ОГЛАВЛЕНИЕ

ГЛАВА 1. О тестировании программного обеспечения и модульном тестировании

Чтобы получить максимальную выгоду от модульного тестирования, необходимо понимать его цель и как его применение поможет в улучшении программного обеспечения. В этой главе вы узнаете о существовании «вселенной» тестирования программного обеспечения, частью которой является модульное тестирование, а также с его преимуществами и недостатками.

Зачем тестировать программное обеспечение?

Обычно конечной целью многих проектов разработки программного обеспечения является получение прибыли. Типичными путями достижения этой цели являются прямые продажи программного обеспечения, реализация через Интернет-магазин или некоторая схема лицензирования его использования. Программы, создаваемые разработчиками для внутреннего использования, часто приносят прибыль косвенным путем, увеличивая производительность труда и уменьшая время, затрачиваемое на разработку. Если экономия, в терминах эффективности труда, больше стоимости разработки программы, проект можно считать выгодным. Разработчики открытых проектов часто продают услуги по поддержке пакетов или сами используют свои программы: в этих случаях предыдущий аргумент остается справедливым.

Итак, с экономической точки зрения все просто: если целью программного проекта является получение прибыли (будь то программный продукт на продажу или для внутреннего использования), его

ценность для пользователя должна быть выше стоимости. Я понимаю, что не сказал ничего нового, но это утверждение имеет следствия, важные для тестирования программного обеспечения.

Если тестирование (также известное, как ***контроль качества***) рассматривать как нечто, обеспечивающее поддержку программного проекта, оно должно служить цели получения прибыли. Это важное обстоятельство, потому что оно автоматически накладывает ограничения на то, как должен тестироваться программный продукт: если тестирование является настолько дорогостоящим, что приносит убытки, значит выбранный способ тестирования не соответствует цели. Однако тестирование может доказать работоспособность продукта, то есть, доказать, что продукт обладает ценными качествами, ожидаемыми клиентами. Если не продемонстрировать эти качества, клиент может отказаться от покупки продукта.

Обратите внимание, что ***целью тестирования*** является демонстрация работоспособности продукта, а не вскрытие ошибок. Это ***контроль качества***, а не ***увеличение качества***. Обнаружение ошибок – это плохо. Почему? Потому что устранение ошибок стоит денег и эти деньги тратятся впустую, потому что вам заплатили за создание программы без ошибок. В идеальном мире разработчики просто писали бы безошибочные программы, выполняли небольшое тестирование, чтобы убедиться в отсутствии ошибок, выгружали бы свои программы на iTunes Connect и ждали, пока деньги посыплются на них. Но постойте: такая организация труда может повлечь за собой убытки. Насколько дольше разработчику придется писать программу, чтобы до тестирования быть в полной уверенности в отсутствии ошибок? Сколько это будет стоить?

Таким образом, уровень тестирования программ представляет собой компромисс между необходимой степенью контроля и степенью убежденности в работоспособности программы, без значительного удорожания проекта. Как определить этот компромисс? Он основывается на снижении до приемлемого уровня рисков, связанных с продажей продукта. Первыми должны тестироваться наиболее «рискованные» компоненты, которые являются наиболее важными для функционирования программы или где по вашему мнению может скрываться большая часть ошибок. Затем компоненты, следующие по степени риска, и так далее, пока степень риска не снизится до уровня, когда не имеет смысла тратить время и деньги на дальнейшее его снижение. Конечной целью должна быть демонстрация клиенту возможностей программы, за которые он захочет заплатить.

Кто должен тестировать программы?

На начальном этапе развития вычислительной техники разработка программного обеспечения велась в соответствии с «каскадной моделью» (рис. 1.1)[1]. Согласно этой модели каждая часть процесса разработки является отдельной «фазой», результат выполнения которой становится начальной точкой следующей фазы. То есть, главные конструкторы или бизнес-аналитики должны были определить требования к продукту и затем эти требования вручались конструкторам и архитекторам для выработки технических условий. Разработчики получали технические условия и на их основе создавали программный код. Далее программный код передавался тестерам для контроля качества. В заключение протестированный программный продукт передавался заказчикам (обычно приемщикам программного обеспечения, известным как ***бета-тестеры***).

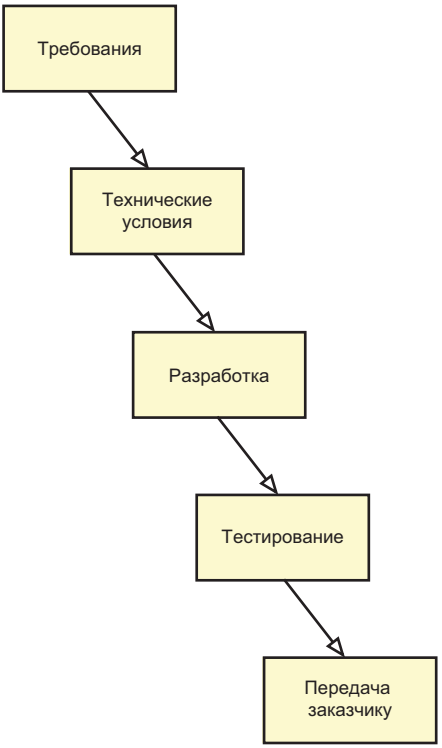

Рис. 1.1. Фазы разработки программного обеспечения в каскадной модели

Такой подход к управлению проектом предполагает отделение программистов от тестеров, что имеет свои достоинства и недостатки с точки зрения тестирования. Достоинство заключается в том, что при разделении на фазы разработки и тестирования, доступ к программному коду получает большее количество людей, что увеличивает вероятность обнаружения ошибок. Глаз разработчика может элементарно

1 В действительности многие программные проекты, включая приложения для iOS, по-прежнему продолжают развиваться по этой модели. Этот факт не может служить опровержением, что каскадная модель является исторической ошибкой.

«замылиться» и может потребоваться свежий взгляд, чтобы указать на недостатки. Аналогично, если какая-то часть требований или технических условий неоднозначна, есть вероятность, что разработчик и тестер будут интерпретировать ее по-разному, что повышает шанс обнаружить ее.

Главным недостатком является стоимость. Табл. 1.1, взятая из книги «Code Complete, 2nd Edition» Стива Макконнела (Steve McConnell) (Microsoft Press, 2004)[2], показывает результаты исследований, согласно которым стоимость исправления ошибки является функцией от времени, которое она «бездействовала» в продукте. В таблице видно, что исправление ошибки в конце проекта стоит значительно дороже, что вполне объяснимо: тестер обнаруживает ошибку и сообщает о ней, затем разработчик должен правильно интерпретировать сообщение и попытаться отыскать ошибку в исходном коде. Если ошибка была обнаружена после того, как разработчик закончил работу над проектом, ему потребуется некоторое время, чтобы повторно ознакомиться с техническими условиями и просмотреть исходный код. Затем исправленная версия должна быть передана на повторное тестирование, чтобы убедиться, что проблема была ликвидирована.

Таблица 1.1. Средняя стоимость исправления дефектов в зависимости от времени их внесения и обнаружения

Стоимость исправления	**Время обнаружения**				
Время внесения	**Выработка требований**	**Проектирование архитектуры**	**Разработка**	**Тестирование**	**После выпуска ПО**
Выработка требований	1	3	5 – 10	10	10 – 100
Проектирование архитектуры	–	1	10	15	25 – 100
Разработка	–	–	1	10	10 – 25

Откуда берется эта дополнительная ***стоимость***? Большая ее часть порождается в ходе взаимодействий различных групп людей, участвующих в разработке: разработчики и тестеры могут пользоваться различными определениями для описания одних и тех же понятий, или вообще по-разному представлять себе различные особенности

2 «Совершенный код. Практическое руководство по разработке программного обеспечения», Питер, 2005, ISBN 5-7502-0064-7. – *Прим. перев.*

программы. Всякий раз, когда происходит взаимодействие, приходится тратить некоторое время на устранение неоднозначностей или проблем, их вызвавших.

Таблица также наглядно показывает, что цена исправления шибки, обнаруженной на последних этапах проекта, зависит от того, насколько рано эта ошибка была допущена: Ошибку, возникшую на этапе составления требований, в конце можно исправить, только полностью переписав фрагмент программы, что является весьма дорогостоящим удовольствием. Это заставляет сторонников каскадной модели использовать очень консервативные подходы на ранних стадиях проекта и не передавать требования или технические условия на следующую стадию, пока не будет полной уверенности, что все точки над «i» и палочки в «t» были расставлены. Это состояние известно как «паралич анализа» (analysis paralysis) и существенно увеличивает стоимость проекта.

Разделение разработчиков и тестеров также оказывает отрицательное влияние на этап тестирования, даже при том, что здесь никаких ограничений не вводится. Поскольку тестеры не имеют такого глубокого понимания внутреннего устройства приложения, как разработчики, они обычно воспринимают тестируемый ими программный продукт, как своеобразный «черный ящик», воздействуя на него только извне. Маловероятно, что сторонние тестеры изберут подход к тестированию типа «стеклянный ящик» , позволяющий исследовать внутреннюю работу программного кода и изменить его, чтобы обеспечить проверку поведения программы.

При подходе к приложению как к «черному ящику» обычно применяется ***системное*** или ***интеграционное тестирование***. Этот формальный термин подразумеваетсборку программного продукта и его тестирование, как единого целого. Обычно эти виды тестирования выполняются в соответствии с предопределенным планом, за создание которого тестеры получают зарплату: они берут технические условия и на их основе создают серию испытательных тестов, каждый из которых описывает шаги, которые необходимо выполнить при подготовке и в процессе тестирования, а также ожидаемые результаты. Такие тесты часто выполняются вручную, особенно когда результат требует интерпретации человеком из-за зависимости от внешних факторов, таких как текущая дата или особенности функционирования сетевой службы. Даже когда такое тестирование можно автоматизировать, оно часто занимает продолжительное время: перед каждым тестом программный продукт и его окружение необходимо

привести в определенное состояние, при этом для отдельных шагов может потребоваться выполнить длительные операции с базой данных, файловой системой или сетевой службой.

Бета-тестирование (эксплуатационные испытания, или опытная эксплуатация), которое в некоторых командах называется ***тестированием в условиях эксплуатации у потребителя***, в действительности является особой разновидностью системного тестирования. Особенность этого вида тестирования заключается в том, что человек, проводящий тестирование, не является профессиональным тестером программного обеспечения. Если между системами или окружениями тестера и потребителя имеются какие-либо различия или испытательные тесты, которые собирается применить пользователь, не рассматривались командой проекта, это вскроется на этапе бета-тестирования, о чем может быть сообщено разработчикам. Для небольших коллективов, особенно тех, кто не может позволить себе воспользоваться услугами профессиональных тестеров, этап бета-тестирования представляет первый шанс опробовать программу в различных ситуациях и окружениях.

Поскольку бета-тестирование выполняется непосредственно перед передачей продукта потребителю, обратная связь с разработчиками затруднена, так как команда проекта уже чувствует, что конец не за горами и предвкушает будущее застолье на презентации. Однако, бессмысленно проводить тестирование, если вы не собираетесь исправлять обнаруженные ошибки.

Разработчики могут также проводить собственное тестирование. Если вам когда-либо доводилось щелкать на кнопке **Build & Debug** (Сборка и отладка) в Xcode, значит вы уже проводили тестирование по типу «стеклянного ящика»: вы исследовали внутреннюю работу программного кода, пытаясь определить, насколько правильно он действует (или, точнее, почему он действует неправильно). Предупреждения компилятора, статический анализатор и прочие инструменты среды разработки помогают разработчику провести тестирование.

Преимущества и недостатки тестирования разработчиком практически совершенно противоположны таковым для независимого тестирования: когда разработчик обнаруживает проблему, он, как правило, легко может ее исправить, потому что он знаком с программным кодом и представляет, где может скрываться ошибка. В действительности разработчик в состоянии проводить тестирование прямо в процессе работы, поэтому значительная часть ошибок обычно обнаружи-

вается сразу после их внесения. Однако, если ошибка заключается в неправильном понимании разработчиком технических условий или предметной области, она не будет обнаружена без посторонней помощи.

Правильное понимание требований

Самая вопиющая ошибка, которую мне приходилось допускать (до настоящего момента и, я надеюсь, до конца профессиональной карьеры), относится к категории «разработчик не понял требований». Я занимался разработкой инструмента системного администрирования для Mac, и поскольку он работал за пределами какой-либо учетной записи, я не мог видеть настройки пользователя, чтобы определить, какой язык использовать для журналирования. Программа читала настройки из файла, содержимое которого выглядело так:

```
LANGUAGE=English
```

Достаточно просто. Проблема состояла в том, что некоторые пользователи, говорящие на других языках, сообщали, что инструмент выводит записи в файл журнала на английском языке, то есть, он неправильно понимал настройки языка. Я обнаружил, что программный код, выполняющий чтение файла с настройками, тесно связан с другой частью инструмента, поэтому потребовалось разорвать эту связь и добавить модульные тесты, чтобы выяснить, как будет вести себя программный код. В конечном итоге я выявил проблему, вызывавшую сбой в функции определения языка, и устранил ее. Если все модульные тесты выполняются, это свидетельствует о правильной работе программного кода, правильно? Нет, неправильно: как оказалось, я не предполагал, что содержимое файла может выглядеть иначе, например:

```
LANGUAGE=en
```

Не только я не знал этого, но и мои тестеры тоже. В действительности, чтобы увидеть проблему, пришлось провести испытания в системе потребителя, даже при том, что программный код с успехом проходил все тесты.

Когда следует тестировать программы?

Ответ на этот вопрос отчасти уже был дан в предыдущем разделе – чем раньше начнется тестирование продукта, тем дешевле обойдется устранение ошибок. Чем раньше удастся убедиться в надежной работе отдельных фрагментов приложения, тем меньше проблем будет возникать при их объединении или добавлении новых фрагментов на последующих стадиях, в сравнении с ситуацией, когда тестирование производится в самом конце. Однако, в предыдущем разделе также было показано, что программные продукты традиционно при-

нято тестировать после окончания разработки: явная фаза контроля качества следует за фазой разработки, перед передачей программы бета-тестерам и перед выпуском окончательной версии.

Современными подходами к управлению программными проектами признается несовершенство этой модели и во главу угла ставится цель обеспечить непрерывное тестирование всех частей приложения. В этом заключается основное различие «гибкого» и традиционного подхода к управлению проектами. При гибком управлении развитие проекта выполняется небольшими шагами, которые называются ***итерациями***. В каждой итерации производится пересмотр требований – устаревшие требования отбрасываются и вносятся новые изменения и дополнения. В каждой итерации проектируются, реализуются и тестируются наиболее важные требования. В конце итерации оценивается продвижение проекта, и принимаются решения о добавлении в продукт новых особенностей или внесении изменений в требования для рассмотрения в следующей итерации. Очень важно, чтобы в принятии решений участвовал заказчик или его представитель, потому что манифест гибкой разработки (http://agilemanifesto.org/iso/ru) утверждает: «люди и взаимодействие важнее процессов и инструментов». Необязательно корпеть над длинным документом с техническими условиями, когда можно просто спросить у заказчика как должна работать программа и получить подтверждение, что программа работает именно так, как требуется.

При использовании методологии гибкой разработки все аспекты проекта тестируются постоянно. В каждой итерации у заказчика постоянно выясняются наиболее важные требования, а разработчики, аналитики и тестеры сотрудничают друг с другом в направлении удовлетворения этих требований. Одна из методологий гибкой разработки, которая называется ***экстремальным программированием*** (ЭП), требует даже, чтобы разработчики постоянно проводили модульное тестирование и работали парами, когда один из них «управляет» клавиатурой, а другой предлагает изменения, улучшения и следит за потенциальными ловушками.

Таким образом, ответ на поставленный вопрос звучит так: программа должна тестироваться постоянно. Нельзя полностью устранить вероятность, что пользователи будут использовать ваш продукт неожиданными способами и вскроют ошибки, незаметные изнутри, по независящим от вас причинам. Однако всегда можно реализовать автоматическое тестирование наиболее важных функций и оставить команде контроля качества и бета-тестерам право экспериментиро-

вать с испытательными тестами, чтобы попытаться нарушить работу приложения самыми неожиданными способами. И в каждой итерации можно спрашивать, добавит ли ценности продукту то, что вы собираетесь сделать, и увеличить степень удовлетворенности заказчика продуктом от его соответствия рекламным заявлениям.

Примеры из практики тестирования

Я уже упоминал прием системного тестирования, когда профессиональные тестеры берут приложение целиком и методично опробуют все варианты его использования, пытаясь получить неожиданное поведение. В случае с приложениями для iOS, этот вид тестирования можно автоматизировать до определенной степени с помощью инструмента **UI Automation** (Автоматизация пользовательского интерфейса), входящего в комплект инструментов профилирования от Apple.

Системное тестирование не всегда выполняется по какому-то универсальному шаблону – иногда тестеры могут преследовать вполне определенные цели. При ***тестировании на возможность вторжения*** тестеры ищут бреши в системе безопасности, подавая на вход приложения неправильно сформированные данные, выполняют операции не в той последовательности или как-то иначе пытаются нарушить ожидания приложения. При ***тестировании на удобство использования*** тестеры наблюдают за действиями пользователя, отмечая, что они делают неправильно, на какие операции тратят больше всего времени или что их смущает. В частности, один из приемов тестирования на удобство использования называется A/B-тестирование: различным пользователям даются различные версии приложения и затем выполняется сравнение по статистическим показателям. В Google, одной из известных компаний, использующих этот подход при тестировании своих приложений, проверяется даже влияние теней различных оттенков на удобство пользовательского интерфейса. Обратите внимание, что для тестирования на удобство использования не требуется иметь полное приложение: чтобы увидеть реакцию пользователя на интерфейс приложения, достаточно иметь макет в **Interface Builder**, **Keynote** или даже просто на бумаге. Неполноценная версия интерфейса не позволит получить полное представление об особенностях взаимодействия с действующим устройством iPhone, но она является одним из наиболее дешевых способов получить первые результаты.

Разработчики, особенно в больших коллективах, передают свой код коллегам для беглой оценки, прежде чем интегрировать его в про-

дукт. Это своего рода тестирование по принципу «стеклянного ящика» – другие разработчики могут видеть, как действует программный код, исследовать его реакцию на определенные условия и оценить, учтены ли все возможные ситуации. При оценке кода коллегами не всегда обнаруживаются логические ошибки. Чаще результатом такой оценки являются рекомендации, касающиеся стиля оформления исходных текстов или других аспектов, которые могут быть исправлены без изменения поведения программного кода. Если рецензенту подсказать, что искать (например, дать ему список из пяти-шести наиболее типичных ошибок – в такие списки для Mac и iOS часто включают проблему, связанную с подсчетом ссылок на объекты), вероятность обнаружения таких ошибок возрастает, однако при этом он может упустить из виду проблемы, не связанные с рекомендованной целью поиска.

Как в общую картину вписывается модульное тестирование?

Модульное тестирование – еще один инструмент, который разработчики могут использовать для тестирования разрабатываемых программ. Подробнее о проектировании и разработке модульных тестов рассказывается в главе 3, «Как писать модульные тесты», а пока достаточно будет отметить, что модульные тесты представляют собой небольшие фрагменты программного кода, который тестирует другой программный код. Они определяют предварительные условия, выполняют тестируемый программный код и ***сравнивают*** полученные результаты с ожидаемыми. Если проверяемые условия удовлетворяются, тесты считаются пройденным. Любое отклонение от ожидаемого результата интерпретируется как ошибка, включая исключения, которые прерывают процесс тестирования до его завершения[3].

Таким образом, модульные тесты представляют собой миниатюрные версии испытательных тестов, применяемых при интеграционном тестировании: они определяют шаги, выполняемые при тестировании, и ожидаемые результаты, но реализованы в программном коде. Это позволяет переложить тестирование на компьютер и не

3 Используемый комплект тестов может отдельно выводить сообщения об «ошибках» и несовпадениях с ожидаемым результатом, но в этом нет ничего плохого. Главное, что проблемы, возникающие при выполнении теста не останутся незамеченными.

заставлять разработчика выполнять весь процесс вручную. Однако, хороший тест – это еще и хорошая документация: он описывает, какие результаты ожидается получить в результате тестирования. Разработчик, пишущий класс для приложения, может также написать тесты, позволяющие убедиться, что класс действует, как требуется. Фактически, как будет показано в следующей главе, разработчик может писать тесты еще до того, как будет написан тестируемый класс.

Модульные тесты получили такое название, потому что каждый из них тестирует отдельную программную единицу, или «модуль», каковой в объектно-ориентированном программировании является класс. Этот термин исходит из термина «единица трансляции» (translation unit), используемого при описании компиляторов, и обозначает единственный файл, передаваемый компилятору. Это означает, что модульные тесты по своей природе являются реализацией подхода к тестированию по типу «стеклянного ящика», потому что они извлекают класс из контекста приложения и тестируют его поведение независимо. Интерпретировать ли класс, как «черный ящик» и взаимодействовать с ним исключительно через общедоступный API, – это личный выбор, но при этом тест по-прежнему будет взаимодействовать с небольшой частью приложения.

Высокая степень избирательности при модульном тестировании делает возможным очень быстро решать проблемы, вскрытые модульными тестами. Разработчик, занимающийся реализацией класса, часто параллельно работает над тестами для этого класса, то есть, код класса стоит у него перед глазами, когда он пишет тесты. У меня даже были ситуации, когда я обнаруживал и исправлял ошибки еще только при создании модульных тестов, потому что я продолжал думать о тестируемом классе. Сравните это с ситуацией, когда другой человек тестирует сценарий использования программного продукта, работу над которым разработчик закончил много месяцев назад. Даже при том, что используя модульное тестирование разработчику приходится писать программный код, который не попадет в приложение, этот труд окупается выгодой от обнаружения и исправления проблем до того, как они окажутся в руках тестеров.

Этап исправления ошибок – это кошмар для любого руководителя проекта: устранение ошибок требует времени, продукт невозможно передать заказчику, пока не будут устранены ошибки, при этом невозможно определить точные сроки, потому что количество ошибок заранее неизвестно, как и неизвестно, сколько времени потребуется разработчику на их устранение. Вернитесь к табл. 1.1 и вы увидите,

что ошибки, исправляемые в конце проекта, являются самыми дорогостоящими и в стоимости их исправления наблюдается самый большой разброс. Выделяя в графике разработки время на создание модульных тестов, некоторые из этих ошибок можно исправить в процессе работы и уменьшить неопределенность касательно даты готовности продукта.

Модульные тесты практически всегда пишутся разработчиками, потому что использование платформы тестирования подразумевает необходимость писать программный код, работающий с прикладными программными интерфейсами и выражающий низкоуровневую логику, то есть, все то, что знает и умеет разработчик. Однако, совершенно необязательно, чтобы модульные тесты писал разработчик, создавший класс, и существуют определенные выгоды от разделения этих задач.

Старший разработчик может определить API класса для реализации младшим разработчиком, выражая желаемое поведение в виде набора тестов. Имея необходимые тесты, младший разработчик может реализовать класс, успешно проходящий все тесты в наборе.

При желании можно использовать обратный порядок взаимодействий. Разработчики, получившие некоторый класс для использования или оценки, но пока не знающие как он должен работать, могут написать тесты, выражающие их предположения о классе, чтобы определить их верность. В процессе разработки тестов они получают более полное представление о возможностях и особенностях поведения класса. Однако, писать тесты для существующего кода обычно сложнее, чем создавать тесты и код параллельно. Классы, реализация которых основана на каких-либо предположениях об окружении, могут не работать в тестовом окружении без приложения существенных усилий, из-за необходимости заменить или устранить зависимости от окружающих объектов. Особенности применения методики модульного тестирования для проверки существующего программного кода рассматриваются в главе 11, «Проектирование при разработке через тестирование».

Разработчики, работающие в паре, могут даже меняться ролями: сначала один пишет тесты для проверки реализации, которую пишет другой, затем они меняются ролями и уже второй пишет тесты для первого. Однако совершенно неважно, какой порядок сотрудничества выберут программисты. В любом случае модульный тест или набор модульных тестов может выступать в качестве документации, раскрывающий замысел одного разработчика для другого.

Одним из основных преимуществ модульного тестирования является автоматизация процесса тестирования. Создание хорошего теста может занять столько же времени, сколько занимает разработка плана тестирования вручную, но затем компьютер может выполнять сотни тестов в секунду. Разработчики могут хранить все свои тесты в системе управления версиями, вместе с программным кодом приложения, и выполнять их в любой момент. Это существенно удешевляет тестирование ***регрессионных*** ошибок: ошибок, которые были исправлены и вновь внесены при последующих изменениях программного кода. После каждого изменения приложения необходимо уделить несколько секунд на выполнение всех тестов, чтобы убедиться, что в результате изменений не было внесено регрессионных ошибок. Можно даже организовать автоматический запуск тестирования сразу после сохранения изменений в репозиторий, с помощью ***системы непрерывной интеграции***, как описывается в главе 4, «Инструменты для тестирования».

Повторное тестирование не только предупредит о появлении регрессионных ошибок. Оно также обеспечит надежную опору, когда возникнет желание изменить исходный программный код без изменения его поведения, то есть, провести ***рефакторинг*** исходных текстов приложения. Цель рефакторинга – привести в порядок исходные тексты приложения или реорганизовать их некоторым образом, что может пригодиться в будущем, но без внедрения новой функциональности или ошибок! Если программный код, подвергаемый процедуре рефакторинга, в достаточной мере охвачен модульными тестами, можно быть уверенными, что любые изменения в поведении будут тут же обнаружены. Это позволяет немедленно исправлять проблемы, не дожидаясь выхода следующей версии.

Однако модульное тестирование не является панацеей от всех бед. Как обсуждалось выше, не существует способа убедиться, что разработчик правильно понял требования, предъявляемые к приложению. Если тесты и тестируемый ими программный код писал один и тот же человек, они оба будут отражать одни и те же взгляды и интерпретацию задачи, решаемой программным кодом. Следует также понимать, что не существует достаточно хороших способов оценить успех применения стратегии модульного тестирования. Популярностью пользуются такие показатели, как степень охвата тестами и количество проходимых тестов, однако каждый из них может существенно изменяться, не отражая фактическое качество тестируемого продукта.

Возвращаясь к идее, согласно которой тестирование должно уменьшить риск выявления ошибок после передачи программы клиенту, было бы очень полезно иметь некий инструмент, с помощью которого можно было бы оценить, насколько благодаря тестированию уменьшился этот риск. В действительности невозможно точно оценить риск в каждой конкретной программе – можно только приблизительно оценить уровень риска.

Подсчет количества проходимых тестов – очень наивный способ измерить эффективность набора тестов. Представьте, что размер годовой премии зависит от количества проходимых тестов – в этом случае можно написать единственный тест и скопировать его множество раз. Он даже не должен проверять программный код приложения – тест, проверяющий результат выражения `"1==1"` с успехом увеличит количество тестов, проходимых вашей программой. Какое число тестов можно считать удовлетворительным для любого приложения? Можете ли вы назвать число, к которому должны стремиться все разработчики приложений для iOS? Наверняка нет – я тоже не смогу. Даже два разработчика, работавших над созданием одного и того же приложения могут обнаружить разные проблемы в разных его частях и таким образом столкнуться с разными уровнями риска.

Оценка степени охвата программного кода частично решает проблему измерения эффективности тестов, отражая объем программного кода, который проверяется при выполнении тестирования. Это означает, что разработчики уже не смогут увеличивать свои премии за счет создания бессмысленных тестов, но они все еще могут «срывать плоды, что висят пониже» и добавлять тесты для простого программного кода. Представьте, что степень охвата увеличена за счет поиска всех определений свойства `@synthesize` в приложении и тестировании его методов доступа. Несомненно, как будет показано далее, эти тесты имеют определенную ценность, но все-таки они не самый лучший способ потратить свое время.

В действительности, инструменты оценки степени охвата программного кода тестами не учитывают сложность кода. Здесь под словом «сложный» подразумевается понятие ***цикломатической сложности***, используемое в информатике. В двух словах, цикломатическая сложность функции или метода зависит от количества циклов и ветвлений, иными словами, – от количества различных путей, которыми может идти выполнение программы.

Возьмем для сравнения два метода: метод `-methodOne` содержит двадцать строк кода и ни одной инструкции `if`, `switch`, `?:`

или цикла (то есть, имеет минимальную сложность). Другой метод, `-methodTwo:(BOOL)flag` содержит инструкцию `if` с десятью строками кода в каждой ее ветке. Чтобы полностью охватить метод `-methodOne`, достаточно написать один тест, но чтобы полностью охватить метод `-methodTwo:` придется написать два теста. Каждый тест исследует код в одной из двух ветвей инструкции `if`. Инструмент оценки степени охвата просто сообщит количество выполненных строк – по двадцать в обоих случаях – но охватить тестами более сложный код труднее, к тому же в сложном коде проще допустить ошибку.

Аналогично эти инструменты плохо справляются с оценкой особых случаев. Если метод принимает объект в виде параметра, инструменту оценки будет безразлично, сравнивает ли тест этот объект с другим инициализированным объектом или с пустым указателем. Фактически, полезными могут быть обе проверки, независимо от заинтересованности в оценке степени охвата. Однако в любом случае будет охвачено одно и то же количество строк, и оценка охвата тестами не изменится.

В конечном счете вам (и, возможно, вашим клиентам) придется самим определять степень риска того или иного участка программы и насколько эти участки должны быть охвачены тестами. Даже если инструменты будут давать достоверную оценку охвата кода тестами, это все равно не снимает с вас ответственность. Ваша цель состоит в том, чтобы убедиться в полезности тестов и наоборот, не создавать тесты, не несущие никакой пользы. На вопрос: «Какие части приложения следует тестировать?», – программист и эксперт по модульному тестированию Кент Бек (Kent Beck) отвечает: «Только те, которые должны работать.».

Что все это означает для разработчиков iOS?

Главное преимущество модульного тестирования для разработчиков приложений для iOS заключается в возможности получения большей выгоды за меньшую цену. Поскольку многие сотни тысяч приложений в App Store производятся мелкими производителями программного обеспечения, все, что способствует повышению качества приложений без значительных финансовых затрат, достойно применения. Инструменты, необходимые для включения модульного тестирования в проекты приложений для iOS, распространяются бесплатно. Факти-

чески, как описывается в главе 4, основная функциональность уже включена в пакет iOS SDK. Тесты можно писать и запускать самостоятельно, то есть, чтобы получать выгоды от модульного тестирования не обязательно привлекать специалиста по контролю качества.

Для выполнения тестов требуется совсем немного времени, поэтому единственными существенными затратами на модульное тестирование является время, необходимое на проектирование и создание испытательных тестов. Взамен этих затрат вы получите более глубокое понимание, как должен действовать ваш код ***уже во время его создания***. Это понимание поможет вам избежать многих ошибок в процессе разработки и иметь больше уверенности в конце проекта, благодаря меньшему количеству ошибок, обнаруженных бета-тестерами.

Не забывайте, что как разработчик приложений для iOS, вы не контролируете сроки передачи новых версий приложения своим клиентам: эту функцию взяла на себя компания Apple. Если серьезная ошибка вынудит выпустить новую версию приложения, вам придется ждать, пока компания Apple утвердит ее (если она это сделает), прежде чем оно попадет в App Store и станет доступным для ваших пользователей. Одно это должно служить поводом для разработки новой процедуры тестирования. Выпуск некачественного программного обеспечения является достаточно серьезной проблемой – невозможность быстро исправить ошибку может иметь весьма неприятные последствия.

Применение методики разработки через тестирование добавит ощущение комфорта – одновременная разработка программного кода и тестов для него поможет повысить производительность труда, потому что размышления об организации программного кода и условиях, в которых он выполняется, станут неотъемлемой частью процесса разработки. Вскоре вы обнаружите, что разработка кода вместе с тестами занимает то же время, что и разработка одного только кода, но при этом вы будете более уверены в нем. В следующей главе будут представлены концепции, на которых основывается методика разработки через тестирование: эти концепции будут использоваться на протяжении всей книги.