспечение должно быть *мягким*. Оно должно легко модифицироваться.

Важно помнить, зачем вообще появилось программное обеспечение. Мы придумали его, чтобы упростить управление поведением машин. Так что трудное в редактировании ПО противоречит самой причине своего существования.

Ценность программного обеспечения заключается в двух факторах. Во-первых, это его поведение, во-вторых, его «мягкость». Клиенты и пользователи ожидают, что мы сможем легко и без больших затрат менять это поведение.

Какой из этих факторов более весом? Для ответа на этот вопрос проведем простой мысленный эксперимент.

Представьте две программы. Одна работает идеально, но мы не можем ничего в ней изменить. Другая ничего не делает правильно, зато легко модифицируется. Какая из них ценнее?

Ненавижу напоминать прописные истины, но если вы помните, требования к программному обеспечению имеют тенденцию меняться. А при изменении требований первая программа станет бесполезной, причем навсегда.

Зато вторую программу мы можем заставить работать именно благодаря легкости ее модификации. Для этого могут потребоваться время и деньги, зато потом она будет работать вечно при минимальном количестве затрат.

Следовательно, во всех ситуациях, кроме самых неотложных, следует отдавать приоритет «мягкости».

Что я подразумеваю под *неотложными* ситуациями? Например, катастрофу, из-за которой компания теряет 10 миллионов долларов в минуту. Здесь нужно действовать немедленно.

Ничего подобного не может случиться в начале создания программного обеспечения. Это ни в коем случае не экстренная ситуация. Более того, на старте можно быть абсолютно уверенным только в том, что вы создаете не тот продукт.

Потому что ни один продукт не выдерживает контакта с пользователем. Как только продукт попадает к людям в руки, сразу же выясняется, что вы не так реализовали сотни различных аспектов. И если это программное обеспечение невозможно легко модифицировать, то оно обречено.

Это одна из самых больших проблем новых проектов. Предприниматели считают, что действовать нужно как можно быстрее и ради этого вполне допустимо пренебрегать любыми правилами и оставлять любой беспорядок. В большинстве случаев такой подход замедляет работу задолго до первого развертывания. Создание ПО пойдет бы-

стрее, а проживет оно намного дольше, если сохранять его структуру неповрежденной.

Спешка при создании программного обеспечения никогда не окупается.

Брайан Марик

Тесты

Все начинается с тестов. Вы пишете их первыми и очищаете первыми. Вы можете быть уверены, что каждая строка кода работает, благодаря тестам, которые позволяют это доказать.

Как предотвратить нарушение поведения кода без тестов, удостоверяющих его работоспособность?

Как предотвратить повреждение структуры кода без тестов, позволяющих его очистить?

И как гарантировать полноту набора тестов, не следуя трем законам разработки через тестирование (TDD)?

Является ли разработка через тестирование необходимым условием профессионализма? Неужели я считаю, что вы не можете быть профессиональным разработчиком программного обеспечения, если не практикуете TDD?

Да, я думаю, что это так. Вернее, мы постепенно к этому идем. Мне кажется, что относительно быстро настанет момент, когда большинство программистов согласятся с тем, что практика TDD входит в минимальный набор дисциплин и моделей поведения, отличающих профессионального разработчика.

Почему я так считаю?

Потому что, как я уже говорил, мы правим миром! Мы пишем правила, заставляющие мир функционировать.

В нашем обществе без программного обеспечения ничто не покупается и не продается. Почти вся переписка осуществляется с помощью ПО.

Создаются почти все документы. Принимаются и применяются законы. В повседневной жизни мы не делаем практически ничего, что не было бы связано с программным обеспечением.

Без ПО наша жизнь останавливается. Оно стало важнейшим инфраструктурным компонентом нашей цивилизации.

Общество этого пока не осознает. Впрочем, до конца этого не осознаем даже мы, программисты. Но постепенно приходит понимание. Понимание того, что от программного обеспечения зависят как человеческие жизни, так и огромные деньги. При этом его пишут люди, не претендующие даже на минимальный уровень дисциплины.

Поэтому да, я считаю, что TDD или какая-то похожая на нее практика в конечном итоге начнет считаться минимальным стандартом поведения профессиональных разработчиков. Думаю, наши клиенты и пользователи будут настаивать на этом.

ЛУЧШАЯ РАБОТА

> *Обещание 2. Я буду писать самый лучший код, на который способен. Я не позволю себе намеренно увеличивать количество кода с дефектами в поведении или структуре.*

Кент Бек однажды сказал: «Сначала заставьте его работать. А затем перепишите его правильно».

Написать работающую программу — первый и самый простой шаг. Куда сложнее следующий — очистить код.

К сожалению, слишком многие программисты считают, что как только программа начала работать, все готово. И тут же переходят к написанию следующей программы.

В результате появляется множество запутанного, нечитаемого кода, замедляющего прогресс всей команды разработчиков. Программисты так поступают, поскольку уверены, что самое главное — скорость. Они знают, что им много платят, и чувствуют себя обязанными предоставить много функционала за короткий промежуток времени.

Но создание программного обеспечения представляет собой сложный и долгий процесс, поэтому таким программистам кажется, что они работают слишком медленно. Внутреннее ощущение того, что они подводят команду, заставляет их торопиться. Эти программисты поспешно заставляют код заработать и объявляют, что все готово. И все равно у них остается чувство, что потрачено слишком много времени.

Я веду множество семинаров, во время которых предлагаю студентам выполнить небольшие проекты. Таким способом я пытаюсь дать им опыт программирования, возможность попробовать новые методы и практики. Меня не волнует, закончат ли они проект, ведь этот код нигде не будет использоваться.

Но люди все равно спешат. Некоторые даже остаются после занятий, чтобы заставить работать этот никому не нужный код.

Так что торопливость возникает совсем не из-за давления со стороны руководства. Настоящее давление у человека в голове, если его самооценка зависит от скорости разработки.

Делаем это правильно

Как я упоминал в начале этого раздела, ценность программного обеспечения заключается в двух вещах. Во-первых, в его поведении, во-вторых, в его структуре. Причем, как я уже отмечал, второе важнее первого. Это связано с тем, что в долгосрочной перспективе программные системы должны уметь реагировать на изменения в требованиях.

Сложности в модификации программного обеспечения означают, что его непросто привести в соответствие с новыми требованиями. А значит, оно быстро устареет.

Чтобы можно было идти в ногу с требованиями, структура программного обеспечения должна быть достаточно чистой и допускающей изменения. Именно это позволит при изменении требований поддерживать ПО в актуальном состоянии, то есть сохранять свою ценность с минимальными усилиями со стороны разработчиков.

Когда обычно происходит изменение требований? Чаще всего в начале проекта, сразу после того, как пользователи посмотрят на работу исходного функционала. Именно в этот момент видно, что система делает на самом деле, и есть возможность сравнить ее поведение со своими ожиданиями.

Поэтому если от вас требуют быстрой разработки, структура системы должна быть чистой с самого начала. Устроенный на старте бардак затормозит выпуск даже первой версии.

Хорошая структура позволяет обеспечить нужное поведение, поэтому профессиональные разработчики отдают более высокий приоритет именно структуре кода.

Таким образом, важно сохранять структуру системы как можно более чистой на протяжении всего жизненного цикла проекта.

Что такое хорошая структура

Хорошая структура упрощает тестирование, изменение и повторное использование кода. Изменения в одной части кода не затрагивают остальные части. Редактирование одного модуля не требует массовых перекомпиляций и повторных развертываний. Политики высокого уровня хранятся отдельно и не зависят от низкоуровневых деталей.

Плохая структура делает систему жесткой, хрупкой и неподвижной. Это традиционные *признаки плохого проекта*.

Речь о жесткости заходит, когда из-за незначительных изменений приходится повторно выполнять компиляцию, сборку и развертывание больших частей системы. Систему называют жесткой, когда усилия, затрачиваемые на интеграцию изменения, намного превосходят стоимость самого изменения.

О хрупкости говорят, когда незначительные изменения в поведении системы вызывают изменения в большом количестве модулей. В такой ситуации мы рискуем, меняя одно поведение, одновременно изменить какое-то другое. Фактически это означает отсутствие кон-

троля над программным обеспечением — вы понятия не имеете, к чему приведут те или иные ваши действия.

Неподвижностью системы называется ситуация, когда из-за запутанного кода вы не можете извлечь модуль с необходимым поведением для его использования в другой системе.

Все это структурные, а не поведенческие проблемы. Система может проходить все тесты и соответствовать всем функциональным требованиям, но оказаться почти бесполезной, поскольку ею слишком сложно манипулировать.

Это своего рода парадокс. Многие системы, корректно реализующие ценное поведение, при этом имеют настолько плохую структуру, что ценность системы фактически сходит на нет.

Называя систему *бесполезной*, я не слишком сильно выражаюсь. Вы когда-нибудь участвовали в огромной *перестройке*? Это ситуация, когда разработчики сообщают руководству, что единственный способ добиться прогресса — полностью перепроектировать всю систему. Фактически это означает, что разработчики считают текущую систему бесполезной.

Согласие руководства на эту полную переделку опять же означает, что руководство согласно с оценкой разработчиков, то есть с тем, что текущая система бесполезна.

Что порождает эти плохо спроектированные и по этой причине бесполезные системы? Зависимости исходного кода! Существует ли лекарство от этой напасти? Конечно. Это управление зависимостями!

Как осуществляется управление зависимостями? Получить систему, не имеющую признаков плохого проекта, позволяют принципы SOLID[1] — пять основных принципов объектно-ориентированного программирования и проектирования.

---

[1] Эти принципы описаны в книгах: *Мартин Р.* Быстрая разработка программ. Принципы, примеры, практика; *Мартин Р.* Чистый код. Создание, анализ и рефакторинг.

Поскольку структура имеет бóльшую ценность, чем поведение, а для ее обеспечения необходимо хорошее управление зависимостями, вытекающее из принципов SOLID, получается, что общая ценность системы зависит от правильного применения принципов SOLID.

Громкое заявление, не так ли? Возможно, вам будет трудно в это поверить. Ценность системы зависит от принципов проектирования. Но именно к такому выводу приводит логическая цепочка, да и у многих из вас есть опыт, подтверждающий этот тезис. Так что имеет смысл самым серьезным образом обдумать его.

Матрица Эйзенхауэра

Генерал Дуайт Д. Эйзенхауэр однажды сказал: «У меня есть два типа проблем: срочные и важные. Срочные — не важные, а важные — всегда несрочные».

Это утверждение глубоко истинно, особенно когда речь заходит об инженерно-технических работах. Можно даже назвать это девизом инженера:

Чем выше срочность, тем меньше важность.

На рис. 12.1 представлена матрица решений Эйзенхауэра: срочность откладывается по вертикали, а важность — по горизонтали. В итоге мы получаем четыре варианта: срочные и важные задачи, срочные и неважные, важные, но несрочные и неважные и несрочные.

Теперь определим приоритетность задач. Два очевидных случая: важные и срочные вверху и неважные, несрочные внизу.

Вопрос в том, как определить приоритетность двух средних вариантов: срочная, но неважная задача и важная, но несрочная? За какую из них нужно браться в первую очередь?

Очевидно, что важные вещи приоритетнее неважных. Я вообще бы сказал, что если что-то неважно, то за это вообще не нужно браться. Заниматься неважными делами — пустая трата времени.

Срочные важные	**Срочные неважные**	
Несрочные важные	**Несрочные неважные**	

Рис. 12.1. Матрица решений Эйзенхауэра

Но при таком подходе остается всего два варианта. В первую очередь мы делаем важные и срочные дела. А потом — важные, но несрочные.

При этом обратите внимание, что такой параметр, как срочность, связан со временем. А вот важность с ним не связана. Вещи, которые важны, рассчитываются на долгосрочную перспективу. А срочные дела касаются сиюминутных аспектов. Структура надолго определяет поведение системы. Поэтому она относится к важным задачам. Поведение же определяет систему здесь и сейчас, то есть это срочная задача.

Итак, структура, как важная задача, ставится на первое место, а поведение оказывается на втором.

Ваш начальник может не согласиться с такой расстановкой приоритетов, поскольку забота о структуре не входит в его обязанности. Это ваша обязанность. Начальник же просто ожидает, что вы будете поддерживать чистоту структуры, занимаясь срочной задачей реализации поведений.

Выше я цитировал Кента Бека: «Сначала заставьте его [код] работать. А затем перепишите его правильно». А теперь утверждаю, что структура имеет более высокий приоритет, чем поведение. Получилась проблема курицы и яйца, не так ли?

Первым делом мы заставляем код работать потому, что структура должна поддерживать поведение; следовательно, сначала нужно реализовать поведение, а затем можно придать ему правильную структуру. Но структура важнее поведения. Поэтому мы придаем ей более высокий приоритет и начинаем разбираться сначала со структурными проблемами, а только потом с поведенческими.

Для разрешения этого парадокса нужно разбить проблему на крошечные части. Например, мы берем пользовательскую историю и реализуем ее в виде кода. Заставляем этот код работать, а затем придаем ему правильную структуру. Мы не начинаем писать код для следующей истории *до тех пор, пока* код текущей не приобретет нужную структуру. Структура кода текущей истории имеет более высокий приоритет, чем поведение следующей.

Единственный момент: истории слишком большие. Нам нужны меньшие фрагменты. Здесь куда больше подойдут тесты. Вот они имеют идеальный размер.

Сначала мы добиваемся прохождения кодом теста, а затем правим структуру этого кода. И только потом можно браться за следующий тест.

Фактически мы подошли к *моральному* обоснованию цикла TDD «красный → зеленый → рефакторинг».

Именно этот цикл помогает предотвратить нарушение как поведения, так и структуры. Именно он позволяет поставить структуру выше поведения. И именно поэтому TDD считается методом *проектирования*, а не техникой тестирования.

Программисты как заинтересованные лица

Запомните. Мы заинтересованы в успехе программного обеспечения. Мы, программисты, — тоже заинтересованная сторона.

Вы когда-нибудь смотрели на свою работу с этой точки зрения? Воспринимали себя как заинтересованную в успехе проекта сторону?

Впрочем, это риторические вопросы. Ведь успех проекта напрямую влияет на вашу карьеру и репутацию. Поэтому очевидно, что вы являетесь заинтересованной стороной.

И как заинтересованная сторона, вы имеете право голоса при выборе вариантов разработки и структурирования системы. Ведь вы в этой ситуации тоже рискуете.

Но вы больше чем просто заинтересованная сторона. Вы инженер. Вас наняли, поскольку вы умеете создавать программные системы и структурировать их таким образом, чтобы они работали долго. И именно поэтому вы отвечаете за производство самого лучшего продукта.

Как заинтересованное лицо, вы имеете право убедиться, что создаваемые вами системы не приносят вреда по причине плохого поведения или плохой структуры. Как инженер, вы обязаны это сделать.

Многие программисты не хотят такой ответственности. Им больше нравится, когда им указывают, что делать. Это совершенно непрофессионально. Программисты с такими взглядами должны получать минимальную заработную плату, поскольку именно столько стоит их работа.

Если ответственность за структуру системы не возьмете на себя вы, то кто это сделает? Ваш начальник?

А он знаком с принципами SOLID? А с шаблонами проектирования? Как насчет объектно-ориентированного проектирования и принципа инверсии зависимостей? Знает ли ваш начальник практики TDD? Понимает ли, что такое самошунтирование, связанный с тестом подкласс или скромный объект? Что вещи, которые меняются вместе, должны быть сгруппированы, а вещи, которые меняются по разным причинам, должны быть разделены?

Ваш босс понимает структуру? Или его понимание ограничивается поведением?

Структура имеет значение. Если о ней не будете заботиться вы, то кто этим займется?

Представьте, что начальник приказывает игнорировать структуру и полностью сосредоточиться на поведении. Вы просто обязаны отказаться. Вы заинтересованная сторона, и у вас есть на это право. Кроме того, вы инженер с определенными обязанностями, выполнения которых начальник отменить не может.

Возможно, вы боитесь, что за отказом последует увольнение. Но скорее всего, нет. Большинство руководителей привыкло к тому, что им приходится отстаивать вещи, которые они считают необходимыми и в которые верят, соответственно, они уважают тех, кто готов делать то же самое.

Разумеется, последует борьба, даже противостояние, и вы будете чувствовать себя некомфортно. Но вы заинтересованная сторона и инженер. Нельзя просто отступить и смириться. Это непрофессионально.

Большинство программистов не любят вступать в конфронтацию. Но взаимодействие с конфликтными начальниками — это навык, которому следует учиться. Мы должны научиться бороться за вещи, которые считаем правильными. Брать на себя ответственность за то, что имеет значение, и бороться за это — поведение профессионала.

Делать все возможное

В Клятве программиста вы обещаете делать все возможное.

Очевидно, что для программиста это вполне разумное обещание. Конечно, вы приложите все усилия и, конечно же, не будете сознательно выпускать вредоносный код.

И конечно же, это обещание не всегда категорично. Бывают моменты, когда приходится подстраиваться под график. Например, если для проведения коммерческого просмотра нужно придумать какое-то быстрое и грязное решение, то в этом нет ничего страшного.

Обещание не мешает даже поставлять клиентам код с далеко не идеальной структурой. Если структура почти правильная, а клиенты ждут программное обеспечение уже завтра, то так тому и быть.

Впрочем, это же обещание *обязывает* вас решить все проблемы поведения и структуры, прежде чем добавлять новые виды поведения. Недопустимо нагромождать новые поведения на заведомо плохую структуру, поскольку это ведет к *накоплению* дефектов.

Что делать, если начальник все равно приказывает двигаться вперед? Вот как следует выстроить разговор в этом случае.

Начальник: Сегодня вечером нужно добавить этот новый функционал.

Программист: Извините, но я не могу этого сделать. Сначала мне нужно почистить структуру кода.

Начальник: Почистишь завтра. А добавить функционал нужно сегодня вечером.

Программист: Я уже сделал так в прошлый раз, и теперь приходится подчищать в два раза больше. Мне действительно нужно закончить очистку кода, прежде чем добавлять что-то новое.

Начальник: Ты, кажется, не понимаешь. Это бизнес. Мы все время должны двигаться вперед. И если мы не сможем реализовать новый функционал, то потеряем деньги.

Программист: Я все это понимаю. И я согласен с тем, что мы должны реализовать этот функционал. Но если я не устраню накопившиеся за последние несколько дней структурные проблемы, то работа замедлится, и мы добавим еще меньше нового функционала.

Начальник: Знаешь, раньше ты мне нравился. Раньше я говорил, что ты замечательный работник. Но сейчас я уже так не думаю. Возможно, тебя вообще лучше уволить.

Программист: Это ваше право. Но я почти уверен, что вам нужна быстрая и корректная реализация функционала. А если я не выполню очистку кода сегодня вечером, то работа начнет замедляться. И мы будем предоставлять все меньше и меньше функций.

Поверьте, я так же, как и вы, хочу двигаться вперед быстро. Вы наняли меня, поскольку я знаю, как этого добиться. Поэтому позвольте мне делать мою работу.

Начальник: Ты действительно думаешь, что все замедлится, если сегодня не сделать очистку кода?

Программист: Я знаю, что будет. Я уже сталкивался с этим. Как, кстати, и вы.

Начальник: И это обязательно нужно делать сегодня вечером?

Программист: Я не чувствую себя в безопасности, когда бардак усугубляется.

Начальник: Сможешь ли ты добавить функционал завтра?

Программист: Да, и после очистки структуры это будет намного проще.

Начальник: Хорошо. Завтра. Но не позже. Иди, работай.

Программист: Спасибо.

Начальник: [в сторону] Мне нравится этот парень. У него есть мужество. У него есть смекалка. Он не отступил даже под угрозой увольнения. Он далеко пойдет, поверь мне, но не говори ему, что я так сказал.

ПОВТОРЯЕМОЕ ДОКАЗАТЕЛЬСТВО

Обещание 3. Для каждой версии я буду предоставлять быстрое, надежное и воспроизводимое доказательство того, что все элементы кода работают должным образом.

Может быть, это обещание кажется вам чрезмерным? Но разве чрезмерно ожидать, что написанный вами код действительно работает?

Позвольте представить вам Эдсгера Вибе Дейкстру.

Дейкстра

Эдсгер Вибе Дейкстра (Edsger Wybe Dijkstra) родился в Роттердаме в 1930 году. Он пережил немецкую оккупацию и бомбардировку родного города и в 1948 году окончил среднюю школу с высшими баллами по математике, физике, химии и биологии.

В марте 1952-го, в возрасте 21 года, всего за девять месяцев до моего рождения, он устроился программистом в Математический центр Амстердама. Эдсгер был самым первым программистом в Нидерландах.

В 1957 году он вступил в брак с Марией Дебетс. Тогда в Нидерландах перед бракосочетанием нужно было заполнять анкету. В графе «профессия» Дейкстра написал «программист», но его заставили заполнять все заново, заявив, что такой профессии не существует. В результате Дейкстра указал, что он «физик-теоретик».

К 1955 году, еще студентом, он пришел к выводу, что задачи программирования требуют более высокого интеллектуального напряжения, чем задачи теоретической физики, и решил, что будет заниматься программированием.

Решение выбрать карьеру программиста Дейкстра обсудил со своим руководителем, Адрианом ван Вейнгаарденом. Дейкстру волновало, что программирование в то время не признавалось ни профессией, ни наукой. Он опасался, что по этой причине его никто не будет воспринимать всерьез. Адриан ответил, что Дейкстра вполне может стать одним из основателей профессии и превратить программирование в науку.

Для достижения этой цели Дейкстра решил показать, что программное обеспечение — это формальная система, такая же, как математика. Он полагал, что программное обеспечение можно представить примерно в таком же виде, как Евклидовы *Начала* — то есть как систему постулатов, доказательств, теорем и лемм. Поэтому он занялся созданием языка и методики написания доказательств.

Доказательство правильности

Для доказательства правильности алгоритмов Дейкстра выбрал три методики: перечисление, индукцию и абстракцию. Перечисление применялось для доказательства правильности двух утверждений в последовательности или двух утверждений, выбранных с помощью логического выражения. Индукция применялась для доказательства корректности циклов. А абстракции позволяли разбирать группы утверждений на более мелкие доказуемые фрагменты.

Звучит сложно, не так ли?

Чтобы вы убедились, насколько это сложно, я хочу показать простую программу на Java для вычисления остатка от деления целого числа (листинг 12.1) и страницу с доказательством этого алгоритма (рис. 12.2)[1].

Листинг 12.1. Простая программа на языке Java

```
public static int remainder(int numerator, int denominator) (
  assert(numerator > 0 && denominator > 0);
  int r = numerator;
  int dd = denominator;
  while(dd<=r)
    dd *= 2;
  while(dd != denominator) {
    dd /= 2;
    if(dd<= r)
      r -=dd;
  }
  return r;
}
```

Думаю, вы сразу поняли, в чем состоит проблема такого подхода. И Дейкстра действительно на это жаловался:

> *Разумеется, я не считаю (по крайней мере, сейчас!), что программист должен проводить такое доказательство при любом написании простого цикла. Потому что в этом случае он вообще не сможет написать даже небольшую программу.*

Дейкстра надеялся, что такие доказательства станут более практичными, если создать библиотеку теорем.

Но он и представить себе не мог, насколько распространенным и повсеместным станет программное обеспечение. Первый программист Нидерландов не предвидел, что количество компьютеров превысит количество людей. Если бы Дейкстра знал, какое количество ПО будет работать в наших домах, в наших карманах и на наших запястьях,

[1] Это переведенный на язык Java фрагмент из работы Дейкстры.

Рис. 12.2. Страница с доказательством алгоритма

то понял бы, что библиотека теорем, которую он мечтал создать, оказалась бы слишком обширной. И у обычного человека просто не было бы шанса ее освоить.

Итак, мечта Дейкстры о явных математических доказательствах корректности программ канула в Лету. Впрочем, до сих пор есть те, кто вопреки всему надеется на возрождение формальных доказательств, но их взгляды не оказывают заметного влияния на индустрию программного обеспечения.

Но пусть мечта эта никуда не привела, она породила нечто основательное. То, чем мы пользуемся сегодня, почти не задумываясь об этом.

Структурное программирование

На заре программирования, в 1950-х и 1960-х, мы писали на таких языках, как Fortran. Вы когда-нибудь видели код на Фортране? Позвольте мне его продемонстрировать.

```
      WRITE(4,99)
99    FORMAT(" NUMERATOR:")
      READ(4,100)NN
      WRITE(4,98)
98    FORMAT(" DENOMINATOR:")
      READ(4,100)ND
100   FORMAT(I6)
      NR=NN
      NDD=ND
1     IF(NDD-NR)2,2,3
2     NDD=NDD*2
      GOTO 1

3     IF(NDD-ND)4,10,4
4     NDD=NDD/2
      IF(NDD-NR)5,5,6
5     NR=NR-NDD
6     GOTO 3

10    WRITE(4,20)NR
20    FORMAT(" REMAINDER:",I6)
      END
```

Эта небольшая программа реализует тот же алгоритм вычисления остатка от деления, что и предыдущая программа на языке Java.

Обратите внимание на оператор `GOTO`. Вы, скорее всего, не часто встречали что-то подобное. Дело в том, что в настоящее время его применение не одобряется. Более того, он отсутствует в большинстве современных языков.

Откуда взялось такое отношение к оператору `GOTO`? Почему языки программирования его больше не поддерживают? Потому что в 1968 году Дейкстра написал в журнале *Communications of the ACM* статью с заголовком «Доводы против оператора GOTO»[1].

Почему Дейкстра выступил против оператора `GOTO`? Чтобы это понять, нужно вспомнить три методики доказательства правильности функции: перечисление, индукцию и абстракцию.

Для успешного выполнения перечисления нужно, чтобы каждое последовательное утверждение можно было анализировать независимо от остальных и чтобы результат одного утверждения передавался в следующее. Очевидно, что перечисление является эффективным методом доказательства правильности функции только при условии, что каждое перечисляемое выражение имеет единственную точку входа и единственную точку выхода. В противном случае у нас не будет уверенности ни во входных, ни в выходных данных оператора.

Более того, индукция — это просто особая форма перечисления, когда мы предполагаем, что утверждение истинно для x, а затем методом перечисления доказываем, что оно истинно для $x + 1$.

Это означает, что тело цикла должно быть перечислимым. И иметь один вход и один выход.

Оператор `GOTO` считается вредным, поскольку позволяет перейти в середину перечисляемой последовательности или выйти из нее.

---

[1] *Dijkstra E. W.* Go To Statement Considered Harmful // Communications of the ACM 11, no. 3 (1968), 147–148. (Статья была опубликована под названием «Оператор GOTO считается вредным». С переводом статьи на русский язык можно ознакомиться здесь: http://hosting.vspu.ac.ru/~chul/dijkstra/goto/goto.htm. — *Примеч. ред.*)

Такое поведение делает невозможным доказательство правильности алгоритма путем перечисления или индукции.

Чтобы корректность кода оставалась проверяемой, Дейкстра рекомендовал составлять код из трех стандартных строительных блоков.

- **Последовательность**, которая выглядит как два или более операторов, упорядоченных во времени. Это все неразветвленные строки кода.
- **Ветвление** выглядит как два или более утверждений, выбираемых с помощью предиката. Это операторы `if/else` и `switch/case`.
- **Итерация** выглядит как оператор, повторяемый под управлением предиката. Это циклы `while` и `for`.

Дейкстра показал, что программа любой сложности может быть составлена только из этих трех блоков и что структурированные таким способом программы доказуемы.

Он назвал эту парадигму *структурным программированием*.

Зачем это нужно, если мы не собираемся заниматься доказательствами корректности? Дело в том, что если что-то доказуемо, значит, об этом можно рассуждать. И наоборот, если что-то недоказуемо, значит, об этом нельзя рассуждать. Но без возможности рассуждать мы не можем должным образом осуществить тестирование.

Функциональная декомпозиция

Идеи Дейкстры не сразу стали популярными. В 1968 году предпочитали языки, в которых оператор `GOTO` использовался очень активно, поэтому идея отказаться от него или отрегулировать его применение вызвала непонимание.

Споры вокруг идей Дейкстры бушевали несколько лет. Интернета в то время не было, поэтому программисты писали письма в редакции основных журналов по программному обеспечению. Разыгрывались настоящие битвы. Некоторые называли Дейкстру богом. Другие

утверждали, что он глупец. Словом, всё как в современных социальных сетях, только медленнее.

Но со временем дебаты сходили на нет, а позиция Дейкстры получала все большую поддержку, пока мы не пришли к тому, что имеем сейчас. А сейчас в большинстве языков программирования просто нет оператора `GOTO`.

В настоящее время структурное программирование практикуют все. Потому что языки программирования не дают нам выбора. Мы все составляем программы из последовательностей, ветвлений и итераций. И очень немногие до сих пор регулярно пользуются ничем не ограниченным оператором `GOTO`.

Непреднамеренным побочным эффектом такого подхода стало появление метода, называемого *функциональной декомпозицией*. Это процесс, при котором программа, начиная с верхнего уровня, рекурсивно разбивается на все более мелкие доказуемые блоки. Это выполняемый сверху вниз процесс рассуждений, стоящий за структурным программированием.

Связь между структурным программированием и функциональной декомпозицией стала основой структурной революции 1970-х и 1980-х годов. В этот период Эд Йордан, Ларри Константин, Том Демарко и Мейлер Пейдж-Джонс популяризировали методы структурного анализа и структурного проектирования.

Разработка через тестирование

Цикл TDD «красный → зеленый → рефакторинг» представляет собой функциональную декомпозицию. В итоге все сводится к написанию тестов для небольших фрагментов задачи. Для этого нужно функционально разложить задачу на доступные для тестирования элементы.

В результате каждая созданная путем разработки через тестирование система состоит из полученных благодаря функциональной декомпозиции элементов, соответствующих блокам структурного программирования. А значит, система, которую они составляют, доказуема.

И тесты тому подтверждение.

Или, скорее, тесты — это *теория*.

Тесты, которые создаются в процессе TDD, не являются формальным математическим доказательством, как того хотел Дейкстра. Более того, он говорил, что тестированием можно доказать только неправильность программы, но невозможно доказать ее правильность.

Но, на мой взгляд, Дейкстра ошибался. Он рассматривал программное обеспечение как своего рода математику. Он хотел, чтобы мы сформировали надстройку из постулатов, теорем, следствий и лемм.

Вместо этого мы поняли, что программное обеспечение — это своего рода наука, которую мы подтверждаем экспериментами. Мы создаем теоретическую надстройку на основе прохождения тестов, как это делают все другие науки.

Доказана ли теория эволюции, или теория относительности, или теория Большого взрыва, или любая другая из основных научных теорий? Нет. Их невозможно доказать математически.

Тем не менее в определенных пределах мы считаем эти теории истинными. Каждый раз, садясь в машину или самолет, мы верим в истинность законов движения Ньютона. Каждый раз, используя систему GPS, мы уверены, что теория относительности Эйнштейна верна.

Тот факт, что мы математически не доказали правильность этих теорий, не означает отсутствия доказательств, позволяющих полагаться на них, причем даже в ситуациях, когда речь идет о нашей жизни.

Такое же доказательство дает нам TDD. Не формальное математическое, а экспериментальное эмпирическое. Такое же, как и те доказательства, от которых каждый день зависит наша жизнь.

И это возвращает нас к третьему обещанию в Клятве программиста:

> *Для каждой версии я буду предоставлять быстрое, надежное и воспроизводимое доказательство того, что все элементы кода работают должным образом.*

Быстро, надежно и воспроизводимо. *Быстро* означает, что набор тестов должен выполняться за очень короткое время. За минуты, а не за часы.

Надежно означает, что после успешного прохождения набора тестов продукт можно передавать заказчику.

Воспроизводимо означает, что эти же тесты могут быть запущены кем угодно в любое время, чтобы убедиться, что система работает правильно. Более того, мы хотим, чтобы тесты запускались много раз в день.

Некоторым может показаться, что просить такого уровня доказательств — это чересчур и что к программистам не следует предъявлять столь высокие требования. Но я не могу представить себе никакой другой стандарт, который имел бы какой-либо смысл.

Когда клиент платит за разработку ПО, разве мы не обязаны, насколько это возможно, доказать, что наши программы делают именно то, за что он заплатил?

Конечно, мы должны. И мы обещаем это нашим клиентам, нашим работодателям и нашим товарищам по команде. Мы в долгу перед бизнес-аналитиками, тестировщиками и руководителями проектов. Но главным образом мы должны это самим себе. Ибо как можно считаться профессионалом, если ты не в состоянии доказать, что результат твоего труда делает ровно то, за что платил клиент?

Давая это обещание, вы должны предоставить не формальное математическое доказательство, о котором мечтал Дейкстра, а скорее набор тестов, который охватывает все необходимые способы поведения, выполняется за секунды или минуты и дает один и тот же четкий результат «годен/не годен» при каждом запуске.

13 ВЕРНОСТЬ СВОИМ ПРИНЦИПАМ

Некоторые обещания в Клятве предполагают верность своим принципам и честность.

МАЛЫЕ ЦИКЛЫ

> *Обещание 4. Я буду делать частые небольшие релизы, чтобы не мешать работе других.*

Создание небольших релизов означает, что мы каждый раз меняем небольшое количество кода. Даже в огромной системе эти дополнительные изменения должны иметь небольшой размер.

История управления исходным кодом

Вернемся ненадолго в 1960-е. Как выглядела система управления исходным кодом, когда этот код создавали, пробивая отверстия в перфокартах (рис. 13.1)?

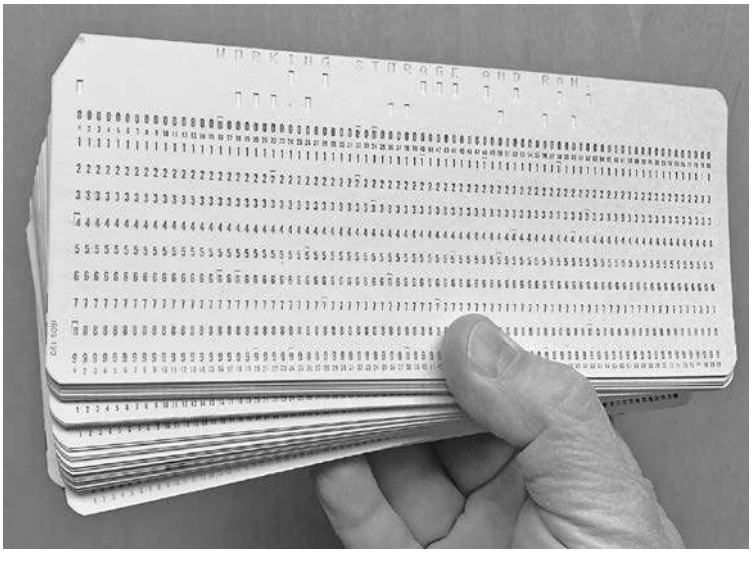


Рис. 13.1. Перфокарта

Тогда исходный код не хранился на диске. Его не было внутри компьютера. Вы в буквальном смысле слова держали его в руках.

Как в то время выглядела система контроля версий? Это был ящик стола.

Каждый программист лично *владел* своим исходным кодом, так что необходимости в «контроле версий» не было. К этому коду все равно не мог прикоснуться никто другой.

Так было на протяжении большей части 1950-х и 1960-х. Никто и не мечтал о чем-то вроде системы управления версиями. Исходный код держали под контролем, положив его в ящик или шкаф.

Если требовалось отредактировать исходный код, его просто брали из шкафа. А закончив, клали обратно.

Проблемы слияния попросту не существовало. Два программиста физически не могли одновременно внести изменения в один модуль.

Но в 1970-х ситуация начала меняться. Появилась привлекательная идея хранить исходный код на магнитной ленте или даже на диске. Мы написали программы для редактирования строк, которые позволяли добавлять, заменять и удалять строки в исходных файлах на ленте. Об экранных редакторах речь пока не шла. Директивы добавления, изменения и удаления *пробивались* на перфокартах. Читая исходную ленту, редактор добавлял изменения с этих карт и записывал новую ленту с кодом.

Вы можете подумать, что это ужасно. В принципе, да. Вспоминая те годы, я соглашаюсь с тем, что это было ужасно. Но лучше, чем работа с программами, пробитыми на картах! Вы не поверите, сколько весили 6000 строк кода. Почти 13 килограммов. Что делать, если эти карты разлетятся по полу, оказываясь под мебелью и даже падая в вентиляционные отверстия?

Упавшую ленту по крайней мере можно просто поднять.

При этом обратите внимание на такой факт. В процессе редактирования кода на ленте мы получали вторую ленту. При этом старая никуда не девалась. Если она просто возвращалась на стойку, то

в нее мог внести изменения кто-то другой, что порождало проблему слияния.

Чтобы предотвратить такую ситуацию, до завершения редактирования и тестирования исходную ленту программист хранил у себя. Затем лента с новой мастер-версией кода возвращалась на стойку. То есть контроль исходного кода физически осуществлял владелец ленты.

Защита исходного кода в то время требовала соблюдения предписанных процедур и соглашений. Программного обеспечения для этой цели не существовало, были только установленные нами правила. Но тем не менее уже тогда концепция управления исходным кодом отделилась от самого исходного кода.

По мере роста систем росло количество программистов, одновременно работающих над одним кодом. Если кто-то удерживал у себя мастер-ленту дольше пары дней, то для всех остальных это становилось настоящей проблемой.

Поэтому мы решили извлекать из мастер-ленты модули. В то время идея модульного программирования была достаточно новой. А идея программы, составленной из множества файлов с исходным кодом, была революционной.

В обиход вошли информационные доски. Такие, как те, что показаны на рис. 13.2.

На такую доску прикреплялись этикетки с именами всех модулей системы. У каждого программиста была булавка своего цвета. Например, у меня синяя. У Кена красная. У моего приятеля СК желтая, и т. д.

Если я хотел отредактировать модуль Trunk Manager, то первым делом должен был посмотреть на информационную доску. Если в этикетке с именем этого модуля не было ничьей булавки, то я втыкал туда свою. После этого можно было взять со стойки мастер-ленту и скопировать ее.

На этой отдельной ленте я редактировал модуль Trunk Manager, причем не трогая ничего другого. Можно было спокойно заниматься

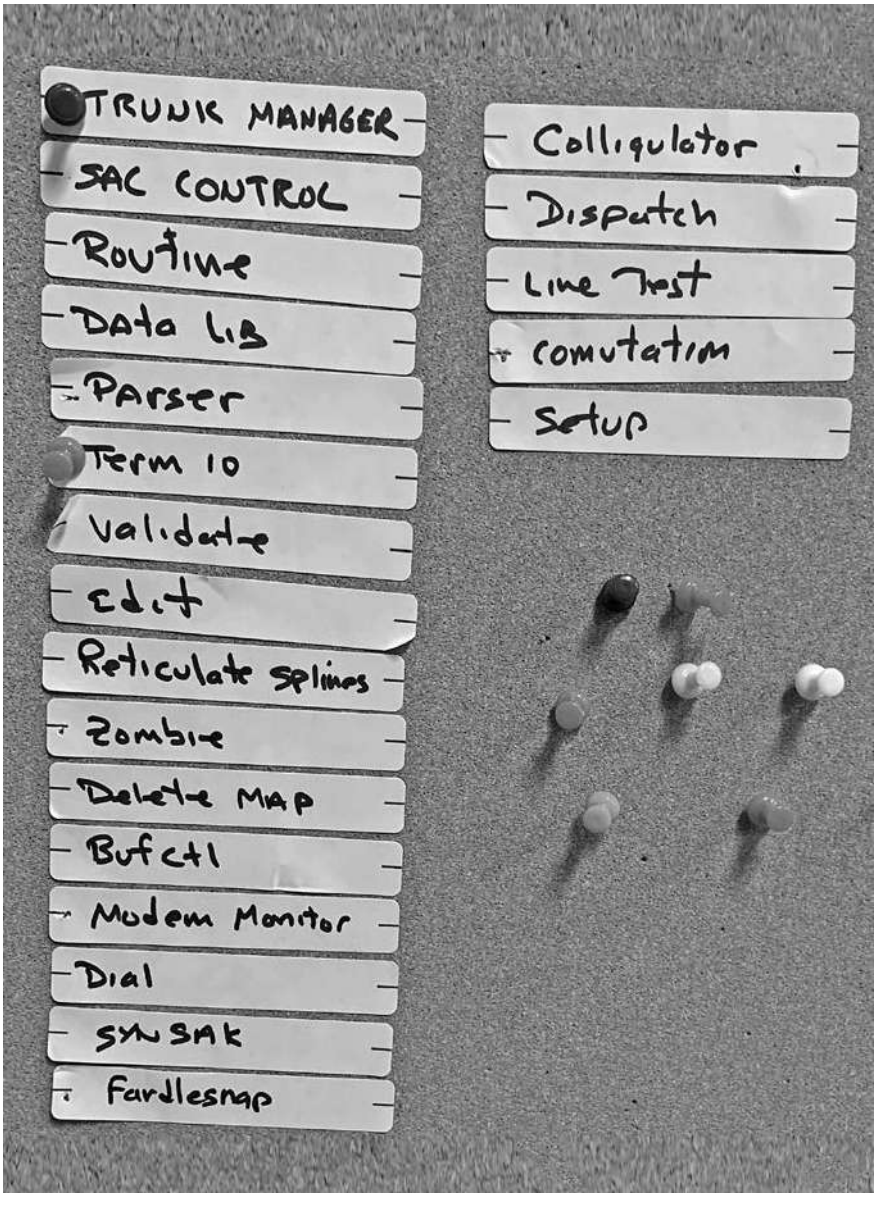


Рис. 13.2. Информационная доска

компиляцией, тестированием и очисткой, пока изменения не начинали работать. После того как в мастер-ленту со стойки копировались внесенные в модуль Trunk Manager изменения, можно было удалить с информационной доски свою булавку.

Этот подход работал, но исключительно потому, что мы работали в одном офисе и каждый из нас знал, чем занимаются остальные. И мы все время *обменивались* информацией.

Я кричал через всю лабораторию: «Кен, я собираюсь редактировать модуль Trunk Manager». Он говорил: «Вставь в него булавку». Я отвечал: «Уже».

Булавки были просто напоминанием. Мы все знали статус каждого фрагмента кода и кто над чем работает. Поэтому система функционировала.

Более того, она функционировала просто отлично. Осведомленность о работе коллег означала, что мы могли помогать друг другу. Вносить предложения. Предупреждать о проблемах, с которыми недавно столкнулись. И избегать слияний.

В то время слияния были не слишком веселым процессом.

Затем, в 1980-х, появились магнитные диски. Они были *большими* и *постоянными*. Их емкость составляла сотни мегабайтов, и уже не приходилось бегать за мастер-лентами к стойке.

Кроме того, у нас появились такие машины, как PDP11 и VAX. В обиход вошли экранные редакторы, настоящие операционные системы и несколько терминалов. Редактированием одновременно могли заниматься несколько человек.

Эпоха булавок и информационных досок завершилась.

Во-первых, к тому времени в лаборатории работало уже 20 или 30 программистов. У нас попросту не было такого разнообразия цветных булавок. Во-вторых, теперь в работе находились сотни модулей, и места на информационной доске перестало хватать.

К счастью, мы нашли решение.

В 1972 году Марк Рочкинд написал на языке SNOBOL[1] первую программу управления версиями. Она называлась SCCS (Source Code Control System).

Позже он переписал ее на C, и она вошла в дистрибутив UNIX для PDP11. SCCS работала только с одним файлом за раз, зато позволяла заблокировать данный файл, чтобы до завершения вашей работы никто не мог получить доступ к его редактированию. Это стало просто спасением.

В 1982 году Вальтер Тичи создал систему управления версиями RCS. Она тоже работала только с файлами и ничего не знала о проектах, но считалась улучшением по сравнению с SCCS и быстро стала стандартной.

Затем, в 1986 году, появилась система одновременных версий CVS. Она расширила возможности RCS до работы с целыми проектами и представила концепцию *оптимистической блокировки*.

До этого времени большинство систем управления исходным кодом напоминали наши цветные булавки. Если кто-то брал модуль в работу, то все остальные лишались доступа к нему. Такое поведение называется *пессимистической блокировкой*.

В CVS два программиста могут одновременно редактировать один файл. Система пытается объединить внесенные изменения и показывает предупреждение в случае, когда они конфликтуют друг с другом.

После этого новые системы управления версиями начали появляться сотнями и даже стали коммерческим продуктом. Часть из них использовала оптимистическую, а часть — пессимистическую блокировку. Стратегия блокировки породила в отрасли своего рода религиозные разногласия.

В 2000 году была создана централизованная система управления версиями Subversion. По своим характеристикам она значительно

[1] Использовавшийся в 1960-х годах прекрасный маленький язык обработки строк, в котором было множество средств сопоставления с шаблонами, присущих нашим современным языкам.

превосходила CVS и изрядно поспособствовала отходу от использования пессимистической блокировки. Кроме того, Subversion стала первой системой, которая применялась в облаке. Кто-нибудь помнит SourceForge?

До этого момента все системы управления версиями строились по принципу мастер-ленты, который применялся еще во времена информационных досок. Исходный код хранится в едином центральном репозитории, извлекается оттуда для редактирования, а сделанные коммиты записываются обратно.

Но мы уже стояли на пороге изменений.

Git

Шел 2005 год. Емкость дисков в наших ноутбуках измерялась гигабайтами. Данные по сети передавалась очень быстро, и скорость их передачи росла. Тактовая частота процессора стабилизировалась на уровне 2,6 ГГц.

Казалось бы, мы уже далеко ушли от управления версиями с помощью информационной доски. Тем не менее концепция мастер-ленты была по-прежнему в ходу. До сих пор существовал центральный репозиторий, из которого программисты брали код и после редактирования возвращали его на место. Каждая фиксация, каждая реверсия, каждое слияние требовали сетевого подключения к этому репозиторию.

Но тут появился git.

Нет, конечно, его появление предвосхитили такие системы контроля версий, как BitKeeper и monotone, но именно git привлек всеобщее внимание и изменил мир программирования.

Ведь именно git избавил нас от мастер-ленты.

Разумеется, нам все еще требовалась окончательная официальная версия исходного кода. Но git не создавал ее автоматически, вы сами выбирали, куда ее поместить.

Система контроля версий git хранит всю историю исходного кода на локальном компьютере. Фиксировать изменения, создавать ветки, проверять старые версии и вообще делать все, что можно было делать с централизованной системой, такой как Subversion, теперь можно было на своей машине. Для этого не требовалось подключаться к какому-то центральному серверу.

Появилась возможность в любой момент подключиться к другому пользователю и передать ему любые сделанные вами изменения. Или добавить внесенные другими изменения в свой локальный репозиторий. При этом мастер-версии кода не существует. Распределенная система управления версиями git следует принципу одноранговой сети.

Даже то место, где вы храните производственные версии, — всего лишь еще один узел, куда другие пользователи могут в любой момент добавить свой код или извлечь оттуда ваш.

Такой подход позволяет перед отправкой изменений сделать столько небольших коммитов, сколько вы считаете нужным. Ничто не мешает создавать коммит при каждом прохождении модульного теста.

И это подводит нас к сути этой исторической справки.

Если посмотреть со стороны на процесс эволюции систем управления версиями, то понятно, что он шел, возможно, бессознательно, к достижению одной определенной цели.

Короткие циклы

Еще раз рассмотрим все сначала. Сколько длился цикл, когда состояние исходного кода контролировалось путем физического удержания у себя стопки перфокарт?

Для работы с исходным кодом перфокарты доставались из шкафа и хранились у программиста до завершения проекта. После фиксации внесенных изменений стопка перфокарт возвращалась в шкаф. Время цикла равнялось времени работы над проектом.

Во времена булавок и информационной доски применялось такое же правило. Булавка оставалась в этикетке с названием модуля, над которым работал программист, пока он не завершал проект.

Даже в конце 1970-х и начале 1980-х, когда в обиход вошли SCCS и RCS, мы продолжали использовать пессимистическую стратегию блокировки, не позволяя другим прикасаться к модулям, пока не закончим их редактирование.

Ситуацию изменила CVS, по крайней мере для некоторых из нас. Оптимистическая блокировка означала, что один программист не мог заблокировать остальным доступ к модулю. Коммит по-прежнему создавался только после завершения проекта, но уже появилась возможность одновременной работы над кодом. Следовательно, среднее время между фиксациями в проекте резко сократилось. Но за это пришлось платить таким неудобством, как слияния.

Как же мы их ненавидели. Слияния ужасны, особенно при отсутствии модульных тестов! Это утомительная, отнимающая много времени и опасная процедура.

Отвращение к слияниям привело нас к новой стратегии.

Непрерывная интеграция

К 2000 году, когда в ходу еще были инструменты наподобие Subversion, мы начали вводить такую практику, как коммиты через каждые несколько минут.

Обоснование было простым. Чем чаще происходят коммиты, тем меньше вероятность, что возникнет необходимость слияния. И когда она все-таки возникнет, слияние оказывается тривиальным.

Мы назвали эту практику *непрерывной интеграцией*.

Конечно, возможность непрерывной интеграции принципиально зависит от наличия очень надежного набора модульных тестов. Без хороших модульных тестов легко сделать ошибку слияния и сломать

чужой код. Таким образом, непрерывная интеграция идет рука об руку с разработкой через тестирование (TDD).

Благодаря такому инструменту, как git, практически исчез предел, до которого можно сократить цикл. Но возникает вопрос: почему мы так озабочены сокращением цикла?

Потому, что длинные циклы мешают прогрессу команды.

Чем дольше промежуток между фиксациями, тем выше вероятность того, что кому-то в команде (а может быть, и всем) придется вас ждать. Что входит в противоречие с обещанием.

Вы можете подумать, что вышесказанное относится только к окончательному релизу. Но нет, речь идет о любом цикле. Об итерациях и спринтах. О цикле редактирование/компиляция/тестирование. О времени между коммитами. Обо *всем*.

Я напомню подоплеку происходящего: важно не мешать прогрессу других.

Ветки и переключатели

Раньше я был активным противником веток. Во времена использования CVS и Subversion я не разрешал членам моей команды разветвлять код. Я хотел, чтобы все изменения как можно чаще возвращались в основную линию разработки.

Обосновывал я это просто. Любая ветка — это код, который извлекли на большой срок. И как вы уже видели, долгосрочное извлечение препятствует прогрессу остальных членов команды, увеличивая время между интеграциями.

Но потом мы перешли на git — и за одну ночь все изменилось.

В то время я руководил проектом с открытым исходным кодом FitNesse, над которым работали около десяти человек. Я только что переместил репозиторий FitNesse из Subversion (Source Forge) в git (GitHub). И внезапно повсюду начали появляться ветки.

Первые несколько дней они приводили меня в замешательство. Должен ли я был пересмотреть свою нетерпимую позицию? Следовало ли отказаться от непрерывной интеграции и просто позволить всем желающим создавать ветки, забыв о проблемах, связанных с длительностью циклов?

Но потом мне пришло в голову, что ветки, которые меня так озадачили, имеют другую природу. Это поток коммитов, сделанных разработчиком между отправками их на удаленный сервер репозитория. На самом деле git всего лишь фиксировал действия разработчика между циклами непрерывной интеграции.

Так что я решил продолжить практику ограничения веток. Просто теперь в основную линию разработки сразу же возвращались не отдельные коммиты, а их наборы. Непрерывная интеграция по-прежнему осуществлялась.

Но если добавление в основную линию происходит примерно раз в час, то там появляется куча недописанного функционала.

Существует два способа решения этой проблемы: ветки и переключатели.

Стратегия ветвления проста. Для разработки функционала создается новая ветка исходного кода. После завершения код возвращается в основной репозиторий.

Но если ветка существует несколько дней или недель, то, скорее всего, ее возвращение потребует большого слияния. Кроме того, оно задержит работу остальной команды.

Хотя в некоторых случаях новый функционал настолько изолирован от остального кода, что его ветвление вряд ли приведет к большому слиянию. В этих обстоятельствах имеет смысл позволить разработчикам спокойно заниматься своим делом без постоянной интеграции.

Фактически подобная ситуация сложилась при разработке инструмента FitNesse несколько лет назад. Мы полностью переписывали синтаксический анализатор. Это был большой проект, занявший несколько человеко-недель. Возможности реализовать его пошагово не было.

Поэтому мы создали отдельную ветку и изолировали ее от остальной системы до момента готовности анализатора.

В итоге нам потребовалось слияние, но это был не самый плохой вариант. Мы хорошо изолировали анализатор от остальной системы. К тому же, к счастью, мы обладали полным набором модульных и приемочных тестов.

Несмотря на успешное завершение этого проекта, я считаю, что разработку нового функционала лучше все-таки оставлять в основной ветке и использовать переключатели, просто отключая его, пока он не будет готов.

Иногда в роли переключателей выступают флаги, но чаще мы исключаем возможность запуска частично написанного функционала в производственной среде с помощью шаблона «Команда», шаблона «Декоратор» и специальных версий шаблона «Фабричный метод».

Почти всегда мы просто отбираем у пользователей возможность запустить новый функционал. Я имею в виду, например, отсутствие нужной кнопки на веб-странице.

Впрочем, в большинстве случаев новый функционал добавляется в рамках текущей итерации — или, по крайней мере, к следующему релизу. Соответственно, нет необходимости использовать переключатели.

Переключатель требуется только в ситуациях, когда вы собираетесь выпускать продукт с частично незавершенным функционалом. Как часто они возникают?

Непрерывное развертывание

Представьте, как хорошо было бы устранить задержки между выпусками продукта. Что, если бы мы могли отправлять код в работу несколько раз в день? В конце концов, задержка релизов тормозит остальных членов команды.

Я хочу, чтобы вы могли смело выпускать свой код в производство несколько раз в день. Буквально при каждой отправке результатов работы в основной репозиторий.

Это, конечно, зависит от автоматизированного тестирования. От автоматизированных тестов, написанных программистами для проверки каждой строки кода, и автоматизированных тестов, написанных бизнес-аналитиками и тестировщиками для проверки всех желаемых вариантов поведения.

Помните, в предыдущей главе мы говорили о том, что тесты можно считать научным *доказательством* корректной работы кода? А если код работает как нужно, то логично выполнить его развертывание.

Кстати, именно таким способом можно понять, достаточно ли хороши ваши тесты. Если после их прохождения вы не готовы выполнить развертывание кода, значит, они далеки от совершенства.

Скорее всего, вы считаете, что настолько частое развертывание приведет к хаосу. Но дело в том, что *ваша* готовность не означает готовности *заказчиков*. Просто готовность в любой момент передать заказчикам готовый к развертыванию код — стандарт для команды разработчиков.

Более того, такой подход позволяет максимально сократить цикл введения программного обеспечения в эксплуатацию. В конце концов, чем большим количеством церемоний и ритуалов оно сопровождается, тем дороже стоит. Любой бизнес хотел бы избавиться от этих расходов.

Конечная цель любого бизнеса — непрерывное, безопасное и простое развертывание, которое происходит как бы между делом, не привлекая к себе особого внимания.

Поскольку развертывание часто требует большого объема работы с настройкой серверов и загрузкой баз данных, эту процедуру необходимо *автоматизировать*. А так как сценарии развертывания являются частью системы, для них пишутся тесты.

Скорее всего, идея непрерывного развертывания настолько далека от вашего текущего рабочего процесса, что кажется просто немыслимой. Но это не значит, что ничего нельзя сделать для сокращения цикла.

И кто знает? Если вы продолжите его сокращать месяц за месяцем, год за годом, то, возможно, однажды вы обнаружите, что выполняете развертывание непрерывно.

Непрерывная сборка

Очевидно, что для коротких циклов развертывания необходимы короткие циклы сборки. А для непрерывного развертывания — непрерывная сборка.

Возможно, у некоторых из вас сборка происходит медленно. Если это так, ускорьте ее. Я говорю серьезно. Современные системы работают настолько быстро, что оправдания медленной сборки не существует. Никакого. Увеличьте ее скорость. Считайте это задачей проектировщика.

Кроме того, я рекомендую приобрести инструмент для непрерывной сборки, например Jenkins, Buildbot или Travis. Настройте его так, чтобы сборка запускалась при каждой отправке сделанных изменений в удаленный репозиторий, и следите за тем, чтобы она проходила корректно.

Сбой сборки — это чрезвычайная ситуация. Действовать надо срочно. Я хочу, чтобы в этом случае всем членам команды рассылались электронные письма и текстовые сообщения. В буквальном смысле слова должна завыть сирена, а на столе генерального директора — загореться мигающая красная лампочка. Нужно, чтобы все срочно оставили свои дела и разобрались с чрезвычайной ситуацией.

Избежать такой ситуации достаточно просто. Перед отправкой изменений в удаленный репозиторий запускайте сборку вместе со всеми тестами в своей локальной среде. Отправлять код можно только после прохождения всех тестов.

Если даже в этом случае происходит сбой, значит, вы обнаружили какую-то проблему с рабочей средой, которую необходимо решить в кратчайшие сроки.

Оставлять произошедшее без внимания недопустимо, поскольку при таком подходе постепенно происходит привыкание к сбоям в процессе сборки и их игнорирование. Но чем чаще вы игнорируете эти сбои, тем более раздражающими становятся предупреждения об ошибках. И тем выше соблазн отключить тесты, которые система не проходит.

Вы обязательно включите их позже, когда исправите приводящие к сбоям ошибки. По крайней мере, вы так думаете.

Но после этого тестирование уже не дает истинных результатов.

После удаления тестов, которые не проходят, сборка снова осуществляется без проблем. И все чувствуют себя хорошо. Но это самообман.

Поэтому выполнять сборку следует непрерывно. И никогда не позволяйте этому процессу завершаться неудачно.

НЕУСТАННОЕ УЛУЧШЕНИЕ

> *Обещание 5. При каждой возможности я буду бесстрашно и неустанно улучшать свои творения. Я никогда не позволю своему коду деградировать.*

В своем посмертном послании основатель скаутского движения Роберт Баден-Пауэлл призвал: «Уходя, попытайтесь оставить этот мир немного чище, чем он был до вашего прихода». Именно этот принцип лег в основу моего правила: возвращайте в систему контроля версий код в лучшем состоянии, чем он был, когда вы начали с ним работать.

Каким образом? Делая небольшое доброе дело перед тем, как вернуть код в репозиторий.

Одним из таких небольших добрых дел может стать увеличение тестового покрытия.

Покрытие тестами

Следите ли вы за тем, какую часть вашего кода покрывают тесты? За тем, какой процент строк и веток покрыт?

Измерить этот показатель можно с помощью множества инструментов. Большая часть из них встроена в IDE и запускается тривиально,

поэтому нет никаких оправданий тому, что вы не знаете процент покрытия.

Что делать с этими цифрами? Для начала расскажу вам, чего с ними *не* следует делать. Не превращайте их в инструмент управления. Не делайте провальную сборку при слишком низком тестовом покрытии. Тестовое покрытие — сложная концепция, которой не стоит пользоваться так наивно.

Ведь в подобных случаях возникает искушение схитрить. Обмануть тестовое покрытие очень легко. Напомню, что инструменты охвата измеряют только объем *выполненного* кода, а не тот код, который был на самом деле протестирован. Поэтому показатель покрытия можно очень сильно увеличить, убирая утверждения из провальных тестов. Понятно, что метрика при этом становится бесполезной.

Показатели покрытия лучше всего использовать в качестве инструмента, который поможет улучшить код. Необходимо работать над *сознательным* приближением этой метрики к 100 процентам путем написания настоящих тестов.

Полный охват — это желательная, но в то же время асимптотическая цель. Большинство систем никогда не достигает 100 процентов, но это не должно удерживать вас от постоянного стремления к этому показателю.

Вот для чего нужны показатели покрытия. Это всего лишь средство измерения, которое поможет вам стать лучше, а не дубинка, с помощью которой можно наказать команду и создать неработающую сборку.

Мутационное тестирование

Стопроцентное покрытие тестами означает, что любое семантическое изменение кода должно вызывать сбой тестирования. Практика TDD хорошо подходит для приближения к этой цели, поскольку при неукоснительном следовании ей каждая строка кода будет написана таким образом, чтобы обеспечить прохождение провального теста.

Но идеальное соблюдение всех правил зачастую трудно реализуемо. Программисты тоже люди, а все практики подчиняются прагматическим соображениям. В реальности даже самый усердный и ориентированный на тестирование разработчик все равно оставит пробелы в тестовом покрытии.

Мутационное тестирование позволяет выявить эти пробелы. Существуют инструменты, помогающие его провести. Тестировщик запускает предоставленный вами набор тестов и измеряет охват. Затем он определенным образом меняет код и снова запускает набор тестов. Практикуются такие вещи, как замена `>` на `<`, или `==` на `!=`, или `x=<something>` на `x=null`. Каждое такое семантическое изменение называется *мутантом*.

Если набор тестов способен обнаружить изменение, то мутант считается убитым. Соответственно, мутанты, не обнаруженные при тестировании, называются выжившими. Как легко догадаться, наша цель состоит в том, чтобы гарантировать отсутствие *выживших мутантов*.

Проведение такого тестирования — порой крайне затратная по времени операция. Даже относительно небольшие системы могут занять несколько часов, поэтому подобные тесты лучше всего проводить по выходным или в конце месяца. Честно скажу, я изумлен тем, что могут обнаружить инструменты мутационного тестирования. Так что их обязательно нужно использовать время от времени.

Семантическая стабильность

Увеличением тестового покрытия и мутационным тестированием мы занимаемся, чтобы создать набор тестов, обеспечивающий *семантическую стабильность*. Семантика системы — это ее требуемое поведение. Набор обеспечивающих семантическую стабильность тестов дает сбой при любом нарушении требуемого поведения. Такие наборы тестов позволяют избавиться от страха перед рефакторингом и очисткой кода. Без них программистам зачастую очень страшно вмешиваться в систему.

Разработка через тестирование дает хороший старт для создания нужного набора тестов, но его недостаточно. Для продвижения в сторону

полной семантической стабильности следует применять такие инструменты, как тестовое покрытие, мутационное тестирование и приемочное тестирование.

Очистка

Возможно, самым эффективным из небольших добрых дел по улучшению кода является простая очистка — рефакторинг с целью улучшения.

Какие улучшения мы можем внести? Самое очевидное — устранение проблем в структуре кода. Но я часто чищу код, даже если такие проблемы в нем отсутствуют.

Я немного улучшаю имена, структуру, архитектуру. Эти изменения малозаметны. Некоторым даже может показаться, что код становится менее чистым. Но моя цель — не просто улучшить состояние кода. Раз за разом проводя незначительную очистку, я *знакомлюсь* с кодом. Мне становится удобнее с ним работать. Возможно, мои действия не дают объективного улучшения кода, зато улучшается мое понимание и моя способность работать с этим кодом. Очистка улучшает *меня* как разработчика этого кода.

Очистка обеспечивает еще одно преимущество, которое нельзя недооценивать. В процессе ее проведения я *зондирую* код. Один из лучших способов убедиться, что код остается гибким, — это регулярно его сгибать. Каждая маленькая чистка, которую я делаю, на самом деле является проверкой гибкости кода. Если она оказывается сложной, значит, я нашел область негибкости, которую теперь могу исправить.

Помните, что программное обеспечение должно быть мягким? Но как узнать, что оно мягкое? Регулярными проверками. Проводя небольшую очистку и внося небольшие улучшения и *чувствуя*, насколько легко или сложно происходит внесение изменений.

Творения

В обещании 5 фигурирует слово *творения* (creations). В этой главе я рассматриваю главным образом код, но он — не единственное, что

создают программисты. Мы пишем проекты и документы, графики и планы. Все эти артефакты являются творениями, которые следует постоянно улучшать.

Мы люди. Люди со временем улучшают вещи. Мы постоянно совершенствуем все, над чем работаем.

ПОДДЕРЖАНИЕ ВЫСОКОЙ ПРОДУКТИВНОСТИ

> *Обещание 6. Я приложу все усилия для поддержания своей и чужой продуктивности на как можно более высоком уровне. Я не буду делать ничего из того, что снижает эту продуктивность.*

Производительность. Это отдельная большая тема, не так ли? Как часто у вас возникает ощущение, что это единственное, что имеет значение в вашей работе? Если подумать, то продуктивность — это то, чему посвящена данная книга и все мои книги по программному обеспечению.

Все они о том, как двигаться быстро.

За семь десятилетий существования программного обеспечения мы узнали следующее: чтобы двигаться быстро, нужно двигаться хорошо.

Единственный способ быстро продвигаться вперед — делать все хорошо.

Именно поэтому вы поддерживаете чистоту своего кода и своих проектов. Вы пишете тесты, проверяющие семантическую стабильность, и поддерживаете высокий процент тестового покрытия. Вы знакомы с различными шаблонами проектирования и умеете ими пользоваться. Вы делаете методы небольшими, а имена описательными.

Но все это *косвенные* методы поддерживания производительности. А существуют и более прямые способы.

1. Устранение вязкости — поддержание эффективности вашей среды разработки.

2. Переключение внимания — умение фокусироваться на рабочих вопросах.
3. Управление временем — умение эффективно отделять рабочее время от времени, затрачиваемого на все остальные задачи.

Вязкость

Когда дело доходит до производительности, программисты часто демонстрируют поразительную близорукость. Основным компонентом продуктивности они считают свою способность быстро писать код.

Но написание кода — очень маленькая часть процесса. Даже если научиться писать код *бесконечно быстро*, общая производительность увеличится ненамного.

Потому что программный процесс — это еще и:

- сборка;
- тестирование;
- отладка;
- развертывание.

Не считая требований, анализа, проектирования, совещаний, исследований, инфраструктуры, инструментов и всего прочего, что входит в программный проект.

Поэтому такой навык, как быстрое создание кода, конечно, важен, но практически не помогает в решении стоящей перед нами задачи.

Взглянем на остальные части процесса разработки.

Сборка

Если после пятиминутного редактирования на сборку уходит 30 минут, то о высокой продуктивности речь уже не идет, не так ли?

Нет никаких причин, по которым в третьем десятилетии XXI века сборка должна занимать больше минуты или двух.

Прежде чем возражать, хорошенько подумайте. Как можно ускорить сборку? Вы абсолютно уверены, что в век облачных вычислений нет способа этого достичь? Найдите факторы, тормозящие процесс сборки, и устраните их. Считайте это задачей на поиск лучшего варианта дизайна.

Тестирование

Ваши тесты замедляют сборку? Могу только повторить свой совет. Ускорьте процедуру тестирования.

Посмотрим на ситуацию вот с какого ракурса. Мой бедный маленький ноутбук имеет четыре ядра с тактовой частотой 2,8 ГГц. Это означает, что в секунду он может выполнять *около 10 миллиардов инструкций*.

В вашей системе вообще где-то есть 10 миллиардов инструкций? Если нет, значит, вы можете протестировать ее целиком менее чем за секунду.

Если, конечно, некоторые инструкции не выполняются более одного раза. Например, сколько раз вам нужно проверить процедуру входа в систему, чтобы убедиться, что она работает? Вообще говоря, одного раза вполне достаточно. Сколько ваших тестов проверяет эту процедуру? Если их больше одного, то это пустая трата ресурсов!

Если для проведения каждого теста приходится сначала входить в систему, значит, эту процедуру на время тестирования следует как-то обойти. Воспользуйтесь шаблоном с подставными объектами. Или вообще удалите из системы, созданной для тестирования, процедуру входа.

Ненужных повторений в тестах следует всеми силами избегать, поскольку именно они могут сделать тестирование ужасно медленным.

Или другой пример. Сколько раз ваши тесты проходят через меню пользовательского интерфейса? Сколько тестов начинается сверху и проходит по длинной цепочке ссылок, чтобы наконец привести систему в состояние, в котором эти тесты можно запустить?

Более одного раза проходить через систему навигации — пустая трата времени! Поэтому создайте специальный API тестирования,

позволяющий тестам быстро привести систему в нужное состояние, без процедуры входа и прохода по всем пунктам меню.

Сколько раз нужно выполнить запрос, чтобы убедиться, что он работает? Один! Поэтому для тестирования потребуется подставной объект, имитирующий базу данных. Не позволяйте одним и тем же запросам выполняться снова и снова.

Медленно могут работать периферийные устройства. Диски. Вебсокеты. Экраны пользовательского интерфейса. Не позволяйте им замедлять тестирование. Используйте подставные объекты. Обходите их. Убирайте их с пути ваших тестов.

Не будьте терпимы к медленным тестам. Тесты должны работать быстро!

Отладка

Уходит много времени на отладку? Почему? По какой причине отладка идет медленно?

Вы используете для написания модульных тестов TDD, не так ли? И обязательно пишете приемочные тесты, верно? И измеряете хорошим инструментом тестовое покрытие? И периодически доказываете семантическую стабильность ваших тестов с помощью мутационного тестирования?

Если вы делаете все эти вещи или хотя бы *некоторые из них*, то время отладки может быть сведено к минимуму.

Развертывание

Процедура развертывания продолжается целую вечность? Почему? Надеюсь, вы *пользуетесь* сценариями развертывания, верно? Вы же не проводите развертывание вручную, не так ли?

Помните: вы программист. Развертывание — это процедура. Так автоматизируйте ее! И обязательно напишите для нее тесты!

У вас должна быть возможность в любой момент развернуть систему одним щелчком кнопки мыши.

Управление отвлекающими факторами

Один из самых пагубных разрушителей продуктивности — отвлечение от работы. Существует множество отвлекающих факторов. Важно уметь их распознавать и защищаться.

Встречи

Вашу работу тормозят встречи?

У меня есть очень простое правило. Оно выглядит так:

Когда встреча становится скучной, уходите.

Делать это нужно очень вежливо. Дождитесь паузы в разговоре и сообщите участникам, что, на ваш взгляд, ваш вклад больше не требуется. После чего спросите, не будут ли они возражать, если вы вернетесь к работе.

Не бойтесь уходить со встреч или собраний. Если этого не делать, то можно впустую потратить целую вечность.

По большому счету, имеет смысл отклонять большинство приглашений на встречи. Вежливый отказ от приглашения — лучший способ не попасть на долгое и скучное мероприятие. Не бойтесь пропустить что-то нужное. Если вы действительно понадобитесь, то к вам подойдут отдельно.

Получая приглашение на встречу, убедитесь, что вам действительно нужно туда идти. Сразу проинформируйте, что у вас всего несколько минут и что вы, скорее всего, уйдете до окончания беседы.

И обязательно садитесь поближе к двери.

Если вы ведущий группы или руководитель, то помните, что одна из ваших основных обязанностей — защищать продуктивность вашей группы, не пуская ее членов на собрания.

Музыка

Когда-то давно, перед тем как сесть за написание кода, я включал музыку. Мне *казалось*, что она способствует моей концентрации. Но, как

выяснилось, я ошибался. Со временем я понял, насколько обманчиво впечатление, что музыка помогает сконцентрироваться. На самом деле она отвлекает внимание.

Однажды, просматривая какой-то код годичной давности, я обнаружил, как на него повлияла музыка. Разбросанные по коду комментарии содержали слова из песни, которую я слушал в момент его написания.

С тех пор я пишу код в тишине, и я обнаружил, что мне гораздо больше нравится как результат, так и внимание к деталям, которое больше ни на что не отвлекается.

Программирование — это акт упорядочивания элементов процедуры с помощью последовательностей, ветвлений и итераций. Музыка состоит из тональных и ритмических элементов, соединенных с помощью последовательностей, ветвлений и итераций. Я допускаю, что прослушивание музыки задействует те же части мозга, что и программирование, тем самым частично поглощая способность писать код. Это моя личная теория, и мне кажется, что она не лишена оснований.

Возможно, у вас дела обстоят по-другому. Возможно, музыка вам помогает. Но может быть, это не так. Я бы посоветовал в течение недели попробовать программировать без музыки и посмотреть, как это скажется на количестве и качестве вашего кода.

Настроение

Важно понимать, что для продуктивной работы требуется умение управлять своим эмоциональным состоянием. Стресс может убить способность программировать. Он способствует нарушению концентрации и рассеянному состоянию ума.

Например, вы когда-нибудь замечали, что не можете программировать после серьезной ссоры с партнером? Нет, вы в состоянии напечатать что-то в своей IDE, но это нельзя назвать работой. Притворяться продуктивным можно, разве что зависая на каком-то скучном совещании, где не нужно уделять особого внимания происходящему.

Вот какое средство восстановления продуктивности в такой ситуации я обнаружил.

Нужно действовать. Попробуйте воздействовать на причину эмоции. Не пытайтесь писать код. Не пытайтесь заглушить свои чувства музыкой или встречами. Это не сработает. Действуйте, чтобы снизить интенсивность эмоциональных переживаний.

Если вы слишком грустны или подавлены, например, из-за ссоры с женой, то позвоните ей и попытайтесь поговорить. Даже если эта попытка ни к чему не приведет, вы обнаружите, что предпринятого действия иногда достаточно, чтобы очистить свой разум от посторонних мыслей и сосредоточиться на написании кода.

Понятно, что проблемы далеко не всегда легко разрешимы. Однако на самом деле все, что нужно сделать, — это убедить себя, что вы предприняли правильные действия. Обычно мне этого хватает, чтобы направить свои мысли в сторону работы.

Поток

Существует измененное состояние ума, которым наслаждаются многие программисты. Это состояние предельной концентрации сознания, когда кажется, что код сам вытекает из-под ваших пальцев. В этом состоянии программисты ощущают себя сверхэффективными и непогрешимыми.

Несмотря на ощущение эйфории, постепенно я обнаружил, что код, который я создаю в этом измененном состоянии, как правило, не очень хорош. В обычном состоянии внимания и сосредоточенности я показываю намного лучшие результаты. Поэтому сейчас я изо всех сил сопротивляюсь попаданию в поток. Парное программирование отлично помогает от таких вещей. Кажется, потоку мешает сама необходимость общаться и сотрудничать с кем-то.

Отсутствие музыки также помогает мне сопротивляться потоку, поскольку позволяет окружающей среде удерживать меня в реальном мире.

Если я обнаруживаю, что начинаю слишком сильно концентрироваться, то отрываюсь от написания кода и какое-то время занимаюсь чем-то другим.

Управление временем

Один из самых важных способов справиться с отвлекающими факторами — практика управления временем. Мне особенно нравится *техника Помидора*[1]. Помидор — итальянское слово, а английские команды обычно называют ее *техникой Томата*.

Цель техники — помочь в управлении временем и концентрацией в течение рабочего дня. Других назначений у нее нет.

В ее основе лежит очень простая идея. Перед началом работы вы ставите стандартный кухонный таймер (традиционно такие таймеры делались в форме помидора) на 25 минут.

Затем вы приступаете к работе и занимаетесь ею, пока не прозвенит таймер.

После этого делается перерыв на 5 минут, чтобы очистить сознание и размяться.

Затем все повторяется.

В 25 минутах нет ничего волшебного. Более того, подойдет любой промежуток времени от 15 до 45 минут. Но как только вы выбрали продолжительность рабочего отрезка, придерживайтесь его. Не меняйте размер помидоров!

Разумеется, если в момент срабатывания таймера я нахожусь в 30 секундах от завершения теста, я дождусь этого завершения. Но следовать заведенному ритуалу тоже немаловажно, так что больше чем на минуту я ни разу не задерживался.

---

[1] *Cirillo F.* The Pomodoro Technique: The Life-Changing Time-Management System. — Virgin Books, 2018.

Пока все звучит обыденно, но основное достоинство этой техники состоит в способе обработки прерываний, таких как телефонные звонки. Ваша задача — *защищать свой «помидор»*!

Скажите тому, кто пытается вас прервать, что вы уделите ему внимание в течение 25 минут или другого выбранного вами промежутка времени. Устраните прерывание как можно быстрее и вернитесь к работе.

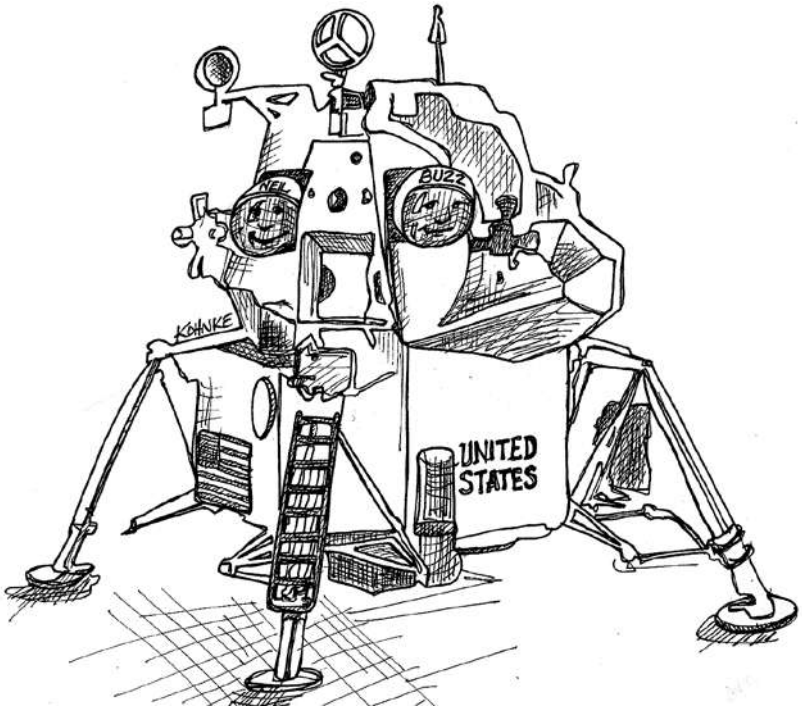
Во время паузы обработайте прерывание.

Естественно, в некоторых случаях время между «помидорами» может оказаться достаточно долгим, ведь далеко не все вопросы с прервавшим вас человеком можно решить быстро.

Но в этом и состоит прелесть этой техники. В конце дня при подсчете количества «съеденных помидоров» вы получаете оценку своей продуктивности.

Научившись разбивать день на «помидоры» и защищаться от помех, вы сможете планировать свой день, оценивая затраты времени на задачи в «помидорах» и вычисляя еженедельную «помидорную скорость».

14 РАБОТА В КОМАНДЕ



Остальные части Клятвы программиста связаны с приверженностью команде.

РАБОТАТЬ КАК ОДНА КОМАНДА

> *Обещание 7. Я буду постоянно заботиться о том, чтобы мои коллеги могли подменить меня, а я их.*

Разделение знаний между подразделениями чрезвычайно вредно как для команды, так и для организации в целом. При таком подходе потеря отдельного человека может означать потерю целого сегмента знаний. Еще это означает, что у членов команды недостаточно контекста, чтобы понять друг друга. Часто они друг друга попросту не слышат.

Эта проблема решается распространением знаний. Важно сделать так, чтобы каждый член команды много знал о работе своих коллег.

Лучше всего знания передаются путем совместной работы — в паре или в группе.

Истина в том, что едва ли найдется лучший способ повысить продуктивность команды, чем совместное программирование. Команда, понимающая, как связаны между собой обязанности каждого в коллективе, обязательно получится более продуктивной, чем группа разрозненных сотрудников.

Открытый/виртуальный офис

Важно и то, чтобы члены команды очень часто виделись и взаимодействовали друг с другом. Лучший способ добиться этого — посадить их в одну комнату.

В начале 2000-х я владел компанией, которая помогала организациям внедрять гибкие методологии разработки. Наши инструкторы и коучи приезжали в эти компании для оказания помощи. Перед началом

каждого мероприятия мы просили перепланировать офисные помещения таким образом, чтобы каждая команда могла работать в своей комнате. Не раз случалось так, что еще до начала занятий руководители сообщали нам, что команды стали намного продуктивнее просто потому, что оказались в одном помещении.

Я пишу этот абзац в первом квартале 2021 года. История с COVID-19 начинает понемногу затухать, и мы все с нетерпением ждем возвращения к нормальной жизни. Но в результате этих событий большое количество команд разработчиков все равно останется работать удаленно.

Удаленная работа никогда не сможет быть такой же продуктивной, как совместная работа в одном помещении. Даже с лучшими электронными помощниками видеть друг друга на экранах не так эффективно, как вживую. Тем не менее современные средства для совместной работы дают отличные результаты. Так что если вы работаете удаленно, *используйте их*.

Создайте для команды виртуальную комнату. Держите лицо каждого в поле зрения, а аудиоканал каждого максимально открытым. Нужно создать иллюзию комнаты, в которой идет совместная работа.

В настоящее время в Сети отлично поддерживается парное и групповое программирование. Даже находясь в разных городах или странах, можно легко обменяться экранами и программировать вместе. Все время держите видео и звук включенными. В процессе совместной работы важно видеть и слышать друг друга.

Удаленные команды должны изо всех сил стараться поддерживать одинаковое рабочее время. Это очень сложно, когда программисты живут в разных уголках земного шара. Постарайтесь добиться минимального разброса часовых поясов в каждой команде. Нужно сделать так, чтобы по крайней мере шесть часов подряд все могли присутствовать в виртуальной комнате.

Вы когда-нибудь замечали, как легко разозлиться на другого водителя? Это эффект лобового стекла. Когда людей отделяет друг от друга лобовое стекло, легко увидеть в другом глупца, невежу и даже врага.

Вы уже не видите в другом человека. Этот эффект наблюдается и при общении в Сети.

Чтобы избежать подобного, нужно несколько раз в год собирать команду в одном физическом помещении. Я рекомендую делать так по неделе в каждом квартале. Это поможет сохранить команду. Сложно попасть под эффект лобового стекла в отношении людей, с которыми вы обедали и физически сотрудничали две недели назад.

ЧЕСТНАЯ И СПРАВЕДЛИВАЯ ОЦЕНКА

> *Обещание 8. Я всегда буду давать честную и точную оценку. Я не буду давать обещания, которые не могу гарантированно выполнить.*

А теперь поговорим об оценке проектов и крупных задач, то есть вещей, реализация которых занимает много дней или недель. Об оценке небольших задач и историй я рассказывал в книге «Чистый Agile».

Умение оценивать — важный для каждого разработчика программного обеспечения навык, которым большинство из нас владеет крайне плохо. Этот навык важен, поскольку каждый бизнес, прежде чем выделить ресурсы на очередной проект, должен примерно представлять, во что тот обойдется.

К сожалению, непонимание сути оценок и неумение их делать привели к почти катастрофической потере доверия между программистами и бизнесом.

Известно множество сбоев программного обеспечения с миллиардными потерями. Часто причиной неудачи становилась неверная оценка. Нередки случаи, когда оценки отличались в два, три, даже четыре и пять раз. Но почему возникает такая ситуация?

В основном потому, что мы не понимаем, что такое оценки и как они создаются. Видите ли, чтобы оценки были полезными, они должны

быть честными, достоверными и точными. Но большинство оценок не обладают этими качествами. Более того, большинство их них — ложь.

Ложь

Большинство оценок некорректно, поскольку дается исходя из конкретной даты окончания проекта.

Например, вспомним историю портала HealthCare.gov. В законе, подписанном президентом США, была указана конкретная дата его запуска.

Честно говоря, от этой нелогичности тошнит. Невозможно передать, как это абсурдно. Программистов никто не просил оценить сроки реализации проекта; их просто уведомили, к какой дате все должно быть готово!

Надо ли говорить, что все связанные с проектом оценки не имели никакого отношения к реальности? Да и как они могли быть другими?

Как-то раз я консультировал одну команду, когда в комнату вошел руководитель проекта, молодой парень лет двадцати пяти. Он только что встречался со своим начальством и был заметно взбудоражен. И сразу же начал рассказывать, насколько важна дата окончания проекта. Он повторял: «Мы просто обязаны уложиться в этот срок. Я имею в виду, что у нас *действительно* нет другого выхода».

Понятно, что присутствующие только закатили глаза. Все равно их мнения никто не спрашивал. Их просто поставили перед фактом.

Оценки в такой ситуации — всего лишь ложь, поддерживающая график.

И это напоминает мне другого клиента, у которого на стене висел огромный план производства программного обеспечения, полный кругов, стрелок, меток и задач. Программисты называли его *закадровым смехом*.

В этом разделе мы поговорим о реальных, честных, аккуратных и точных оценках. Оценках, которые дают профессионалы.

Честность, безошибочность, точность

Самый важный аспект при оценке ситуации — честность. Оценки никому не принесут пользы, если не будут честными.

Я: Разреши спросить. Какую самую честную оценку ты можешь дать?

Программист: Я не знаю.

Я: Правильно.

Программист: Правильно что?

Я: Я не знаю.

Программист: Подождите. Вы попросили меня дать честную оценку.

Я: Правильно.

Программист: А я сказал, что я не знаю.

Я: Правильно.

Программист: Так какая она?

Я: Я не знаю.

Программист: Хорошо, каким образом я должен ее узнать?

Я: Ты же уже произнес ее.

Программист: Произнес что?

Я: Я не знаю.

Самая честная оценка, которую вы можете дать, — «я не знаю». Ее нельзя назвать особенно точной. В любом случае у вас есть *какие-то сведения* о ситуации. Задача состоит в том, чтобы количественно определить, что вы знаете, а что нет.

Прежде всего, ваша оценка должна быть безошибочной и точной. Это не означает, что вы обязаны назвать конкретную дату. Достаточно указать диапазон дат, в котором вы уверены.

Например, оценка между сегодня и через десять лет в ответ на вопрос, сколько времени вам понадобится, чтобы написать программу «Hello World», безошибочная. Но ей не хватает точности.

С другой стороны, ответ «вчера в 2:15 ночи» очень точный, но его нельзя назвать безошибочным, ведь вы еще даже не приступали к работе.

Видите разницу? Когда вы даете оценку, она должна быть честной как с точки зрения безошибочности, так и с точки зрения точности. Чтобы не ошибиться, нужно называть диапазон дат, в котором вы уверены. Чтобы быть точным, нужно сузить этот диапазон до уровня, в котором вы опять же уверены.

И для обеих этих операций необходима беспощадная честность.

Быть честным в таких вещах можно только в случае, если у вас есть представление о том, насколько вы можете ошибаться. Поэтому мне хотелось бы рассказать вам две истории о своих ошибках.

История 1. Проект «Векторизация»

Шел 1978 год. Я работал в компании Teradyne в Дирфилде, штат Иллинойс. Мы создавали автоматизированное испытательное оборудование для телефонной компании.

Я был молодым программистом и работал над прошивкой для встроенного измерительного устройства, которое болтами крепилось к стойкам в телефонных станциях. Это устройство называлось COLT — тестер линий в телефонных станциях.

В COLT использовался восьмибитный микропроцессор Intel 8085. У нас было 32 Кбайт твердотельной оперативной памяти и еще 32 Кбайт ПЗУ. Основу ПЗУ составляла микросхема Intel 2708 размером 1K × 8, поэтому мы использовали 32 такие микросхемы, кото-

рые вставлялись в разъемы на платах памяти. Одна плата вмещала 12 микросхем, так что нам потребовалось три платы.

Программа была написана на ассемблере 8085. Исходный код хранился в наборе файлов, скомпилированных как единое целое. Компилятор создал один двоичный файл размером чуть меньше 32 Кбайт.

Этот файл мы разделили на 32 фрагмента по одному килобайту каждый. Каждый из них был записан на одну из микросхем, которые вставлялись в разъемы на платах ПЗУ.

Понятно, что каждая микросхема вставлялась в конкретный разъем на конкретной плате. Поэтому приходилось их очень тщательно маркировать.

Мы продали сотни таких устройств. Они были установлены на телефонных станциях по всей стране и, по сути, по всему миру.

Как вы думаете, что происходило при внесении изменений в программу? Например, при замене одной строки?

Добавление или удаление строки меняло все адреса находящихся ниже подпрограмм. А так как они вызывались другими подпрограммами, расположенными выше, изменения затрагивали каждую микросхему. В итоге замена даже одной строки в программе означала повторную прошивку всех 32 микросхем!

Это был кошмар. Приходилось создавать сотни наборов микросхем и рассылать их всем торговым представителям. Они объезжали все телефонные станции в своем районе, вскрывали наши блоки, вытаскивали все платы памяти, вынимали оттуда старые микросхемы и вставляли новые.

Не знаю, в курсе ли вы, что процедура извлечения и вставки микросхемы в разъем не совсем надежна. Выводы микросхемы могут изгибаться и ломаться. Поэтому инженерам службы эксплуатации приходилось носить с собой множество запасных микросхем и заниматься отладкой, пока устройство наконец не начинало работать.

Однажды мой начальник сказал, что мы должны решить эту проблему, сделав микросхемы независимыми. Разумеется, он использовал

другие слова, но его намерение было именно таким. Каждую микросхему требовалось превратить в независимо компилируемую и развертываемую единицу. Это позволило бы редактировать программу без замены всех 32 микросхем.

Не буду утомлять вас подробностями реализации. Достаточно сказать, что нам потребовалось создать массивы векторов, использовать косвенные вызовы через соответствующие векторы и разбить программы на независимые фрагменты размером менее одного килобайта каждый[1].

Мы обсудили стратегию с начальником, а затем он спросил меня, сколько времени потребуется на реализацию.

Я назвал срок в две недели.

Но за две недели я не закончил. Как не закончил ни за шесть, ни за восемь, ни за десять недель. Работа заняла 12 недель — она оказалась намного сложнее, чем я предполагал.

Так что я ошибся в шесть раз. В шесть!

К счастью, начальник не рассердился. Он видел, как я каждый день работаю над проектом. И регулярно получал от меня информацию о состоянии дел. Он понимал, с какими сложностями я сталкиваюсь.

Но все равно. В шесть раз? Как я мог так ошибиться?

История 2. pCCU

А в начале 1980-х мне пришлось сотворить чудо. Мы обещали клиенту новый продукт. Он назывался CCU-CMU.

Медь — ценный металл. Он редкий и дорогой. Телефонные компании владели тоннами этого металла в виде огромной сети медных проводов, проложенных по всей стране еще в прошлом веке. Для экономии эти провода решили заменить гораздо более дешевой широкополосной

[1] То есть превратить каждую микросхему в полиморфный объект.

сетью из коаксиального кабеля и волокна, передающего цифровые сигналы. Это называлось *цифровым соединением*.

Продукт CCU-CMU представлял собой новую версию нашей измерительной технологии, соответствующую новой архитектуре цифровой коммутации телефонных компаний.

Пару лет назад мы пообещали этот продукт телефонным компаниям. И по нашим оценкам получалось, что на его создание уйдет примерно человеко-год. Но тогда никто так и не приступил к работе.

Сами знаете, как это происходит. Телефонные компании задерживали процедуру внедрения цифровых коммутаторов, поэтому мы отложили разработку. На повестке дня всегда стояло множество других, более срочных вопросов.

Но в один прекрасный день меня позвал начальник и сообщил, что мы забыли об одном мелком клиенте, который уже установил первый цифровой коммутатор. Теперь этот клиент ожидает CCU/CMU в течение следующего месяца, как и было обещано.

Мне нужно было менее чем за месяц создать программное обеспечение, требующее человеко-года работы.

Я объяснял, что это невозможно. Я никак не мог построить полностью функционирующий комплекс CCU/CMU за один месяц.

Начальник ухмыльнулся и сказал, что мы можем схитрить. Дело в том, что телефонная компания, установившая цифровой коммутатор, была очень маленькой. Это давало нам возможность обойтись минимально возможной конфигурацией. Более того, конфигурация их оборудования практически не имела тех сложностей, ради которых мы собирались писать CCU/CMU.

Короче, за две недели я создал специальное, единственное в своем роде устройство для конкретного клиента. Мы назвали его pCCU.

Уроки

Эти две истории показывают, насколько огромным может оказаться диапазон оценок. С одной стороны, я в шесть раз недооценил

сроки векторизации микросхем. С другой стороны, для комплекса CCU/CMU мы нашли решение за одну двадцатую от ожидаемого времени.

И здесь в игру вступает честность. Потому что, честно говоря, когда что-то идет не так, по закону подлости возникают дополнительные осложнения. А когда дела идут хорошо, все может сложиться как бы само собой.

И все это крайне затрудняет оценку.

Безошибочность

Теперь уже понятно, что конкретная дата *не может* выступать в роли оценки. Такая оценка слишком точна для процесса, длительность которого может отличаться в шесть или даже двадцать раз.

Оценка — это не дата, а диапазон. Она представляет собой *распределение вероятностей*.

Распределение вероятностей имеет среднее значение и ширину, иногда называемую *стандартным отклонением*, или *сигмой*. Мы должны уметь выражать оценки как через среднее значение, так и через сигму.

Начнем со среднего значения.

Чтобы получить ожидаемое среднее время выполнения сложной задачи, нужно сложить среднее время выполнения всех подзадач. Разумеется, это рекурсивный процесс. Подзадачи тоже разбиваются на более мелкие задачи. В результате формируется дерево задач, которое часто называют иерархической структурой работ (work breakdown structure, WBS).

Основная проблема состоит в нашей неспособности идентифицировать все подзадачи, подподзадачи и подподподзадачи. Как правило, некоторые из них пропускаются. Иногда пропускается половина всех более мелких задач.

Для компенсации этого фактора полученная сумма умножается на два. Иногда на три. Или даже на большее число.

Кирк: Сколько вам потребуется на ремонт?

Скотти: Восемь недель. Но так как этого времени у нас нет, я сделаю за две.

Кирк: Мистер Скотт, вы же всегда умножали свои оценки времени ремонта на четыре?

Скотти: Конечно! А как еще я могу сохранить свою репутацию чудотворца?

После этого диалога коэффициент 2, 3 или даже 4 звучит как мошенничество. Впрочем, так оно и есть. Но по такому же принципу происходит и вся процедура оценки.

Существует всего один реальный способ узнать, сколько времени займет выполнение какой-то работы. Взять и сделать эту работу. Все остальное — это допущения и в некотором роде обман.

Итак, признайте, мы собираемся мошенничать. Сформировать WBS и умножить на некий поправочный коэффициент F, значение которого колеблется от 2 до 4, в зависимости от вашей уверенности и продуктивности. В результате мы получаем оценку среднего времени завершения проекта.

На вопрос руководителя, как вы пришли к именно такой оценке, придется сказать правду. И узнав о поправочном коэффициенте, они, как правило, просят уменьшить его, потратив больше времени на составление WBS.

В принципе, это совершенно справедливое требование, но имеет смысл предупредить руководство, что стоимость разработки полной иерархической структуры эквивалентна стоимости выполнения самой задачи. Потому что к моменту, когда вы *получите* полную WBS, проект будет завершен. Ведь единственный способ по-настоящему перечислить все задачи — начать выполнять все известные вам задачи, чтобы рекурсивно обнаружить все остальные.

Так что просто сосредоточьте усилия на как можно более точном определении временных рамок, а начальству скажите, что определение более точного поправочного коэффициента будет стоить очень дорого.

Существует множество методов оценки подзадач на листьях дерева в WBS. Можно воспользоваться функциональными точками или аналогичным критерием сложности. Но я всегда сталкивался с тем, что лучше всего эти задачи оцениваются интуитивно.

Обычно я делаю оценку путем сравнения с задачами, которые я уже выполнил. Если мне кажется, что новая задача в два раза сложнее, то я умножаю время на два.

Оценив все листья дерева, суммируйте полученные оценки. Это и будет среднее значение для проекта.

И не стоит слишком беспокоиться о зависимостях. Программное обеспечение — забавная штука. Даже если A зависит от B, B часто не нужно выполнять перед A. Другими словами, ничто *не мешает* реализовать сначала выход из системы, а только потом вход в нее.

Точность

Любая оценка — допущение. Оно может оказаться и неверным. Вопрос только в том, насколько мы промахнулись. Так что часть процедуры оценки составляет определение степени ее ошибочности.

Я люблю пользоваться тремя оценками: для наилучшего случая (best case), для наихудшего случая (worst case) и для обычного случая (normal case).

Для обычного случая вы определяете, сколько времени, по вашему мнению, займет выполнение задачи, если среднее количество выполняемых действий пойдет не так, как *обычно*. Воспринимайте это как задачку на интуицию. Обычный случай — это оценка, которую дают реалисты.

Если брать более строгое определение, то это оценка, которая с вероятностью 50 процентов может оказаться слишком оптимистичной или слишком пессимистичной. Другими словами, вы укладываетесь в оценочное время выполнения работы в половине случаев.

Худший случай описывается законом Мерфи. Предполагается, что все, что может пойти не так, идет не так. Оценка такого случая с 95-процент-

ной вероятностью оказывается слишком длинной. Другими словами, вы промахиваетесь со своей оценкой только в одном случае из двадцати.

Наилучшая оценка — это когда все, что только возможно, идет правильно. Каждое утро вы получаете идеальный завтрак. Все коллеги вежливы и дружелюбны. Не происходит ни катастроф, ни совещаний, ни телефонных звонков, словом, нет никаких отвлекающих факторов.

Ваши шансы попасть в такую ситуацию составляют 5 процентов: один случай из двадцати.

В результате мы получаем три числа. Это наилучший случай с 5-процентным шансом на успех, обычный случай с 50-процентным шансом на успех и наихудший случай с 95-процентным шансом на успех. Они дают нам кривую нормального распределения. Это и есть фактическая оценка.

Обратите внимание, что это не дата. Даты завершения мы не знаем. У нас есть только грубое представление о вероятностях.

При отсутствии конкретной информации это единственный логичный способ оценки.

Называя конкретную дату, вы не даете оценку, а берете на себя *обязательство*. Разумеется, иногда приходится поступать и так. В этом случае вы фактически обещаете все сделать к указанной дате. А давать обещание можно только в случае, когда вы уверены, что сможете его выполнить. Иначе это просто нечестно.

Так что если вы не знаете *наверняка*, что сможете сдать проект к определенной дате, то ни в коем случае не оперируйте в разговоре датами. Гораздо честнее предложить диапазон дат с вероятностями.

Обобщение

Итак, мы оценили задачи, из которых состоит проект, для наилучшего (*B*), обычного (*N*) и наихудшего (*W*) случаев. Как теперь оценить время выполнения всего проекта?

Нужно взять вероятности для каждой задачи и суммировать их, используя стандартные статистические методы.

Первым делом представляем каждую задачу с точки зрения ожидаемого времени ее выполнения и стандартного отклонения.

Теперь запомните, что шесть стандартных отклонений (по три с каждой стороны от среднего) соответствуют вероятности выше 99 процентов. Поэтому наше стандартное отклонение (сигма) будет вычисляться по формуле $(W - B) / 6$.

Немного сложнее обстоят дела с ожидаемым временем завершения (*mu*). Обратите внимание, что *N*, вероятно, не совпадет со средней точкой $(W - B)$. Скорее средняя точка будет находиться после *N*, поскольку высока вероятность, что проект займет больше времени, чем мы рассчитываем. Как же определить среднее время выполнения задачи? Каким будет *ожидаемое* время завершения?

Вероятно, лучше всего использовать вот такое средневзвешенное значение: $mu = (2N + (B + W) / 2) / 3$.

Теперь мы рассчитали *mu* и сигму для набора задач. Ожидаемое время завершения проекта — всего лишь сумма всех *mu*.

Сигма проекта — квадратный корень из суммы квадратов всех сигм.

Это просто базовая статистическая математика.

Эта процедура оценки изобретена еще в конце 1950-х для управления программой баллистических ракет Polaris Fleet. С тех пор она успешно использовалась в тысячах проектов.

Она называется PERT — метод оценки и анализа программ.

Честность

Мы начинали с честности. Потом поговорили о безошибочности и точности. Пора вернуться к тому, с чего мы начали.

Вариант оценки, который я описал, честен по своей сути. Это способ проинформировать заинтересованных лиц об уровне вашей неуверенности.

И это честно, ведь вы действительно не уверены. Те, кто отвечает за управление проектом, должны осознавать риски. Ведь только в этом случае рисками можно управлять.

Но людей редко устраивает неопределенность. Клиенты и руководители почти наверняка потребуют от вас большей определенности.

Как я уже упоминал, единственный способ узнать все наверняка — начать выполнять части проекта. Полная уверенность в сроках выполнения наступит только после завершения проекта. Поэтому клиентов и руководителей нужно проинформировать о цене повышения определенности.

Впрочем, иногда для повышения определенности руководство использует другую тактику. Вас могут попросить взять на себя обязательство.

Нужно научиться распознавать такие ситуации. Руководство пытается управлять своими рисками, перекладывая их на вас. Просьба взять на себя обязательство — это просьба взять на себя риски, управление которыми — работа руководства.

В этом нет ничего плохого. Руководители имеют на это полное право. И во многих ситуациях нужно соглашаться. Но — и я подчеркиваю это — только в том случае, если вы достаточно уверены, что *сможете* выполнить обещанное.

Если начальник спрашивает, можете ли вы что-то сделать к пятнице, то хорошенько подумайте, насколько это приемлемо. Если это приемлемо и целесообразно, то скажите «да»!

Но ни при каких обстоятельствах не говорите «да», когда вы не уверены.

В этом случае вы *должны* сказать «нет», а затем описать свою неуверенность. Совершенно нормально сказать: «В пятницу я не смогу. Мне нужно время до следующей среды».

На самом деле крайне важно отказываться от обязательств, исполнение которых вы не можете гарантировать. Потому что ответ «да» сформирует длинную цепочку провалов для вас, вашего начальника и многих других. Люди будут рассчитывать на вас, а вы их подведете.

Итак, когда вас просят взять на себя обязательство и вы можете это сделать, говорите «да». Но если не можете, то говорите «нет» и описывайте свою неуверенность.

Будьте готовы обсудить варианты и обходные пути. Будьте готовы к тому, что придется искать способы сказать «да». Никогда не стремитесь говорить «нет». Но и бояться этого не нужно.

Видите ли, вас наняли за умение говорить «нет». Согласиться может кто угодно. Но только люди, обладающие навыками и знаниями, знают, когда и как сказать «нет».

Вы ценны своей способностью распознавать ситуации, в которых требуется отрицательный ответ. Говоря «нет» в такие моменты, вы экономите деньги своего работодателя.

И последнее. Часто руководители пытаются уговорить вас принять на себя обязательство. Остерегайтесь таких ситуаций.

Вас могут обвинить в том, что вы не болеете за общее дело, или сказать, что у других членов группы еще больше обязательств, чем у вас. Не поддавайтесь на эти уловки.

Будьте готовы обсудить совместный поиск решения, но не позволяйте вас запугивать, заставляя сказать «да», когда вы точно знаете, что этого делать не следует.

И будьте очень осторожны, когда вас уговаривают попробовать. Предложение начальника «Ну, может, ты хотя бы попытаешься?» может звучать вполне разумно, но на него нужно ответить так:

> *Нет! Я уже пытаюсь. Почему вы этого не видите? Я пытаюсь изо всех сил, и нет никакого способа заставить меня пытаться сильнее. У меня нет волшебной палочки, с помощью которой я могу творить чудеса.*

Возможно, вы скажете это своими словами, главное — передать суть.

И помните, что, говоря: «Да, я попытаюсь», — вы лжете. Потому что вы понятия не имеете, как добиться того, чего от вас просят. Вы не планируете менять свое поведение. Слово «да» звучит только для того, чтобы от вас отстали. А значит, ваше согласие — ложь.

УВАЖЕНИЕ

> *Обещание 9. Я буду уважать своих коллег-программистов за их этические принципы, стандарты, принятые практики и навыки. Никакие другие качества или характеристики не будут влиять на мое отношение к коллегам.*

Мы, профессионалы в области программного обеспечения, принимаем на себя тяжкое бремя нашего ремесла. Среди нас есть мужчины и женщины, представители различных рас, люди с разной сексуальной ориентацией, разными политическими взглядами и религиозными воззрениями.

Единственным условием для вступления в наше сообщество и получения признания и уважения со стороны каждого его члена являются навыки, принятые практики, стандарты и этика нашей профессии. Никакие другие человеческие качества не играют роли. Никакая дискриминация по любому другому признаку не допустима.

Я все сказал.

НИКОГДА НЕ ПЕРЕСТАВАЙ УЧИТЬСЯ

> *Обещание 10. Я никогда не перестану учиться и совершенствовать свои профессиональные навыки.*

Программист никогда не перестает учиться.

Я уверен, вы уже слышали, что должны каждый год изучать новый язык. И это действительно так. Хороший программист должен знать около десятка языков.

Речь не про разновидности одного и того же языка. Не только C, C++, Java и C#. Скорее нужно знать языки из разных семей.

Вы должны знать язык со статической типизацией, такой как Java или C#. Процедурный язык, такой как C или Pascal. Язык логического

программирования, такой как Prolog. Язык со стековой нотацией, такой как Forth. Язык с динамической типизацией, такой как Ruby. Языки для функционального программирования, такие как Clojure или Haskell.

Вы также должны знать несколько фреймворков, несколько методологий проектирования и несколько процессов разработки. Понятно, что стать экспертом во всех этих вещах невозможно, но крайне желательно познакомиться с ними поглубже.

Список вещей, которые нужно изучать, практически бесконечен. За прошедшие десятилетия наша отрасль быстро менялась, и изменения, вероятно, продолжатся. Вы должны идти в ногу с ними.

А это значит, что вы должны продолжать учиться. Читать книги и блоги. Смотреть видео. Посещать конференции и группы пользователей. Ходить на обучающие курсы. Делайте все, чтобы продолжать учиться.

Обратите внимание на сокровища прошлого. Книги, написанные в 1960-х, 1970-х и 1980-х годах — замечательные источники знаний и информации. Не нужно думать, что они устарели. В нашей отрасли устаревает не так уж много. Уважайте усилия и достижения тех, кто был до вас, изучайте их советы и выводы.

Не думайте, что ваше обучение — задача работодателя. Речь идет о *вашем* карьерном росте, и взять на себя ответственность за него должны только вы сами. Приобретать новые знания — ваша работа. Вы сами должны искать, чему еще можно научиться.

Если вам посчастливилось работать в компании, которая покупает книги и отправляет на конференции и тренинги, то в полной мере используйте эти возможности. Если нет — оплачивайте книги, конференции и курсы самостоятельно.

И запланируйте время, которое вы будете на это тратить. Еженедельно. Своему работодателю вы должны отдать от 35 до 40 часов в неделю. А своей карьере — еще от 10 до 20. Так поступают профессионалы. Профессионалы тратят время на самосовершенствование и вкладываются в свою карьеру. Это значит, что вам придется работать от 50 до 60 часов в неделю. В основном на работе, но и дома тоже.

Роберт Мартин

Идеальная работа. Программирование без прикрас

Перевела с английского *И. Рузмайкина*

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.05.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 30,960. Тираж 2000. Заказ 0000.

Роберт Мартин

ЧИСТЫЙ КОД: СОЗДАНИЕ, АНАЛИЗ И РЕФАКТОРИНГ. БИБЛИОТЕКА ПРОГРАММИСТА

Плохой код может работать, но он будет мешать развитию проекта и компании-разработчика, требуя дополнительные ресурсы на поддержку и «укрощение».

Каким же должен быть код? Эта книга полна реальных примеров, позволяющих взглянуть на код с различных направлений: сверху вниз, снизу вверх и даже изнутри. Вы узнаете много нового о коде. Более того, научитесь отличать хороший код от плохого, узнаете, как писать хороший код и как преобразовать плохой код в хороший.

Книга состоит из трех частей. Сначала вы познакомитесь с принципами, паттернами и приемами написания чистого кода. Затем приступите к практическим сценариям с нарастающей сложностью — упражнениям по чистке кода или преобразованию проблемного кода в менее проблемный. И только после этого перейдете к самому важному — концентрированному выражению сути этой книги — набору эвристических правил и «запахов кода». Именно эта база знаний описывает путь мышления в процессе чтения, написания и чистки кода.

Роберт Мартин

ЧИСТАЯ АРХИТЕКТУРА. ИСКУССТВО РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

«Идеальный программист» и «Чистый код» — легендарные бестселлеры Роберта Мартина — рассказывают, как достичь высот профессионализма. «Чистая архитектура» продолжает эту тему, но не предлагает несколько вариантов в стиле «решай сам», а объясняет, что именно следует делать, по какой причине и почему именно такое решение станет принципиально важным для вашего успеха.

Роберт Мартин дает прямые и лаконичные ответы на ключевые вопросы архитектуры и дизайна. «Чистую архитектуру» обязаны прочитать разработчики всех уровней, системные аналитики, архитекторы и каждый программист, который желает подняться по карьерной лестнице или хотя бы повлиять на людей, которые занимаются данной работой.

Роберт Мартин

ИДЕАЛЬНЫЙ ПРОГРАММИСТ. КАК СТАТЬ ПРОФЕССИОНАЛОМ РАЗРАБОТКИ ПО

Всех программистов, которые добиваются успеха в мире разработки ПО, отличает один общий признак: они больше всего заботятся о качестве создаваемого программного обеспечения. Это основа для них. Потому что они являются профессионалами своего дела.

В этой книге легендарный эксперт Роберт Мартин (более известный в сообществе как «Дядюшка Боб»), автор бестселлера «Чистый код», рассказывает о том, что значит «быть профессиональным программистом», описывая методы, инструменты и практики разработки «идеального ПО». Книга насыщена практическими советами в отношении всех аспектов программирования: от оценки проекта и написания кода до рефакторинга и тестирования. Эта книга — больше, чем описание методов, она о профессиональном подходе к процессу разработки.