Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

И. А. Мурашко, В. В. Комраков

ВЕРИФИКАЦИЯ И АТТЕСТАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
для студентов специальностей
1-40 05 01 «Информационные системы
и технологии (по направлениям)» и 1-40 80 04
«Информатика и технологии программирования»
дневной и заочной форм обучения

Электронный аналог печатного издания

Гомель 2021

УДК 004.415.5(075.8)
ББК 32.972я73
М91

Рекомендовано к изданию научно-методическим советом факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого
(*протокол № 1 от 12.03.2019 г.*)

Учебно-методическое пособие выполнено в ходе реализации проекта MaCICT (Modernisation of Master Curriculum in Information Computer Technologies)в рамках Erasmus+ Программы Европейского Союза. В этом проекте передаются лучшие практики вузов-партнеров из Евросоюза для модернизации учебного плана и дисциплин с целью внедрения в учебный процесс изучения гибких навыков, позволяющих повысить конкурентоспособность на рынке труда выпускников ИТ-специальностей.

Рецензент: зав. каф. «Информатика» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *Т. А. Трохова*

Мурашко, И. А.

М91 Верификация и аттестация программного обеспечения : учеб.-метод. пособие для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» днев. и заоч. форм обучения / И. А. Мурашко, В. В. Комраков. – Гомель : ГГТУ им. П. О. Сухого, 2021. – 191 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

ISBN 978-985-535-464-3.

В книге особое внимание уделяется верификации программного обеспечения, жизненному циклу проекта и использованию тестирования на различных его этапах, разработке тестовой документации, нефункциональным видам тестирования, аттестации программного обеспечения.

Может быть полезна при решении прикладных задач, а также при выполнении курсовых, дипломных работ и подготовке магистерских диссертаций.

УДК 004.415.5(075.8)
ББК 32.972я73

ISBN 978-985-535-464-3

Оглавление

Введение

Аттестация программных средств (ПС) – это авторитетное подтверждение качества ПС. Обычно для аттестации ПС создается представительная комиссия из экспертов, представителей заказчика и представителей разработчика. Эта комиссия проводит испытания ПС с целью получения необходимой информации для оценки его качества. Под испытанием ПС мы будем понимать процесс проведения комплекса мероприятий, исследующих пригодность ПС для успешной его эксплуатации в соответствии с требованиями заказчика. Этот комплекс включает проверку полноты и точности программной документации, изучение и обсуждение других ее свойств, а также необходимое тестирование программ, входящих в состав ПС, и, в частности, соответствия этих программ имеющейся документации.

На основе информации, полученной во время испытаний ПС, прежде всего должно быть установлено, что ПС выполняет декларированные функции, а также должно быть установлено, в какой степени ПС обладает декларированными примитивами и критериями качества. Таким образом, оценка качества ПС является основным содержанием процесса аттестации. Произведенная оценка качества ПС фиксируется в соответствующем решении аттестационной комиссии.

Верификацией и аттестацией называются процессы проверки и анализа, в ходе которых проверяется соответствие программного обеспечения (ПО) своей спецификации и требованиям заказчиков. Верификация и аттестация охватывают весь цикл жизни ПО – они начинаются на этапе анализа требований и завершаются проверкой программного кода на этапе тестирования программной системы.

Верификация и аттестация – абсолютно разные понятия, однако часто их путают. Для того чтобы различать их, выведем главное различие между этими терминами. Верификация отвечает на вопрос, правильно ли создана система, а аттестация отвечает на вопрос, правильно ли работает система. Из этого следует, что верификация проверяет соответствие ПО системной спецификации, в частности функциональным и нефункциональным требованиям. Аттестация – это более общий процесс. Во время аттестации цель инженера – доказать заказчику, что продукт оправдывает ожидания последнего. Аттестация проводится после верификации.

На ранних этапах разработки ПО очень важна аттестация системных требований. В требованиях очень часто встречаются ошибки, недочеты, упущения, что может привести к несоответствию продукта

замыслу заказчика. Инженер должен справляться с этой проблемой. Однако, как известно, сложно искоренить все погрешности в требованиях. Отдельные ошибки могут обнаружиться в том случае, если программный продукт реализован.

В процессах верификации и аттестации используются две основные методики проверки и анализа систем: инспектирование ПО и тестирование ПО. Инспектирование ПО подразумевает анализ и проверку различных представлений системы, например, документации. Инспектирование происходит на всех этапах разработки программной системы. Параллельно с инспектированием может проводиться автоматический анализ исходного кода программ и соответствующих документов. Инспектирование и автоматический анализ – это статические методы верификации и аттестации, поскольку им не требуется исполняемая система. Тестирование ПО есть анализ выходных данных и рабочих характеристик программного продукта для проверки правильности работы системы. Тестирование – динамический метод верификации и аттестации, так как применяется к исполняемой системе.

О тестировании программ по-настоящему серьезно заговорили в США в 90-е гг. Это были сумасшедшие для High Tech индустрии годы. Появившиеся на рынке мощные персональные компьютеры, развитые операционные системы и средства программирования позволяли решать все новые и новые, существенно более сложные задачи. Массовое внедрение Интернета, широкое использование RAD-технологий, позволяющих быстро проектировать прикладные программы, повлекли за собой создание огромных по прежним меркам программных систем. И сложность этих систем все растет день ото дня. Соответственно, количество потенциальных проблем также растет с неимоверной быстротой.

В 70–80-е гг. тестирование на практике использовалось только в организациях, создающих большие программные комплексы (операционные системы, уникальные базы данных, системы реального времени, как правило, военного назначения). Подобных организаций было относительно немного. В наше время без хорошо оттестированных программ на рынок не могут выйти сотни и тысячи фирм и компаний.

В настоящее время компьютеры внедряются во все большее количество областей. Они окружают нас повсюду. С каждым днем нам все труднее представлять нашу жизнь без компьютера. И мы все чаще предъявляем повышенные требования к качеству программ, используемых не только в таких важных областях, как медицина, военная

промышленность, космонавтика, финансы, но и в повседневной жизни. Попытайтесь подсчитать, сколько раз вы вспоминали нехорошим словом разработчиков той или иной программы, которой Вам приходилось пользоваться. Думаем, это будет сделать непросто.

Как следствие, все острее и чаще возникают вопросы качества программ у все большего числа компаний, производящих разного рода ПО. Эти компании вкладывают все большее количество средств в обеспечение качества своих продуктов, создавая собственные группы или даже отделы, занимающиеся тестированием, или просто передавая тестирование сторонним организациям. Наиболее крупные и уважающие себя, а также дорожащие своей репутацией компании, создают системы управления качеством (Quality Management System), направленные на постоянное совершенствование процессов и технологий создания программных продуктов, на постоянное повышение качества программ.

Глава 1. Верификация и тестирование программного обеспечения

1.1. Тестирование и качество программного обеспечения

Тестирование программного обеспечения (software testing) – это процесс анализа или эксплуатации программного обеспечения с целью выявления дефектов. Тестирование – это плановая и упорядоченная деятельность.

Большинство стандартов определяют ПО (software) как программу (совокупность программ) плюс программная документация (program documentation). Следовательно, тестировать можно не только программы, но и различные документы. Итак, мы можем тестировать:

- Программы при их непосредственном запуске и исполнении.
- Код программ без их запуска и исполнения.
- Прототип программного продукта (product prototype).
- Различную проектную документацию (project documentation):
  – требования к программному продукту (product requirements);
  – функциональные спецификации к программному продукту (functional specifications);
  – документы, описывающие архитектуру (product architecture), дизайн (product design);
  – план проекта (project plan) и тестовый план (test plan);
  – тестовые сценарии (test cases).
- Сопроводительную документацию (или документацию для конечных пользователей):
  – интерактивную помощь (on-line help);
  – руководства по установке (Installation guide) и использованию программного продукта (user manual).

Качество программного продукта можно оценить некоторым набором характеристик, определяющих, насколько продукт «хорош» с точки зрения всех потенциально заинтересованных в нем сторон. Такими сторонами являются:

– заказчик продукта;
– спонсор;
– конечный пользователь;
– разработчики продукта;

– тестировщики продукта;
– инженеры поддержки;
– отдел обучения;
– отдел продаж и т. п.

Каждый из участников может иметь различное представление о продукте и по-разному судить о том, насколько он хорош или плох, т. е. насколько высоко качество продукта. С точки зрения разработчика, продукт может быть настолько хорош, насколько хороши заложенные в нем алгоритмы и технологии. Пользователю продукта, скорее всего, безразличны детали внутренней реализации, его в первую очередь волнуют вопросы функциональности и надежности. Спонсора интересует цена и совместимость с будущими технологиями. Таким образом, задача обеспечения качества продукта выливается в задачу определения заинтересованных лиц, согласования их критериев качества и нахождения оптимального решения, удовлетворяющего этим критериям.

В рамках подобной задачи группа тестирования рассматривается не просто как еще одна заинтересованная сторона, но и как сторона, способная оценить удовлетворение выбранных критериев и сделать вывод о качестве продукта с точки зрения других участников. К сожалению, далеко не все критерии могут быть оценены группой тестирования. Поэтому ее внимание в основном сосредоточено на критериях, определяющих качество программного продукта с точки зрения конечного пользователя.

Тестирование, с технической точки зрения, есть процесс выполнения приложения на некоторых входных данных и проверка получаемых результатов с целью подтвердить их корректность по отношению к результату.

Тестирование не позиционируется в качестве единственного способа обеспечения качества. Оно является частью общей системы обеспечения качества продукта, элементы которой выбираются по критерию наибольшей эффективности применения в конкретном проекте.

В каждом конкретном проекте элементы системы должны быть выбраны так, чтобы обеспечить приемлемое качество, исходя из приоритетов и имеющихся ресурсов. Выбирая элементы для системы обеспечения качества конкретного продукта, можно применить комбинированное тестирование, обзоры кода, аудит. При подобном выборе некоторые качества, например, легкость модификации и исправления дефектов, не будут оценены и, возможно, выполнены. Задачей

тестирования в рассматриваемом случае будет обнаружение дефектов и оценка удобства использования продукта, включая полноту функциональности. Исходя из задач, поставленных перед группой тестирования в конкретном проекте, выбирается соответствующая стратегия тестирования. Так, в данном примере, ввиду необходимости оценить удобство использования и полноту функциональности, преимущественный подход к разработке тестов следует планировать на основе использования сценариев.

Итак, основная последовательность действий при выборе и оценке критериев качества программного продукта включает:

1) определение всех лиц, так или иначе заинтересованных в исполнении и результатах данного проекта;

2) определение критериев, формирующих представление о качестве для каждого из участников;

3) приоритезацию критериев, с учетом важности конкретного участника для компании, выполняющей проект, и важности каждого из критериев для данного участника;

4) определение набора критериев, которые будут отслежены и выполнены в рамках проекта, исходя из приоритетов и возможностей проектной команды. Постановка целей по каждому из критериев;

5) определение способов и механизмов достижения каждого критерия;

6) определение стратегии тестирования исходя из набора критериев, попадающих под ответственность группы тестирования, выбранных приоритетов и целей.

Процесс тестирование напрямую связан с понятием качества программного продукта. В RUP (Rational Unified Process – методология разработки ПО) качество программного продукта определяется следующим образом:

Качество программного обеспечения – это характеристика, демонстрирующая то, что продукт удовлетворяет установленным требованиям или превышает их. При этом качество определяется на основе использования точно установленных метрик и критериев.

Качество рассматривается с двух точек зрения: качество продукта и качества процесса.

Качество продукта – это качество основного производимого изделия и всех элементов, входящих в него (компоненты, подсистемы, архитектура и т. д.).

Качество процесса относится к оценке того, как процесс разработки был осуществлен (включая метрики и критерии качества) и насколько твердо организация-разработчик ему следовала для производства требуемого продукта. Дополнительно качество процесса также характеризуется качеством документов и моделей (планов итераций, планов тестирования, реализации функций предметной области, модели проекта и т. д.), созданных для поддержки разработки основного продукта.

Обеспечение качества – это совокупность планируемых и систематически проводимых мероприятий для подтверждения того, что программный продукт удовлетворяет определенным требованиям к качеству.

Приведем несколько концепций хорошего качества программного продукта, которые предлагаются в RUP. Они основаны на нижеизложеных утверждениях:

1. *Не слишком плохо*. Качество программного продукта должно позволить оставаться этому продукту на рынке.

2. *Непогрешимость*. Следует считать, что проектная команда, выпускающая продукт, является лучшей в мире, поэтому и продукт, который она выпускает, есть лучший в мире.

3. *Совершенство*. Проектная команда делает все, чтобы создать совершенный программный продукт.

4. *Клиент всегда прав*. Если клиенту нравится создаваемый продукт, то этого достаточно.

5. *Хороший процесс разработки*. Качество продукта определяется хорошим процессом разработки.

6. *Удовлетворение требований*. Если продукт удовлетворяет требованиям – это хороший продукт.

7. *Ответственность*. Качество продукта определяется контрактом. Если проектная команда выполнила контракт, то качество продукта хорошее.

8. *Защита*. Проблемы, возникающие при разработке программного продукта, должны быть определены, зафиксированы и предотвращены.

9. *Учет многих факторов*. Продукт является хорошим, когда он имеет достаточно преимуществ и с ним не возникает никаких критических проблем.

В большинстве проектов критерии к качеству определяются в основном пунктами 4–6 и 9. То, что перечислено выше, это своего рода критерии качества. Они, конечно же, сильно размыты и их нужно детализировать (что и делается в конкретных проектах).

Именно во многом на основании критериев качества и делается заключение о том, достаточно тестировать продукт или еще нет.

1.2. Классификация видов тестирования

Тестирование можно классифицировать по очень большому количеству признаков. Соответствующий материал достаточно объемен и сложен, поэтому приведем на рис. 1.1 самую простую, минимальную классификацию.

Классификация методов тестирования по уровню детализации приложения приводится в параграфе 1.3.

Согласно определению тестирования этот вид деятельности предусматривает «анализ» и «эксплуатацию» программного продукта.

Процесс, связанный с анализом разработки программного обеспечения, называется *статическим тестированием* (static testing). Статическое тестирование предусматривает проверку любых рабочих продуктов, например, таких как программный код, требования к программному продукту, функциональная спецификация, архитектура, дизайн и т. д. Статическое тестирование является одним из наиболее эффективных способов выявления дефектов на ранних стадиях работы над проектом, благодаря чему достигается существенная экономия времени и затрат на разработку. Статическое тестирование по существу есть все, что можно сделать для выявления дефектов без прогона программного кода.

В отличие от статического тестирования, тестовая деятельность, предусматривающая эксплуатацию программного продукта, носит название *динамического тестирования* (dynamic testing). В отличие от статического тестирования, динамическое тестирование без прогона кода программного продукта обойтись не может. Другими словами, динамическое тестирование состоит из запуска программы, прогона всех ее функциональных модулей и сравнения ее фактического поведения с ожидаемым, используя пользовательский интерфейс.

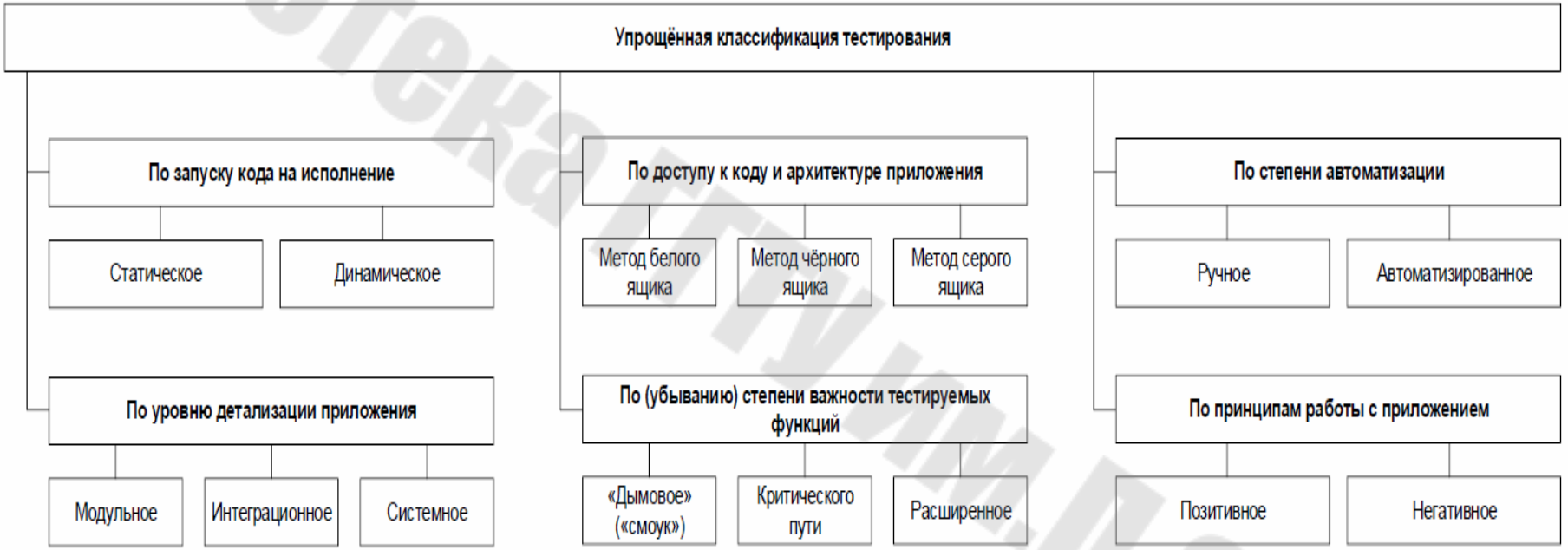

Рис. 1.1. Классификация тестирования

Для тестирования программного кода без его непосредственного запуска применяется *метод белого ящика* (white box testing method). Говоря о структурном тестировании (structured testing), мы понимаем как раз тестирование данным методом. Его тесты основаны на знании кода приложения и его внутренних механизмов. Соответственно концепция структурного тестирования связана с тестированием внутренней структуры исходного кода программного обеспечения. Метод белого ящика обычно применяется на стадии, когда приложение еще не собрано воедино, но необходимо проверить каждый из его компонентов, модулей, процедур и подпрограмм. Следовательно, структурное тестирование тесно взаимосвязано с компонентным или модульным тестированием (unit testing), которое чаще всего выполняет программист, хорошо понимающий код.

Метод черного ящика (black box testing), тесты которого разработаны исходя из знаний функциональных и бизнес требований к тестируемому продукту, используется для тестирования программы при ее запуске на исполнение. Тестировщик тестирует программу так, как с ней будет работать конечный пользователь, и он ничего не знает о внутренних механизмах и алгоритмах, по которым работает код программы. Другими словами, он запускает приложение на выполнение и тестирует его функциональность, используя пользовательский интерфейс для ввода входных данных и получая выходные. Но как при этом обрабатываются входные данные, он не знает. Цель данного метода – проверить работу всех функций приложения на соответствие функциональным требованиям.

Основная разница между тестированиями по методу черного и белого ящиков в том, что метод черного ящика может скрыть проблемы, которые метод белого ящика обнаружит. Так, метод черного ящика может не сообщить о некорректном функционировании объекта, потому что проблемы в работе оказались незаметны. Метод белого ящика может обнаружить этот некорректный объект или функцию, проведя их по специальному пути исполнения кода.

Известен метод *серого ящика* (gray box testing method), который сочетает в себе нечто среднее между методами белого и черного ящиков. Также существуют следующие классификации методов тестирования:

• По степени автоматизации:

– *ручное тестирование* – тест-кейсы выполняет человек;

– *автоматизированное тестирование* – тест-кейсы частично или полностью выполняет специальное инструментальное средство.

• По (убыванию) степени важности тестируемых функций (по уровню функционального тестирования):

– *дымовое тестирование* – проверка самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения;

– *тестирование критического пути* – проверка функциональности, используемой типичными пользователями в типичной повседневной деятельности;

– *расширенное тестирование* – проверка всей (остальной) функциональности, заявленной в требованиях.

• По принципам работы с приложением:

– *позитивное тестирование* – все действия с приложением выполняются строго по инструкции без каких-либо недопустимых действий, некорректных данных и т. д. Можно образно представить, что приложение исследуется в «тепличных условиях»;

– *негативное тестирование* – в работе с приложением выполняются некорректные операции и используются данные, потенциально приводящие к ошибкам (классика жанра – деление на нуль). Негативные тесты не предполагают возникновения в приложении ошибки. Напротив, они предполагают, что верно работающее приложение даже в критической ситуации поведет себя правильным образом (в примере с делением на нуль отобразит сообщение «Делить на нуль запрещено»).

1.3. Жизненный цикл проекта и его связь с тестированием. Фазы процесса тестирования

Выделяются четыре основных фазы разработки ПО:

1) inception (начало);

2) elaboration (уточнение);

3) construction (имплементация, собственно разработка);

4) transition (переход к сдаче продукта и передаче его в поддержку).

Диаграмма с видами и объемами работ на каждой фазе разработки программного обеспечения показана на рис. 1.2.

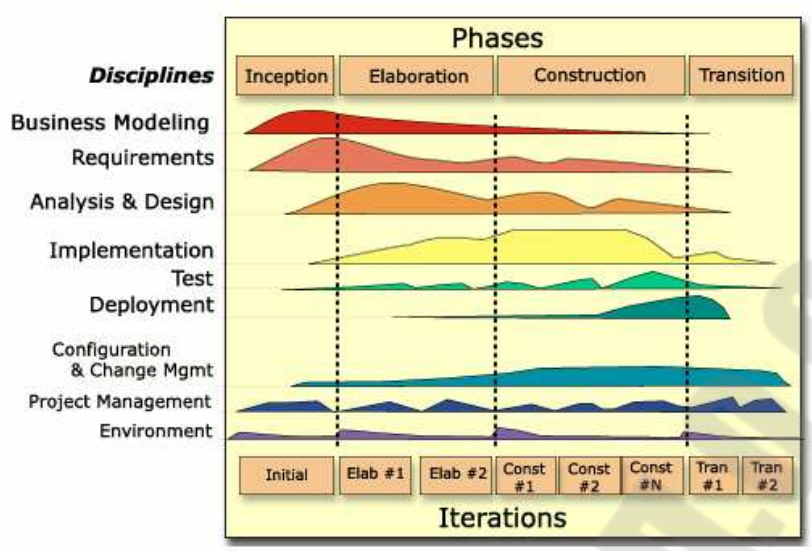

Рис. 1.2. Фазы разработки программного обеспечения

Давайте разберемся, что происходит на каждой фазе.

На фазе ***inception*** происходит сбор пожеланий заказчика, формулируются так называемые бизнес требования (дисциплина Business Modeling), которые затем перерастают в функциональные требования (Requirements). На этой же фазе чуть позже стартует анализ этих самых требований с точки зрения построения архитектуры и дизайна будущего приложения (analysis and design). Как только появляются первые документы, тестировщики начинают их анализировать и готовиться к тестированию (дисциплина Test) и чуть позже разработчики начинают кодировать.

На фазе ***elaboration*** активность разработчиков постоянно растет. Пишутся отдельные компоненты, проводятся юнит и интеграционные тесты. Идет активное уточнение и доработка требований, дизайна.

На фазе ***constraction*** активность разработчиков очень высокая, они вовсю имплементируют функциональность. Активность тестировщиков тоже постоянно растет, и ближе к концу фазы достигает пика. А вот работа с требованиями постепенно угасает, так как основное уже сделано. Происходят некоторые доработки изменения, но уже в несколько меньшей степени, хотя все еще активно.

Наконец на фазе ***transition*** в основном все активности ведутся в рамках развертывания системы у заказчика (дисциплина Deployment) и приемочного тестирования. Активность разработчиков и тестировщиков, соответственно, снижается.

Рассмотрим тестирование в процессе разработки ПО.

Цикл разработки ПО начинается с идентификации требований к ПО и заканчивается тестированием и передачей в эксплуатацию.

Традиционно использовались модели последовательного типа, где работы велись одна за другой. Рассмотрим две характерные модели последовательного типа: V-модель (рис. 1.3) и водопадную модель (рис. 1.4).

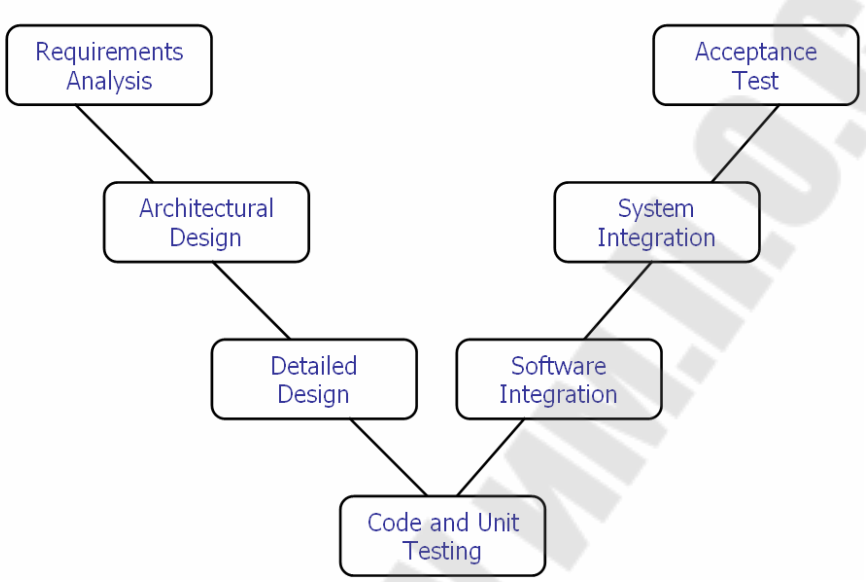

Рис. 1.3. V-модель

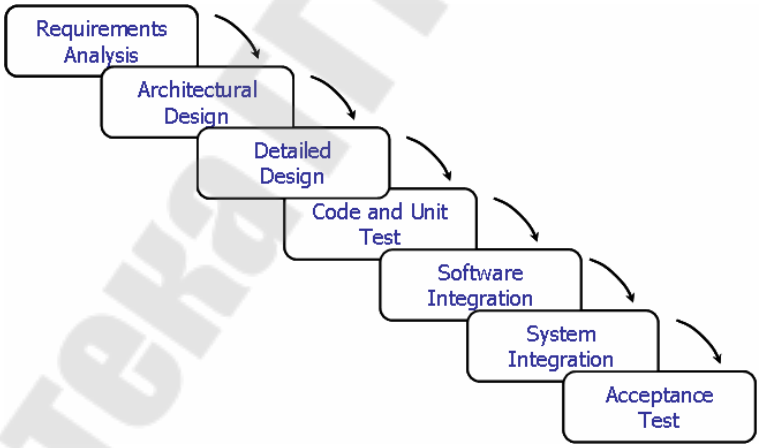

Рис. 1.4. Водопадная модель

Характерная особенность последовательных моделей заключается в том, что спецификации создаются на протяжении первых трех фаз и только потом начинаются разработка и тестирование и продолжаются в течение последующих четырех фаз. Причем V-модель более прогрессивная, чем водопадная. В водопадной, пока не закончилась предыдущая фаза, следующая не начинается. Время и практика пока-

зали, что все последовательные модели менее эффективны, поскольку не позволяют обнаруживать ошибки и выполнять тестирование на ранних стадиях, когда спецификации только пишутся. Изначально большинство ошибок скрыто как раз в требованиях к программному продукту. И только потом они попадают в следующие фазы. И чем позже ошибка будет найдена, тем труднее ее исправить и тем больше требуется времени на исправление.

Постепенно мир перешел на другие модели – итеративные (рис. 1.5).

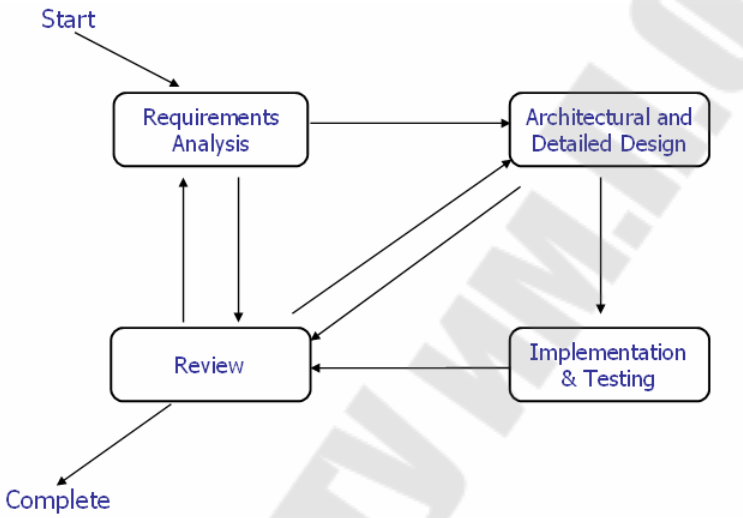

Рис. 1.5. Итеративная модель

Итеративная модель предполагает совершенствование продукта через итерации. В каждой итерации коллектив разработчиков выполняет сборку программы. Каждая сборка является потенциальным кандидатом для тестирования. Для каждой сборки могут разрабатываться или уточняться тесты. Поскольку процесс разработки является итерационным, некоторые тесты, используемые при тестировании ранних сборок, могут быть использованы и при тестировании последующих сборок (регрессионное тестирование).

1.4. Планирование тестирования

При тестировании ПО выделяются три основных уровня: модульное тестирование; интеграционное тестирование; системное тестирование.

Рассмотрим все три уровня тестирования более подробно.

Модульное тестирование – это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель мо-

дульного тестирования состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. Модульное тестирование проводится по принципу «белого ящика», т. е. основывается на знании внутренней структуры программы и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и тому подобное обычно пропускаются на уровне модульного тестирования и выявляются на более поздних стадиях тестирования.

Именно эффективность обнаружения тех или иных типов дефектов должна определять стратегию модульного тестирования, т. е. расстановку акцентов при определении набора входных значений. У организации, занимающейся разработкой программного обеспечения, как правило, имеется историческая база данных (Repository) разработок, хранящая конкретные сведения о разработке предыдущих проектов: о версиях и сборках кода (build), зафиксированных в процессе разработки продукта, о принятых решениях, допущенных просчетах, ошибках, успехах и т. п. Проведя анализ характеристик прежних проектов, подобных заказанному организации, можно предохранить новую разработку от старых ошибок, например, определив типы дефектов, поиск которых наиболее эффективен на различных этапах тестирования.

В данном случае анализируется этап модульного тестирования. Если анализ не дал нужной информации, например, в случае проектов, в которых соответствующие данные не собирались, то основным правилом становится поиск локальных дефектов, у которых код, ресурсы и информация, вовлеченные в дефект, характерны именно для данного модуля. В этом случае на модульном уровне ошибки, связан-

ные, например, с неверным порядком или форматом параметров модуля, могут быть пропущены, поскольку они вовлекают информацию, затрагивающую другие модули (а именно спецификацию интерфейса), в то время как ошибки в алгоритме обработки параметров довольно легко обнаруживаются.

Интеграционное тестирование – это тестирование части системы, состоящей из двух и более модулей. Основная задача интеграционного тестирования – поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки (Stub) на месте отсутствующих модулей. Основная разница между модульным и интеграционным тестированием состоит в целях, т. е. в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с тестированием интерфейсов, например, вызовов функций или методов, или анализ использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой.

Предположим, имеется система, состоящая из оттестированных на этапе модульного тестирования модулей. Задача, решаемая методом интеграционного тестирования, – тестирование межмодульных связей, реализующихся при исполнении программного обеспечения этой системы. Интеграционное тестирование использует модель «белого ящика» на модульном уровне. Поскольку тестировщику текст программы известен с детальностью до вызова всех модулей, входящих в тестируемый комплекс, применение структурных критериев на данном этапе возможно и оправданно.

Интеграционное тестирование применяется на этапе сборки модульно оттестированных модулей в единый комплекс. Известны два метода сборки модулей:

– *монолитный*, характеризующийся одновременным объединением всех модулей в тестируемый комплекс;

– *инкрементальный*, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с пошаговым тестиро-

ванием собираемого комплекса. В инкрементальном методе выделяют две стратегии добавления модулей:

– «сверху вниз» и соответствующее ему восходящее тестирование;

– «снизу вверх» и соответственно нисходящее тестирование.

Особенности монолитного тестирования заключаются в следующем: для замены неразработанных к моменту тестирования модулей, кроме самого верхнего, необходимо дополнительно разрабатывать драйверы (test driver) и (или) заглушки (stub), замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение монолитного и инкрементального подхода дает следующее:

• Монолитное тестирование требует больших трудозатрат, связанных с дополнительной разработкой драйверов и заглушек и со сложностью идентификации ошибок, проявляющихся в пространстве собранного кода.

• Пошаговое тестирование связано с меньшей трудоемкостью идентификации ошибок за счет постепенного наращивания объема тестируемого кода и соответственно локализации добавленной области тестируемого кода.

• Монолитное тестирование предоставляет большие возможности распараллеливания работ, особенно на начальной фазе тестирования.

Особенности нисходящего тестирования заключаются в следующем: организация среды для исполняемой очередности вызовов оттестированными модулями тестируемых модулей, постоянная разработка и использование заглушек, организация приоритетного тестирования модулей, содержащих операции обмена с окружением, или модулей, критичных для тестируемого алгоритма.

Недостатки нисходящего тестирования:

• Проблема разработки достаточно «интеллектуальных» заглушек, т. е. заглушек, способных к использованию при моделировании различных режимов работы комплекса, необходимых для тестирования.

• Сложность организации и разработки среды для реализации исполнения модулей в нужной последовательности.

• Параллельная разработка модулей верхних и нижних уровней приводит к не всегда эффективной реализации модулей из-за подстройки (специализации) еще не тестированных модулей нижних уровней к уже оттестированным модулям верхних уровней.

Особенности восходящего тестирования заключаются в организации порядка сборки и перехода к тестированию модулей, соответствующему порядку их реализации.

Недостатки восходящего тестирования:

• Запаздывание проверки концептуальных особенностей тестируемого комплекса.

• Необходимость в разработке и использовании драйверов.

Системное тестирование качественно отличается от интеграционного и модульного уровней. *Системное тестирование* рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов, в отличие от последних фаз интеграционного тестирования, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций. Основная задача *системного тестирования* – в выявлении дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и т. п.

Системное тестирование производится над проектом в целом с помощью метода «черного ящика». Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю. Тестированию подлежат коды и пользовательская документация.

Категории тестов системного тестирования:

• Полнота решения функциональных задач.

• Стрессовое тестирование – на предельных объемах нагрузки входного потока.

• Корректность использования ресурсов (утечка памяти, возврат ресурсов).

• Оценка производительности.

• Эффективность защиты от искажения данных и некорректных действий.

• Проверка инсталляции и конфигурации на разных платформах.

• Корректность документации.

Поскольку системное тестирование проводится на пользовательских интерфейсах, создается иллюзия того, что построение специальной системы автоматизации тестирования не всегда необходимо. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная автоматизация тестирования, что приводит к созданию тестовой системы гораз-

до более сложной, чем система тестирования, применяемая на уровне тестирования модулей или их комбинаций.

Задача планирования активности тестирования состоит в оптимальном распределении ресурсов между всеми уровнями тестирования.

В процессе тестирования выделяют следующие фазы:

1. Определение целей (требований к тестированию), включающее следующую конкретизацию: какие части системы будут тестироваться, какие аспекты их работы будут выбраны для проверки, каково желаемое качество и т. п.

2. Планирование: создание графика (расписания) разработки тестов для каждой тестируемой подсистемы; оценка необходимых человеческих, программных и аппаратных ресурсов; разработка расписания тестовых циклов. Важно отметить, что расписание тестирования обязательно должно быть согласовано с расписанием разработки создаваемой системы, поскольку наличие исполняемой версии разрабатываемой системы (Implementation Under Testing (IUT) или Application Under Testing (AUT) – часто употребляемые обозначения для тестируемой системы) является одним из необходимых условий тестирования, что создает взаимозависимость в работе команд тестировщиков и разработчиков.

3. Разработка тестов, т. е. тестового кода для тестируемой системы, если необходимо – кода системы автоматизации тестирования и тестовых процедур (выполняемых вручную).

4. Выполнение тестов: реализация тестовых циклов.

5. Анализ результатов.

После анализа результатов возможно повторение процесса тестирования, начиная с пунктов 3, 2 или даже 1.

Тестовый цикл – это цикл исполнения тестов, включающий фазы 4 и 5 тестового процесса. Тестовый цикл заключается в прогоне разработанных тестов на некотором однозначно определяемом срезе системы (build). Тестовый цикл включает следующую последовательность действий:

1. Проверка готовности системы и тестов к проведению тестового цикла, включающая:

- Проверку того, что все тесты, запланированные для исполнения на данном цикле, разработаны и помещены в систему версионного контроля.
- Проверку того, что все подсистемы, запланированные для тестирования на данном цикле, разработаны и помещены в систему версионного контроля.

• Проверку того, что разработана и задокументирована процедура определения и создания среза системы, или build.

• Проверки некоторых дополнительных критериев.

2. Подготовка тестовой машины в соответствии с требованиями, определенными на этапе планирования (например, полная очистка и переустановка системного программного обеспечения). Конфигурация тестовой машины так же, как и срез системы, должны быть однозначно воспроизводимыми.

3. Воспроизведение среза системы.

4. Прогон тестов в соответствии с задокументированными процедурами.

5. Сохранение тестовых протоколов (test log). Test log может содержать вывод системы в STDOUT, список результатов сравнения, полученных при исполнении данных с эталонными или любые другие выходные данные тестов, с помощью которых можно проверить правильность работы системы.

6. Анализ протоколов тестирования и принятие решения о том прошел или не прошел каждый из тестов (Pass/Fail).

7. Анализ и документирование результатов цикла.

Последний перед выпуском продукта тестовый цикл не должен включать изменений кода build или кода продукта тестируемой системы. Этот цикл называется «финальным». Таким образом обеспечивается ситуация, когда финальный цикл полностью повторяем, а выпускаемый продукт полностью совпадает с продуктом, который прошел тестирование. Финальный цикл необходим для гарантии достоверности результатов тестирования.

Тестовый план – это документ, или набор документов, содержащий следующую информацию:

1) тестовые ресурсы;

2) перечень функций и подсистем, подлежащих тестированию;

3) тестовую стратегию, включающую:

– анализ функций и подсистем с целью определения наиболее слабых мест, т. е. областей функциональности тестируемой системы, где появление дефектов наиболее вероятно;

– определение стратегии выбора входных данных для тестирования. Так как множество возможных входных данных программного продукта, как правило, практически бесконечно, выбор конечного подмножества, достаточного для проведения исчерпывающего тестирования, является сложной задачей. Для ее решения могут быть применены такие методы, как покрытие классов входных и выходных

данных, анализ крайних значений, покрытие модели использования, анализ временной линии и т. п. Выбранную стратегию необходимо обосновать и задокументировать;

– определение потребности в автоматизированной системе тестирования и дизайн такой системы;

4) расписание тестовых циклов;

5) фиксацию тестовой конфигурации: состава и конкретных параметров аппаратуры и программного окружения;

6) определение списка тестовых метрик, которые на тестовом цикле необходимо собрать и проанализировать. Например, метрик, оценивающих степень покрытия тестами набора требований, степень покрытия кода тестируемой системы, количество и уровень серьезности дефектов, объем тестового кода и другие характеристики.

1.5. Разработка требований

Требования являются отправной точкой для определения того, что проектная команда будет проектировать, реализовывать и тестировать. Если в требованиях что-то «не то», то и реализовано будет «не то», т. е. колоссальная работа множества людей будет выполнена впустую. Эту мысль иллюстрирует рис. 1.6.

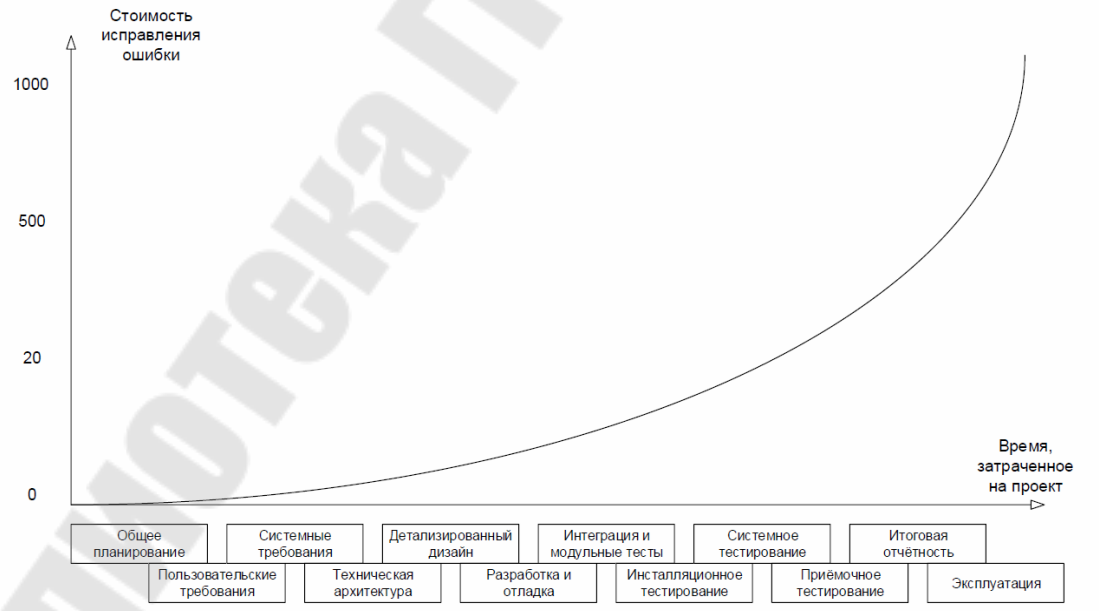

Рис. 1.6. Стоимость исправления ошибки в зависимости от момента ее обнаружения

Брайан Хэнкс, описывая важность требований, подчеркивает, что они:

– позволяют понять, что и с соблюдением каких условий система должна делать;

– предоставляют возможность оценить масштаб изменений и управлять изменениями;

– являются основой для формирования плана проекта (в том числе плана тестирования);

– помогают предотвращать или разрешать конфликтные ситуации;

– упрощают расстановку приоритетов в наборе задач;

– позволяют объективно оценить степень прогресса в разработке проекта.

Вне зависимости от того, какая модель разработки ПО используется на проекте, чем позже будет обнаружена проблема, тем сложнее и дороже будет ее решение. А в самом начале («водопада», «спуска по букве v», «итерации», «витка спирали») идет планирование и работа с требованиями.

Если проблема в требованиях будет выяснена на этой стадии, ее решение может свестись к исправлению пары слов в тексте, в то время как недоработка, вызванная пропущенной проблемой в требованиях и об обнаруженная на стадии эксплуатации, может даже полностью уничтожить проект.

Рассмотрим уровни и типы требований (рис. 1.7).

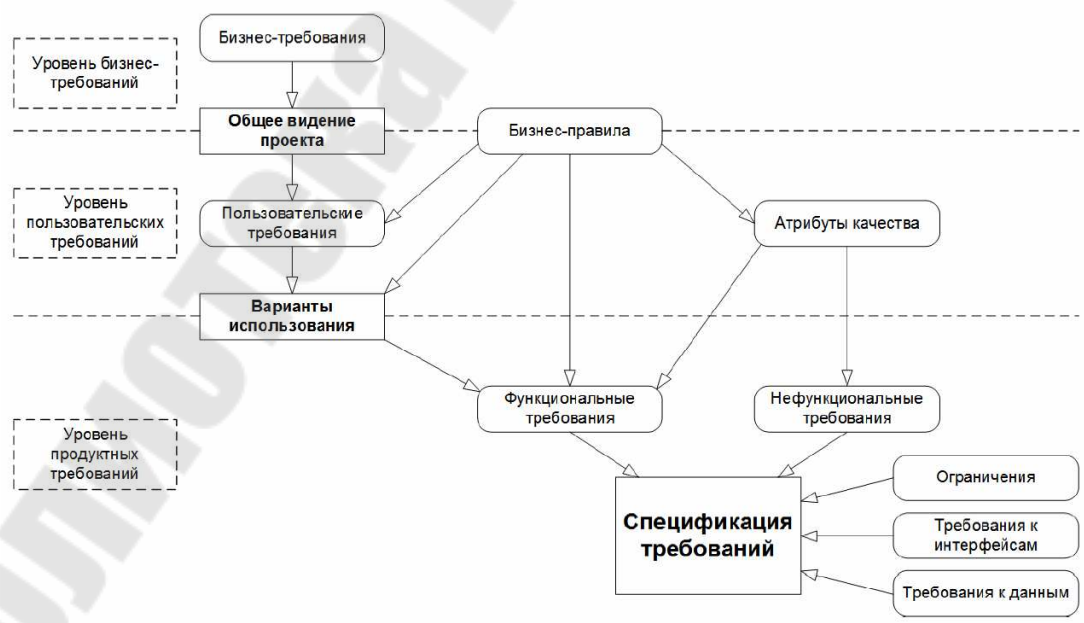

Рис. 1.7. Уровни и типы требований

Бизнес-требования (business requirements) выражают цель, ради которой разрабатывается продукт (зачем вообще он нужен, какая от него ожидается польза, как заказчик с его помощью будет получать прибыль). Результатом выявления требований на этом уровне является общее видение (vision and scope) – документ, который, как правило, представлен простым текстом и таблицами. Здесь нет детализации поведения системы и иных технических характеристик, но вполне могут быть определены приоритеты решаемых бизнес-задач, риски и т. п.

Пользовательские требования (user requirements) описывают задачи, которые пользователь может выполнять с помощью разрабатываемой системы (реакцию системы на действия пользователя, сценарии работы пользователя). Поскольку здесь уже появляется описание поведения системы, требования этого уровня могут быть использованы для оценки объема работ, стоимости проекта, времени разработки и т. д. Пользовательские требования оформляются в виде вариантов использования (use cases), пользовательских историй (user stories), пользовательских сценариев (user scenarios).

Бизнес-правила (business rules) описывают особенности принятых в предметной области (и (или) непосредственно у заказчика) процессов, ограничений и иных правил. Эти правила могут относиться к бизнес-процессам, правилам работы сотрудников, нюансам работы ПО и т. д.

Атрибуты качества (quality attributes) расширяют собой нефункциональные требования и на уровне пользовательских требований могут быть представлены в виде описания ключевых для проекта показателей качества (свойств продукта, не связанных с функциональностью, но являющихся важными для достижения целей создания продукта – производительность, масштабируемость, восста-навливаемость). Атрибутов качества очень много, но для любого проекта реально важными является лишь некоторое их подмножество.

Функциональные требования (functional requirements) описывают поведение системы, т. е. ее действия (вычисления, преобразования, проверки, обработку и т. д.) В контексте проектирования функциональные требования в основном влияют на дизайн системы.

Стоит помнить, что к поведению системы относится не только то, что система должна делать, но и то, что она не должна делать (например: «приложение не должно выгружать из оперативной памяти фоновые документы в течение 30 минут с момента выполнения с ними последней операции»).

Нефункциональные требования (non-functional requirements) описывают свойства системы (удобство использования, безопасность, надежность, расширяемость и т. д.), которыми она должна обладать при реализации своего поведения. Здесь приводится более техническое и детальное описание атрибутов качества. В контексте проектирования нефункциональные требования в основном влияют на архитектуру системы.

Ограничения (limitations, constraints) представляют собой факторы, ограничивающие выбор способов и средств (в том числе инструментов) реализации продукта.

Требования к интерфейсам (external interfaces requirements) описывают особенности взаимодействия разрабатываемой системы с другими системами и операционной средой.

Требования к данным (data requirements) описывают структуры данных (и сами данные), являющиеся неотъемлемой частью разрабатываемой системы. Часто сюда относят описание базы данных и особенностей ее использования.

Спецификация требований (software requirements specification, SRS) объединяет в себе описание всех требований уровня продукта и может представлять собой весьма объемный документ (сотни и тысячи страниц).

Поскольку требований может быть очень много, а их приходится не только единожды написать и согласовать между собой, но и постоянно обновлять, работу проектной команды по управлению требованиями значительно облегчают соответствующие инструментальные средства (requirements management tools).

1.6. Тестирование качества требований

В процессе тестирования требований проверяется их соответствие определенному набору свойств.

Завершенность (completeness). Требование является полным и законченным с точки зрения представления в нем всей необходимой информации, ничто не пропущено по соображениям «это и так всем понятно»:

– отсутствуют нефункциональные составляющие требования или ссылки на соответствующие нефункциональные требования;

– указана лишь часть некоторого перечисления (например: «экспорт осуществляется в форматы PDF, PNG и т. д.» – что мы должны понимать под «и т. д.»?);

– приведенные ссылки неоднозначны.

Атомарность, *единичность* (atomicity). Требование является атомарным, если его нельзя разбить на отдельные требования без потери завершенности и оно описывает одну и только одну ситуацию:

– в одном требовании фактически содержится несколько независимых;

– требование допускает разночтение в силу грамматических особенностей языка;

– в одном требовании объединено описание нескольких независимых ситуаций.

Непротиворечивость, *последовательность* (consistency). Требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам:

– противоречия внутри одного требования;

– противоречия между двумя и более требованиями, между таблицей и текстом, рисунком и текстом, требованием и прототипом и т. д.;

– использование неверной терминологии или использование разных терминов для обозначения одного и того же объекта или явления.

Недвусмысленность (unambiguousness, clearness). Требование описано без использования жаргона, неочевидных аббревиатур и расплывчатых формулировок и допускает только однозначное объективное понимание. Требование атомарно в плане невозможности различной трактовки сочетания отдельных фраз:

– использование терминов или фраз, допускающих субъективное толкование;

– использование неочевидных или двусмысленных аббревиатур без расшифровки;

– формулировка требований из соображений, что нечто должно быть всем очевидно.

Выполнимость (feasibility). Требование технологически выполнимо и может быть реализовано в рамках бюджета и сроков разработки проекта:

– так называемое «озолочение» (gold plating) — требования, которые крайне долго и (или) дорого реализуются и при этом практически бесполезны для конечных пользователей;

– технически нереализуемые на современном уровне развития технологий требования;

– в принципе нереализуемые требования.

Обязательность, *нужность* (obligatoriness) и *актуальность* (up-to-date). Если требование не является обязательным к реализации, оно должно быть просто исключено из набора требований. Если требование нужное, но «не очень важное», для указания этого факта используется указание приоритета. Также исключены (или переработаны) должны быть требования, утратившие актуальность. Частые проблемы с этими требованиями следующие:

– требование было добавлено «на всякий случай», хотя реальной потребности в нем не было и нет;

– требованию выставлены неверные значения приоритета по критериям важности и (или) срочности;

– требование устарело, но не было переработано или удалено.

Прослеживаемость (traceability). Прослеживаемость бывает *вертикальной* (vertical traceability) и *горизонтальной* (horizontal traceability). Вертикальная позволяет соотносить между собой требования на различных уровнях требований, горизонтальная позволяет соотносить требование с тест-планом, тест-кейсами, архитектурными решениями и т. д.

Для обеспечения прослеживаемости часто используются специальные инструменты по управлению требованиями (requirements management tool) и (или) матрицы прослеживаемости (traceability matrix). Частые проблемы с этими требованиями следующие:

– требования не пронумерованы, не структурированы, не имеют оглавления, не имеют работающих перекрестных ссылок;

– при разработке требований не были использованы инструменты и техники управления требованиями;

– набор требований неполный, носит обрывочный характер с явными «пробелами».

Модифицируемость (modifiability). Это свойство характеризует простоту внесения изменений в отдельные требования и в набор требований. Можно говорить о наличии модифицируемости в том случае, если при доработке требований искомую информацию легко найти, а ее изменение не приводит к нарушению иных описанных в этом перечне свойств:

– требования неатомарны и непрослеживаемы, а потому их изменение с высокой вероятностью порождает противоречивость;

– требования изначально противоречивы. В такой ситуации внесение изменений (не связанных с устранением противоречивости)

только усугубляет ситуацию, увеличивая противоречивость и снижая прослеживаемость;

– требования представлены в неудобной для обработки форме (например, не использованы инструменты управления требованиями, и в итоге команде приходится работать с десятками огромных текстовых документов).

Проранжированность по важности, *стабильности*, *срочности* (ranked for importance, stability, priority). Важность характеризует зависимость успеха проекта от успеха реализации требования. Стабильность характеризует вероятность того, что в обозримом будущем в требование не будет внесено никаких изменений. Срочность определяет распределение во времени усилий проектной команды по реализации того или иного требования. Частые проблемы с этими требованиями следующие:

– проблемы с проранжированностью по важности повышают риск неверного распределения усилий проектной команды, направления усилий на второстепенные задачи и конечного провала проекта из-за неспособности продукта выполнять ключевые задачи с соблюдением ключевых условий;

– проблемы с проранжированностью по стабильности повышают риск выполнения бессмысленной работы по совершенствованию, реализации и тестированию требований, которые в самое ближайшее время могут претерпеть кардинальные изменения (вплоть до полной утраты актуальности);

– проблемы с проранжированностью по срочности повышают риск нарушения желаемой заказчиком последовательности реализации функциональности и ввода этой функциональности в эксплуатацию.

Корректность (correctness) и *проверяемость* (verifiability). Фактически эти свойства вытекают из соблюдения всех вышеперечисленных (или можно сказать, что они не выполняются, если нарушено хотя бы одно из вышеперечисленных). В дополнение можно отметить, что проверяемость подразумевает возможность создания объективного тест-кейса (тест-кейсов), однозначно показывающего, что требование реализовано верно и поведение приложения в точности соответствует требованию. Частые проблемы с этими требованиями следующие:

– опечатки (особенно опасны опечатки в аббревиатурах, превращающие одну осмысленную аббревиатуру в другую также осмысленную, но не имеющую отношения к некоему контексту; такие опечатки крайне сложно заметить);

– наличие неаргументированных требований к дизайну и архитектуре;

– плохое оформление текста и сопутствующей графической информации, грамматические, пунктуационные и иные ошибки в тексте;

– неверный уровень детализации;

– требования к пользователю, а не к приложению.

Тестирование документации и требований относится к разряду нефункционального тестирования (non-functional testing). Основные техники такого тестирования в контексте требований таковы.

Взаимный просмотр (peer review). Взаимный просмотр («рецензирование») является одной из наиболее активно используемых техник тестирования требований и может быть представлен в одной из трех следующих форм (по мере нарастания его сложности и цены):

• Беглый просмотр может выражаться как в показе автором своей работы коллегам с целью создания общего понимания и получения обратной связи, так и в простом обмене результатами работы между двумя и более авторами с тем, чтобы коллега высказал свои вопросы и замечания. Это самый быстрый, дешевый и часто используемый вид просмотра. Для запоминания: аналог беглого просмотра – это ситуация, когда вы в школе с одноклассниками проверяли перед сдачей сочинения друг друга, чтобы найти описки и ошибки.

• Технический просмотр выполняется группой специалистов. В идеальной ситуации каждый специалист должен представлять свою область знаний. Тестируемый продукт не может считаться достаточно качественным, пока хотя бы у одного просматривающего остаются замечания. Для запоминания: аналог технического просмотра – это ситуация, когда некий договор визирует юридический отдел, бухгалтерия и т. д.

• Формальная инспекция представляет собой структурированный, систематизированный и документируемый подход к анализу документации. Для его выполнения привлекается большое количество специалистов, само выполнение занимает достаточно много времени,

и потому этот вариант просмотра используется достаточно редко (как правило, при получении на сопровождение и доработку проекта, созданием которого ранее занималась другая компания). Для запоминания: аналог формальной инспекции – это ситуация генеральной уборки квартиры (включая содержимое всех шкафов, холодильника, кладовки и т. д.).

Вопросы. Следующей очевидной техникой тестирования и повышения качества требований является (повторное) использование техник выявления требований, а также (как отдельный вид деятельности) – задавание вопросов. Если хоть что-то в требованиях вызывает у вас непонимание или подозрение – задавайте вопросы. Можно спросить представителей заказчика, можно обратиться к справочной информации. По многим вопросам можно обратиться к более опытным коллегам при условии, что у них имеется соответствующая информация, ранее полученная от заказчика. Главное, чтобы ваш вопрос был сформулирован таким образом, чтобы полученный ответ позволил улучшить требования.

Тест-кейсы и чек-листы. Мы помним, что хорошее требование является проверяемым, а значит, должны существовать объективные способы определения того, верно ли реализовано требование. Продумывание чек-листов или даже полноценных тест-кейсов в процессе анализа требований позволяет нам определить, насколько требование проверяемо. Если можно придумать несколько пунктов чек-листа, это еще не признак того, что с требованием все хорошо (например, оно может противоречить каким-то другим требованиям). Но если никаких идей по тестированию требования в голову не приходит – это тревожный знак.

Рекомендуется для начала убедиться, что вы понимаете требование (в том числе прочесть соседние требования, задать вопросы коллегам и т. д.). Также можно пока отложить работу с данным конкретным требованием и вернуться к нему позднее – возможно, анализ других требований позволит вам лучше понять и это конкретное. Но если ничто не помогает – скорее всего, с требованием что-то не так. Справедливости ради надо отметить, что на начальном этапе проработки требований такие случаи встречаются очень часто – требования сформированы очень поверхностно, расплывчато и явно нуждаются в доработке, т. е. здесь нет необходимости проводить сложный анализ, чтобы констатировать непроверяемость требования. На стадии же, когда требования уже хорошо сформулированы и протестированы, вы

можете продолжать использовать эту технику, совмещая разработку тест-кейсов и дополнительное тестирование требований.

Исследование поведения системы. Эта техника логически вытекает из предыдущей (продумывания тест-кейсов и чек-листов), но отличается тем, что здесь тестированию подвергается, как правило, не одно требование, а целый набор. Тестировщик мысленно моделирует процесс работы пользователя с системой, созданной по тестируемым требованиям, и ищет неоднозначные или вовсе неописанные варианты поведения системы. Этот подход сложен, требует достаточной квалификации тестировщика, но способен выявить нетривиальные недоработки, которые почти невозможно заметить, тестируя требования по отдельности.

Рисунки (графическое представление). Чтобы увидеть общую картину требований целиком, очень удобно использовать рисунки, схемы, диаграммы, интеллект-карты и т. д. Графическое представление удобно одновременно своей наглядностью и краткостью (например, UML-схема базы данных, занимающая один экран, может быть описана несколькими десятками страниц текста). На рисунке очень легко заметить, что какие-то элементы «не стыкуются», что где-то чего-то не хватает и т. д. Если для графического представления требований необходимо использовать общепринятую нотацию (например, уже упомянутый UML), получаем дополнительные преимущества: эту схему смогут без труда понимать и дорабатывать коллеги, а в итоге может получиться хорошее дополнение к текстовой форме представления требований.

Прототипирование. Можно отметить, что прототипирование часто является следствием создания графического представления и анализа поведения системы. С использованием специальных инструментов можно очень быстро сделать наброски пользовательских интерфейсов, оценить применимость тех или иных решений и даже создать не просто «прототип ради прототипа», а заготовку для дальнейшей разработки, если окажется, что реализованное в прототипе (возможно, с небольшими доработками) устраивает заказчика.

Здесь есть несколько важных моментов, на которые стоит обратить внимание:

- Ответы заказчика могут быть менее структурированными и последовательными, чем наши вопросы. Это нормально. Он может позволить себе такое, мы – нет.

• Ответы заказчика могут содержать противоречия. Это тоже нормально, так как заказчик мог что-то забыть или перепутать. Наша задача – свести эти противоречивые данные воедино (если это возможно) и задать уточняющие вопросы (если это необходимо).

• В случае если с нами общается технический специалист, в его ответах вполне могут проскакивать технические жаргонизмы (например, слово «хелп»). Не надо переспрашивать его о том, что это такое, если жаргонизм имеет однозначное общепринятое значение, но при доработке текста наша задача – написать то же самое строгим техническим языком. Если жаргонизм все же непонятен – тогда лучше спросить (так, «хелп» – это всего лишь краткая помощь, выводимая консольными приложениями как подсказка о том, как использовать ПО).

Рассмотрим типичные ошибки при анализе и тестировании требований.

Отметка того факта, что с требованием все в порядке. Если у вас не возникло вопросов и (или) замечаний к требованию – не надо об этом писать. Любые пометки в документе подсознательно воспринимаются как признак проблемы, и такое «одобрение требований» только раздражает и затрудняет работу с документом – сложнее становится заметить пометки, относящиеся к проблемам.

Описание одной и той же проблемы в нескольких местах. Помните, что ваши пометки, комментарии, замечания и вопросы тоже должны обладать свойствами хороших требований (настолько, насколько эти свойства к ним применимы). Если вы много раз в разных местах пишете одно и то же об одном и том же, вы нарушаете как минимум свойство модифицируемости. Постарайтесь в таком случае вынести ваш текст в конец документа, укажите в нем же (в начале) перечень пунктов требований, к которым он относится, а в самих требованиях в комментариях просто ссылайтесь на этот текст.

Написание вопросов и комментариев без указания места требования, к которым они относятся. Если ваше инструментальное средство позволяет указать часть требования, к которому вы пишете вопрос или комментарий, сделайте это (например, Word позволяет

выделить для комментирования любую часть текста – хоть один символ). Если это невозможно, цитируйте соответствующую часть текста. В противном случае вы порождаете неоднозначность или вовсе делаете вашу пометку бессмысленной, так как становится невозможно понять, о чем вообще идет речь.

Задавание плохо сформулированных вопросов. Есть три вида таких вопросов:

• Первый вид возникает из-за того, что автор вопроса не знает общепринятой терминологии или типичного поведения стандартных элементов интерфейса (например, «что такое чек-бокс?», «как в списке можно выбрать несколько пунктов?», «как подсказка может всплывать?»).

• Второй вид плохих вопросов похож на первый из-за формулировок: вместо того, чтобы написать «что вы имеете в виду под {чем-то}?», автор вопроса пишет «что такое {что-то}?» То есть вместо вполне логичного уточнения получается ситуация, очень похожая на рассмотренную в предыдущем пункте.

• Третий вид сложно привязать к причине возникновения, но его суть в том, что к некорректному и (или) невыполнимому требованию задается вопрос наподобие «что будет, если мы это сделаем?» Ничего не будет, так как мы это точно не сделаем. И вопрос должен быть совершенно иным (каким именно – зависит от конкретной ситуации, но точно не таким). И еще раз напомним о точности формулировок: иногда одно-два слова могут на корню уничтожить отличную идею, превратив хороший вопрос в плохой. Сравните: «Что такое формат даты по умолчанию?» и «Каков формат даты по умолчанию?» Первый вариант просто показывает некомпетентность автора вопроса, тогда как второй – позволяет получить полезную информацию. К этой же проблеме относится непонимание контекста. Часто можно увидеть вопросы в стиле «о каком приложении идет речь?», «что такое система?» и им подобные. Чаще всего автор таких вопросов просто вырвал требование из контекста, по которому было совершенно ясно, о чем идет речь.

Написание очень длинных комментариев и (*или*) *вопросов.* История знает случаи, когда одна страница исходных требований превращалась в 20–30 страниц текста анализа и вопросов. Это плохой подход. Все те же мысли можно выразить значительно более кратко, чем сэкономить как свое время, так и время автора исходного документа. Тем более стоит учитывать, что на начальных стадиях работы с тре-

бованиями они весьма нестабильны, и может получиться так, что ваши 5–10 страниц комментариев относятся к требованию, которое просто удалят или изменят до неузнаваемости.

Критика текста или даже его автора. Помните, что ваша задача – сделать требования лучше, а не показать их недостатки (или недостатки автора). Потому комментарии вида «плохое требование», «неужели вы не понимаете, как глупо это звучит», «надо переформулировать» неуместны и недопустимы.

Категоричные заявления без обоснования. Как продолжение ошибки «критика текста или даже его автора» можно отметить и просто категоричные заявления наподобие «это невозможно», «мы не будем этого делать», «это не нужно». Даже если вы понимаете, что требование бессмысленно или невыполнимо, эту мысль стоит сформулировать в корректной форме и дополнить вопросами, позволяющими автору документа самому принять окончательное решение. Например, «это не нужно» можно переформулировать так: «Мы сомневаемся в том, что данная функция будет востребована пользователями. Какова важность этого требования? Уверены ли вы в его необходимости?».

Указание проблемы с требованиями без пояснения ее сути. Помните, что автор исходного документа может не быть специалистом по тестированию или бизнес-анализу. Потому просто пометка в стиле «неполнота», «двусмысленность» и так далее могут ничего ему не сказать. Поясняйте свою мысль. Сюда же можно отнести небольшую, но досадную недоработку, относящуюся к противоречивости: если вы обнаружили некие противоречия, сделайте соответствующие пометки во всех противоречащих друг другу местах, а не только в одном из них. Например, вы обнаружили, что требование 20 противоречит требованию 30. Тогда в требовании 20 отметьте, что оно противоречит требованию 30, и наоборот. И поясните суть противоречия.

Плохое оформление вопросов и комментариев. Старайтесь сделать ваши вопросы и комментарии максимально простыми для восприятия. Помните не только о краткости формулировок, но и об оформлении текста. Если вопросы структурированы в виде списка – такая структура воспринимается намного лучше, чем сплошной текст. Перечитайте свой текст, исправьте опечатки, грамматические и пунктуационные ошибки и т. д.

Описание проблемы не в том месте, к которому она относится. Классическим примером может быть неточность в сноске, приложении или рисунке, которая почему-то описана не там, где она нахо-

дится, а в тексте, ссылающемся на соответствующий элемент. Исключением может считаться противоречивость, при которой описать проблему нужно в обоих местах.

Ошибочное восприятие требования как «требования к пользователю». Выше было изложено, что требования в стиле «пользователь должен быть в состоянии отправить сообщение» являются некорректными. И это так. Но бывают ситуации, когда проблема намного менее опасна и состоит только в формулировке. Например, фразы в стиле «пользователь может нажать на любую из кнопок», «пользователю должно быть видно главное меню» на самом деле означают «все отображаемые кнопки должны быть доступны для нажатия» и «главное меню должно отображаться». Да, эту недоработку тоже стоит исправить, но не следует отмечать ее как критическую проблему.

Скрытое редактирование требований. Эту ошибку можно смело отнести к разряду крайне опасных. Ее суть состоит в том, что тестировщик произвольно вносит правки в требования, никак не отмечая этот факт. Соответственно, автор документа, скорее всего, не заметит такой правки, а потом будет очень удивлен, когда в продукте что-то будет реализовано совсем не так, как когда-то было описано в требованиях. Потому простая рекомендация: если вы что-то правите, обязательно отмечайте это (средствами вашего инструмента или просто явно в тексте). И еще лучше отмечать правку как предложение по изменению, а не как свершившийся факт, так как автор исходного документа может иметь совершенно иной взгляд на ситуацию.

Анализ, не соответствующий уровню требований. При тестировании требований следует постоянно помнить, к какому уровню они относятся, так как в противном случае появляются следующие типичные ошибки:

- Добавление в бизнес-требования мелких технических подробностей.
- Дублирование на уровне пользовательских требований части бизнес-требований (если вы хотите увеличить прослеживаемость набора требований, имеет смысл просто использовать ссылки).
- Недостаточная детализация требований уровня продукта (общие фразы, допустимые, например, на уровне бизнес-требований, здесь уже должны быть предельно детализированы, структурированы и дополнены подробной технической информацией).

Глава 2. Разработка тестовой документации. Поиск и документирование дефектов

2.1. Тестирование программного обеспечения: разработка тестов

Итак, говоря о тестировании методом черного ящика, мы говорим о функциональном тестировании (functional testing). Функциональное тестирование еще называют поведенческим, или тестированием на поведенческом уровне. Тестировщики, занимающиеся функциональным тестированием, используют преимущественно метод черного ящика, а программисты, перед тем как передать приложение в тестирование, проверяют свои модули методом белого ящика.

Функциональное тестирование — один из процессов жизненного цикла программного продукта, который проводится с целью получения объективных доказательств функционирования программного продукта в соответствии с установленными либо подразумеваемыми заказчиком требованиями к программному продукту. В процессе функционального (а также любого другого) тестирования тестировщик ищет дефекты (defect or bug).

Функциональное тестирование подразделяется на ручное (manual testing) и автоматическое или автоматизированное (automated testing). Ручное тестирование подразумевает выполнение тестов вручную. В свою очередь автоматическое тестирование подразумевает привлечение каких-либо средств или инструментов для автоматизирования тестирования.

Цели функционального тестирования

Тестирование осуществляется с тем, чтобы:

– обнаружить ошибки и задокументировать их;

– определить, соответствует ли приложение предъявляемым к нему требованиям;

– принять объективное заключение о возможности поставки программного продукта заказчику и документирование этого заключения.

Тестировщики не принимают окончательного решения о готовности программного продукта. Как правило, это делает менеджер проекта или решает сам заказчик. Однако тестировщик может повли-

ять на принятие решения, предоставляя полную и максимально объективную информацию о том, в каком состоянии находится продукт и каков уровень его качества на данный момент.

Уровни функционального тестирования

Приемочный тест – это самый первый и короткий тест, проверяющий работу основной функциональности программного продукта. Данный тест длится от получаса до двух-трех часов максимум в зависимости сложности программы, по результатам которого ведущий инженер по тестированию принимает решение о целесообразности дальнейшего тестирования. Если программа не прошла приемочный тест, она отправляется на доработку к программистам.

Критический тест – основной вид теста, во время которого проверяется основная функциональность программного продукта, критичная для конечного пользователя при стандартном его использовании. В рамках данного тестирования, как правило, проверяется большинство требований, предъявляемых к программному продукту.

Расширенный тест – это углубленный тест, при котором проверяется нестандартное использование программного продукта. Прогоняются различные сложные, логически запутанные сценарии, совершаются действия, которые конечный пользователь будет совершать очень редко. Тестирование этого уровня требуется далеко не для всех типов приложений и во многих случаях могут не проводиться или проводиться ограниченно.

Типы (виды) тестирования

Инсталляционное тестирование (Installation testing). В процессе инсталляционного тестирования проверяется корректность установки и деинсталляции программного продукта в среде, максимально приближенной к эксплуатационной. Проверка правильности установки программного продукта должна быть обязательным элементом проекта по тестированию любого продукта.

Регрессионное тестирование (*Regression testing*). Повторное выполнение тестов для проверки того, что изменения, внесенные в программу в результате разработки новой или изменении существующей функциональности, или в результате устранения ошибок, не повлияли на функциональность, которая не изменялась (т. е. текущая версия ведет себя идентично предыдущей, за исключением измененных областей, и новых ошибок в уже оттестированной ранее функциональности не появилось).

Тестирование новой функциональности (New Feature testing). В данном виде тестирования акцент делается на тестирование новой

функциональности, появившейся в конкретном выпуске (build) программного продукта.

Конфигурационное тестирование (Configuration testing). С помощью конфигурационных тестов проверяется совместимость продукта с различным программным (software) и аппаратным (hardware) обеспечением. Как правило, программный продукт делается с тем расчетом, чтобы он сразу работал в максимально разнообразной внешней среде. Если же речь идет о коробочном продукте, то фактор совместимости приобретает еще более важное значение. Для того чтобы выяснить реакцию продукта на окружение и соседство с другим программным обеспечением, и проводят данные тесты.

Тестирование на совместимость (Compatibility testing). Тестирование совместимости помогает убедиться в функциональных возможностях и надежности работы продукта в поддерживаемых браузерах (если речь идет о Веб-приложениях) и операционных системах.

Тестирование на удобство эксплуатации или практичности (*Usability testing*). Тестирование интерфейса человек/машина производится в отношении таких моментов, как внешний вид пользовательского интерфейса, удобство навигации (преимущественно для Веб-сайтов). Практичность – очень важная характеристика программного продукта. Например, программа может вполне соответствовать всем предъявляемым к ней требованиям с точки зрения функциональности. Но функции реализованы неудобно: например, некоторые шаги приходится повторять много раз, тогда как по логике достаточно выполнить однажды. В результате пользование программой утомляет пользователя. Для выявления такого рода «недочетов» и применяют тесты на удобство использования. Часто эта группа тестов относится к категории некритичных, но когда речь идет, например, о рыночном готовом продукте, пренебрегать удобством эксплуатации весьма опасно.

Перечисленные выше группы тестов – базовый набор, но далеко не полный. В зависимости от назначения системы испытаниям подвергаются различные аспекты ее функциональности, в соответствии с приоритетами задач, которые система должна решать.

Тестирование интернационализации (Internationalization testing). Этот вид тестирует насколько продукт готов к тому, чтобы быть адаптированным для работы в других локалях с другим языком пользовательского интервейса, отличным от языка по умолчанию (как правило, это английский). Например, мы сделали продукт для англоязычных пользователей. Но при этом, мы подготовились к тому, чтобы

быстро его адаптировать для японцев. То есть за короткий срок мы сможем перевести весь текст на экранах, учесть индивидуальные особенности данной страны (тип валюты, разделители чисел и дат, адреса, телефоны и т. д.).

Локализационное тестирование (Localization testing). Локализационное тестирование, в свою очередь, проверяет правильно ли локализован продукт. То есть переведен на другой язык и корректно работает с учетом национальных особенностией страны или региона, в котором будет продаваться и использоваться наш продукт.

Positive/Negative testing. Позитивное тестирование проверяет то, что приложение не делает того, что должно делать в соответствии с требованиями.

В свою очередь, негативное – проверяет, что программа делает что-то из того, что она НЕ должна делать.

Вернемся к нашему примеру про числовое поле. Ввод чисел от 1 до 99 (т. е. допустимых значений) – это позитивное тестирование. Если вдруг программа не будет принимать этих значений, значит она НЕ будет делать то, что должна.

Тесты из остальных трех эквивалентных классов будут негативными. С помощью всех них мы будем проверять, как реагирует программа на попытку ввести недопустимые или ошибочные данные, т. е. целенаправленно заставить ее дать сбой. Таким образом, мы пытаемся найти что-то, что программа делает из того, что не должна делать. Например, спокойно обрабатывает значение 100, хотя по-хорошему, должна выдать сообщение об ошибке.

Исследовательское тестирование (Exploratory testing). Очень полезно, когда мы хотим выяснить информацию об уже полностью или частично сделанном продукте. Мы просто запускаем его и начинаем в нем копаться, разбираясь что к чему. Этот вид тестирования позволяет одновременно делать несколько вещей: изучать продукт; изучать пути, по которым можно привести продукт к неправильной работе; изучить слабый места продукта, и, соответственно, сосредоточиться на их тестировании; собственно, одновременно можно находить и документировать ошибки, т. е. проводить тестирование; разработать новые тесты, которые можно было бы использовать в будущем в качестве регрессионных.

Отдельно можно перечислить такие виды тестирования, как:

– *тестирование безопасности* (Security testing);

– *тестирование производительности* (Performance testing);

– *нагрузочное тестирование* (Load testing);

– *стрессовое тестирование* (Stress testing).

Все эти виды используются для тестирования приложений в многопользовательской среде.

Рассмотрим наиболее распространенный способ тестирования – *тестирование на основе тест-кейсов* – формализованный подход, в котором тестирование производится на основе заранее подготовленных тест-кейсов, наборов тест-кейсов и иной документации.

Тест-кейс (тестовый случай) – набор тестовых данных, условий выполнения теста и последовательность действий тестировщика, а также ожидаемый результат, которые разрабатывается с целью проверки тех или иных аспектов работы программы.

В зависимости от инструмента управления тест-кейсами внешний вид их записи может немного отличаться, могут быть добавлены или убраны отдельные поля, но концепция остается неизменной.

Общий вид всей структуры тест-кейса представлен в табл. 2.1.

Рассмотрим каждый атрибут подробно.

Идентификатор (identifier) представляет собой уникальное значение, позволяющее однозначно отличить один тест-кейс от другого и используемое во всевозможных ссылках. В общем случае идентификатор тест-кейса может представлять собой просто уникальный номер, но (если позволяет инструментальное средство управления тест-кейсами) может быть и куда сложнее: включать префиксы, суффиксы и иные осмысленные компоненты, позволяющие быстро определить цель тест-кейса и часть приложения (или требований), к которой он относится (например: UR216_S12_DB_Neg).

Приоритет (priority) показывает важность тест-кейса. Он может быть выражен буквами (A, B, C, D, E), цифрами (1, 2, 3, 4, 5), словами («крайне высокий», «высокий», «средний», «низкий», «крайне низкий») или иным удобным способом. Количество градаций также не фиксировано, но чаще всего лежит в диапазоне от трех до пяти.

Таблица 2.1

Общий вид структуры тест-кейса

Идентификатор	**Приоритет**	**Связанное с тест-кейсом требование**	**Модуль приложения**	**Подмодуль приложения**	**Заглавие (суть) тест-кейса и его шаги**	**Ожидаемый результат по каждому шагу тест-кейса**
UG_U1.12	A	R97	Галерея	Загрузка файла	Галерея, загрузка файла, имя со спецсимволами	
					Приготовления: создать непустой файл с именем #$%^&.jpg	
					1. Нажать кнопку «Загрузить картинку»	1. Появляется окно загрузки картинки
					2. Нажать кнопку «Выбрать»	2. Появляется диалоговое окно браузера выбора файла для загрузки
					3. Выбрать из списка приготовленный файл	3. Имя выбранного файла появляется в поле «Файл»
					4. Нажать кнопку «OK»	4. Диалоговое окно файла закрывается, в поле «Файл» появляется полное имя файла
					5. Нажать кнопку «Добавить в галерею»	5. Выбранный файл появляется в списке файлов галереи

Приоритет тест-кейса может коррелировать:

– с важностью требования, пользовательского сценария или функции, с которыми связан тест-кейс;

– потенциальной важностью дефекта, на поиск которого направлен тест-кейс;

– степенью риска, связанного с проверяемым тест-кейсом требованием, сценарием или функцией.

Основная задача этого атрибута – упрощение распределения внимания и усилий команды (более высокоприоритетные тест-кейсы получают их больше), а также упрощение планирования и принятия решения о том, чем можно пожертвовать в некоей форс-мажорной ситуации, не позволяющей выполнить все запланированные тест-кейсы.

Связанное с тест-кейсом требование (requirement) показывает то основное требование, проверке выполнения которого посвящен тест-кейс (основное – потому, что один тест-кейс может затрагивать несколько требований). Наличие этого поля улучшает такое свойство тест-кейса, как прослеживаемость.

Частые вопросы, связанные с заполнением этого поля, таковы:

• Можно ли его оставить пустым? Да. Тест-кейс вполне мог разрабатываться вне прямой привязки к требованиям, и (пока?) значение этого поля определить сложно. Хоть такой вариант и не считается хорошим, он достаточно распространен.

• Можно ли в этом поле указывать несколько требований? Да, но чаще всего стараются выбрать одно самое главное или «более высокоуровневое» (например, вместо того, чтобы перечислять R56.1, R56.2, R56.3 и т. д., можно просто написать R56). Чаще всего в инструментах управления тестами это поле представляет собой выпадающий список, где можно выбрать только одно значение, и этот вопрос становится неактуальным. К тому же многие тест-кейсы все же направлены на проверку строго одного требования, и для них этот вопрос также неактуален.

Модуль и подмодуль приложения (module and submodule) указывают на части приложения, к которым относится тест-кейс, и позволяют лучше понять его цель.

Идея деления приложения на модули и подмодули проистекает из того, что в сложных системах практически невозможно охватить взглядом весь проект целиком, и вопрос «как протестировать это приложение» становится недопустимо сложным. Тогда приложение логически разделяется на компоненты (модули), а те, в свою очередь, на

более мелкие компоненты (подмодули). И вот уже для таких небольших частей приложения создать хорошие тест-кейсы становится намного проще.

Как правило, иерархия модулей и подмодулей создается как единый набор для всей проектной команды, чтобы исключить путаницу из-за того, что разные люди будут использовать разные подходы к такому разделению или даже просто разные названия одних и тех же частей приложения.

Теперь – самое сложное: как выбираются модули и подмодули. В реальности проще всего отталкиваться от архитектуры и дизайна приложения.

Но что делать, если мы не знаем «внутренностей» приложения (или не очень разбираемся в программировании)? Модули и подмодули можно выделять на основе графического интерфейса пользователя (крупные области и элементы внутри них), на основе решаемых приложением задач и подзадач и т. д. Главное, чтобы эта логика была одинаковым образом применена ко всему приложению. Наличие полей «Модуль» и «Подмодуль» улучшает такое свойство тест-кейса, как прослеживаемость.

Заглавие (*суть*) *тест-кейса* (*title*) призвано упростить и ускорить понимание основной идеи (цели) тест-кейса без обращения к его остальным атрибутам. Именно это поле является наиболее информативным при просмотре списка тест-кейсов.

Заглавие тест-кейса может быть полноценным предложением, фразой, набором словосочетаний – главное, чтобы выполнялись следующие условия:

– информативность;

– хотя бы относительная уникальность (чтобы не путать разные тест-кейсы).

Исходные данные, *необходимые для выполнения тест-кейса* (*precondition*, *preparation*, *initial data*, *setup*), позволяют описать все то, что должно быть подготовлено до начала выполнения тест-кейса, например:

– состояние базы данных;

– состояние файловой системы и ее объектов;

– состояние серверов и сетевой инфраструктуры.

Шаги тест-кейса (steps) описывают последовательность действий, которые необходимо реализовать в процессе выполнения тест-кейса. Общие рекомендации по написанию шагов таковы:

• Начинайте с понятного и очевидного места, не пишите лишних начальных шагов (запуск приложения, очевидные операции с интерфейсом и т. п.).

• Даже если в тест-кейсе всего один шаг, нумеруйте его (иначе возрастает вероятность в будущем случайно «приклеить» описание этого шага к новому тексту).

• Если вы пишете на русском языке, используйте безличную форму (например, «открыть», «ввести», «добавить» вместо «откройте», «введите», «добавьте»), в английском языке не надо использовать частицу «to» (т. е. «запустить приложение» будет «start application», не «to start application»).

• Соотносите степень детализации шагов и их параметров с целью тест-кейса, его сложностью, уровнем и т. д. – в зависимости от этих и многих других факторов степень детализации может варьироваться от общих идей до предельно четко прописанных значений и указаний.

• Ссылайтесь на предыдущие шаги и их диапазоны для сокращения объема текста (например, «повторить шаги 3–5 со значением…»).

• Пишите шаги последовательно, без условных конструкций вида «если… то…».

Ожидаемые результаты (expected results) по каждому шагу тест-кейса описывают реакцию приложения на действия, описанные в поле «шаги тест-кейса». Номер шага соответствует номеру результата.

По написанию ожидаемых результатов можно порекомендовать следующее:

• Описывайте поведение системы так, чтобы исключить субъективное толкование (например, «приложение работает верно» – плохо, «появляется окно с надписью…» – хорошо).

• Пишите ожидаемый результат по всем шагам без исключения, если у вас есть хоть малейшие сомнения в том, что результат некоего шага будет совершенно тривиальным и очевидным (если вы все же пропускаете ожидаемый результат для какого-то тривиального действия, лучше оставить в списке ожидаемых результатов пустую строку – это облегчает восприятие).

• Пишите кратко, но не в ущерб информативности.

• Избегайте условных конструкций вида «если… то…».

2.2. Тестирование качества тест-кейсов

Даже правильно оформленный тест-кейс может оказаться некачественным, если в нем нарушено одно из нижеприведенных свойств.

Правильный технический язык, точность и единообразие формулировок. Это свойство в равной мере относится и к требованиям, и к тест-кейсам, и к отчетам о дефектах – к любой документации. Из самого общего и важного напомним и добавим:

– пишите лаконично, но понятно;

– используйте безличную форму глаголов (например, «открыть» вместо «откройте»);

– обязательно указывайте точные имена и технически верные названия элементов приложения;

– не объясняйте базовые принципы работы с компьютером (предполагается, что ваши коллеги знают, что такое, например, «пункт меню» и как с ним работать);

• везде называйте одни и те же вещи одинаково (например, нельзя в одном тест-кейсе некий режим работы приложения назвать «графическое представление», а в другом тот же режим – «визуальное отображение», так как многие люди могут подумать, что речь идет о разных вещах);

– следуйте принятому на проекте стандарту оформления и написания тест-кейсов (иногда такие стандарты могут быть весьма жесткими: вплоть до регламентации того, названия каких элементов должны быть приведены в двойных кавычках, а каких – в одинарных).

Баланс между специфичностью и общностью. Тест-кейс считается тем более специфичным, чем более детально в нем расписаны конкретные действия, конкретные значения и так далее, т. е. чем в нем больше конкретики. Соответственно, тест-кейс считается тем более общим, чем в нем меньше конкретики. Рассмотрим поля «шаги» и «ожидаемые результаты» двух тест-кейсов (табл. 2.2 и 2.3).

Таблица 2.2

Конвертация из всех поддерживаемых кодировок

Шаги	Ожидаемые результаты
Приготовления: • Создать папки C:/A, C:/B, C:/C, C:/D. • Разместить в папке C:/D файлы 1.html, 2.txt, 3.md из прилагаемого архива. 1. Запустить приложение, выполнив команду «php converter.php c:/a c:/b c:/c/converter.log». 2. Скопировать файлы 1.html, 2.txt, 3.md из папки C:/D в папку C:/A. 3. Остановить приложение нажатием Crtl+C	1. Отображается консольный журнал приложения с сообщением «текущее_время started, source dir c:/a, destination dir c:/b, log file c:/c/converter.log», в папке C:/C появляется файл converter.log, в котором появляется запись «текущее_время started, source dir c:/a, destination dir c:/b, log file c:/c/converter.log». 2. Файлы 1.html, 2.txt, 3.md появляются в папке C:/A, затем пропадают оттуда и появляются в папке C:/B. В консольном журнале и файле C:/C/converter.log появляются сообщения (записи) «текущее_время processing 1.html (KOI8-R)», «текущее_время processing 2.txt (CP-1251)», «текущее_время processing 3.md (CP-866)». 3. В файле C:/C/converter.log появляется запись «текущее_время closed». Приложение завершает работу

Таблица 2.3

Конвертация из всех поддерживаемых кодировок

Шаги	Ожидаемые результаты
1. Выполнить конвертацию трех файлов допустимого размера в трех разных кодировках всех трех допустимых форматов	1. Файлы перемещаются в папку-приемник, кодировка всех файлов меняется на UTF-8

Если вернуться к вопросу «какой тест-кейс вы бы посчитали хорошим, а какой – плохим и почему», то ответ таков: оба тест-кейса плохие потому, что первый является слишком специфичным, а второй – слишком общим. Можно сказать, что здесь до абсурда доведены идеи низкоуровневых и высокоуровневых тест-кейсов.

Почему плоха излишняя специфичность (табл. 2.2):

• При повторных выполнениях тест-кейса всегда будут выполняться строго одни и те же действия со строго одними и теми же данными, что снижает вероятность обнаружения ошибки.

• Возрастает время написания, доработки и даже просто прочтения тест-кейса.

• В случае выполнения тривиальных действий опытные специалисты тратят дополнительные мыслительные ресурсы в попытках понять, что же они упустили из виду, так как они привыкли, что так описываются только самые сложные и неочевидные ситуации.

Почему плоха излишняя общность (табл. 2.3):

• Тест-кейс сложен для выполнения начинающими тестировщиками или даже опытными специалистами, лишь недавно подключившимися к проекту.

• Недобросовестные сотрудники склонны халатно относиться к таким тест-кейсам.

• Тестировщик, выполняющий тест-кейс, может понять его иначе, чем было задумано автором (в итоге будет выполнен фактически совсем другой тест-кейс).

Выход из этой ситуации состоит в том, чтобы придерживаться золотой середины (хотя, конечно же, какие-то тесты будут чуть более специфичными, какие-то – чуть более общими). Ниже приведен пример такого срединного подхода (табл. 2.4).

Таблица 2.4

Конвертация из всех поддерживаемых кодировок

Шаги	Ожидаемые результаты
Приготовления: • Создать в корне любого диска четыре отдельные папки для входных файлов, выходных файлов, файла журнала и временного хранения тестовых файлов. • Распаковать содержимое прилагаемого архива в папку для временного хранения тестовых файлов. 1. Запустить приложение, указав в параметрах соответствующие пути из приготовления к тесту (имя файла журнала – произвольное). 2. Скопировать файлы из папки для временного хранения в папку для входных файлов. 3. Остановить приложение	1. Приложение запускается и выводит сообщение о своем запуске в консоль и файл журнала. 2. Файлы из папки для входных файлов перемещаются в папку для выходных файлов, в консоли и файле журнала отображаются сообщения о конвертации каждого из файлов с указанием его исходной кодировки. 3. Приложение выводит сообщение о завершении работы в файл журнала и завершает работу

В этом тест-кейсе есть все необходимое для понимания и выполнения, но при этом он стал короче и проще для выполнения, а отсутствие строго указанных значений приводит к тому, что при многократном выполнении тест-кейса (особенно – разными тестировщиками) конкретные параметры будут менять свои значения, что увеличивает вероятность обнаружения ошибки.

Подчеркнем, что сами по себе специфичность или общность тест-кейса не являются чем-то плохим, но резкий перекос в ту или иную сторону снижает качество тест-кейса.

Баланс между простотой и сложностью. Здесь не существует академических определений, но принято считать, что простой тест-кейс оперирует одним объектом (или в нем явно виден главный объект), а также содержит небольшое количество тривиальных действий; сложный тест-кейс оперирует несколькими равноправными объектами и содержит много нетривиальных действий.

Преимущества простых тест-кейсов:

- Их можно быстро прочесть, легко понять и выполнить.
- Понятны начинающим тестировщикам и новым людям на проекте.
- Делают наличие ошибки очевидным (как правило, в них предполагается выполнение повседневных тривиальных действий, проблемы с которыми видны невооруженным взглядом и не вызывают дискуссий).
- Упрощают начальную диагностику ошибки, так как сужают круг поиска.

Преимущества сложных тест-кейсов:

- При взаимодействии многих объектов повышается вероятность возникновения ошибки.
- Пользователи, как правило, используют сложные сценарии, а потому сложные тесты более полноценно эмулируют работу пользователей.
- Программисты редко проверяют такие сложные случаи (и они совершенно не обязаны это делать).

В табл. 2.5 приведен пример слишком простого тест-кейса, а в табл. 2.6 – слишком сложного.

Таблица 2.5

Запуск приложения

Шаги	Ожидаемые результаты
1. Запустить приложение	1. Приложение запускается

Таблица 2.6

Повторная конвертация

Шаги	Ожидаемые результаты
Приготовления: • Создать в корне любого диска три отдельные папки для входных файлов, выходных файлов, файла журнала. • Подготовить набор из нескольких файлов максимального поддерживаемого размера поддерживаемых форматов с поддерживаемыми кодировками, а также нескольких файлов допустимого размера, но недопусти-мого формата. 1. Запустить приложение, указав в параметрах соответствующие пути из приготовления к тесту (имя файла журнала – произвольное). 2. Скопировать в папку для входных файлов несколько файлов допустимого формата. 3. Переместить сконвертированные файлы из папки с результирующими файлами в папку для входных файлов. 4. Переместить сконвертированные файлы из папки с результирующими файлами в папку с набором файлов для теста. 5. Переместить все файлы из папки с набором файлов для теста в папку для входных файлов. 6. Переместить сконвертированные файлы из папки с результирующими файлами в папку для входных файлов	2. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов. 3. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов. 5. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов допустимого формата и сообщения об игнорировании файлов недопустимого формата. 6. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов допустимого формата и сообщения об игнорировании файлов недопустимого формата

Этот тест-кейс одновременно является слишком сложным по избыточности действий и по спецификации лишних данных и операций.

Пример хорошего сложного тест-кейса представлен в табл. 2.7.

Таблица 2.7

Много копий приложения, конфликт файловых операций

Шаги	Ожидаемые результаты
Приготовления: • Создать в корне любого диска три отдельные папки для входных файлов, выходных файлов, файла журнала. • Подготовить набор из нескольких файлов максимального поддерживаемого размера поддерживаемых форматов с поддерживаемыми кодировками. 1. Запустить первую копию приложения, указав в параметрах соответствующие пути из приготовления к тесту (имя файла журнала – произвольное). 2. Запустить вторую копию приложе-ния с теми же параметрами (см. шаг 1). 3. Запустить третью копию приложения с теми же параметрами (см. шаг 1). 4. Изменить приоритет процессов второй («high») и третьей («low») копий. 5. Скопировать подготовленный набор исходных файлов в папку для входных файлов	3. Все три копии приложения запускаются, в файле журнала появляются последовательно три записи о запуске приложения. 5. Файлы постепенно перемещаются из входной в выходную папку, в консоли и файле журнала появляются сообщения об успешной конвертации файлов, а также (возможно) сообщения вида: a. «source file inaccessible, retrying». b. «destination file inaccessible, retrying». c. «log file inaccessible, retrying». Ключевым показателем корректной работы является успешная конвертация всех файлов, а также появление в консоли и файле журнала сообщений об успешной конвертации каждого файла (от одной до трех записей на каждый файл). Сообщения (предупреждения) о недоступности входного файла, выходного файла или файла журнала также являются показателем корректной работы приложения, однако их количество зависит от многих внешних факторов и не может быть спрогнозировано заранее

Иногда более сложные тест-кейсы являются также и более специфичными, но это лишь общая тенденция, а не закон. Также нельзя по сложности тест-кейса однозначно судить о его приоритете (в нашем примере хорошего сложного тесткейса он явно будет иметь очень низкий приоритет, так как проверяемая им ситуация является искусственной и крайне маловероятной, но бывают и сложные тесты с самым высоким приоритетом). Как и в случае специфичности и общности, сами по себе простота или сложность тест-кейсов не являются чем-то плохим (более того, рекомендуется начинать разработку и выполнение тест-кейсов с простых, а затем переходить ко все более и более сложным), однако излишняя простота и сложность также снижают качество тест-кейса.

«*Показательность*» (*высокая вероятность обнаружения ошибки*). Начиная с уровня тестирования критического пути, можно утверждать, что тест-кейс является тем более хорошим, чем он более показателен (с большей вероятностью обнаруживает ошибку). Именно поэтому мы считаем непригодными слишком простые тест-кейсы – они непоказательны.

Пример непоказательного (плохого) тест-кейса рассмотрен в табл. 2.8.

Таблица 2.8

Запуск и остановка приложения

Шаги	Ожидаемые результаты
1. Запустить приложение с корректными параметрами. 2. Завершить работу приложения	1. Приложение запускается. 2. Приложение завершает работу.

Пример показательного (хорошего) тест-кейса демонстрирует табл. 2.9.

Таблица 2.9

Запуск с некорректными параметрами, несуществующие пути

Шаги	Ожидаемые результаты
1. Запустить приложение со всеми тремя параметрами (SOURCE_DIR, DESTINATION_DIR, LOG_FILE_NAME), значения которых указывают на несуществующие в фай-ловой системе пути (например: z:\src\, z:\dst\, z:\log.txt при условии, что в системе нет логического диска z)	1. В консоли отображаются нижеуказанные сообщения, приложение завершает работу. Сообщения: a. Сообщение об использовании. b. SOURCE_DIR [z:\src\]: directory not exists or inaccessible. c. DESTINATION_DIR [z:\dst\]: directory not exists or inaccessible. d. LOG_FILE_NAME [z:\log.txt]: wrong file name or inaccessible path

Обратите внимание, что показательный тест-кейс по-прежнему остался достаточно простым, но он проверяет ситуацию, возникновение ошибки в которой несравненно более вероятно, чем в ситуации, описываемой плохим непоказательным тест-кейсом. Также можно отметить, что показательные тест-кейсы часто выполняют какието «интересные действия», т. е. такие действия, которые едва ли будут выполнены просто в процессе работы с приложением (например: «со-

хранить файл» — это обычное тривиальное действие, которое явно будет выполнено не одну сотню раз даже самими разработчиками, а вот «сохранить файл на носитель, защищенный от записи», «сохранить файл на носитель с недостаточным объемом свободного пространства», «сохранить файл в папку, к которой нет доступа» — это уже гораздо более интересные и нетривиальные действия).

Последовательность в достижении цели. Суть этого свойства выражается в том, что все действия в тест-кейсе направлены на следование единой логике и достижение единой цели и не содержат никаких отклонений. Примерами правильной реализации этого свойства могут служить представленные в этой главе в избытке варианты хороших тест-кейсов. А нарушение может выглядеть следующим образом (табл. 2.10).

Таблица 2.10

Конвертация из всех поддерживаемых кодировок

Шаги	Ожидаемые результаты
Приготовления: • Создать в корне любого диска четыре отдельные папки для входных файлов, выходных файлов, файла журнала и временного хранения тестовых файлов. • Распаковать содержимое прилагаемого архива в папку для временного хранения тестовых файлов. 1. Запустить приложение, указав в параметрах соответствующие пути из приготовления к тесту (имя файла журнала – произвольное). 2. Скопировать файлы из папки для временного хранения в папку для входных файлов. 3. *Остановить приложение*. 4. *Удалить файл журнала*. 5. *Повторно запустить приложение с теми же параметрами*. 6. Остановить приложение	1. Приложение запускается и выводит сообщение о своем запуске в консоль и файл журнала. 2. Файлы из папки для входных файлов перемещаются в папку для выходных файлов, в консоли и файле журнала отображаются сообщения о конвертации каждого из файлов с указанием его исходной кодировки. 3. *Приложение выводит сообщение о завершении работы в файл журнала и завершает работу.* 5. *Приложение запускается и выводит сообщение о своем запуске в консоль и заново созданный файл журнала.* 6. Приложение выводит сообщение о завершении работы в файл журнала и завершает работу

Шаги 3–5 никак не соответствуют цели тест-кейса, состоящей в проверке корректности конвертации входных данных, представленных во всех поддерживаемых кодировках.

Отсутствие лишних действий. Чаще всего это свойство подразумевает, что не нужно в шагах тест-кейса долго и по пунктам расписывать то, что можно заменить одной фразой (табл. 2.11).

Таблица 2.11

Плохо	Хорошо
1. Указать в качестве первого параметра приложения путь к папке с исходными файлами. 2. Указать в качестве второго параметра приложения путь к папке с конечными файлами. 3. Указать в качестве третьего параметра приложения путь к файлу журнала. 4. Запустить приложение	1. Запустить приложение со всеми тремя корректными параметрами (например, c:\src\, c:\dst\, c:\log.txt при условии, что соответствующие папки существуют и доступны приложению)

Вторая по частоте ошибка – начало каждого тест-кейса с запуска приложения и подробного описания по приведению его в то или иное состояние. В наших примерах мы рассматриваем каждый тест-кейс как существующий в единственном виде в изолированной среде, и потому вынуждены осознанно допускать эту ошибку (иначе тест-кейс будет неполным), но в реальной жизни на запуск приложения будут свои тесты, а длинный путь из многих действий можно описать как одно действие, из контекста которого понятно, как это действие выполнить.

Следующий пример тест-кейса не относится к нашему «Конвертеру файлов», но очень хорошо иллюстрирует эту мысль (табл. 2.12).

Таблица 2.12

Плохо	Хорошо
1. Запустить приложение. 2. Выбрать в меню пункт «Файл». 3. Выбрать подпункт «Открыть». 4. Перейти в папку, в которой находится хотя бы один файл формата DOCX с тремя и более страницами	1. Открыть DOCX-файл с тремя и более страницами

И сюда же можно отнести ошибку с повторением одних и тех же приготовлений во множестве тест-кейсов (да, по описанным выше причинам в примерах мы снова вынужденно делаем так, как в жизни

делать не надо). Удобнее объединить тесты в набор и указать приготовления один раз, подчеркнув, нужно или нет их выполнять перед каждым тест-кейсом в наборе.

Неизбыточность по отношению к другим тест-кейсам. В процессе создания множества тест-кейсов очень легко оказаться в ситуации, когда два и более тест-кейса фактически выполняют одни и те же проверки, преследуют одни и те же цели, направлены на поиск одних и тех же проблем. Если вы обнаруживаете несколько тест-кейсов, дублирующих задачи друг друга, лучше всего или удалить все, кроме одного, самого показательного, или перед удалением остальных на их основе доработать этот выбранный самый показательный тест-кейс. Ожидаемые результаты должны быть подобраны и сформулированы таким образом, чтобы любое отклонение от них сразу же бросалось в глаза и становилось очевидным, что произошла ошибка. Сравните выдержки из двух тест-кейсов (табл. 2.13 и 2.14).

Таблица 2.13

Выдержка из недемонстративного тест-кейса

Шаги	Ожидаемые результаты
5. Разместить в файле текст «Пример длинного текста, содержащего символы русского и английского алфавита вперемешку.» в кодировке KOI8-R (в слове «Пример» буквы «р» – английские). 6. Сохранить файл под именем «test.txt» и отправить файл на конвертацию. 7. Переименовать файл в «test.txt»	6. Приложение игнорирует файл. 7. Текст принимает корректный вид в кодировке UTF-8 с учетом английских букв

Таблица 2.14

Выдержка из демонстративного тест-кейса

Шаги	Ожидаемые результаты
5. Разместить в файле текст «Е╥╒═┼╥ ┕┼╖╨ ┕┴.» (Эти символы представляют собой словосочетание «Пример текста.» в кодировке KOI8-R, прочи-танной как CP866). 6. Отправить файл на конвертацию	6. Текст принимает вид: «Пример текста.» (кодировка UTF8)

В первом случае тест-кейс плох не только расплывчатостью формулировки «корректный вид в кодировке UTF-8 с учетом английских букв», там также очень легко допустить ошибки при выполнении:

– забыть сконвертировать вручную входной текст в KOI8-R;

– не заметить, что в первый раз расширение начинается с пробела;

– забыть заменить в слове «Пример» буквы «р» на английские;

– из-за расплывчатости формулировки ожидаемого результата принять ошибочное, но выглядящее правдоподобно поведение за верное.

Второй тест-кейс четко ориентирован на свою цель по проверке конвертации (не содержит странной проверки с игнорированием файла с неверным расширением) и описан так, что его выполнение не представляет никаких сложностей, а любое отклонение фактического результата от ожидаемого будет сразу же заметно.

Прослеживаемость. Из содержащейся в качественном тест-кейсе информации должно быть понятно, какую часть приложения, какие функции и какие требования он проверяет. Частично это свойство достигается через заполнение соответствующих полей тест-кейса («Ссылка на требование», «Модуль», «Подмодуль»), но и сама логика тест-кейса играет не последнюю роль, так как в случае серьезных нарушений этого свойства можно долго с удивлением смотреть, например, на какое требование ссылается тест-кейс, и пытаться понять, как же они между собой связаны.

Возможность повторного использования. Это свойство редко выполняется для низкоуровневых тест-кейсов, но при создании высокоуровневых тест-кейсов можно добиться таких формулировок, при которых:

– тест-кейс будет пригодным к использованию с различными настройками тестируемого приложения и в различных тестовых окружениях;

– тест-кейс практически без изменений можно будет использовать для тестирования аналогичной функциональности в других проектах или других областях приложения.

Примером тест-кейса, который тяжело использовать повторно, может являться практически любой тест-кейс с высокой специфичностью. Не самым идеальным, но очень наглядным примером тест-кейса, который может быть легко использован в разных проектах, может служить следующий тест-кейс (табл. 2.15).

Таблица 2.15

Шаги	Ожидаемые результаты
Запуск, все параметры некорректны 1. Запустить приложение, указав в качестве всех параметров заведомо некорректные значения	1. Приложение запускается, после чего выводит сообщение с описанием сути проблемы с каждым из параметров и завершает работу

Повторяемость. Тест-кейс должен быть сформулирован таким образом, чтобы при многократном повторении он показывал одинаковые результаты. Это свойство можно разделить на два подпункта:

- Во-первых, даже общие формулировки, допускающие разные варианты выполнения тест-кейса, должны очерчивать соответствующие явные границы (например: «ввести какое-нибудь число» – плохо, «ввести целое число в диапазоне от –273 до +500 включительно» – хорошо).
- Во-вторых, действия (шаги) тест-кейса по возможности не должны приводить к необратимым (или сложно обратимым) последствиям (например, удалению данных, нарушению конфигурации окружения и т. д.) – не стоит включать в тест-кейс такие «разрушительные действия», если они не продиктованы явным образом целью тест-кейса; если же цель тест-кейса обязывает нас к выполнению таких действий, в самом тест-кейсе должно быть описание действий по восстановлению исходного состояния приложения (данных, окружения).

Соответствие принятым шаблонам оформления и традициям. С шаблонами оформления, как правило, проблем не возникает: они строго определены имеющимся образцом или вообще экранной формой инструментального средства управления тест-кейсами. Что же касается традиций, то они отличаются даже в разных командах в рамках одной компании, и тут невозможно дать иного совета, кроме как «почитайте уже готовые тест-кейсы перед тем как писать свои». В данном случае обойдемся без отдельных примеров, так как аналогичные примеры приведены выше.

2.3. Поиск и документирование дефектов

Процесс функционального тестирования.

Рассмотрим более подробно каждую стадию итеративной модели (рис. 2.1), приведенной выше.

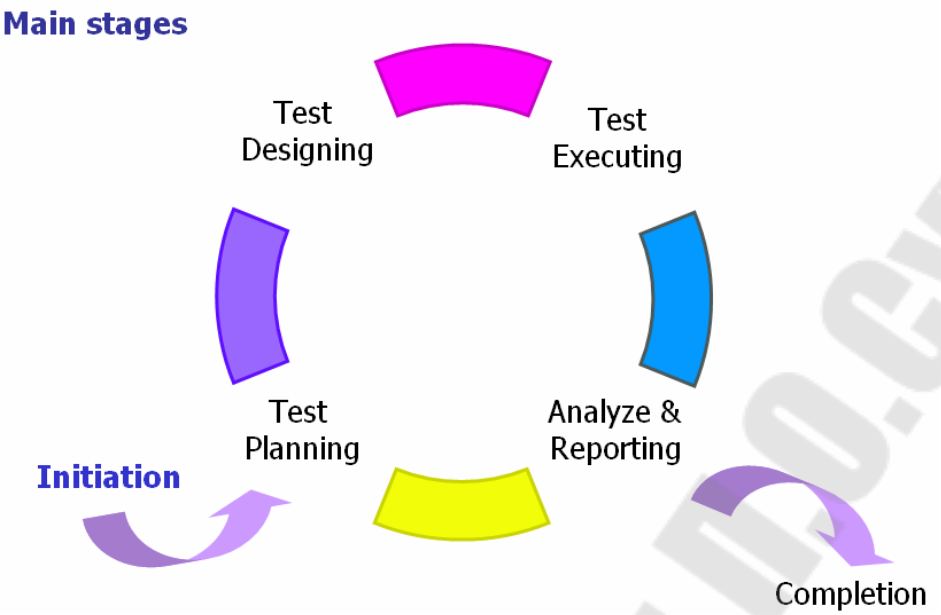

Рис. 2.1. Процесс функционального тестирования

Инициация. Процесс начинается с понимания и постановки целей и задач проекта, формирования команды тестировщиков.

После согласования команды назначенные специалисты по функциональному тестированию принимают участие во вступительном собрании (в случае их назначения в начале проекта) и приступают к анализу проектной документации, на основании которого ведущий специалист по тестированию (ВСТ) при необходимости разрабатывает рекомендации по корректировке документации для обеспечения возможности полноценного тестирования программного продукта и посылает их менеджеру проекта.

Планирование теста. Назначенный ведущий специалист по тестированию разрабатывает на основе анализа проектной документации тестовый план, согласовывает его с менеджером проекта и публикует в системе хранения документов, тем самым делает доступным для проектной команды и представителей Заказчика.

Разработка теста. Ведущий специалист по тестированию распределяет обязанности по тестированию программного продукта среди команды специалистов, выделенных для данного проекта, в соответствии с которыми все участники тестирования разрабатывают тестовые сценарии для закрепленных за ними областей тестирования программного продукта. Тестовые сценарии должны быть утверждены до начала тестирования программного продукта. Тестовые сценарии могут выполняться как вручную, так и с использованием средств

автоматического тестирования. Часто приемочный тест (или smoke test) может выполняться автоматическим способом. В зависимости от реализации процесса в конкретном проекте тестовые сценарии могут содержать следующие типы тестов:

– приемочный тест (smoke test);

– критический тест (critical path test);

– расширенный тест (extended test).

Выполнение теста. При получении сообщения о выпуске новой версии программного продукта ВСТ проверяет соответствие конфигурации версии программного продукта проектной документации, команда тестирования приступает к инсталляции продукта на выделенном тестовом оборудовании, уточняет конфигурационную информацию (проверяет номер версии ПП, наличие требуемых компонент и т. п.) и проводит приемочный тест. Приемочный и другие типы тестов выполняются, если они предусмотрены тестовым планом. Как минимум один вид тестов должен быть предусмотрен в тестовом плане.

В ходе приемочного и последующих типов тестов специалисты по тестированию проверяют соответствие исправленных дефектов, заявленных группой разработки программного продукта, их реальному состоянию и отображают их состояние в системе хранения отчетов об ошибках (Bugtracking tool). Кроме того, как и во время остальных видов тестирования, каждый обнаруженный в программном продукте дефект должен незамедлительно заносится в «систему багтрэкинга», откуда автоматически поступает уведомление о новом дефекте в группу разработки.

По результатам приемочного теста ведущий специалист по тестированию принимает решение о целесообразности дальнейшего тестирования. Если тест не пройден, то версия программного продукта (ПП) не прошла приемочный тест (smoke test failed) и процесс тестирования данной версии приостанавливается полностью или частично. При положительном решении посылается сообщение о прохождении теста (smoke test passed) и команда тестирования приступает к критическому и расширенному тестам, которые могут длиться продолжительное время (вплоть до нескольких рабочих дней и недель). Как правило, данный процесс продолжается до момента выпуска новой версии программного продукта.

Анализ и отчетность. Как правило, в конце каждой рабочей недели ВСТ (СТ в случае закрепления за ним отдельного приложения из

состава ПП) создает отчет о результатах тестирования версии (нескольких версий) программного продукта и опубликовывает в систему хранения документации или рассылает по электронной почте. В случае, когда в течение рабочей недели версии ПП не выпускались, допускается не создавать данный отчет, при этом вся статистическая информация за данный период должна быть включена в следующий отчет за общий период. Данные результаты обсуждаются на еженедельном общем собрании отдела функционального тестирования и (или) на совещании проектной команды. Кроме того, при необходимости, ВСТ корректирует тестовые план и сценарий, согласует вносимые изменения в описанном ранее порядке и публикует обновленные версии документов в систему хранения документации.

Ведущий специалист по тестированию принимает участие в регулярных, как правило, еженедельных, хотя менеджер проекта может установить иную периодичность, обсуждениях состояния проекта, а также в завершающем собрании (postmortem meeting) по проекту.

Завершение. После того как продукт оттестирован и группа тестирования рекомендует его к поставке заказчику, процесс тестирования завершается.

В зависимости от инструментального средства управления отчетами о дефектах внешний вид их записи может немного отличаться, могут быть добавлены или убраны отдельные поля, но концепция остается неизменной.

Общий вид всей структуры отчета о дефекте представлен в табл. 2.16.

Идентификатор (identifier) представляет собой уникальное значение, позволяющее однозначно отличить один отчет о дефекте от другого и используемое во всевозможных ссылках. В общем случае идентификатор отчета о дефекте может представлять собой просто уникальный номер, но (если позволяет инструментальное средство управления отчетами) может быть и куда сложнее: включать префиксы, суффиксы и иные осмысленные компоненты, позволяющие быстро определить суть дефекта и часть приложения (или требований), к которой он относится.

Краткое описание (summary) должно в предельно лаконичной форме давать исчерпывающий ответ на вопросы: «Что произошло?» «Где это произошло»? «При каких условиях это произошло?» Например: «Отсутствует логотип на странице приветствия, если пользователь является администратором».

Таблица 2.16

Структура отчета о дефекте

Идентификатор	Краткое описание	Подробное описание	Шаги по воспроизведению	Воспроизводимость	Важность	Срочность	Симптом	Возможность обойти	Комментарий	Приложения
19	Бесконечный цикл обработки входного файла с атрибутом «только для чтения»	Если у входного файла выставлен атрибут «только для чтения», после обработки приложению не удается переместить его в каталог-приемник: создается копия файла, но оригинал не удаляется, и приложение снова и снова обрабатывает этот файл и безуспешно пытается переместить его в каталог-приемник. **Ожидаемый результат:** после обработки файл перемещен из каталога-источника в каталог-приемник. **Фактический результат:** обработанный файл копируется в каталог-приемник, но его оригинал остается в каталоге-источнике. **Требование:** ДС-2.1	1. Поместить в каталог-источник файл допустимого типа и размера. 2. Установить данному файлу атрибут «только для чтения». 3. Запустить приложение. Дефект: обработанный файл появляется в каталоге-приемнике, но не удаляется из каталога-источника, файл в каталоге-приемнике непрерывно обновляется (видно по значению времени последнего изменения)	Всегда	Средняя	Обычная	Некорректная операция	Нет	Если заказчик не планирует использовать установку атрибута «только для чтения» файлам в каталоге-источнике для достижения своих целей, можно просто снимать этот атрибут и спокойно перемещать файл	

• Что произошло? Отсутствует логотип.

• Где это произошло? На странице приветствия.

• При каких условиях это произошло? Если пользователь является администратором.

Одной из самых больших проблем для начинающих тестировщиков является именно заполнение поля «краткое описание», которое одновременно должно:

– содержать предельно краткую, но в то же время достаточную для понимания сути проблемы информацию о дефекте;

– отвечать на вышеизложенные вопросы («что, где и при каких условиях случилось») или как минимум на те один-два вопроса, которые применимы к конкретной ситуации;

– быть достаточно коротким, чтобы полностью помещаться на экране (в тех системах управления отчетами о дефектах, где конец этого поля обрезается или приводит к появлению скроллинга);

– при необходимости содержать информацию об окружении, под которым был обнаружен дефект;

– по возможности не дублировать краткие описания других дефектов (и даже не быть похожими на них), чтобы дефекты было сложно перепутать или посчитать дубликатами друг друга;

– быть законченным предложением русского или английского (или иного) языка, построенным по соответствующим правилам грамматики.

Подробное описание (description) представляет в развернутом виде необходимую информацию о дефекте, а также (обязательно!) описание фактического результата, ожидаемого результата и ссылку на требование (если это возможно).

В отличие от краткого описания, которое, как правило, является одним предложением, здесь можно и нужно давать подробную информацию. Если одна и та же проблема (вызванная одним источником) проявляется в нескольких местах приложения, можно в подробном описании перечислить эти места.

Шаги по воспроизведению (steps to reproduce, STR) описывают действия, которые необходимо выполнить для воспроизведения дефекта. Это поле похоже на шаги тест-кейса, за исключением одного важного отличия: здесь действия прописываются максимально подробно, с указанием конкретных вводимых значений и самых мелких деталей, так как отсутствие этой информации в сложных случаях может привести к невозможности воспроизведения дефекта.

Воспроизводимость (reproducibility) показывает, при каждом ли прохождении по шагам воспроизведения дефекта удается вызвать его проявление. Это поле принимает всего два значения: всегда (always) или иногда (sometimes).

Можно отметить, что воспроизводимость «иногда» означает, что тестировщик не нашел настоящую причину возникновения дефекта. Это приводит к серьезным дополнительным сложностям в работе с дефектом.

Как легко догадаться, такая ситуация является крайне неприятной, а потому рекомендуется один раз потратить время на тщательную диагностику проблемы, найти ее причину и перевести дефект в разряд воспроизводимых всегда.

Важность (severity) показывает степень ущерба, который наносится проекту существованием дефекта. В общем случае выделяют следующие градации важности:

– *критическая* (critical) – существование дефекта приводит к масштабным последствиям катастрофического характера, например: потеря данных, раскрытие конфиденциальной информации, нарушение ключевой функциональности приложения и т. д.;

– *высокая* (major) – существование дефекта приносит ощутимые неудобства многим пользователям в рамках их типичной деятельности, например: недоступность вставки из буфера обмена, неработоспособность общепринятых клавиатурных комбинаций, необходимость перезапуска приложения при выполнении типичных сценариев работы;

– *средняя* (medium) – существование дефекта слабо влияет на типичные сценарии работы пользователей, и (или) существует обходной путь достижения цели, например: диалоговое окно не закрывается автоматически после нажатия кнопок «OK»/«Cancel», при распечатке нескольких документов подряд не сохраняется значение поля «Двусторонняя печать», перепутаны направления сортировок по некоему полю таблицы;

– *низкая* (minor) – существование дефекта редко обнаруживается незначительным процентом пользователей и (почти) не влияет на их работу, например: опечатка в глубоко вложенном пункте меню настроек, некое окно сразу при отображении расположено неудобно (нужно перетянуть его в удобное место), неточно отображается время до завершения операции копирования файлов.

Срочность (priority) показывает, как быстро дефект должен быть устранен. В общем случае выделяют следующие градации срочности:

– *наивысшая* (ASAP, as soon as possible) срочность указывает на необходимость устранить дефект настолько быстро, насколько это возможно. В зависимости от контекста «настолько быстро, насколько возможно» может варьироваться от «в ближайшем билде» до единиц минут;

– *высокая* (high) срочность означает, что дефект следует исправить вне очереди, так как его существование или уже объективно мешает работе, или начнет создавать такие помехи в самом ближайшем будущем;

– *обычная* (normal) срочность означает, что дефект следует исправить в порядке общей очередности. Такое значение срочности получает большинство дефектов;

– *низкая* (low) срочность означает, что в обозримом будущем исправление данного дефекта не окажет существенного влияния на повышение качества продукта.

Несколько дополнительных рассуждений о важности и срочности целесообразно рассмотреть отдельно.

Один из самых частых вопросов относится к тому, какая между ними связь. Никакой. Для лучшего понимания этого факта можно сравнить важность и срочность с координатами *X* и *Y* точки на плоскости. Хоть «на бытовом уровне» и кажется, что дефект с высокой важностью следует исправить в первую очередь, в реальности ситуация может выглядеть совсем иначе.

Чтобы проиллюстрировать эту мысль подробнее, вернемся к перечню градаций: заметили ли вы, что для разных степеней важности примеры приведены, а для разных степеней срочности – нет? И это не случайно.

Зная суть проекта и суть дефекта, его важность определить достаточно легко, так как мы можем проследить влияние дефекта на критерии качества, степень выполнения требований той или иной важности и т. д. Но срочность исправления дефекта можно определить только в конкретной ситуации.

Симптом (symptom) – позволяет классифицировать дефекты по их типичному проявлению. Не существует никакого общепринятого списка симптомов. Более того, далеко не в каждом инструментальном средстве управления отчетами о дефектах есть такое поле, а там, где оно есть, его можно настроить. В качестве примера рассмотрим следующие значения симптомов дефекта:

– *косметический дефект* (cosmetic flaw) – визуально заметный недостаток интерфейса, не влияющий на функциональность приложения (например, надпись на кнопке выполнена шрифтом не той гарнитуры);

– *повреждение/потеря данных* (data corruption/loss) – в результате возникновения дефекта искажаются, уничтожаются (или не сохраняются) некоторые данные (например, при копировании файлов копии оказываются поврежденными);

– *проблема в документации* (documentation issue) – дефект относится не к приложению, а к документации (например, отсутствует раздел руководства по эксплуатации);

– *некорректная операция* (incorrect operation) – некоторая операция выполняется некорректно (например, калькулятор показывает ответ 17 при умножении 2 на 3);

– *проблема инсталляции* (installation problem) – дефект проявляется на стадии установки и (или) конфигурирования приложения;

– *ошибка локализации* (localization issue) – что-то в приложении не переведено или переведено неверно на выбранный язык интерфейса;

– *нереализованная функциональность* (missing feature) – некая функция приложения не выполняется или не может быть вызвана (например, в списке форматов для экспорта документа отсутствует несколько пунктов, которые там должны быть);

– *проблема масштабируемости* (scalability) – при увеличении количества доступных приложению ресурсов не происходит ожидаемого прироста производительности приложения;

– *низкая производительность* (low performance) – выполнение неких операций занимает недопустимо большое время;

– *крах системы* (system crash) – приложение прекращает работу или теряет способность выполнять свои ключевые функции (также может сопровождаться крахом операционной системы, веб-сервера и т. д.);

– *неожиданное поведение* (unexpected behavior) – в процессе выполнения некоторой типичной операции приложение ведет себя необычным (отличным от общепринятого) образом (например, после добавления в список новой записи активной становится не новая запись, а первая в списке);

– *недружественное поведение* (unfriendly behavior) – поведение приложения создает пользователю неудобства в работе (например, на разных диалоговых окнах в разном порядке расположены кнопки «OK» и «Cancel»);

– *расхождение с требованиями* (variance from specs) – этот симптом указывают, если дефект сложно соотнести с другими симптомами, но тем не менее приложение ведет себя не так, как описано в требованиях;

– *предложение по улучшению* (enhancement) – во многих инструментальных средствах управления отчетами о дефектах для этого случая есть отдельный вид отчета, так как предложение по улучшению формально нельзя считать дефектом: приложение ведет себя согласно требованиям, но у тестировщика есть обоснованное мнение о том, как ту или иную функциональность можно улучшить.

Часто встречается вопрос о том, может ли у одного дефекта быть сразу несколько симптомов. Да, может. Например, крах системы очень часто ведет к потере или повреждению данных. Но в большинстве инструментальных средств управления отчетами о дефектах значение поля «Симптом» выбирается из списка, и потому нет возможности указать два и более симптома одного дефекта. В такой ситуации рекомендуется выбирать либо симптом, который лучше всего описывает суть ситуации, либо «наиболее опасный» симптом (например, недружественное поведение, состоящее в том, что приложение не запрашивает подтверждения перезаписи существующего файла, приводит к потере данных; здесь «потеря данных» куда уместнее, чем «недружественное поведение»).

Возможность обойти (workaround) – показывает, существует ли альтернативная последовательность действий, выполнение которой позволило бы пользователю достичь поставленной цели (например, клавиатурная комбинация Ctrl + P не работает, но распечатать документ можно, выбрав соответствующие пункты в меню). В некоторых инструментальных средствах управления отчетами о дефектах это поле может просто принимать значения «Да» и «Нет», в некоторых при выборе «Да» появляется возможность описать обходной путь. Традиционно считается, что дефектам без возможности обхода стоит повысить срочность исправления.

Комментарий (comments, additional info) – может содержать любые полезные для понимания и исправления дефекта данные. Иными словами, сюда можно писать все то, что нельзя писать в остальные поля.

Приложение (attachments) – представляет собой не столько поле, сколько список прикрепленных к отчету о дефекте приложений (копий экрана, вызывающих сбой файлов и т. д.).

Общие рекомендации по формированию приложений таковы:

- Если вы сомневаетесь, делать или не делать приложение, лучше сделайте.
- Обязательно прилагайте так называемые «проблемные арте-

факты» (например, файлы, которые приложение обрабатывает некорректно).

Если вы прилагаете видеоролик с записью происходящего на экране, обязательно оставляйте только тот фрагмент, который относится к описываемому дефекту (это будет буквально несколько секунд или минут из возможных многих часов записи). Старайтесь подобрать настройки кодеков так, чтобы получить минимальный размер ролика при сохранении достаточного качества изображения.

2.4. Жизненный цикл дефекта

Отчет о дефекте (и сам дефект вместе с ним) проходит определенные стадии жизненного цикла, которые схематично показаны на рис. 2.2.

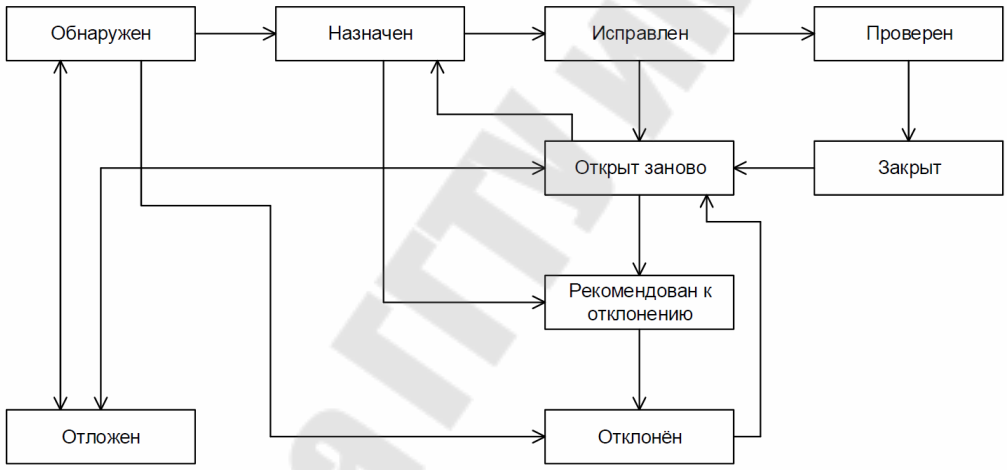

Рис. 2.2. Жизненный цикл дефекта

На рис. 2.2 приводятся следующие понятия:

• *Обнаружен* (submitted) – начальное состояние отчета (иногда называется «Новый» (new)), в котором он находится сразу после создания. Некоторые средства также позволяют сначала создавать черновик (draft) и лишь потом публиковать отчет.

• *Назначен* (assigned) – в это состояние отчет переходит с момента, когда кто-то из проектной команды назначается ответственным за исправление дефекта. Назначение ответственного производится или решением лидера команды разработки, или коллегиально, или по добровольному принципу, или иным принятым в команде способом или выполняется автоматически на основе определенных правил.

• *Исправлен* (fixed) – в это состояние отчет переводит ответственный за исправление дефекта член команды после выполнения соответствующих действий по исправлению.

• *Проверен* (verified) – в это состояние отчет переводит тестировщик, удосто-верившийся, что дефект на самом деле был устранен. Как правило, такую проверку выполняет тестировщик, изначально написавший отчет о дефекте.

• *Закрыт* (closed) – состояние отчета, означающее, что по данному дефекту не планируется никаких дальнейших действий. Здесь есть некоторые расхождения в жизненном цикле, принятом в разных инструментальных средствах управления отчетами о дефектах

• *Открыт заново* (reopened) – в это состояние (как правило, из состояния «Исправлен») отчет переводит тестировщик, удостоверившийся, что дефект по-прежнему воспроизводится на билде, в котором он уже должен быть исправлен.

• *Рекомендован к отклонению* (to be declined) – в это состояние отчет о дефекте может быть переведен из множества других состояний с целью вынести на рассмотрение вопрос об отклонении отчета по той или иной причине. Если рекомендация является обоснованной, отчет переводится в состояние «Отклонен» (см. следующий пункт).

• *Отклонен* (declined) – в это состояние отчет переводится в случаях, подробно описанных в пункте «Закрыт», если средство управления отчетами о дефектах предполагает использование этого состояния вместо состояния «Закрыт» для тех или иных резолюций по отчету.

• *Отложен* (deferred) – в это состояние отчет переводится в случае, если исправление дефекта в ближайшее время является нерациональным или не представляется возможным, однако есть основания полагать, что в обозримом будущем ситуация исправится (выйдет новая версия библиотеки, вернется из отпуска специалист по некоей технологии, изменятся требования заказчика и т. д.).

2.5. Особенности тестирования Веб-приложений

Веб-приложение – это клиент-серверное приложение, в котором клиентом выступает браузер, а сервером – веб-сервер (в широком смысле). Основная часть приложения, как правило, находится на стороне веб-сервера, который обрабатывает полученные запросы в соответствии с бизнес-логикой продукта и формирует ответ, отправляе-

мый пользователю. На этом этапе в работу включается браузер, именно он преобразовывает полученный ответ от сервера в графический интерфейс, понятный рядовому пользователю.

Итак, первой и одной из ключевых особенностей веб-приложений является их архитектура. Давайте более детально рассмотрим этот вопрос, так как он представляет особую ценность для тестирования.

Веб-приложение представлено следующими составляющими («сторонами»):

1. Клиент. Как правило, клиент – это браузер, но встречаются и исключения (в тех случаях, когда один веб-сервер (ВС1) выполняет запрос к другому (ВС2), роль клиента играет веб-сервер ВС1). В классической ситуации (когда роль клиента выполняет браузер) для того, чтобы пользователь увидел графический интерфейс приложения в окне браузера, последний должен обработать полученный ответ веб-сервера, в котором будет содержаться информация, реализованная с применением HTML, CSS, JS (самые используемые технологии). Именно эти технологии «дают понять» браузеру, как именно необходимо «отрисовать» все, что он получил в ответе.

2. Сервер. Веб-сервер – это сервер, принимающий HTTP-запросы от клиентов и выдающий им HTTP-ответы. Для того чтобы избежать возможной путаницы, отметим, что веб-сервером называют как программное обеспечение, выполняющее функции веб-сервера, так и непосредственно компьютер, на котором это программное обеспечение работает. Наиболее распространенными видами ПО веб-серверов являются Apache, IIS и NGINX. На веб-сервере функционирует тестируемое приложение, которое может быть реализовано с применением самых разнообразных языков программирования: PHP, Python, Ruby, Java, Perl и пр.

3. База данных. В классической теории речь идет о двух «сторонах» веб-приложения, однако, если внимательно посмотреть на весь процесс работы приложений, мы можем отметить, что в алгоритме работы веба незримо, но довольно активно принимает участие еще одна «сторона» – база данных. Фактически она не является частью веб-сервера, но большинство приложений просто не могут выполнять все возложенные на них функции без нее, так как именно в базе данных хранится вся динамическая информация приложения (учетные, пользовательские данные и пр.).

База данных – довольно широкое понятие, которое используется не только в сфере информационных технологий. В этом контексте ба-

за данных – это информационная модель, позволяющая упорядоченно хранить данные об объекте или группе объектов, обладающих набором свойств, которые можно категоризировать. Базы данных функционируют под управлением так называемых систем управления базами данных (далее – СУБД). Самыми популярными СУБД являются MySQL, MS SQL Server, PostgreSQL, Oracle (все – клиент-серверные).

Также существуют встраиваемые и файл-серверные СУБД. Например, одна популярная встраиваемая СУБД – SQLite, которая используется в некоторых браузерах, Android API, Skype и других известных приложениях. Взаимодействие с перечисленными СУБД основано на специальном языке структурированных запросов – SQL.

Теперь, собрав в голове определенный архитектурный пазл, предлагаем рассмотреть его с точки зрения тестирования ПО. Несколько позже мы рассмотрим и то, как все составляющие «общаются» между собой.

Особенности архитектуры клинтского приложения

Мы указали клиента как первую составляющую архитектуры. В классической ситуации клиент представлен браузером, а потому вопрос тестирования кроссбраузерности (ввиду многообразия браузеров) весьма актуален. Мы также рассмотрим тестирование заполняемых форм и текста как основного источника информации, получаемой через клиента.

Кроссбраузерность: разнообразие клиентов

Что мы понимаем под тестированием на кроссбраузерность? Это проверка на правильность (соответствие требованиям и стандартам) отображения и функционирования веб-приложения в разных браузерах и на разных операционных системах. В современном мире стандартизация принимает глобальные масштабы, а потому большинство популярных браузеров одинаково обрабатывают код. При этом необходимость в кроссбраузерном тестировании не исчезает, так как далеко не все проблемы решаются стандартизацией.

Перед началом работ я рекомендую выяснять у ответственных лиц необходимый для тестируемого ПО набор браузеров. Можно самим собрать статистику по целевой аудитории вашего продукта и ограничить кроссбраузерное тестирование набором наиболее популярных в этой аудитории браузеров. В условиях регрессионного тестирования, когда ограничены временные и человеческие ресурсы, также можно применять практику распределения браузеров: закрепляя за каждым тестировщиком определенный браузер, вы сможете

«покрыть» большой процент кроссбраузерных дефектов. Этот метод имеет и свои минусы: он наиболее результативен в большой команде тестирования, но становится практически неприменимым при наличии одного тестировщика.

При кроссбраузерном тестировании необходимо проверять:

– функциональные возможности продукта, реализуемые на стороне клиента;

– правильность отображения элементов графики;

– шрифты и размеры текстовых символов;

– доступность и функциональность разнообразных форм, включая их интерактивность.

В тестировании нуждаются все основные (среди пользовательской аудитории) браузеры, но особое внимание необходимо уделить IE, если он входит в их число. Именно в нем очень часто возникают проблемы, которые отсутствуют в других браузерах: так, в IE дополнительно рекомендуется обращать особое внимание на масштабируемость, фокус полей и работу JS.

Отдельно рекомендую не забывать о всякого рода валидаторах верстки, например, https://validator.w3.org/. Даже если у вас недостаточно знаний, чтобы оценить соответствие верстки стандартам, можно использовать для этого автоматические средства и, проанализировав результат, указать разработчикам на самые серьезные «оплошности». Не стоит забывать, что иногда валидаторы обращают внимание на самые «мелочные мелочи», которые никто и никогда исправлять не будет. Если вы и заводите баг-репорты на подобного рода замечания, то удобнее будет собрать их в единый документ и прикрепить к репорту. К такого рода «мелочам» можно отнести всевозможные рекомендации, которые не имеют своего влияния на отображение и функционирование контента.

Веб-формы на стороне клиента

Одной из важных составляющих интернет-приложений являются формы для заполнения, взаимодействие с которыми пользователь осуществляет с помощью все того же пристально рассмотренного нами клиента. Однако данные формы очень часто служат источником дефектов, которые, обосновавшись в «продакшене», могут принести большие финансовые и репутационные убытки компании.

Как же не пропустить дефекты в формах на продакшен? Рассмотрим несколько простых шагов:

1. Тщательно проверяем обязательность заполнения полей и наличие соответствующей маркировки у них.

2. Заполнив и отправив форму, убеждаемся в том, что с данными происходит именно то, что запланировано. Если данные должны быть внесены в базу данных, проверяем, корректно ли завершился процесс (в конце концов, об этом можно попросить разработчика, если не хватает своих знаний SQL или прав доступа к БД).

3. Используем чит-листы для тестирования форм, например, чит-лист регистрации от Алексея Лупана или чит-лист по Web UI контролам от Игоря Любина.

4. Проверяем, выводятся ли понятные пользователю информационные сообщения о необходимости заполнения пустых полей после попытки отправить форму.

5. Обращаем пристальное внимание на реализацию экранирования символов в полях форм, являющихся потенциальным источником уязвимостей для приложения и пользователей. Экранирование должно осуществляться на уровне не только клиента, но и сервера, отключить который в клиенте довольно просто (например, с помощью специальных плагинов, снимающих все возможные ограничения в несколько кликов, таких как Web Developer Toolbar – Forms).

6. Убеждаемся, что после заполнения формы пользователю приходит подтверждающее письмо с указанием соответствующего отправителя, а само тело письма соответствует требованиям (в том числе и на работоспособность ссылок).

7. Используем вспомогательные специальные инструменты для тестирования форм (например, Web Developer Toolbar).

Текст как основной источник информации при работе через клиента

Как не крути, но особая ценность сети Интернет заключается в том, что она является практически безграничным источником информации. Часть этой информации представлена в виде текстов, с которыми, опять же, пользователь взаимодействует посредством клиента. Большинство веб-ресурсов в том или ином объеме требуют проверки текстов на предмет отсутствия грамматических ошибок и опечаток.

Конечно, значимость этого тестирования не так велика по сравнению с функциональным направлением, но пренебрегать им не стоит. На практике довелось встретиться с очень серьезной опечаткой: на продакшене одного из крупнейших интернет-гипермаркетов России была допущена ошибка, изменившая гордое название величественного города Ярославль на прискорбное оскорбительное слово. Название служило частью суб-меню, а потому было заметно всем жителям города с полумиллионным населением.

Да, мы не всегда имеем достаточно времени для вычитывания всех текстов, но в таких ситуациях на помощь приходят «SpellChekerы» (программы для проверки орфографии, онлайн или в виде плагинов для браузеров), например, Яндекс. Спеллер.

Веб-сервер: «долой клиент, тестим без него»

Тестирование части веб-приложения, размещенной на веб-сервере, можно провести и минуя графический (клиентский) интерфейс, однако это требует от специалиста определенного уровня знаний и навыков технического характера, а также применения дополнительных инструментов. Рассмотрим веб-сервер с точки зрения нагрузочного и инсталляционного тестирования.

Инсталляция на веб-сервер

Итак, прежде чем приступить к тестированию, мы должны установить (инсталлировать) веб-приложение на веб-сервер. Собственно, в этом есть сходство с проверкой десктоп-приложений, но существует и различие в нюансах, которые необходимо учесть и протестировать, особенно если это касается ПО, распространяемого для локальной инсталляции на веб-серверы пользователей.

В чем же заключаются эти нюансы?

1. Большая часть веб-приложений требует для инсталляции специфических знаний в администрировании ОС. Попробуйте установить приложение на нескольких веб-серверах. В оптимальном случае это будут самые популярные технологии среди ваших пользователей, для установления списка которых потребуется предварительное исследование.

2. Инсталлируя веб-приложение, обращайте внимание на то, действительно ли приложение устанавливается в указанную вами директорию, базу данных, использует выбранный вами префикс и соблюдает прочие конфигурационные моменты.

3. Убедитесь, что приложение можно установить как из localhost, так и удаленно.

4. Если процесс инсталляции является интерактивным, и каждый выбор пользователя на определенном шаге влияет на последующие действия, то необходимо будет пройти все ветви, так как именно инсталляция является первым шагом пользователя в работе с вашим приложением.

5. Не забывайте о негативных тестах: проверки в условиях недостаточности ресурсов (места на диске, оперативной памяти) или привилегий, прерывание процесса установки во время активной его фазы (например, отправляя сервер в перезагрузку).

Нагрузочное тестирование

Нагрузочное тестирование имитирует работу с приложением определенного количества пользователей. Этот вид тестирования осуществляется при помощи специальных инструментов (например, jMeter), главная цель которых – определить профили нагрузки и искусственно создать для них нагрузку, выявляющую граничные возможности приложения (или сервера) в условиях работы с ним того или иного количества пользователей.

Полученная информация подвергается тщательному анализу с последующим выявлением «узких горлышек» и граничных программных и аппаратных возможностей, которые в дальнейшем используются для обеспечения стабильности веб-сервера и самого приложения, работающего на нем.

Приведем пример из практики функционирования большого коммерческого продукта, который долгое время работал с разнообразными типами договоров. Однажды в релиз выпустили очередной особо ожидаемый тип договора, и на следующий день система полностью перестала работать, а служба поддержки была завалена огромным количеством обращений. Всему виной стал просчет в тестировании: команда проверяла одновременную работу с десятками тысяч договоров, но никто не смог предугадать, что на практике речь пойдет о сотнях тысяч, а иногда и о миллионах договоров. Нагрузочное тестирование позволяет обнаружить потенциальные проблемы такого характера еще на этапе тестирования.

База данных: «хранить нельзя удалить»

Еще одной ранее рассмотренной составляющей веб-приложений является база данных, в которой приложение хранит всю необходимую информацию. Для того чтобы база данных служила достойным хранилищем информации для вашего приложения, при тестировании необходимо обращать внимание на следующие основные моменты:

1. Вносимая через интерфейс информация должна быть сохранена в базе данных в неизменном (первоначальном) виде.

2. Сохраненная в базе данных информация должна отображаться в любой части приложения одинаково (если иного не требует бизнес-логика приложения).

3. Названия таблиц и структура базы данных должны соответствовать проектной документации.

4. Нужно следить за тем, чтобы запросы не обрабатывались слишком долго, а количество соединений было достаточным. Мониторинг состояния базы данных – один из важных моментов тестирования.

5. Стоит учитывать, что удаление записи в базе данных не всегда сопровождается полным удалением сущности. Иногда используется так называемое «псевдоудаление», и нужно проверить, правильно ли оно выполняется.

6. Пустую базу данных на тестовом стенде рекомендуется либо заполнять сгенерированными случайными данными, либо снимать дамп с продакшена и после обфускации данных «заливать» их в тестируемую базу данных. Иногда квалификация тестировщика (или иная причина) не позволяет выполнить этот процесс самостоятельно: в таком случае рекомендуется обратиться за помощью к специалистам инфраструктуры или разработчикам.

Запросы: do you speak computish?

Все составляющие веб-приложения должны взаимодействовать между собой, и происходит это благодаря HTTP(s). Без HTTP наша многосторонняя система не функционировала бы в принципе, так как HTTP – это протокол передачи данных, занимающий одно из основных мест в нашей клиент-серверной архитектуре.

Взаимодействие осуществляется через сообщения (запросы и ответы): на отправленный запрос от клиента должен прийти ответ сервера. Классический запрос/ответ состоит из трех составляющих:

– стартовая строка;

– заголовок;

– тело сообщения.

При работе с ответами специалист по тестированию в первую очередь должен обращать внимание на методы и коды состояния, которые присутствуют в стартовой строке.

Самыми популярными методами являются GET, HEAD и POST.

Метод GET. Используется для запроса содержимого, размещенного на сервере (например, GET /path/resource?param1=value1 &param2=value2 HTTP/1.1).

Метод HEAD. По своей сути не отличается от вышеуказанного метода, однако ответ сервера на такой запрос лишен «тела», а практическое применение ориентировано на облегченное использование с целью получения минимальной информации о сервере (продукте) или его статусе.

Метод POST. Данный метод используется для передачи данных на сервер, однако его основа «прячется» в тело, что отличает его от GET. Во время публикации этой статьи, например весь текст будет помещен в тело POST-запроса; после обработки его сервером на сайт будет добавлена статья.

Существуют и другие методы: PUT, DELETE, CONNECT, TRACE, PATCH и т. д.

То, что составляющие веб-приложения взаимодействуют между собой посредством HTTP, является хорошей новостью для специалиста по тестированию, так как это позволяет не просто отслеживать общение, вникая в логику работы и сверяя ее с техническими требованиями, но и влиять непосредственно на приложение, прикладывая свою руку к отправляемым запросам.

Классическими приложениями, которые можно использовать для генерации запросов, является Fiddler или Postman. Используя Fiddler, можно с легкостью отслеживать все запросы от клиента и ответы, просматривать их детали, а также вносить свои изменения и отправлять модифицированные запросы на сервер, оценивая поведение системы в таком случае.

На что следует обращать внимание в запросах в первую очередь:

1. Правильный ли метод используется для того или иного запроса? Если вы отправляете сообщение, а на сервер уходит только GET-запрос, то что-то здесь явно не так.

2. Казалось бы, клик по одной и той же функциональной клавише генерирует один и тот же запрос. Но есть прецеденты, когда клик по кнопке приводил к ситуации, в которой JavaScript переписывал запрос рядом находящейся кнопки, таким образом меняя ее предназначение.

3. Вникайте в отправляемые запросы. В них довольно легко разобраться, особенно если это GET – они логически понятны даже рядовому пользователю. Анализ запросов – это возможность обнаружить спрятавшийся дефект гораздо быстрее, чем осуществляя его поиск в интерфейсе.

4. Мониторьте трафик на предмет запросов на другие (не ваши) сервера. Пример из жизни: фронтэнд сайта делал фриланс-разработчик, в завершение работы которого мы принялись за тестирование. Я имею привычку мониторить весь трафик тестируемого приложения: это позволило обнаружить, что вышеуказанный разработчик без зазрения совести спрятал в «фронт» сайта запросы, которые работали на благо его личного интернет-магазина.

5. Мониторя трафик, внимательно следите за кодами состояний. Мы подробнее остановимся на этом пункте.

Коды состояния глазами адепта качества

Каждый ответ на любой запрос несет в себе массу полезной для разработчиков и специалистов по тестированию ПО информации. Одним из источников такой информации может стать код состояния:

по этому коду клиент узнает о результатах его запроса и определяет, какие действия ему предпринять дальше. Набор кодов состояния является стандартом, он описан в соответствующих документах RFC. Каждый раз, когда вы создаете баг-репорт о неработающих ссылках, иных элементах веб-страницы или системных ошибках, обязательно указывайте ответы сервера: они сэкономят время разработчику при определении причин дефекта.

Все коды можно поделить на группы (сотые, двухсотые, трехсотые, четырехсотые и пятисотые) каждая группа-«сотня» несет свой тип информации.

Более детально с кодами состояния можно ознакомиться на Wikipedia.

На практике, используя при тестировании специальные приложения (тот же Fiddler), вы без труда сможете отсортировать свои запросы и ответы по коду состояния и отобрать, например, все 400-е и 500-е с последующим их анализом. Таким образом очень быстро «отлавливаются» дефекты с «отвалившимися» стилями, скриптами, файлами, функциями приложения и т. п.

Чем еще отличается веб-приложение от десктопного: больше особенностей – больше проблем!

Многопользовательская сущность веб-приложений

Широта аудитории приложений накладывает свой отпечаток на специфику работы.

1. Одно приложение одновременно может использоваться огромным количеством людей. Мы уже рассматривали вопрос нагрузочного тестирования, но также следует обратить внимание на то, что в число пользователей могут входить представители разных культур, языков и религий. Нам необходимо помнить об этом, особенно если речь идет о тестировании международного приложения.

2. Каждый пользователь может иметь свои уровни доступа. В идеальном варианте тестировщик создает для себя матрицу уровней доступа и тестирует каждый доступ в отдельности.

3. Пользователи с одним уровнем доступа могут обращаться к одним и тем же сущностям, что приводит к конкурентному доступу. Тестируется это довольно просто. Для примера рассмотрим систему, имеющую дело с договорами, которые можно создавать, публиковать, редактировать, аннулировать. Алгоритм работы таков: под несколькими окнами в режиме инкогнито авторизуемся в приложении под пользователями с разными уровнями доступа; далее выбранную для теста сущность открываем на редактирование, а под второй учетной

записью эту же сущность пробуем перевести в статус «Аннулировано» – на этом этапе должен сработать контроль на конкурентный доступ. Операция аннулирования блокируется, а пользователю выдается сообщение о том, что сущность редактируется другим пользователем (поведение и приоритет действий определяются в соответствии с требованиями и особенностями продукта, но логика не меняется).

4. Широта аудитории говорит о том, что за монитором может находиться человек, имеющий злой умысел в отношении вашего ПО.

Сетевые страсти: веб-приложение в разных условиях передачи данных

Веб-приложения активно используют сеть, и это является источником возможных проблем. Таковой, например, является использование приложения в условиях низкой скорости передачи данных (в браузер Google Chrome, например, встроена функция Throttling, которая позволяет сильно занижать скорость передачи данных), в условиях потери пакетов или при отключении сети во время активной фазы работы приложения (способ имитации: сначала делаем скорость передачи данных с помощью Throttling минимальной, а потом прерываем сетевое соединение во время обработки запроса).

В любом из описанных выше случаев приложение должно работать корректно. При «падении» запроса (time out) или иной проблеме мы должны, перезагрузив страницу, снова получить полностью работающее веб-приложение без какого-либо намека на только что пережитый «урон». Для всех ли функций приложения необходимо подобные тесты? Ни в коем случае! В будущем можете ориентироваться на свой опыт, а на первых этапах в этих вопросах лучше проконсультироваться с разработчиками.

Тестирование безопасности веб-приложения: спаси, сохрани, защити

Тестирование безопасности – отдельное направление тестирования, которое требует от специалиста фундаментальных знаний технического характера и хорошей профильной квалификации. Отметим ряд общих моментов, которые могут помочь любому тестировщику находить классические уязвимости, не допуская их выход на продакшен. Вопросы безопасности приложений регламентируются OWASP Guide, CHECK, ISACA, NIST Guideline, OSSTMM.

Существует ряд принципов безопасности, к которым относятся конфиденциальность, целостность и доступность:

Конфиденциальность – ограничение доступа к той или иной

информации для определенной категории пользователей (или наоборот предоставление доступа только ограниченной категории).

Целостность включает в себя:

а) возможность восстановить данные в полном объеме при их повреждении;

б) доступ на изменение информации только определенной категории пользователей.

Доступность – иерархия уровней доступа и четкое их соблюдение.

Перечислим классические уязвимости современных веб-приложений:

1. XSS – генерация на странице продукта скриптов, представляющих опасность для пользователей продукта.

2. XSRF – уязвимость, при которой пользователь переходит с доверенной страницы на вредоносную, где воруются представляющие ценность пользовательские данные.

3. Code injection (PHP, SQL) – инъекция части исполнительного кода, которая делает возможным получить несанкционированный доступ к программному коду или базе данных и вносить в них изменения.

4. Authorization bypass – это вид уязвимости, при котором можно получить несанкционированный доступ к учетной записи или документам другого пользователя.

5. Переполнение буфера – явление, которого можно достичь во вредоносных целях, по своей сути представляет использование места для записи данных далеко за пределами выделенного буфера памяти.

При тестировании рекомендуем использовать чит-листы уязвимостей XSS Filter Evasion Cheat Sheet и MySQL SQL Injection Cheat Sheet.

Практические советы: еще раз о насущном

Начинаем тестировать не с тестирования: с чего начать?

Акцентируем ваше внимание на необходимости согласовать все ключевые аспекты с ответственными лицами (аналитиками, проектными менеджерами, разработчиками) еще до начала тестирования. Необходимо заранее выяснить следующие моменты:

– цель тестирования (с их точки зрения);

– виды тестирования, которые необходимо провести;

– специфику приложения и его целевую аудиторию;

– перечень устройств, на которых необходимо проводить тестирование;

– список операционных систем (ОС) и браузеров для тестирования;

– разрешения экранов, на которых необходимо проверить приложение;

– предусмотрены ли требования к разного рода формам, есть ли стайл-гайд;

– необходимость предоставления конкретной документации по результатам тестирования (отчет, чек-листы, тест-кейсы и т. д.).

Получив ответы на эти вопросы, о которых часто забывают новички, вы сэкономите свое время. Дальше переходите к непосредственной подготовке окружения и формированию стратегии тестирования.

Некогда думать, нужно тестировать!

Любое тестирование требует содержательного подхода с применением техник тест-анализа и тест-дизайна. В противном случае вы рискуете навсегда остаться «monkey-тестером», ценность труда которого будет мизерна. В целом ключевые положения тест-анализа и тест-дизайна применимы как к тестированию десктоп приложений, так и веб-приложений, но с существенной оговоркой: вы должны учитывать все изложенные в статье нюансы. Отдельно хочу акцентировать внимание на том, что без стойкого понимания методик и способов применения тест-анализа и тест-дизайна тестировать качественно ПО практически невозможно.

Рассмотрим классический набор методик тест-дизайна, которые можно применять при тестировании веб-приложений:

– разбиение на классы эквивалентности;

– попарное тестирование;

– тестирование переходов состояний;

– анализ граничных значений;

– тестирование с помощью таблиц решений;

– методика тест-туров Виттакера и множество других.

Подай ключ «на 13»

В настоящее время существует огромное множество разнообразных инструментов, которые упрощают жизнь всем участникам разработки нового ПО. Следовательно, не стоит забывать о том, что помимо развития личных качеств, технических знаний и навыков, мы должны уметь хорошо пользоваться вспомогательными инструментами, каждый раз испытывая все новые и новые.

Отмечу лишь некоторые из них:

• Ресайзер – оперативное изменение всяческих настроек размеров отображения, просмотр метрик, переключение ориентаций отображения.

• Opera Mobile Classic Emulator – классический одноименный эмулятор.

• Mobile Phone Emulator – классический эмулятор.

• Fiddler – ПО для отслеживания всего трафика. При этом любое тестирование необходимо сопровождать фоновой работой, а потом сортирую по ошибкам и анализирую трафик. Также с помощью этого приложения можно отправлять ложные запросы на сервер с нужными вам параметрами.

Xenu Link Evaluator (альтернатива – Black Widow) – «чекер» веб-приложения на предмет наличия в нем «битых» ссылок. Также можно использовать его для формирования карты приложения.

Skipfish – активный сканер уязвимостей веб-приложений.

OpenVAS – бесплатный сканер уязвимостей.

Nikto – веб-сканер, проверяющий веб-серверы на самые частые ошибки, возникающие обычно из-за человеческого фактора.

Выводы и напутствия

Важно помнить, что тестирование ПО ставит перед каждым вступившим в стройные ряды сферы обеспечения качества ПО такие задачи, которые практически невозможно решать однозначно и по четкому алгоритму. Тестирование – это философия, творчество, полет мысли, основанный на четких и безоговорочных технических аспектах. Пожалуй, только тестировщикам приходится одновременно быть разработчиком, системным аналитиком, пользователем, заказчиком (для которого первоочередной интерес – финансовый или иной успех продукта), специалистом в предметной области и даже DBA.

2.6. Особенности тестирования Desktop приложений

В тестировании веб-сервисов основной клиент – это браузер. Их подразделяют на мобильные/desktop и др. В тестировании desktop приложений главным клиентом выступает операционная система.

Самым главным вопросом в тестировании настольных приложений является наличие большого количества конфигураций, по которым может быть собран персональный компьютер. Для продуктов, которые разрабатываются для конкретного заказчика, разработка и тестирование может быть упрощено, так как продукт может быть разработан под конкретную платформу. Для «коробочных» продуктов, которые могут массово распространяться (примером такого продукта может быть Microsoft Office), вопрос конфигурации персонального компьютера становится особенно важным.

При проверке работоспособности настольного приложения необходимо обратить внимание на следующие особенности.

Для нашего продукта две основные ветки ОС это Windows и Windows Server. Актуальные версии Windows – XP(SP1,SP2, SP3), Windows Vista, Windows 7, Windows 8, Windows 10. Серверные – Windows Server 2003, Windows Sever 2008, Windows Server 2010.

С чего же начать тестирование desktop приложений?

Можно начать со знакомства с HELP – мануалом продукта.

Давайте рассмотрим тест-кейсы на примере всем известного MS Office, также доступного как облачный сервис Office 360.

Таблица 2.17

Системные требования

Компьютер и процессор	ПК: процессор x86 или x64 с тактовой частотой от 1 ГГЦ и поддержкой набора инструкций SSE2. Mac: процессор Intel
Память	ПК: 2 ГБ ОЗУ
	Mac: 4 ГБ ОЗУ
Жесткий диск	ПК: 3,0 ГБ свободного пространства на жестком диске Mac: 6 ГБ свободного пространства на жестком диске формата HFS+ (также известного как Mac OS Extended или HFS Plus)
Экран	ПК: разрешение экрана 1280 x 800
	Mac: разрешение экрана 1280 x 800
Графическая подсистема	ПК: чтобы использовать аппаратное ускорение графики, необходима графическая карта с поддержкой DirectX10
Операционная система	ПК: Windows 10, Windows 8.1, Windows 8, Windows 7 с пакетом обновления 1, Windows 10 Server, Windows Server 2012 R2, Windows Server 2012 или Windows Server 2008 R2 Mac: Mac OS X 10.10 Для оптимальной работы используйте последнюю версию любой операционной системы
Браузер	Текущая или предыдущая версия Internet Explorer, текущая версия Microsoft Edge, Safari, Chrome или Firefox

Тестирование по системным требованиям

Процессор. Минимальный набор инструкций SSE2. Если ваш процессор не поддерживает эти требования, то программа должна выдать сообщение о невозможности установки. В настоящее время

тяжело найти процессоры которые не поддерживают SSE2, есть всего несколько таких типов 2002–2004 года выпуска, по сути это раритет. Сейчас для нашего продукта это не критично, поэтому мы пропускаем такой тест-кейс. Но не забывайте про распространенные процессоры AMD, которые не поддерживают такой тип инструкций. В таких случаях программа будет «крешиться». Если Windows успеет перехватить процесс и выдать ошибку, то есть шанс отловить этот баг.

За время работы в компании был случай, когда после релиза стало поступать много жалоб от клиентов. Но отловить баг было очень проблематично, потому что не нашлось подходящей конфигурации железа. Выловить удалось только программистам при помощи тщательного code review. В релиз был включен переход от старой библиотеки к новой, и поэтому минимальные требования к железу повысились. Как следствие, пользователи со старым железом не смогли обновить приложение.

После багфикса в коде уже учитывались все минимальные требования. Также мы обновили описание продукта, добавив минимальные рекомендуемые характеристики.

Оперативная память компьютера. Проводим положительные и отрицательные тест-кейсы. Смотрим как программа будет действовать при <= 2ГБ ПЗУ, >= 2ГБ ПЗУ. Зачастую можно обхойтись виртуальными образами OracleBox.

Пример sanity тест:

- Поднимаем виртуалку с 2гб ОЗУ.
- Запускаем программу. Все ОК.
- Затем уменьшаем в настройках виртуалки до 1 Гб , 512 Мб и далее.
- Запускаем программу. Бинго!

Жесткий диск. Аналогично ПЗУ. Можно добавить негативный тест, когда нет места на диске и идет процесс установки программы и (или) ее запуска. Еще один кейс, когда программа работает и записывает информацию на диск. Приложение должно корректно обрабатывать ситуацию, когда не хватает места для записи.

Дисплей. Основные кейсы – это различные разрешения: от минимального до 4К. Такие мониторы стали попадаться намного чаще. Особое внимание также нужно обратить на сохранение окна приложения. Попробуйте свернуть/развернуть/восстановить/изменить размер и закрыть приложение. Рассматривайте ситуацию с двумя дисплеями, таких тоже сейчас немало. В этих случаях программа должна запоминать размеры и параметры дисплея, на котором была запущена.

Графическая подсистема. Смотрим минимальные требования и запускаем. Современные компьютеры даже со встроенной видеокартой могут запускать большинство приложений. На моем проекте не требуется больших ресурсов, поэтому мы редко проводим регрессионное тестирование с большим скоупом видеокарт.

Тестирование по операционным системам

Всегда обращайте внимание на разрядность операционки. Она имеет большое значение. Если ваша программа связана с математическими вычислениями, то на ОС с разной разрядностью, результаты могут быть совершенно разными. Всегда тестируйте приложения с учетом обновлений Microsoft. У них принято часто латать дыры. А частые обновления ОС могут навредить вашей программе.

В серверных и обычных версиях Windows есть разграничение прав доступа: администратор, гость и др. Важно учитывать все эти роли и иметь возможность установки программы даже в режиме гостя. Хотя гость и не имеет прав записи в programm files, он должен иметь свой каталог temp и устанавливать программу в эту папку.

Негативные тесты тоже возможны. Например, попробуйте установить вашу программу в папку без права на запись или чтение.

Браузер. Здесь ему тоже нашлось место. Очень часто приложения используют движок нативного браузера Windows, в нашем случае IE. Так же как Android приложения могут быть построены на Webview. В Windows 10 используется MS Edge, Windows 8.1 IE 11, в Windows XP SP3 максимальная версия это IE8. Если минимальные требования к вашему продукту это IE9+, то пользователи с Windows XP будут недовольны.

2.7. Рефакторинг

Рефакторинг (Refactoring) (сущ.) – это изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения.

В основном цель рефакторинга – упростить понимание и модификацию программного обеспечения. Можно выполнить много изменений в программном обеспечении, в результате которых его видимое поведение изменится незначительно или вообще не изменится. Рефакторингом будут только такие изменения, которые сделаны с целью облегчения понимания исходного кода. Противоположным при-

мером может служить оптимизация производительности.

Как и рефакторинг, оптимизация производительности обычно не изменяет поведения компонента (за исключением скорости его работы); она лишь изменяет его внутреннее устройство. Цели, однако, различны.

Оптимизация производительности часто затрудняет понимание кода, но она необходима для достижения желаемого результата.

Главное обстоятельство, которое нужно отметить, заключается в том, что рефакторинг не меняет видимого поведения программного обеспечения. Оно продолжает выполнять прежние функции. Никто – ни конечный пользователь, ни программист – не сможет сказать по внешнему виду, что что-то изменилось.

Применение рефакторинга при разработке программного обеспечения разделяет время между двумя разными видами деятельности – вводом новых функций и изменением структуры. Добавление новых функций не должно менять структуру существующего кода: просто вводятся новые возможности. Прогресс можно оценить, добавляя тесты и добиваясь их нормальной работы. При проведении рефакторинга вы стремитесь не добавлять функции, а только улучшать структуру кода. При этом не добавляются новые тесты (если только не обнаруживается пропущенная ранее ситуация); тесты изменяются только тогда, когда это абсолютно необходимо, чтобы проверить изменения в интерфейсе.

В процессе разработки программного обеспечения может оказаться необходимым часто переключаться между двумя видами работы. Попытавшись добавить новую функцию, можно обнаружить, что это гораздо проще сделать, если изменить структуру кода. Тогда следует на некоторое время переключиться на рефакторинг. Улучшив структуру кода, можно добавлять новую функцию. А добившись ее работы, можно заметить, что она написана способом, затрудняющим ее понимание, тогда вы снова переключаетесь и занимаетесь рефакторингом. Все это может происходить в течение десяти минут, но в каждый момент вы должны понимать, которым из видов работы заняты.

Рефакторинг – это инструмент, который можно и нужно использовать для нескольких целей.

Рефакторинг улучшает композицию программного обеспечения. Без рефакторинга композиция программы приходит в негодность. По мере внесения в код изменений, связанных с реализацией краткосрочных целей или производимых без полного понимания ор-

ганизации кода, последний утрачивает свою структурированность. Разобраться в проекте, читая код, становится все труднее. Рефакторинг напоминает наведение порядка в коде. Убираются фрагменты, оказавшиеся не на своем месте. Утрата кодом структурности носит кумулятивный характер. Чем сложнее разобраться во внутреннем устройстве кода, тем труднее его сохранить и тем быстрее происходит его распад. Регулярно проводимый рефакторинг помогает сохранять форму кода.

Плохо спроектированный код обычно занимает слишком много места, часто потому, что он выполняет в нескольких местах буквально одно и то же. Поэтому важной стороной улучшения композиции является удаление дублирующегося кода. Важность этого связана с модификациями кода в будущем. Сокращение объема кода не сделает систему более быстрой, потому что изменение размера образа программы в памяти редко имеет значение. Однако объем кода играет существенную роль, когда приходится его модифицировать. Чем больше кода, тем труднее правильно его модифицировать, и разбираться приходится в его большем объеме. При изменении кода в некотором месте система ведет себя несоответственно расчетам, поскольку немодифицирован другой участок, который делает то же самое, но в несколько ином контексте. Устраняя дублирование, мы гарантируем, что в коде есть все, что нужно, и притом только в одном месте, в чем и состоит суть хорошего проектирования.

Рефакторинг облегчает понимание программного обеспечения. Во многих отношениях программирование представляет собой общение с компьютером. Программист пишет код, указывающий компьютеру, что необходимо сделать, и тот в ответ делает в точности то, что ему сказано. Со временем разрыв между тем, что требуется от компьютера, и тем, что вы ему говорите, сокращается. В таком режиме суть программирования состоит в том, чтобы точно сказать, что требуется от компьютера. Но программа адресована не только компьютеру. Пройдет некоторое время, и кому-нибудь понадобится прочесть ваш код, чтобы внести какие-то изменения. Об этом пользователе кода часто забывают, но он-то и есть главный. Станет ли кто-нибудь волноваться из-за того, что компьютеру для компиляции потребуется несколько дополнительных циклов? Зато важно, что программист может потратить неделю на модификацию кода, которая заняла бы у него лишь час, будь он в состоянии разобраться в коде.

Рефакторинг помогает найти ошибки. Лучшее понимание

кода помогает выявить ошибки. После прояснения структуры программы некоторые сделанные допущения становятся настолько ясными, что не возможно не увидеть ошибки. Это напоминает высказывание Кента Бека, которое он часто повторяет: «Я не считаю себя замечательным программистом. Я просто хороший программист с замечательными привычками».

Рефакторинг очень помогает писать надежный код.

Рефакторинг позволяет быстрее писать программы.

В конечном счете все перечисленное сводится к одному: рефакторинг способствует ускорению разработки кода.

Хороший дизайн важен для сохранения скорости разработки программного обеспечения. Благодаря рефакторингу программы разрабатываются быстрее, так как он удерживает композицию системы от распада. С его помощью можно даже улучшить дизайн.

Когда следует проводить рефакторинг?

Как правило, лучше не откладывать рефакторинг «на потом». Рефакторингом следует заниматься постоянно понемногу. Надо не решать проводить рефакторинг, а проводить его, потому что необходимо сделать что-то еще, а поможет в этом рефакторинг.

Правило трех ударов

Вот руководящий совет, который дал Дон Роберте (Don Roberts). Делая что-то в первый раз, вы просто это делаете. Делая что-то аналогичное во второй раз, вы морщитесь от необходимости повторения, но все-таки повторяете то же самое. Делая что-то похожее в третий раз, вы начинаете рефакторинг.

Примечание

- После трех ударов начинайте рефакторинг.
- Применяйте рефакторинг при добавлении новой функции.

Чаще всего я начинаю рефакторинг, когда в некоторое программное обеспечение требуется добавить новую функцию. Иногда при этом причиной рефакторинга является желание лучше понять код, который надо модифицировать. Этот код мог написать кто-то другой, а мог и я сам. Всякий раз, когда приходится думать о том, что делает некий код, я задаю себе вопрос о том, не могу ли я изменить его структуру так, чтобы организация кода стала более очевидной. После этого я провожу рефакторинг кода. Отчасти это делается на тот случай, если мне снова придется с ним работать, но в основном потому, что мне большее становится понятным, если в процессе работы я делаю код более ясным.

Еще одна причина, которая в этом случае побуждает к проведе-

нию рефакторинга, – это дизайн, не способствующий легкому добавлению новой функции. Глядя на дизайн кода, я говорю себе: «Если бы я спроектировал этот код так-то и так-то, добавить такую функцию было бы просто». В таком случае я не переживаю о прежних промахах, а исправляю их путем проведения рефакторинга. Отчасти это делается с целью облегчения дальнейших усовершенствований, но в основном потому, что я считаю это самым быстрым способом. Рефакторинг – процесс быстрый и ровный. После рефакторинга добавление новой функции происходит значительно более гладко и занимает меньше времени.

Применяйте рефакторинг, если требуется исправить ошибку.

При исправлении ошибок польза рефакторинга во многом заключается в том, что код становится более понятным. Я смотрю на код, пытаясь понять его, я произвожу рефакторинг кода, чтобы лучше понять. Часто оказывается, что такая активная работа с кодом помогает найти в нем ошибки. Можно взглянуть на это и так: если мы получаем сообщение об ошибке, то это признак необходимости рефакторинга, потому что код не был достаточно ясным и мы не смогли увидеть ошибку.

Применяйте рефакторинг при разборе кода.

В некоторых организациях регулярно проводится разбор кода, а в других – нет, и напрасно. Благодаря разбору кода, знания становятся достоянием всей команды разработчиков. При этом более опытные разработчики передают свои знания менее опытным. Разборы помогают большему числу людей разобраться с большим числом аспектов крупной программной системы. Они также очень важны для написания понятного кода. Мой код может казаться понятным мне, но не моей команде. Это неизбежно – очень трудно поставить себя на место того, кто не знаком с вашей работой. Разборы также дают возможность большему числу людей высказать полезные мысли. Столько хороших идей у меня и за неделю не появится. Вклад, вносимый коллегами, облегчает мне жизнь, поэтому я всегда стараюсь чаще посещать разборы.

Почему рефакторинг приносит результаты

Кент Бек

Ценность программ имеет две составляющие – то, что они могут делать для нас сегодня, и то, что они смогут делать завтра. В большинстве случаев мы сосредоточиваемся при программировании на том, что требуется от программы сегодня. Исправляя ошибку или до-

бавляя новую функцию, мы повышаем ценность программы сегодняшнего дня путем расширения ее возможностей.

Занимаясь программированием долгое время, нельзя не заметить, что функции, выполняемые системой сегодня, отражают лишь одну сторону вопроса. Если сегодня можно выполнить ту работу, которая нужна сегодня, но так, что это не гарантирует выполнение той работы, которая понадобится завтра, то вы проиграли. Хотя, конечно, вы знаете, что вам нужно сегодня, но не вполне уверены в том, что потребуется завтра. Это может быть одно, может быть и другое, а может быть и то, что вы не могли себе вообразить.

Я знаю достаточно, чтобы выполнить сегодняшнюю задачу. Я знаю недостаточно, чтобы выполнить завтрашнюю. Но если я буду работать только на сегодняшний день, завтра я не смогу работать вообще.

Рефакторинг – один из путей решения описанной проблемы. Обнаружив, что вчерашнее решение сегодня потеряло смысл, мы его изменяем. Теперь мы можем выполнить сегодняшнюю задачу. Завтра наше сегодняшнее представление о задаче покажется наивным, поэтому мы и его изменим. Из-за чего бывает трудно работать с программами? В данный момент мне приходят в голову четыре причины:

- Программы, трудные для чтения, трудно модифицировать.
- Программы, в логике которых есть дублирование, трудно модифицировать.
- Программы, которым нужны дополнительные функции, что требует изменений в работающем коде, трудно модифицировать.
- Программы, реализующие сложную логику условных операторов, трудно модифицировать.

Итак, нам нужны программы, которые легко читать, вся логика которых задана в одном и только одном месте, модификация которых не ставит под угрозу существующие функции и которые позволяют выражать условную логику возможно более простым способом.

Рефакторинг представляет собой процесс улучшения работающей программы не путем изменения ее функций, а путем усиления в ней указанных качеств, позволяющих продолжить разработку с высокой скоростью.

Я обнаружил, что рефакторинг помогает мне разобраться в коде, написанном не мной. До того, как я стал применять рефакторинг, я мог прочесть код, в какой-то мере понять его и внести предложения. Теперь, когда у меня возникают идеи, я смотрю, нельзя ли их тут же реализовать с помощью рефакторинга. Если да, то я провожу рефак-

торинг. Проделав это несколько раз, я лучше понимаю, как будет выглядеть код после внесения в него предлагаемых изменений. Не надо напрягать воображение, чтобы увидеть будущий код, это можно сделать уже теперь. В результате появляются уже идеи следующего уровня, которых у меня не возникло бы, не проведи я рефакторинг.

Кроме того, рефакторинг способствует получению более конкретных результатов от разбора кода. В его процессе не только возникают новые предложения, но многие из них тут же реализуются. В результате это мероприятие дает ощущение большей успешности.

Для этой технологии группы разбора должны быть невелики. Мой опыт говорит в пользу того, чтобы над кодом совместно работали один рецензент и сам автор кода. Рецензент предлагает изменения, и они вместе с автором решают, насколько легко провести соответствующий рефакторинг. Если изменение небольшое, они модифицируют код.

При разборе больших проектов часто бывает лучше собрать несколько разных мнений в более представительной группе. При этом показ кода может быть не лучшим способом. Я предпочитаю использовать диаграммы, созданные на UML, и проходить сценарии с помощью CRC-карт (Class Responsibility Collaboration cards). Таким образом, разбор дизайна кода я провожу в группах, а разбор самого кода – с отдельными рецензентами.

Эта идея активного разбора кода доведена до своего предела в практике программирования парами (Pair Programming) в книге по экстремальному программированию [Beck, XP]. При использовании этой технологии все серьезные разработки выполняются двумя разработчиками на одной машине. В результате в процесс разработки включается непрерывный разбор кода, а также рефакторинг.

Косвенность и рефакторинг

Кент Бек

Вычислительная техника – это дисциплина, в которой считается, что все проблемы можно решить благодаря введению одного или нескольких уровней косвенности.

Деннис Де Брюле

Учитывая одержимость разработчиков программного обеспечения косвенностью, не следует удивляться тому, что рефакторинг, как правило, вводит в программу дополнительную косвенность. Рефакторинг обычно разделяет большие объекты, как и большие методы, на несколько меньших. Однако рефакторинг – меч обоюдоострый. Каж-

дый раз при разделении чего-либо надвое количество объектов управления растет. При этом может также быть затруднено чтение программы, потому что один объект делегирует полномочия другому, который делегирует их третьему. Поэтому желательно минимизировать косвенность.

Но не следует спешить. Косвенность может окупиться, например, следующими способами:

• *Позволить совместно использовать логику*. Например, подметод, вызываемый из разных мест, или метод родительского класса, доступный всем подклассам.

• *Изолировать изменения в коде*. Допустим, я использую объект в двух разных местах и мне надо изменить его поведение в одном из этих двух случаев. Если изменить объект, это может повлиять на оба случая, поэтому я сначала создаю подкласс и пользуюсь им в том случае, где нужны изменения. В результате можно модифицировать родительский класс без риска непреднамеренного изменения во втором случае.

• *Кодировать условную логику*. В объектах есть сказочный механизм – полиморфные сообщения, гибко, но ясно выражающие условную логику. Преобразовав явные условные операторы в сообщения, часто можно одновременно уменьшить дублирование, улучшить понятность и увеличить гибкость.

Базы данных

Одной из областей применения рефакторинга служат базы данных. Большинство деловых приложений тесно связано с поддерживающей их схемой базы данных. Это одна из причин, по которым базу данных трудно модифицировать. Другой причиной является миграция данных. Даже если система тщательно разбита по слоям, чтобы уменьшить зависимости между схемой базы данных и объектной моделью, изменение схемы базы данных вынуждает к миграции данных, что может оказаться длительной и рискованной операцией.

В случае необъектных баз данных с этой задачей можно справиться, поместив отдельный программный слой между объектной моделью и моделью базы данных. Благодаря этому можно отделить модификации двух разных моделей друг от друга. Внося изменения в одну модель, не обязательно изменять другую, надо лишь модифицировать промежуточный слой. Такой слой вносит дополнительную сложность, но дает значительную гибкость. Даже без проведения ре-

факторинга это очень важно в ситуациях, когда имеется несколько баз данных или сложная модель базы данных, управлять которой вы не имеете возможности.

Не обязательно начинать с создания отдельного слоя. Можно создать этот слой, заметив, что части объектной модели становятся переменчивыми. Благодаря этому достигаются наилучшие возможности для осуществления модификации.

Объектные модели одновременно облегчают задачу и усложняют ее. Некоторые объектно-ориентированные базы данных позволяют автоматически переходить от одной версии объекта к другой. Это сокращает необходимые усилия, но влечет потерю времени, связанную с осуществлением перехода. Если переход не автоматизирован, его приходится осуществлять самостоятельно, что требует больших затрат. В этом случае следует более осмотрительно изменять структуры данных классов. Можно свободно перемещать методы, но необходимо проявлять осторожность при перемещении полей. Надо применять методы для доступа к данным, чтобы создавать иллюзию их перемещения, в то время как в действительности его не происходило. При достаточной уверенности в том, где должны находиться данные, можно переместить их с помощью одной операции. Модифицироваться должны только методы доступа, что снижает риск появления ошибок.

Изменение интерфейсов

Важной особенностью объектов является то, что они позволяют изменять реализацию программного модуля независимо от изменений в интерфейсе. Можно благополучно изменить внутреннее устройство объекта, никого при этом не потревожив, но интерфейс имеет особую важность – если изменить его, может случиться всякое.

В рефакторинге беспокойство вызывает то, что во многих случаях интерфейс действительно изменяется. Такие простые вещи, как «Переименование метода» (Rename Method), целиком относятся к изменению интерфейса. Как это соотносится с бережно хранимой идеей инкапсуляции?

Поменять имя метода нетрудно, если доступен весь код, вызывающий этот метод. Даже если метод открытый, но можно добраться до всех мест, откуда он вызывается, и модифицировать их, метод можно переименовывать. Проблема возникает только тогда, когда интерфейс используется кодом, который не доступен для изменений. В таких случаях мы говорим, что интерфейс опубликован (на одну ступень дальше открытого интерфейса). Если интерфейс опублико-

ван, изменять его и просто редактировать точки вызова небезопасно. Необходима несколько более сложная технология.

При таком взгляде вопрос изменяется. Теперь задача формулируется следующим образом: как быть с рефакторингами, изменяющими опубликованные интерфейсы?

При изменении опубликованного интерфейса в процессе рефакторинга необходимо сохранять как старый интерфейс, так и новый, по крайней мере, до тех пор, пока пользователи не смогут отреагировать на модификацию. К счастью, это не очень трудно. Обычно можно сделать так, чтобы старый интерфейс продолжал действовать. Попытайтесь устроить так, чтобы старый интерфейс вызывал новый. При этом, изменив название метода, сохраните прежний. Не копируйте тело метода – дорогой дублирования кода вы пройдете прямиком к проклятию. На Java следует также воспользоваться средством пометки кода как устаревшего (необходимо объявить метод, который более не следует использовать, с модификатором deprecated). Благодаря этому, обращающиеся к коду будут знать, что ситуация изменилась.

Хорошим примером этого процесса служат классы коллекций Java. Новые классы Java 2 заменяют собой те, которые предоставлялись первоначально. Однако, когда появились классы Java 2, JavaSoft, было приложено много усилий для обеспечения способа перехода к ним.

Изменения дизайна, вызывающие трудности при рефакторинге

Можно ли с помощью рефакторинга исправить любые недостатки проектирования или в дизайне системы могут быть заложены такие базовые решения, которые впоследствии не удастся изменить путем проведения рефакторинга? Имеющиеся в этой области данные недостаточны. Конечно, часто рефакторинг оказывается эффективным, но иногда возникают трудности. Мне известен проект, в котором, хотя и с трудом, но удалось при помощи рефакторинга привести архитектуру системы, созданной без учета защиты данных, к архитектуре с хорошей защитой.

На данном этапе мой подход состоит в том, чтобы представить себе возможный рефакторинг. Рассматривая различные варианты дизайна системы, я пытаюсь определить, насколько трудным окажется рефакторинг одного дизайна в другой. Если трудностей не видно, то я, не слишком задумываясь о выборе, останавливаюсь на самом простом дизайне, даже если он не охватывает все требования, которые могут возникнуть в дальнейшем. Однако если я не могу найти простых способов рефакторинга, то продолжаю работать над дизайном.

Мой опыт показывает, что такие ситуации редки.

Когда рефакторинг не нужен?

В некоторых случаях рефакторинг вообще не нужен. Основной пример – необходимость переписать программу с нуля. Иногда имеющийся код настолько запутан, что подвергнуть его рефакторингу, конечно, можно, но проще начать все с самого начала. Такое решение принять нелегко, и я признаюсь, что не могу предложить достаточно надежные рекомендации по этому поводу.

Явный признак необходимости переписать код – его неработоспособность. Это обнаруживается только при его тестировании, когда ошибок оказывается так много, что сделать код устойчивым не удается. Помните, что перед началом рефакторинга код должен выполняться в основном корректно.

Рефакторинг и проектирование

Рефакторинг играет особую роль в качестве дополнения к проектированию. Когда я только начинал учиться программированию, я просто писал программу и как-то доводил ее до конца. Со временем мне стало ясно, что если заранее подумать об архитектуре программы, то можно избежать последующей дорогостоящей переработки. Я все более привыкал к этому стилю предварительного проектирования. Многие считают, что проектирование важнее всего, а программирование представляет собой механический процесс. Аналогией проекта служит технический чертеж, а аналогией кода – изготовление узла. Но программа весьма отличается от физического механизма. Она значительно более податлива и целиком связана с обдумыванием. Как говорит Элистер Кокберн (Alistair Cockburn): «При наличии готового дизайна я думаю очень быстро, но в моем мышлении полно пробелов».

Существует утверждение, что рефакторинг может быть альтернативой предварительному проектированию. В таком сценарии проектирование вообще отсутствует. Первое решение, пришедшее в голову, воплощается в коде, доводится до рабочего состояния, а потом обретает требуемую форму с помощью рефакторинга. Такой подход фактически может действовать. Мне встречались люди, которые так работают и получают в итоге систему с очень хорошей архитектурой. Тех, кто поддерживает «экстремальное программирование» [Beck, XP], часто изображают пропагандистами такого подхода.

Подход, ограничивающийся только рефакторингом, применим, но не является самым эффективным. Даже «экстремальные» программисты сначала разрабатывают некую архитектуру будущей сис-

темы. Они пробуют разные идеи с помощью CRC-карт или чего-либо подобного, пока не получат внушающего доверия первоначального решения. Только после первого более или менее удачного «выстрела» приступают к кодированию, а затем к рефакторингу. Смысл в том, что при использовании рефакторинга изменяется роль предварительного проектирования. Если не рассчитывать на рефакторинг, то ощущается необходимость как можно лучше провести предварительное проектирование. Возникает чувство, что любые изменения проекта в будущем, если они потребуются, окажутся слишком дорогостоящими. Поэтому в предварительное проектирование вкладывается больше времени и усилий – во избежание таких изменений впоследствии. С применением рефакторинга акценты смещаются. Предварительное проектирование сохраняется, но теперь оно не имеет целью найти единственно правильное решение. Все, что от него требуется, – это найти приемлемое решение. По мере реализации решения, с углублением понимания задачи становится ясно, что наилучшее решение отличается от того, которое было принято первоначально. Но в этом нет ничего страшного, если в процессе участвует рефакторинг, потому что модификация не обходится слишком дорого.

Рефакторинг позволяет создавать более простые проекты, не жертвуя гибкостью, благодаря чему процесс проектирования становится более легким и менее напряженным. Научившись в целом распознавать то, что легко поддается рефакторингу, о гибкости решений даже перестаешь задумываться. Появляется уверенность в возможности применения рефакторинга, когда это понадобится. Создаются самые простые решения, которые могут работать, а гибкие и сложные решения по большей части не потребуются.

Рефакторинг и производительность

С рефакторингом обычно связан вопрос о его влиянии на производительность программы. С целью облегчить понимание работы программы часто осуществляется модификация, приводящая к замедлению выполнения программы. Это важный момент. Я не принадлежу к той школе, которая пренебрегает производительностью в пользу чистоты проекта или в надежде на рост мощности аппаратной части. Программное обеспечение отвергалось как слишком медленное, а более быстрые машины устанавливают свои правила игры. Рефакторинг, несомненно, заставляет программу выполняться медленнее, но при этом делает ее более податливой для настройки производительности. Секрет создания быстрых программ, если только они не предназначены для работы в жестком режиме реального времени, состоит

в том, чтобы сначала написать программу, которую можно настраивать, а затем настроить ее так, чтобы достичь приемлемой скорости.

Мне известны три подхода к написанию быстрых программ. Наиболее трудный из них связан с ресурсами времени и часто применяется в системах с жесткими требованиями к выполнению в режиме реального времени. В этой ситуации при декомпозиции проекта каждому компоненту выделяется бюджет ресурсов – по времени и памяти. Компонент не должен выйти за рамки своего бюджета, хотя разрешен механизм обмена временными ресурсами. Такой механизм жестко сосредоточен на соблюдении времени выполнения. Это важно в таких системах, как, например, кардиостимуляторы, в которых данные, полученные с опозданием, всегда ошибочны. Данная технология избыточна в системах другого типа, например, в корпоративных информационных системах, с которыми я обычно работаю.

Второй подход предполагает постоянное внимание. В этом случае каждый программист в любой момент времени делает все от него зависящее, чтобы поддерживать высокую производительность программы. Это распространенный и интуитивно привлекательный подход, однако он не так хорош на деле. Модификация, повышающая производительность, обычно затрудняет работу с программой. Это замедляет создание программы. На это можно было бы пойти, если бы в результате получалось более быстрое программное обеспечение, но обычно этого не происходит. Повышающие скорость усовершенствования разбросаны по всей программе, и каждое из них касается только узкой функции, выполняемой программой. С производительностью связано то интересное обстоятельство, что при анализе большинства программ обнаруживается, что большая часть времени расходуется небольшой частью кода. Если в равной мере оптимизировать весь код, то окажется, что 90 % оптимизации произведено впустую, потому что оптимизировался код, который выполняется не слишком часто. Время, ушедшее на ускорение программы, и время, потерянное из-за ее непонятности – все это израсходовано напрасно.

Третий подход к повышению производительности программы основан как раз на этой статистике. Он предполагает создание программы с достаточным разложением ее на компоненты без оглядки на достигаемую производительность вплоть до этапа оптимизации производительности, который обычно наступает на довольно поздней стадии разработки и на котором осуществляется особая процедура настройки программы.

Глава 3. Нефункциональные виды тестирования

3.1. Тестирование безопасности

Основные цели тестирования безопасности:

1. Обеспечение безопасности онлайн-транзакций.

2. Защита конфиденциальной информации от несанкционированного доступа.

3. Минимизация риска утраты, искажения или хищения данных.

4. Увеличение сопротивления DoS-атакам.

Для достижения целей безопасности наши специалисты выполняют аудит потенциальных угроз с учетом специфики программного обеспечения и используют оптимальные методы тестирования безопасности.

Результаты тестирования безопасности

Мы стремимся обеспечить ясность результатов и процессов тестирования для наших клиентов. Вы получите:

– полный перечень угроз безопасности и уязвимостей тестируемого ПО;

– подробный отчет о проведенных тестах.

Аудит защищенности бизнес приложений

Анализ бизнес-приложений позволит получить независимую профессиональную оценку защищенности критичных информационных активов и подробные рекомендации по повышению их безопасности.

Критичная для бизнеса информация, интересная злоумышленникам, обычно хранится и обрабатывается в таких информационных системах, как различные веб-приложения, СУБД, ERP. Это сложные корпоративные приложения, аудит которых должен проводиться отдельно, так как необходимо учитывать специфику каждого приложения и использовать соответствующие модели нарушителя.

Когда следует проводить аудит приложений?

- Для уже используемых критичных приложений – с выбранной периодичностью либо при внесении изменений.
- Перед запуском в эксплуатацию нового бизнес-приложения.
- При добавлении надстроек к существующим приложениям.

В случае инцидента информационной безопасности (ИБ), связанного с функционированием приложения, и при подозрении на некорректную работу приложения с точки зрения ИБ.

Рассмотрим типы приложений, требующих проведения аудита безопасности.

SAP-системы – обеспечение защищенности корпоративных систем управления ресурсами предприятия (ERP-систем) является одной из важнейших задач служб информационной безопасности современных компаний.

СУБД Oracle – анализ защищенности корпоративных сетей все чаще показывает, что в целом уровень безопасности заметно возрос, однако ряду проблем до сих пор не уделяется должного внимания. Одна из них – это защищенность корпоративных систем управления базами данных (СУБД).

Системы ДБО – система дистанционного банковского обслуживания является ключевым ресурсом банка, безопасности которого следует уделять особое внимание.

Мобильные приложения – использование мобильных телефонов стало повсеместным и повседневным. Программное обеспечение, которое раньше было доступно только на компьютере, сейчас доступно и на телефоне.

Интеграционные шины (Enterprise Service Bus, ESB) позволяют реализовать обмен данными как между внутренними системами, такими как ERP или АБС, так и между компаниями-партнерами.

Встроенные системы (Embedded) – компьютерные системы, предназначенные для выполнения определенных специфичных задач.

Исходный код – в настоящее время большинство атак происходит на уровне приложений. Поэтому безопасность программного обеспечения имеет наивысший приоритет. В особенности это относится к заказной разработке.

Веб-приложения больше других подвержены угрозам безопасности, так как по определению доступны из сети Интернет.

Рассмотрим пример группы тестов аудита безопасности сайта (веб-приложения).

Reconnaissance – разведка, подготовительный этап. На данном этапе производится сканирование портов исследуемого ресурса, а также идентификация доступных сервисов.

Configuration Management Testing – тестирование настроек системы. На данном этапе производится поиск уязвимых компонентов серверного ПО и выявление небезопасных настроек хостинга.

Authentication/Authorization bypass – тестирование механизмов аутентификации и авторизации. На данном этапе производится проверка

возможности получения нелегитимного доступа к компонентам системы через обход механизмов аутентификации и авторизации.

Session Management Testing – тестирование механизмов управления сессиями. Проверяется возможность фиксации сессии (Session Fixation), подделки межсайтовых запросов (Cross Site Request Forgery) и др.

Business Logic Testing – тестирование механизмов логики приложения. На данном этапе проводится анализ уязвимых мест в логике веб-приложения, например, возможность совершить действие без соответствующего на то разрешения, проведения двойных транзакций (Race Condition)).

Data Validation Testing – тестирование механизмов проверки входных данных. Проверяется возможность внедрения произвольного кода (SQL Injection, XML Injection, XSS, Code Injection, B/H/S overflows) через очевидные и неочевидные точки входа в приложение.

Insecure File Uploading – небезопасная загрузка файлов на сервер. На данном этапе проводится проверка на возможность загрузки файлов, содержащих код (включая вредоносный), который впоследствии может быть выполнен на сервере.

В настоящее время наиболее распространенными видами *уязвимости в безопасности программного обеспечения* являются:

XSS (*Cross-Site Scripting*) – это вид уязвимости программного обеспечения (Web приложений), при которой, на генерированной сервером странице, выполняются вредоносные скрипты, с целью атаки клиента.

XSRF/*CSRF* (*Request Forgery*) – это вид уязвимости, позволяющий использовать недостатки HTTP протокола, при этом злоумышленники работают по следующей схеме: ссылка на вредоносный сайт установливается на странице, пользующейся доверием у пользователя, при переходе по вредоносной ссылке выполняется скрипт, сохраняющий личные данные пользователя (пароли, платежные данные и т. д.), либо отправляющий СПАМ сообщения от лица пользователя, либо изменяет доступ к учетной записи пользователя, для получения полного контроля над ней.

*Code injectio*ns (*SQL*, *PHP*, *ASP* и т. д.) – это вид уязвимости, при котором становится возможно осуществить запуск исполняемого кода с целью получения доступа к системным ресурсам, несанкционированного доступа к данным либо выведения системы из строя.

Server-Side Includes (*SSI*) *Injection* – это вид уязвимости, использующий вставку серверных команд в HTML код или запуск их напрямую с сервера.

Authorization Bypass – это вид уязвимости, при котором возможно получить несанкционированный доступ к учетной записи или документам другого пользователя

Чек лист проверки безопасности

Контроль доступа – определяет проблемы, связанные с несанкционированным доступом пользователей к информации и функциям в зависимости от предоставленной роли. Тестирование конфигурации ролевой модели.

Аутентификация – позволяет удостовериться в отсутствии возможности обойти процедуру регистрации и авторизации; убедиться в корректности управления пользовательскими данными, исключить возможность получения информации о зарегистрированных пользователях и их учетных данных.

Валидация входных значений – используется для проверки алгоритмов обработки данных, включая некорректные значения прежде, чем на них будет ссылаться приложение.

Криптография – обнаруживает проблемы, связанные с шифрованием, дешифрованием, подписью, верификацией подлинности, в том числе включая уровень сетевых протоколов, работу с временным файлами и cookies.

Механизмы обработки ошибок – включает проверку системных ошибок приложения на отсутствие факта раскрытия информации о внутренних механизмах безопасности (например, посредством демонстрации исключений, программного кода).

Конфигурация сервера – ищет в многопоточных процессах ошибки, связанные с доступностью значений переменных для совместного использования другими приложениями и запросами.

Интеграция со сторонними сервисами – позволяет убедиться в невозможности манипуляции данными, передаваемыми между приложением и сторонними компонентами, например, платежными системами или соцсетями.

Проверка устойчивости к Dos/DDos атакам – проверяет способность приложения обрабатывать незапланированно высокие нагрузки и большие объемы данных, которые могут быть направлены на выведение приложения из строя.

Степень безопасности вашей системы будет оценена в соответствии с международным стандартом подтверждения безопасности приложений OWASP. Мы также проведем рекомендательную работу

по устранению уязвимостей и рисков, опционально можем разработать регламент работы с продуктом для операторов и разработчиков – так вы защитите собственный бизнес и пользователей ресурса.

3.2. Тестирование производительности

Тестирование производительности – обобщенное понятие, которым часто обозначают разные виды проверки ПО.

Первым в череде тестов проводится стресс-тест (Stress Test), цель которого – установить предельный уровень производительности продукта. Стресс-тест позволяет проанализировать зависимость ключевых характеристик системы (времени отклика самых важных бизнес-транзакций, количества запросов в секунду, количества транзакций в секунду) от количества одновременно работающих пользователей.

Во время стресс-теста нагрузка на систему подается непрерывно до тех пор, пока не будет достигнут один из критериев его остановки. Например, стресс-тест банковской системы был остановлен при превышении отметки в 1500 пользователей, когда высокая загруженность процессора (более 80 %) привела к увеличению среднего времени отклика в пять раз и массовому появлению ошибок HTTP(S).

На втором этапе проводится нагрузочный тест (Load Test), с помощью которого оценивается способность системы справляться с длительной нагрузкой (4–8 часов). Количество пользователей для нагрузочного теста определяется в количестве 80 % от результата максимальной производительности, выявленной при стресс-тесте. Уровень нагрузки при *тестировании банковской системы* поддерживался на одном уровне в течение восьми часов и составил 1200 пользователей. Нагрузочный тест показал существенное ухудшение производительности системы с течением времени, а дополнительное профилирование ее компонентов позволило обнаружить дефекты, проявляющиеся только при длительной работе большого количества пользователей (например, утечки памяти).

Как правильно подавать нагрузку на систему?

При проведении нагрузочного тестирования важно аккуратно подойти к установке инструмента нагрузочного тестирования. Инструмент устанавливается на генератор нагрузки – виртуальную или физическую машину, ресурсы которой используются для создания нагрузки на систему. Генератор нагрузки должен располагаться максимально близко к тестовому окружению. Это необходимо для устра-

нения искажений при подаче нагрузки, вызванных задержками сети, величина которых может варьироваться от нескольких миллисекунд до нескольких десятков секунд.

Поскольку задержки, возникающие в локальных сетях при передаче пакетов данных, существенно меньше, мы рекомендуем размещать генератор нагрузки в одной локальной сети с серверами тестируемого приложения.

Так, во время тестирования бразильского видеопортала среднее время отклика от сервера составило 20 секунд при запуске тестов с рабочей машины в европейском регионе. А при запуске с виртуальной машины, развернутой в одной локальной сети с тестируемой системой, – 2 секунды. Таким образом, обеспечение наивысшей скорости обмена данными между клиентом и сервером, позволило протестировать приложение в условиях, приближенных к идеальным.

Иногда стоит обратная задача – измерить ключевые характеристики производительности приложения путем эмуляции работы пользователей из отдаленных регионов.

Например, на одном из проектов мы проводили тестирование крупного новостного портала игровой вселенной, который посещают пользователи со всего мира. Для предоставления объективных результатов мы провели распределенное нагрузочное тестирование. Нагрузка была распределена между несколькими генераторами нагрузки, расположенными в разных географических регионах. В качестве генераторов нагрузки использовались виртуальные машины, арендуемые на платформах облачных сервисов. Результаты распределенного тестирования позволили сделать выводы о необходимости размещения дополнительных серверов в регионах с наихудшими показателями производительности.

По желанию заказчика возможно проведение дополнительных видов тестов.

Дополнительные виды тестов производительности

Задачей объемного теста (Volume Test) является оценка производительности системы при увеличении объемов данных, хранимых в базе данных приложения. Схема подачи нагрузки при данном виде теста такая же, как и при нагрузочном. Для проведения теста требуется база данных, заполненная необходимым объемом данных. Так, при тестировании биллинговой системы для оператора мобильной связи, объем данных был выбран исходя из ее прогнозируемого наполнения через два года после выхода обновленной версии системы в производство.

Тест на стабильность (Stability Test) позволяет оценить работоспособность системы при длительной ожидаемой нагрузке в режиме работы 24/7. К примеру, если веб-сайт посещают пользователи, находящиеся в разных часовых поясах, уровень нагрузки может сохраняться постоянным. Помимо возможных перезапусков серверов системы под продолжительной нагрузкой, при тесте на отказоустойчивость также изучается влияние редких событий на деградацию производительности системы, например, работа сборщиков мусора.

Тест на масштабируемость (Scalability Test) оценивает способность системы увеличивать производительность пропорционально увеличению масштаба нефункциональных возможностей. Так, после проведения нагрузочного теста и замера характеристик производительности веб-приложения к его серверам добавляется дополнительный сервер с аналогичными характеристиками. При повторном запуске нагрузочного теста можно оценить изменение производительности и корректность работы балансировщика нагрузки. Тест на масштабируемость позволяет определить эргономичность расхода ресурсов системы, увеличить ее рабочий потенциал и рационализировать инвестиции в аппаратное обеспечение.

Отличие тестирования клиентской части vs. от тестирования производительности

Исходя из опыта нашей компании, мы заметили, что клиенты, которые обращаются в компанию за *тестированием веб-приложений*, путают тестирование клиентской части и тестирование производительности. Рассмотрим эти понятия и процесс их взаимодействия на примере обычного веб-приложения.

Наилучшим решением проблем со стабильностью и сбоями в работе веб-серверов является обеспечение двухуровневой архитектуры приложения: клиентской, или Front-end, части и серверной, или Back-end, части.

Пользователь открывает браузер и отправляет запрос к странице сайта на Front-end сервер. Запрос принимается и запрашивается у исполнительной части веб-приложения – Back-end сервера, который хранит логику приложения, обеспечивает выполнение PHP-скриптов и генерирует HTML-страницы. Front-end принимает сформированную страницу от Back-end и в качестве ответа на запрос пользователя передает ее в браузер. Получив страницу, браузер пользователя начинает ее отображение, что сопровождается отправкой серии запросов на графический контент и CSS. Эти запросы принимает Front-end и об-

рабатывает без обращений к Back-end'у.

Оценка скорости работы клиентской и серверной частей веб-приложения осуществляется двумя разными видами тестирования: для Front-end применяется тестирование клиентской части, или Client-Side Testing, а для Back-end – тестирование производительности серверной части.

Основная цель тестирования клиентской части состоит в измерении времени, необходимого браузеру, для загрузки HTML-страницы. Наиболее важными показателями здесь являются количество загружаемых данных, их объем, а также количество выполненных запросов.

Собрать данную статистику можно как с использованием встроенных инструментов браузера, так и с помощью специализированных инструментов и онлайн-сервисов, которые позволяют замерить необходимые показатели с учетом интересующего региона.

Помимо общего веса страницы, инструменты предоставляют детализированную информацию по каждому из компонентов. Изучив параметры запросов, можно обнаружить ряд проблем, приводящих к ухудшению скорости отображения страницы. К примеру, подгружается слишком большая картинка и Javascript, или отправляется значительное количество запросов.

Улучшить скорость отображения страницы можно с помощью уменьшения размеров, сжатия элементов (CSS, Javascript и графического контента), а также путем сокращения названий переменных и оптимизации кода страницы.

Другая необходимая проверка направлена на анализ заголовков кэширования, поскольку корректность его выполнения при повторном посещении страницы позволяет повысить скорость загрузки страницы до 80 %.

Тестирование клиентской части также позволяют обнаружить ряд дефектов, например, отсутствие или некорректную работу элементов на странице.

Тестирование производительности серверной части направлено на анализ выполнения запросов и получения соответствующего запроса от Back-end.

Цели данного вида тестирования:

– измерить время отклика самых важных бизнес-транзакций;

– определить предельный уровень допустимой нагрузки;

– выявить «узкие» места в производительности системы;

– составить рекомендации по улучшению производительности;

– найти возможные дефекты, проявляющиеся только при одновременной работе большого количества пользователей.

Заключение

Важно проводить *тестирование производительности* последовательно с измерением конкретных показателей. Такой подход даст полное представление о продуктивности системы и укажет на области для дальнейшей доработки приложения с целью приведения в соответствие с обозначенными бизнес-требованиями.

3.3. Тестирование юзабилити

К основным видам юзабилити тестирования относятся:

- Карточная сортировка (Card Sorting).
- Контекстное исследование (Contextual Inquiry).
- Контрольные листы (Checklists).
- Макетирование (Prototyping).
- Обзоры (Surveys).
- Опросники (Questionnaires).
- Плюралистическая проработка (Pluralistic Walkthroughs).
- Протоколы самоотчета (Self-Reporting Logs).
- Фиксация «мыслей вслух» (Thinking Aloud Protocol).
- Фокусные группы (Focus Groups).

Как определить, насколько удобен веб-сайт для большинства пользователей? У каждого имеются свои предпочтения, и то, что нравится одному, далеко не всегда нравится другому. Существуют ли универсальные критерии, пригодные для оценки практически любого сайта?

Пригодность к использованию и удобство веб-сайтов (web usability) изучается при помощи большого количества специальных методов.

Карточная сортировка (Card Sorting)

Карточная сортировка – это классификационный метод, при котором пользователи сортируют различные элементы разрабатываемого веб-сайта по нескольким категориям.

Для проведения карточной сортировки создается список параметров, которые предполагается подвергнуть классификации, после чего каждый из указанных параметров выписывается на отдельной карточке. Карточки предъявляются пользователям, которых инструктируют сгруппировать наиболее логичным, по их мнению, образом.

Полученную в результате карточной сортировки информацию используют для организации пользовательского интерфейса.

Контекстное исследование (Contextual Inquiry)

Контекстное исследование – это метод структурированного интервью, которое отличается от обычного, например, журналистского интервью, тем, что оно всегда построено на трех базовых принципах:

1. Учет контекста, в котором используется изучаемый сайт.

2. Совместная оценка сайта пользователем и разработчиком.

3. В фокусе оценки сайта находится именно его удобство для пользователя.

Контекстное исследование является одной из альтернатив метода эталонного тестирования, при котором удобство оценивается в лабораторных условиях, а не в привычной для пользователя рабочей обстановке. При контекстном исследовании работа, время, мотивация и социальные факторы, воздействующие на пользователя, остаются такими же, как в реальном мире, в отличие от лабораторных исследований, где эти факторы контролируются экспериментатором.

Контекстное исследование наиболее применимо для того, чтобы оценить ту обстановку, в которой будет использоваться продукт, поэтому оно проводится на ранних стадиях его разработки.

Наиболее предпочтительно использование контекстного исследования совместно с эталонным тестированием, так как каждый из этих методов по отдельности менее эффективен, чем их совокупность.

Контрольные листы (Checklists)

Контрольные листы помогают удостовериться в том, что вебсайт выполнен с учетом принципов функциональности дизайна. Обычно их используют на заключительной стадии работы в дополнение к экспертным методам для того, чтобы структурировать экспертные оценки по каким-то определенным признакам.

Существует большое количество готовых контрольных листов, однако решение об использовании того или иного списка должно зависеть от задач исследования. Зачастую возникает необходимость в разработке собственных критериев качества в определенной области.

Макетирование (Prototyping)

Макетирование – это создание модели конечного продукта (вебсайта), позволяющее протестировать его составляющие на любых стадиях разработки.

В процессе макетирования строится модель, включающая все тестируемые компоненты (дизайн, элементы управления и т. д.). Можно

использовать различные способы ее построения – от изображения элементов интерфейса на бумаге до создания рабочего макета веб-сайта. Различают «горизонтальное» и «вертикальное» макетирование.

«Горизонтальное» макетирование означает представление в макете широкого спектра параметров веб-сайта, но без их детальной проработки. Этот вид макетирования предпочтительно использовать для изучения пользовательских предпочтений в области интерфейса. В процессе исследования выясняют, где, по мнению пользователей, должны находиться те или иные функции, насколько они доступны и т. д. «Горизонтальное» макетирование применяют на ранних стадиях разработки.

При «вертикальном» макетировании исследуют функциональность отдельных компонентов веб-сайта. Например, при «вертикальном» макетировании сетевой базы данных могут изучаться ее поисковые возможности, но проверка ссылок, скорость загрузки, работа в разных браузерах и т. д. при этом не проводится. Так как этот вид макетирования предполагает детальное изучение небольшого сектора веб-сайта, его следует проводить лишь после завершения разработки тестируемого сектора.

Обзоры (Surveys)

Обзоры – это специальное интервью с пользователями, при котором им задаются заранее подготовленные вопросы, а их ответы записываются для обработки. Традиционная методология проведения обзоров может также играть важную роль в исследовании веб-сайтов. Вопросы, входящие в обзор, могут варьироваться в зависимости от целей исследования, но как правило группируются по следующим категориям: демографическая информация о пользователе (место проживания, возраст, род занятий и т. д.); оценка информационного наполнения веб-сайта (какую информацию ожидали найти, соответствует ли наполнение веб-сайта их ожиданиям, что хотели бы добавить и т. д.); оценка дизайна сайта (легкость в использовании, скорость загрузки, оформление и т. д.).

Обзоры используют как на стадиях концептуализации и разработки для проведения маркетинговых исследований, идентификации потенциальных пользователей, установления их информационных нужд и компетентности в использовании компьютеров, так и после реализации веб-сайта для оценки реакций пользователей на информационное наполнение и удобство. Огромное количество образцов и результатов обзоров доступно через любую поисковую систему по клю-

чевым словам «web usability» и «survey».

Опросники (Questionnaires)

Опросники для оценки веб-сайтов составляются по тем же принципам, что и психодиагностические тесты. Их основное отличие от обзоров состоит в форме представления пользователям. Если обзоры – это структурированные интервью, то опросники предъявляются в виде списка вопросов с вариантами ответа.

Существует достаточно много стандартизированных опросников, большинство из которых распространяется на коммерческой основе, однако западные опросники нуждаются не только в переводе, но и в кросс-культурной адаптации. В зависимости от того, какие вопросы входят в опросник, этот метод может применяться на всех стадиях разработки веб-сайта или в процессе его эксплуатации.

Плюралистическая проработка (Pluralistic Walkthroughs)

Плюралистическая проработка проводится большой по размеру группой, в которую помимо экспертов в области оценки веб-продукции входят пользователи, разработчики и специалисты по профессиональному здоровью и организации труда.

Группа шаг за шагом рассматривает сценарии поведения пользователя, обсуждая и прорабатывая каждый элемент его взаимодействия с веб-сайтом. Вовлечение в эксперимент специалистов различных специальностей обеспечивает оценку веб-сайта с самых разнообразных позиций, а их замечания способствуют идентификации имеющихся недостатков.

Метод применяется на ранних стадиях разработки в целях получения обратной связи как от специалистов, так и от конечных пользователей.

Протоколы самоотчета (Self-Reporting Logs)

Протоколы самоотчета это бланки типа «карандаш-бумага», в которых пользователи фиксируют все свои действия и соображения о взаимодействии с веб-сайтом. Этот метод является достаточно экономным, так как специалисты вовлекаются лишь в обработку результатов, не контролируя действия пользователя во время выполнения заданий.

Основными недостатками этого метода, как и других методов, предполагающих самостоятельную работу пользователей, является невозможность контроля и регистрации их эмоциональных реакций на взаимодействие с веб-сайтом и проблема адекватности их отчетов

тому, что они реально делают. Поэтому в данном случае отбор пользователей, участвующих в эксперименте, приобретает принципиальный характер.

Для проведения исследования необходимо обеспечить испытуемых доступом к прототипу веб-сайта, описанием задач, которые они должны решать при его использовании, и стандартизированным бланком для регистрации пользователями своих действий.

Обычно эта техника используется на ранних стадиях планирования или разработки или для выявления пользовательских предпочтений.

Фиксация «мыслей вслух» (Thinking Aloud Protocol)

Фиксация мыслей пользователя, вовлеченного в эксперимент, является одной из самых популярных техник при оценке функциональности веб-сайта. Пользователя просят произносить вслух все мысли, чувства и представления, которые у него возникают в процессе решения задачи.

Пользователя обеспечивают доступом к тестируемому веб-сайту или его прототипу и дают ему задание, которое он должен реализовать в процессе его эксплуатации. Его задача – выполнять задачу, одновременно «озвучивая» все, что приходит в голову по поводу интерфейса. Данные записываются на аудиопленку или фиксируются письменно.

В отличие от большинства других методов, эта техника позволяет оценить непосредственные реакции пользователя на взаимодействие с отдельными компонентами веб-сайта, не отсроченные по времени. И если его ожидания в отношении необходимых для решения задачи операций расходятся с дизайнерским решением веб-сайта, возможно, следует изменить это решение.

Несмотря на то что основной задачей техники является выяснение пользовательских представлений, с ее помощью можно реализовывать и другие цели. Например, терминология, которую употребляет пользователь для обозначения тех или иных элементов интерфейса, может быть использована и в дизайне веб-сайта.

Метод предполагает обобщение данных, полученных от нескольких пользователей. Существует также близкий к этой технике метод ответов на вопросы, использующий не вербализацию мыслей и эмоций, а директивные вопросы экспериментатора с фиксацией ответов пользователя.

Фокусные группы (Focus Groups)

Метод фокусных групп заключается в опросе специально ото-

бранной группы пользователей. В исследование, которое обычно продолжается около двух часов, вовлекается от 6 до 9 пользователей. Основное достоинство фокусных групп состоит в том, что они позволяют выявлять спонтанные реакции и идеи и оценивать отношение к этим идеям группы в целом.

Как правило, участники группы воспринимают происходящее как относительно свободный неструктурированный процесс, но ведущий группы должен иметь предварительный сценарий работы, вытекающий из целей исследования, и следить, чтобы групповая дискуссия не выходила из русла обсуждаемой проблемы. Кроме того, необходимо добиваться равного участия в дискуссии всех членов группы. Достаточно часто бывает, что в группе выделяется лидер, доминирующий над остальными участниками. Такие ситуации мешают свободному выражению мыслей и идей остальных членов группы и также нуждаются в коррекции.

Результаты работы фокусной группы заносятся в специальный протокол для дальнейшей обработки. Сбор детальной информации при этом методе затруднен из-за относительной стихийности группового процесса, поэтому рекомендуется проводить несколько фокусных групп, состоящих из репрезентативных пользователей.

Несмотря на ряд преимуществ фокусных групп, они имеют и свои недостатки. Главным из них является неточность оценки, основанной на утверждениях, мыслях и предпочтениях небольшого количества пользователей. Поэтому при оценке веб-сайта фокусные группы должны использоваться лишь наряду с другими методами.

Фокусные группы могут использоваться как на любой стадии разработки веб-сайта, так и для оценки готового продукта.

3.4. Введение в автоматизацию тестирования

Использование автоматизированного тестирования предоставляет огромные возможности и позволяет существенно повысить надежность кода и безопасность приложения. Поэтому разработка крупных и сложных систем непременно требуют привлечения специалистов в области автоматизированного тестирования. С другой стороны, автоматизированное тестирование – процесс достаточно сложный как с точки зрения написания кода, так и с точки зрения методологии и организации процессов в команде.

Стратегия автоматизации тестирования предполагает модель постоянной поставки с несколькими командами, работающими по методологии Agile.

На этом примере перечислим ключевые моменты, которые необходимо учитывать, чтобы получить максимальный эффект от проведения автоматизированных тестов.

Краткое содержание

Автоматизированное тестирование – ключевой процесс разработки с использованием методологии Agile. При переходе к постоянному развертыванию автоматизация тестирования становится еще более важной из-за возможности быстро информировать разработчиков о состоянии приложения. Чтобы обеспечить постоянный поток обратной связи, автоматические тесты необходимо проводить постоянно и быстро, а их результаты должны быть надежными и достоверными.

Чтобы обеспечить выполнение этих условий, большая часть проверок должна проводиться в рамках разработки новых функциональных возможностей. Другими словами, разработка и тестирование должны быть неразрывно связаны, а обеспечение качества должно быть заложено с самого начала разработки, чтобы новые возможности не нарушали работу существующего функционала.

Это требует «инвертирования пирамиды автоматизации тестирования» с отказом от GUI-тестов, которые занимают много времени, в пользу тестирования на более низких уровнях, например, API, где автоматические тесты можно запустить сразу после unit-тестов как часть сборки, чтобы обеспечить базовый уровень надежности.

Обзор стратегии автоматизации тестирования

Предотвращение вместо обнаружения – разумеется, необходимо приложить все усилия, чтобы предотвратить возникновение недостатков, но техники и методы, которые для этого используются, не являются предметом этой статьи. Здесь нас интересует, как можно обнаружить баги, как только они появляются в системе, и оперативно передать информацию разработчикам.

Качество должно быть превыше количества. В подавляющем большинстве случаев лучше выпустить в релиз одну фичу, надежную, как скала, нежели сразу несколько полусырых возможностей. Минимальным критерием для релиза должно быть полное отсутствие регрессионных дефектов, т. е. новые возможности не должны нарушать работу существующего функционала.

Как было изложено выше, быстрое информирование разработчиков о состоянии приложения имеет огромное значение при непре-

рывной поставке, следовательно, надо найти механизм, который позволит быстро давать обратную связь. Один из способов – увеличить количество unit-тестов, интеграционных тестов и тестов API. Эти низкоуровневые тесты формируют сеть безопасности, которая помогает убедиться, что приложение работает так, как было задумано, и позволяет предотвратить дефекты, возникающие на других уровнях тестирования. Unit-тесты служат основой для автоматизации тестирования на более высоких уровнях.

Вторая возможность для улучшения работы – запускать регрессионные тесты чаще и в параллели с непрерывной поставкой. Автоматизированное тестирование должно быть не изолированной задачей, а непрерывным процессом, неотъемлемо вписанным в жизненный цикл ПО.

Регрессионное тестирование

Автоматические регрессионные тесты – основа стратегии автоматизации тестирования.

«Дымовой» пакет регрессионных тестов нужен для проверки того, что приложение загружается и запускается. В него также входят несколько ключевых сценариев, позволяющих убедиться, что приложение еще работает.

Цель этого пакета тестов в том, чтобы отловить наиболее очевидные проблемы, например, то, что приложение не загружается или не запускается основной поток взаимодействия пользователя с приложением. Поэтому «дымовые» тесты не должны продолжаться больше **5 минут**, их цель – сообщить, что не работает что-то ключевое.

Такие тесты запускаются при каждом развертывании приложения и могут содержать как API, так и GUI-тесты.

Функциональный пакет регрессионных тестов нужен для более детальной проверки работы приложения, чем это позволяют «дымовые» тесты.

Необходимо создать несколько функциональных пакетов для различных целей. Если есть несколько команд, работающих над различными разделами приложения, то в идеале нужны регрессионные пакеты, покрывающие область работы каждой команды.

Эти пакеты должны запускаться в различных окружениях по мере необходимости и проверять, что поведение приложения остается неизменным вне зависимости от окружения. Такие тесты запускаются несколько раз в день и должны продолжаться не дольше 15–30 минут.

Поскольку эти тесты более детализированы и занимают больше времени, важно выносить большую часть функциональных тестов на

уровень API, где тестирование проходит быстрее. Это нужно для того, чтобы не выходить за временные рамки в 15–30 минут.

Полный пакет регрессионных тестов позволяет протестировать приложение как целое. Цель этого пакета тестов – проверить, что различные части приложения, которые обращаются к различным базам данных и другим приложениям, работают корректно.

Этот пакет тестов не предназначен для проверки всех возможностей приложения, поскольку их работа уже проверена функциональными регрессионными пакетами. В любом случае эти тесты более «легкие» и проверяют переходы из одного состояния в другое или несколько наиболее популярных сценариев или путей пользователя.

Такие тесты в основном проводятся с использованием GUI, поскольку они проверяют, как пользователь будет взаимодействовать с системой. Время, которое на них затрачивается, может варьироваться в зависимости от приложения, но обычно такие тесты запускаются один раз за день или за ночь.

Рассмотрим стратегию автоматизации тестирования для нескольких Agile-команд.

Автоматизированные unit-тесты. Автоматизация тестирования начинается на уровне unit-тестов. Эти тесты должны создаваться для каждой новой возможности, находящейся в разработке. Именно они ложатся в основу более широкой практики автоматизации вплоть до системных GUI-тестов. Разработчики обязаны убедиться в том, что для каждой новой функциональной возможности разработан полный набор надежных unit-тестов, позволяющих проверить, что код работает как задумано и отвечает всем требованиям. Unit-тесты наиболее выгодны с точки зрения окупаемости, поскольку их недолго написать, легко поддерживать и изменять (благодаря тому, что нет зависимостей), так что если в коде есть ошибка, то разработчик быстро узнает о ней. Unit-тесты должны запускаться как на компьютере разработчика, так и в среде непрерывной интеграции.

Автоматическая интеграция/API-тесты или сервис-тесты. В то время как unit-тесты основаны на тестировании функций внутри класса, интеграционные тесты формируют следующую ступень, направленную на тестирование классов, образующих компонент, входящий в состав нового функционала. Такие тесты запускаются только после того, как unit-тестирование было успешно завершено.

Сервис-тесты обычно запускаются на уровне API без вовлечения GUI-интерфейса, следовательно, тесты направлены на проверку

функциональности в чистом виде, а поскольку тесты непосредственно обращаются к компонентам, они быстро проводятся и могут быть частью сборки. При необходимости тестирования взаимодействия с внешними сервисами, в случае, если внешние сервисы не доступны либо не могут гарантировать предоставление данных, отвечающих условиям тестирования, можно использовать эмуляторы внешних сервисов, например WireMock. API-тесты и (или) сервис-тесты могут запускаться на компьютере разработчика или быть частью сборки, но если они начинают занимать длительное время, лучше запускать их в среде непрерывной интеграции. Для сервис-тестов можно использовать такие инструменты, как SoapUI.

Тесты приложения. На практике крупное приложение, например, система для электронной коммерции, может быть разбито на несколько приложений, предоставляющих различные возможности. Концепция «тестирования приложений» заключается в том, что группы тестов, направленные на возможности одного приложения, объединяются и прогоняются для этого приложения. Этот пакет можно использовать в случаях, когда команда планирует выпустить индивидуальное приложение и хочет проверить, все ли работает корректно.

Чтобы протестировать приложение в целом, обычно требуется интерфейс для взаимодействия между различными его компонентами, а значит, тестирование лучше проводить с использованием браузера или GUI. Цель этих тестов – убедиться в том, что приложение работает корректно. Такие тесты называют «вертикальными», так как они направлены на проверку работоспособности конкретного приложения или компонента, а не всей системы целиком. Эти тесты отличаются глубиной проработки и большим объемом.

Для проведения таких тестов в браузере можно использовать Selenium WebDriver. Этот инструмент является наиболее популярным для проведения автоматизированного тестирования в браузерах и предоставляет богатые возможности API для проведения сложных проверок.

Полные сценарные тесты. Автоматизированные GUI-тесты, которые запускаются для всей системы, используются как типичные пути пользователей или полные сценарии взаимодействия. Из-за проблем с этим типом тестов (описанных ниже) их количество лучше сократить до минимума. Полные сценарии включены в ночные регрессионные пакеты.

Инвертирование пирамиды автоматизации тестирования

В рамках стратегии автоматизации тестирования нам необходи-

мо минимизировать количество автоматизированных тестов на уровне GUI.

Несмотря на то что проведение автоматизированного GUI-тестирования дает хорошие и значимые результаты с точки зрения симуляции пользовательского взаимодействия с приложением, оно имеет и ряд своих недостатков:

– *хрупкость* – для определения веб-элементов для взаимодействия тесты используют html-локаторы, поэтому как только меняется уникальный ID какого-либо элемента интерфейса, тесты перестают работать, а это влечет за собой значительные расходы на поддержку;

– *ограниченное тестирование* – GUI может не позволить тестировщику полностью проверить функциональность, поскольку он не всегда содержит все детали веб-ответа, необходимые для верификации;

– *низкая скорость* – поскольку тесты проводятся через GUI, время загрузки страницы существенно увеличивает общее время тестирования, и обратная связь разработчикам поступает значительно позже;

– *наименьшая окупаемость* – из-за всех проблем, перечисленных выше, GUI-тесты становятся наименее целесообразными с финансовой точки зрения.

Автоматическое тестирование в браузере нужно сокращать до минимума и применяем для симуляции поведения пользователей в основных потоках взаимодействия и полных сценариях, где используется система в целом.

3.5. Тестирование мобильных приложений

Бурный рост использования мобильных устройств и разработка мобильных приложений делает тестирование ключевым требованием для быстрой доставки конечному пользователю высококачественных мобильных приложений.

Особенности мобильного приложения

Мобильные приложения сильно отличаются от настольных приложений во многих аспектах. Поэтому необходимо учитывать это при планировании процесса тестирования.

Рассмотрим основные различия между мобильными и настольными приложениями:

• Мобильное устройство – это система, которая не обладает производительным аппаратным обеспечением. Таким образом, оно не может работать как персональный компьютер.

• Тестирование мобильных приложений проводится на мобильных процессорах (процессоры архитектур ARM, MIPS), в то время как настольное приложение тестируется на центральном процессоре (в подавляющем большинстве x86, x86-64).

• У мобильных устройств бывают разные разрешения. Размер экрана мобильного телефона меньше, чем у настольных.

• Выполнение и прием вызовов является основной задачей телефона, поэтому приложение не должно вмешиваться в эту важную функцию.

• Широкий спектр конкретных операционных систем и компонентных конфигураций: Android, iOS, BlackBerry, Windows Mobile.

• Операционная система мобильного телефона быстро устаревает.

• Мобильные устройства используют сетевые подключения (3G, 4G, Wi-Fi), широкополосное подключение к настольному ПК или Wi-Fi.

• Мобильные устройства постоянно осуществляют поиск сети. Поэтому необходимо протестировать приложение с разной скоростью передачи данных.

• Инструменты, которые хорошо подходят для тестирования настольных приложений, не полностью подходят для тестирования мобильных приложений.

• Мобильные приложения должны поддерживать несколько входных каналов (клавиатура, голос, жесты и т. д.), мультимедийные технологии и другие функции, повышающие их удобство использования.

Типы приложения

Еще одна важная вещь в процессе тестирования мобильных приложений – это тип приложения.

Существует три основных типа мобильных приложений: мобильные веб-приложения, нативные приложения и гибридные приложения.

Фактически мобильным веб-приложением является веб-сайт, открытый в гаджете (смартфоне или планшете) с помощью мобильного браузера.

Достоинства мобильных веб-приложений:

• Простая разработка.

• Легкий доступ.

• Простое обновление.
• Не требует установки.
• Недостатки мобильных веб-приложений:
• Нет поддержки автономных функций.
• Ограниченная функциональность в сравнении с гибридными и нативными приложениями (нет доступа к файловой системе и локальным ресурсам).
• Проблемы с перераспределением: Google Play и App Store не поддерживают перераспределение мобильных веб-приложений.

Нативное приложение – это приложение, разработанное специально для одной платформы (Android, iOS, BlackBerry, Windows Mobile).

Достоинства нативных приложений:
• Нативное приложение работает в автономном режиме.
• Оно может использовать все функции своего устройства.
• Продвинутый пользовательский интерфейс.
• Push-уведомления для удобства пользователей.

Недостатки нативных приложений:
• Разработка нативных обходится дороже в сравнении с мобильными веб-приложениями.
• Требуются большие затраты на техническое обслуживание.
• Гибридное приложение – это сочетание нативного и мобильного веб-приложений. Его можно определить как отображение содержимого мобильного сайта в формате приложения.

Достоинства гибридных приложений:
• Более рентабельно по сравнению с нативным приложением.
• Простое распространение.
• Встроенный браузер.
• Особенности устройства.

Недостатки гибридных приложений:
• Работает не так быстро, как нативное приложение.
• Графика менее адаптирована к ОС в сравнении с нативным приложением.

Ключевые моменты в стратегии тестирования мобильного сайта

Рассмотрим основные моменты и проблемы, с которыми можно столкнуться в процессе тестирования.

Выбор устройств

Несомненно, реальное устройство – лучшее решение, если нужно протестировать мобильное приложение. Тестирование на реальном устройстве всегда дает максимальную точность результатов.

Фактически действительно нелегко выбрать наиболее подходящее устройство. Перечислим некоторые действия, которые необходимо предпринять при выборе устройства для мобильного тестирования:

Проанализировать и определить самые популярные и используемые гаджеты на рынке.

- Выбрать устройства с разной ОС.
- Выбрать устройства с различными разрешениями экрана.
- Также необходимо обратить внимание на следующие факторы: совместимость, объем памяти, возможность подключения и т. д.

Преимущества для тестирования мобильных приложений на реальных устройствах:

- Высокая точность результата тестирования.
- Простая репликация ошибок.

Такие моменты, как емкость батареи, геолокация, push-уведомления, встроенные датчики устройств легко тестируются.

- Возможность проверки входящих прерываний (звонков, SMS).
- Возможность тестирования мобильного приложения в реальных условиях и условиях.
- Нет ложных срабатываний.

Недостатки для тестирования мобильных приложений на реальных устройствах:

- Существует огромное количество часто используемых устройств.
- Дополнительные расходы на обслуживание устройств.
- Ограниченный доступ к устройствам, часто используемым в зарубежных странах.

Как можно увидеть, тестирование на реальных устройствах является хорошим решением, но также имеет некоторые ограничения.

Эмуляторы или симуляторы?

Несложно догадаться о существовании специальных инструментов, которые эмулируют/моделируют функциональность и поведение мобильных устройств.

Часто путают значения слов «эмулятор» и «симулятор». Несмотря на то, что они почти одинаково произносятся, они не одинаковы.

Фактически эмулятор – это оригинальная замена устройства. Однако нет возможности модифицировать программы и приложения, хотя их можно запускать. Симулятор не копирует аппаратное обеспечение устройства, но есть возможность настроить аналогичную среду, такую как в ОС оригинального устройства.

Таким образом, лучше использовать мобильные симуляторы для тестирования мобильного приложения. Эмуляторы больше подходят для тестирования мобильных сайтов.

Преимущества использования симуляторов для тестирования мобильного приложения:

- Простая настройка.
- Быстродействие.
- Помогает проверять и тестировать поведение вашего мобильного приложения.
- Экономически выгодно.

Недостатки использования симуляторов для тестирования мобильного приложения:

- Аппаратное оборудование не учитывается.
- Возможны ложные срабатывания.
- Получение неполных данных о результатах моделирования, что создает определенные трудности для полного анализа результатов тестирований.

Облачное тестирование мобильного приложения

Тестирование мобильных приложений с использованием облачных инструментов, по-видимому, является оптимальным выбором. Это может помочь преодолеть недостатки реальных устройств и симуляторов.

Основные преимущества этого подхода:

- Легкая доступность.
- Возможность запуска мобильных устройств на нескольких системах.
- Возможность не только тестировать, но и обновлять, а также управлять приложениями в облаке.
- Экономически выгодно.
- Высокая масштабируемость.
- Один и тот же скрипт можно запускать на нескольких устройствах параллельно.

Недостатки облачного мобильного тестирования:

- Меньше контроля.
- Нет такого высокого уровня безопасности.
- Зависимость от интернет-соединения.

Некоторые полезные облачные инструменты, которые могут помочь вам протестировать мобильное приложение: Xamarin Test Cloud, Perfecto Mobile Continuous Quality Lab, Keynote Mobile Testing.

Ручное и автоматизированное мобильное тестирование

В настоящее время многие специалисты поддерживают мнение о том, что ручное тестирование в итоге перестанет использоваться. Конечно, это неправда. Невозможно обойтись без автоматизации тестирования, но есть ситуации, когда предпочтительным является ручное тестирование.

Достоинства ручного тестирования мобильных приложений:

- Это более экономически выгодно в краткосрочной перспективе.
- Ручное тестирование более гибкое.
- Лучшее моделирование действий пользователя.

Недостатки ручного тестирования мобильных приложений:

- Ручные тестовые примеры трудно использовать повторно.
- Менее эффективно выполнение определенной постоянной задачи.
- Процесс тестирования медленный.
- Некоторые типы тестовых случаев не могут быть выполнены вручную (нагрузочное тестирование).
- Преимущества автоматизированного тестирования приложений:
- Процесс тестирования занимает мало времени.
- Экономичность в долгосрочной перспективе использования.
- Автоматизированные тестовые случаи легко использовать повторно.
- Единственное решение для некоторых видов тестирования (тестирование производительности).
- Результаты испытаний легко доступны.

Недостатки автоматизированного тестирования приложений:

- У некоторых мобильных средств тестирования есть ограничения.
- Процесс тестирования занимает много времени.

Автоматизированное тестирование наименее эффективно в определении удобства пользования.

Конечно, нет однозначных ответов на то, какую стратегию лучше всего выбрать. Однако сочетание различных вариантов наиболее оптимально. Например, можно использовать симуляторы на самых ранних этапах вашего тестирования. Но лучше использовать реальные устройства (физические или облачные) на заключительных этапах. Автоматизированное тестирование предпочтительнее для нагрузочного и регрессионного тестирования.

Платформы для тестирования мобильных приложений

Одной из основных проблем в тестировании мобильных приложений является огромное количество моделей устройств. Не каждая

компания или предприятие может позволить себе покупку даже нескольких десятков устройств, не говоря уже о сотне или тысячи.

Однако существует решение данной проблемы. Разработчики некоторых операционных систем и систем для прототипирования приложений пошли на встречу разработчикам и предоставляют в условиях аренды так называемые «фермы» устройств – кластеры мобильных телефонов, планшетов с предустановленными операционными системами и сервисами. На данных «фермах» разработчикам разрешено выполнять автоматизированные юнит-тесты, инструментальные тесты (тесты, которые затрагивают взаимодействие приложения или его частей с операционной системой) и интеграционные тесты.

Таких сервисов существует несколько:

1. Firebase Test Lab For Android – сервис для тестирования Android приложений, предоставляемый компанией Google, который предоставляет возможность выполнять автоматизированные тесты на устройствах с помощью инструмента Robo Test, а также выполнять проверку игровых приложений с помощью Android Game Loop Test. Также система предоставляет отчеты по тестам и сбоям, а также снимкам для каждого устройства.

2. Xamarin Test Cloud – сервис для тестирования Android, iOS, Windows Mobile приложений, предоставляемый компанией Xamarin (Microsoft). Данный инструмент работает в рамках программы Xamarin и легко интегрируется в Visual Studio.

3. HockeyApp, AppBlade, Appaloosa, TestFairy – сервисы, интегрующиеся в приложение, которые помогают отслеживать ошибки, падения, а также значимые события, которые так или иначе влияют на работоспособность приложения.

Этапы тестирования мобильных приложений

Основные этапы процесса тестирования мобильных приложений в большей степени похожи на этапы тестирования веб-сайта. Список этапов тестирования приводится в параграфе 2.1.

Советы по тестированию мобильного приложения

Давайте систематизируем наши знания и попытаемся определить основные советы для тестирования мобильных приложений.

- Изучите приложение, которое вы собираетесь тестировать.
- Помните о различиях между настольными и мобильными приложениями.

• Учитывайте особенности операционной системы и оборудования.

• Используйте реальные устройства, когда это возможно.

• Не пытайтесь найти «швейцарский нож» для тестирования. Используйте инструменты, с которыми вы знакомы.

• Используйте преимущества облачного мобильного тестирования.

• Подтверждайте свои результаты с помощью скриншотов, журналов и видео.

• Используйте параметры меню разработки для iOS и Android.

• Не пренебрегайте (но не злоупотребляйте) эмуляторами и симуляторами для тестирования.

• Проверьте работоспособность своего приложения.

• Не автоматизируйте все.

• Попросите реальных пользователей протестировать ваше приложение.

• Учитывайте человеческий фактор.

Глава 4. Аттестация программного обеспечения

4.1. Аттестация программного обеспечения на основе тестирования классов эквивалентности

На четвертый день своего исследования Амазонки Байрон вылез из своей камеры, проверил последние новости о своем личном цифровом помощнике (далее – КПК), оснащенном беспроводной технологией, и понял, что чувствует в животе тяжесть, однако не было страха – нет, он не боялся, а был воодушевлен – и не было напряжения – нет, на самом деле он был довольно расслаблен – так что это, по всей вероятности, это был паразит.

Чак Килан

Тестирование классов эквивалентности – это метод, используемый для сокращения количества тест-кейсов до контролируемого уровня при сохранении разумного охвата тестами. Эта простая техника интуитивно используется почти всеми тестировщиками, даже если они могут не знать о ней как о формальном методе разработки тестов. Многие тестировщики логически вывели его полезность, в то время как другие обнаружили его просто из-за нехватки времени для более тщательного тестирования. Рассмотрим эту ситуацию. Мы пишем модуль для системы управления персоналом, который решает, как мы должны обрабатывать заявления о приеме на работу в зависимости от возраста человека. Правила нашей организации:

0–16 Не нанимать
16–18 Можно нанимать только неполный рабочий день
18–55 Можно нанимать штатных сотрудников
55–99 Не нанимать

Замечание

С этими правилами наша организация не нанимала бы Дуги Хаузера, доктора медицины или полковника Харлана Сандерса, один слишком молодой, а другой слишком пожилой.

Должны ли мы проверять модуль для следующих возрастов: 0, 1, 2, 3, 4, 5, 6, 7, 8, ..., 90, 91, 92, 93, 94, 95, 96, 97, 98, 99? Если бы у нас было много времени (и мы не возражали против многочисленного

повторения и нам платили по часам), мы бы, конечно, могли все проверить. Если программист реализовал этот модуль со следующим кодом, мы должны проверить каждый возраст.

```
If (applicantAge == 0) hireStatus="NO";
If (applicantAge == 1) hireStatus="NO";
…
If (applicantAge == 14) hireStatus="NO";
If (applicantAge == 15) hireStatus="NO";
If (applicantAge == 16) hireStatus="PART";
If (applicantAge == 17) hireStatus="PART";
If (applicantAge == 18) hireStatus="FULL";
If (applicantAge == 19) hireStatus="FULL";
…
If (applicantAge == 53) hireStatus="FULL";
If (applicantAge == 54) hireStatus="FULL";
If (applicantAge == 55) hireStatus="NO";
If (applicantAge == 56) hireStatus="NO";
…
If (applicantAge == 98) hireStatus="NO";
If (applicantAge == 99) hireStatus="NO";
```

Учитывая эту реализацию и тот факт, что любой набор тестов проходит, ничего не говорит нам о следующем тесте, который мы могли бы выполнить. Этот тест может выполниться или потерпеть неудачу. К счастью, программисты не пишут такой код (по крайней мере, не очень часто). Более опытный программист может написать:

```
If (applicantAge >= 0 && applicantAge <=16)
hireStatus="NO";
If (applicantAge >= 16 && applicantAge <=18)
hireStatus="PART";
If (applicantAge >= 18 && applicantAge <=55)
hireStatus="FULL";
If (applicantAge >= 55 && applicantAge <=99)
hireStatus="NO";
```

Учитывая эту типичную реализацию, ясно, что для первого требования нам не нужно проверять 0, 1, 2, ..., 14, 15 и 16. Необходимо

проверить только одно значение. Какое же это значение? Любое число в этом диапазоне так же хорошо, как и любое другое. То же самое верно для каждого из других диапазонов. Диапазоны, подобные описанным здесь, называются *классами эквивалентности*. Класс эквивалентности состоит из набора данных, которые обрабатываются модулем однообразно или которые должны давать одинаковый результат. Любое значение данных в классе эквивалентно с точки зрения тестирования любому другому значению. В частности, мы ожидаем, что:

- Если один тест-кейс в классе эквивалентности обнаруживает дефект, все другие тестовые примеры в том же классе эквивалентности, вероятно, обнаружат тот же дефект.
- Если один тест-кейс в классе эквивалентности не обнаруживает дефект, никакие другие тестовые примеры в том же классе эквивалентности, скорее всего, не обнаружат дефект.

Группа тестов образует класс эквивалентности, если считается, что они все тестируют одно и то же. Если один тест обнаружит ошибку, другие, вероятно, тоже. Если один тест не обнаружит ошибку, другие, вероятно, тоже не обнаружат.

Этот подход предполагает, конечно, что существует задание, которое определяет различные классы эквивалентности для тестирования. Это также предполагает, что программист не сделал ничего странного, как в случае:

```
If (applicantAge >= 0 && applicantAge <=16)
hireStatus="NO";
If (applicantAge >= 16 && applicantAge <=18)
hireStatus="PART";
If (applicantAge >= 18 && applicantAge <=41)
hireStatus="FULL";
// strange statements follow
If (applicantAge == 42 && applicantName == "Lee")
hireStatus="HIRE NOW AT HUGE SALARY";
If (applicantAge == 42 && applicantName <> "Lee")
hireStatus="FULL";
// end of strange statements
If (applicantAge >= 43 && applicantAge <=55)
hireStatus="FULL";
If (applicantAge >= 55 && applicantAge <=99)
hireStatus="NO";
```

Используя подход классов эквивалентности, мы сократили количество тестовых случаев со 100 (тестирование каждого возраста) до четырех (тестирование одного возраста в каждом классе эквивалентности), в результате чего получили значительную экономию. Теперь мы готовы начать тестирование? Возможно, нет. А как насчет входных значений, таких как 969, –42, FRED и & $ #! @? Должны ли мы создавать тест-кейсы для неверного ввода? Ответ, как скажет любой хороший консультант, «это зависит от…». Чтобы понять этот ответ, нам нужно изучить подход, вышедший из объектно-ориентированного мира, который называется *проектирование по контракту*.

По закону договор является юридически обязывающим соглашением между двумя (или более) сторонами, в котором описывается, что каждая сторона обещает делать или не делать. Каждое из этих обещаний приносит пользу другому.

В подходе «проектирование по контракту» модули (называемые «методами» в объектно-ориентированной парадигме, однако «модуль» – более общий термин) определяются в терминах предусловий и постусловий. Постусловия определяют, что модуль обещает сделать (вычислить значение, открыть файл, распечатать отчет, обновить запись в базе данных, изменить состояние системы и т. д.). Предварительные условия определяют, что требуется этому модулю, чтобы он мог соответствовать своим постусловиям. Например, если у нас был модуль с именем openFile, что он обещает сделать? Открыть файл. Каковы должны быть предусловия для openFile?

Во-первых, файл должен существовать; во-вторых, мы должны предоставить имя (или другую идентификационную информацию) файла; в-третьих, файл должен быть «открываемым», т. е. он уже не может быть открыт исключительно другим процессом; в-четвертых, мы должны иметь права доступа к файлу и т. д. Предварительные условия и постусловия устанавливают договор между этим модулем и другими модулями, которые его вызывают.

Тестирование по контракту основано на философии проектирования по контракту. Его подход заключается в создании тест-кейсов только для ситуаций, в которых выполняются предварительные условия. Например, мы бы не смогли протестировать модуль openFile, если файл не существует. Причина проста. Если файл не существует, openFile не обещает работать. Если нет требования работы в определенных условиях, то нет необходимости проводить тестирование в соответствии с этим условием.

В этот момент тестеры обычно протестуют. Да, они согласны, модуль не претендует на работу в этом случае, но что, если предварительные условия нарушаются во время работы? Что делает система? Получаем ли мы сообщение с ошибкой? Другой подход к разработке – это защитный дизайн. В этом случае модуль предназначен для приема любых данных. Если предварительные условия выполнены, модуль достигнет своих обычных постусловий. Если предварительные условия не выполняются, модуль уведомит вызывающего абонента, вернув код ошибки или выдав исключение (в зависимости от используемого языка программирования). Это уведомление на самом деле является еще одним из постусловий модуля. Основываясь на этом подходе, мы могли бы определить защитное тестирование – подход, который проводит тестирование как в нормальных, так и в ненормальных условиях.

Как это относится к тестированию классов эквивалентности? Нужно ли проводить тестирование с такими данными, как –42, FRED и & $ #! @? Если мы используем проектирование по контракту и тестирование по контракту, ответ – нет. Если мы используем защитный дизайн и, следовательно, защитное тестирование, ответ – да. Спросите своих разработчиков, какой подход они используют. Если они отвечают либо «контракт», либо «защита», вы знаете, какой стиль тестирования использовать. Если они ответят «Что?» это означает, что они не думают о том, как модули взаимодействуют. Они не думают о предварительных и постусловных договорах. Вы должны ожидать, что интеграционное тестирование будет основным источником дефектов, которые будут более сложными, и займет больше времени, чем предполагалось.

Технология тестирования

Этапы для тестирования классов эквивалентности просты. Во-первых, сначала определите классы эквивалентности. Во-вторых, создайте контрольный пример для каждого класса эквивалентности. Можно создать дополнительные тест-кейсы для каждого класса эквивалентности, если у вас есть время и деньги. Дополнительные тест-кейсы могут заставить вас чувствовать себя более уверенно, но они редко обнаруживают дефекты, которые не обнаруживают вышеуказанные тест-кейсы.

Для разных типов данных требуются разные типы классов эквивалентности. Давайте рассмотрим четыре варианта. Предположим, что философия защитного тестирования заключается в проверке как

действительного, так и недействительного ввода. Тестирование неверных входных данных часто является серьезным источником дефектов. Если данные являются непрерывным диапазоном значений, то обычно существует один класс допустимых значений и два класса недопустимых значений, один ниже действительного класса и один выше него. Рассмотрим ипотечную компанию. Они дают ипотечные кредиты для людей с доходом от 1 000 до 83 333 долл. США в месяц. Все, у кого доход ниже 1 000 долл. США в месяц не имеют права получить ипотеку. Все, у кого доход больше 83,333 долл. США в месяц не нужно обращаться в эту компанию, просто можно оплатить наличными. Для проверки корректных значений возьмем доход 1 342 долл. США в месяц. Для некорректных значений выберем 123 долл. США в месяц и 90 000 долл. США в месяц (рис. 4.1).

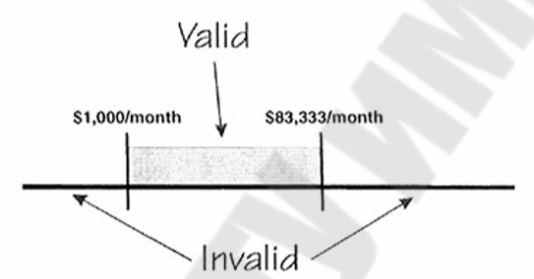

Рис. 4.1. Пример значений для класса непрерывной эквивалентности

Классы дискретной эквивалентности – если входное условие принимает дискретные значения в диапазоне допустимых значений, обычно существует один действительный и два недопустимых класса. Ипотечная компания выдает единую ипотеку на покупку от одного до пяти домов. Нуль или меньше домов не является корректными данными, равно как и шесть или больше. Также нет дробных или десятичных значений, таких как 2 1/2 или 3,14159 (рис. 4.2).

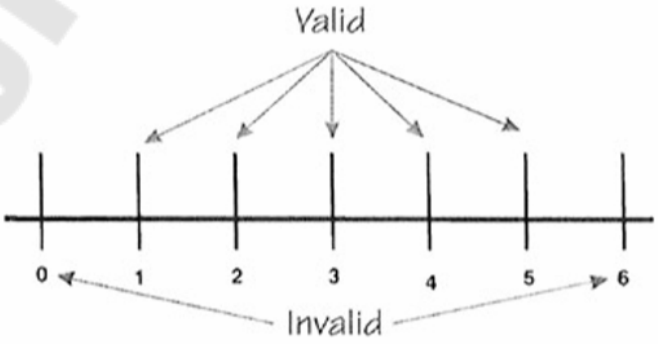

Рис. 4.2. Пример значений класса дискретной эквивалентности

Для правильного ввода мы можем выбрать два дома. Неправильные значения могут быть –2 и 8.

Ипотечная компания будет выдавать ипотечные кредиты только для физических лиц. Они не будут выдавать ипотечные кредиты для корпораций, трастов, партнерств или любого другого юридического лица (рис. 4.3).

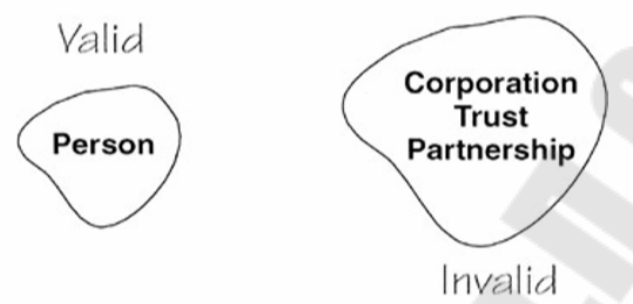

Рис. 4.3. Одиночный выбор для класса эквивалентности

Для правильного ввода мы должны использовать «физическое лицо». Для недействительного мы можем выбрать «корпорация» или «траст» или любую другую случайную текстовую строку. Сколько недействительных случаев мы должны создать? У нас должен быть хотя бы один; мы можем выбрать дополнительные тесты для дополнительной уверенности в себе.

Ипотечная компания будет предоставлять ипотечные кредиты в кондоминиумах, таунхаусах и домах на одну семью. Они не будут делать закладные на дуплексы, мобильные дома, дома на деревьях или любой другой тип жилья (рис. 4.4).

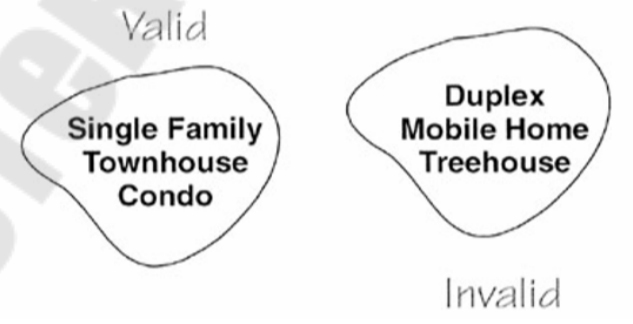

Рис. 4.4. Множественный выбор для класса эквивалентности

Для правильного ввода мы должны выбрать «Кондоминиум», «Таунхаус» или «Дом на одну семью». Хотя правило гласит, что нужно выбрать один контрольный пример из допустимого класса эквива-

лентности, более комплексным подходом будет создание контрольных примеров для каждой записи в допустимом классе. Это имеет смысл, когда список допустимых значений невелик. Но если бы это был список из пятидесяти штатов, округа Колумбия и различных территорий Соединенных Штатов, вы бы проверили каждый из них? Что если бы в списке были все страны мира? Правильный ответ, конечно, зависит от риска для организации, если мы, как тестировщики, упускаем что-то жизненно важное.

Теперь у нас редко будет время для создания отдельных тестов для каждого отдельного класса эквивалентности каждого входного значения, которое входит в нашу систему. Чаще всего мы будем создавать тест-кейсы, которые одновременно проверяют несколько полей ввода. Например, мы могли бы создать один тест-кейс со следующей комбинацией входных данных (табл. 4.1).

Таблица 4.1

Тест-кейс с допустимыми значениями данных

Ежемесячный доход, долл. США	Количество единиц недвижимости	Заявитель	Виды жилья	Результат тестирования
5 000	2	Физическое лицо	Кондоминиум	Правильно

В основном у нас не будет времени для создания отдельных тестов для каждого отдельного класса эквивалентности с каждым входным значением.

Каждое из значений в табл. 4.1 находится в допустимом диапазоне, поэтому мы ожидаем, что система будет работать правильно, а контрольный пример сообщит о прохождении теста.

Заманчиво использовать тот же подход для недопустимых значений. Результат приведен в табл. 4.2.

Таблица 4.2

Тест-кейс со всеми недопустимыми значениями данных

Ежемесячный доход, долл. США	Количество единиц недвижимости	Заявитель	Виды жилья	Результат тестирования
100	8	Товарищество	Дом на дереве	Неверно

Если система принимает эти данные как действительные, очевидно, что система неправильно проверяет четыре поля ввода. Если

система отклоняет эти данные как недействительные, это может быть сделано таким образом, что тестировщик не может определить, какое поле он отклонил.

Например: ОШИБКА: 653X-2.7 НЕПРАВИЛЬНЫЙ ВХОД

Во многих случаях ошибки в одном поле ввода могут отменять или маскировать ошибки в другом поле, поэтому система принимает данные как действительные. Лучшим подходом является проверка одного недопустимого значения за раз, чтобы убедиться, что система правильно его обнаруживает (табл. 4.3).

Таблица 4.3

Тест-кейсы с одним недопустимым значением

Ежемесячный доход, долл. США	**Количество единиц недвижи-мости**	**Заявитель**	**Виды жилья**	**Результат тести-рования**
100	1	Физическое лицо	Дом на одну семью	Неверно
1 342	0	Физическое лицо	Кондоминиум	Неверно
1 342	1	Корпорация	Таунхаус	Неверно
1 342	1	Физическое лицо	Дом на дереве	Неверно

Для дополнительной уверенности входные данные (как действительные, так и недействительные) могут варьироваться.

Набор тест-кейсов, изменяющих недопустимые значения одно за другим, но также варьирующих допустимые значения, приведен в табл. 4.4.

Таблица 4.4

Тест-кейсы с изменяющимися допустимыми и недопустимыми значениями

Ежемесячный доход, долл. США	**Количество единиц недвижи-мости**	**Заявитель**	**Виды жилья**	**Результат тестирования**
100	1	Физическое лицо	Дом на одну семью	Неверно
1 342	0	Физическое лицо	Кондоминиум	Неверно

Окончание табл. 4.4

Ежемесячный доход, долл. США	**Количество единиц недвижимости**	**Заявитель**	**Виды жилья**	**Результат тестирования**
5 432 долл. США	3	Корпорация	Таунхаус	Неверно
10 000 долл. США	2	Физическое лицо	Дом на дереве	Неверно

Другой подход к использованию классов эквивалентности заключается в изучении выходных данных, а не входных данных. Разделите выходы на классы эквивалентности, затем определите, какие входные значения будут вызывать эти выходы. Этот подход имеет то преимущество, что позволяет тестировщику проверять и, следовательно, тестировать каждый вид выходных значений. Но такой подход может быть обманчивым. В предыдущем примере для системы управления персоналом одним из результатов системы было НЕТ, т. е. «Не нанимать». Беглый просмотр входных данных, которые должны вызывать этот вывод, даст {0, 1, ..., 14, 15}. Обратите внимание, что это не полный набор. Кроме того, {55, 56, ..., 98, 99} также должны вызывать выход НЕТ. Важно убедиться, что все потенциальные выходные данные могут быть сгенерированы, но не обманывайте себя при выборе данных класса эквивалентности, в которых пропущены важные входные данные.

Применимость и ограничения

Тестирование классов эквивалентности может значительно сократить количество тестовых случаев, которые должны быть созданы и выполнены. Оно наиболее подходит для систем, в которых большая часть входных данных принимает значения в пределах диапазонов или наборов. Предполагается, что данные в одном и том же классе эквивалентности фактически обрабатываются системой одинаковым образом. Самый простой способ проверить это предположение – спросить программиста об их реализации. Тестирование класса эквивалентности в равной степени применимо на уровне единиц измерения, интеграции, системы и приемочных испытаний. Все, что для этого требуется, – это входы или выходы, которые могут быть разделены на основе требований системы.

Выводы

Тестирование классов эквивалентности – это метод, используемый для сокращения количества тест-кейсов до приемлемого при со-

хранении разумного охвата. Эта простая техника интуитивно используется почти всеми тестировщиками, даже если они могут не знать о ней как о формальном методе разработки тестов. Класс эквивалентности состоит из набора данных, которые обрабатываются модулем одинаково или которые должны давать одинаковый результат. Любое значение данных в классе эквивалентно с точки зрения тестирования любому другому значению.

4.2. Аттестация программного обеспечения на основе тестирования граничных значений

Линн Селла

Принц посмотрел вниз на неподвижную тело Спящей Красавицы, задаваясь вопросом, как ее гибкие губы будут ощущаться своими собственными, и размышляя, достаточно ли силен Алтоид, чтобы противостоять утреннему дыханию, которое может создать столетний сон.

Введение

Тестирование класса эквивалентности является самым основным методом проектирования теста. Это помогает тестировщикам выбирать небольшое подмножество возможных тестовых случаев при сохранении разумного охвата. Тестирование класса эквивалентности имеет также другое преимущество. Это приводит нас к идее тестирования граничных значений – второго ключевого метода проектирования процесса тестирования.

В предыдущей главе были приведены следующие правила, которые указывают, как мы должны обрабатывать заявления о приеме на работу в зависимости от возраста человека. Были приведены следующие правила:

0–16 Не нанимать

16–18 Можно нанимать только на неполный рабочий день

18–55 Можно нанимать в качестве штатного сотрудника

55–99 Не нанимать

Обратите внимание на проблему на границах – «ребрах» каждого класса. Возраст «16» входит в два разных класса эквивалентности (как 18 и 55). Первое правило гласит: не нанимайте 16-летних. Второе правило гласит, что 16-летний подросток может быть принят на работу на неполный рабочий день. Тестирование граничных значений фо-

кусируется на границах просто потому, что именно здесь скрываются многие дефекты. Опытные тестеры сталкивались с такой ситуацией много раз. Неопытные тестировщики могут интуитивно чувствовать, что ошибки будут чаще всего встречаться на границах. Эти дефекты могут быть в требованиях или в коде, как показано ниже:

```
If (applicantAge >= 0 && applicantAge <=16)
hireStatus="NO";
If (applicantAge >= 16 && applicantAge <=18)
hireStatus="PART";
If (applicantAge >= 18 && applicantAge <=55)
hireStatus="FULL";
If (applicantAge >= 55 && applicantAge <=99)
hireStatus="NO";
```

Конечно, ошибка, которую делают программисты, заключается в неправильном кодировании неравенства. Запись > (больше чем) вместо ≥ (больше или равно) является таким примером.

Наиболее эффективный способ обнаружения таких дефектов в требованиях или в коде – это проверка. Однако независимо от того, насколько эффективны наши проверки, мы захотим протестировать код, чтобы убедиться в его правильности. Возможно эти действия хотела получить рассматриваемая организация:

0–15 Не нанимать

16–17 Можно нанимать только на неполный рабочий день

18–54 Можно нанимать сотрудников на полную ставку

55–99 Не нанимать

А как насчет возраста –3 и 101? Обратите внимание, что требования не определяют, как эти значения должны рассматриваться. Мы могли бы догадаться, но «угадывание требований» не является приемлемой практикой. Код, который реализует исправленные правила:

```
If (applicantAge >= 0 && applicantAge <=15)
hireStatus="NO";
If (applicantAge >= 16 && applicantAge <=17)
hireStatus="PART";
If (applicantAge >= 18 && applicantAge <=54)
hireStatus="FULL";
If (applicantAge >= 55 && applicantAge <=99)
```

hireStatus="NO";

Интересные значения на границах или вблизи границ в этом примере: {–1, 0, 1}, {15, 16, 17}, {17, 18, 19}, {54, 55, 56} и {98, 99, 100}. Другие значения, такие как {–42, 1001, FRED,% $ # @}, могут быть включены в зависимости от задокументированных предварительных условий модуля.

Технология тестирования

Этапы для использования граничного тестирования просты. Сначала определите классы эквивалентности. Во-вторых, определите границы каждого класса эквивалентности. В-третьих, создайте контрольные примеры для каждого граничного значения, выбрав одну точку на границе, одну точку чуть ниже границы и одну точку чуть выше границы. «Ниже» и «выше» являются относительными терминами и зависят от единиц измерения данных. Если граница равна 16, а единица измерения является «целым числом», то точка «ниже» равна 15, а точка «выше» равна 17. Если граница равна 5,00 долл. США, а единица измерения – «доллары США и центы», то нижняя точка равна 4,99 долл. США. и вышеуказаный пункт составляет 5,01 долл. США. С другой стороны, если значение равно 5 долл. США, а единица измерения – «доллары США», то нижняя точка равна 4 долл. США, а вышеуказанная точка равна 6 долл. США.

Создайте контрольные примеры для каждого значения границы, выбрав одну точку на границе, одну точку чуть ниже границы и одну точку чуть выше границы.

Обратите внимание, что точка чуть выше одной границы может находиться в другом классе эквивалентности. Нет причин дублировать тест. То же самое можно сказать и о точке чуть ниже границы. Конечно, вы можете создать дополнительные тест-кейсы и дальше от границ (в пределах классов эквивалентности), если у вас есть ресурсы. Как обсуждалось в предыдущей главе, эти дополнительные тестовые примеры могут вызвать у вас ощущение уверенности, но они редко обнаруживают дополнительные дефекты. Тестирование граничных значений является наиболее подходящим, когда входными данными является непрерывный диапазон значений. Возвращаясь снова к ипотечной компании, каковы интересные граничные значения? Для ежемесячного дохода границы составляют 1 000 долл. США в месяц и 83 333 долл. США в месяц (при условии, что единицами являются доллары США) (рис. 4.5).

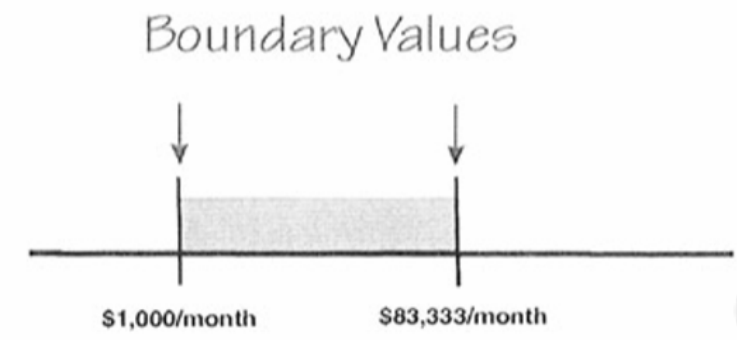

Рис. 4.5. Граничные значения для класса непрерывной эквивалентности

Ввод тестовых данных {$ 999, $ 1 000, $ 1,001} на нижнем уровне и {83,332, $ 83,333, $ 83,334} на верхнем уровне выбирается для проверки границ. Поскольку ипотечная компания будет выписывать ипотеку на количество от одного до пяти домов, нуль или меньше домов не является правильным значением, равно как и шесть или больше. Эти значения определяют границы для тестирования (рис. 4.6).

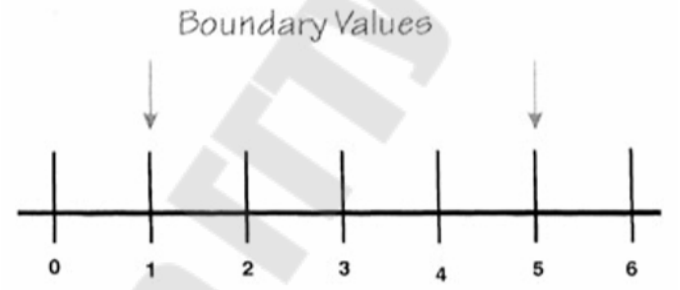

Рис. 4.6. Граничные значения для класса дискретной эквивалентности

Изредка у нас будет время для создания индивидуальных тестов для каждого граничного значения каждого входного значения, которое входит в нашу систему. Чаще всего мы будем создавать контрольные примеры, которые одновременно проверяют несколько полей ввода (табл. 4.5).

Таблица 4.5

Тест-кейсы, содержащие комбинацию допустимых значений (на границах) и недопустимых значений (за границами)

Ежемесячный доход, долл. США	Количество единиц недвижимости	Результат тестирования	Описание
1 000	1	Верно	Минимальный доход, минимальное количество домов
83 333	1	Верно	Максимальный доход, минимальное количество домов
1 000	5	Верно	Минимальный доход, максимальное количество домов
83 333	5	Верно	Максимальный доход, максимальное количество домов
1 000	0	Неверно	Минимальный доход, ниже минимального количества домов
1 000	6	Неверно	Минимальный доход, выше максимального количества домов
83 333	0	Неверно	Максимальный доход, ниже минимального количества домов
83 333	6	Неверно	Максимальный доход, выше максимального количества домов
999	1	Неверно	Ниже минимального дохода, минимальное количество домов
83 334	1	Неверно	Выше максимального дохода, минимальное количество домов
999	5	Неверно	Ниже минимального дохода, максимальное количество домов
83 334	5	Неверно	Выше максимального дохода, максимальное количество домов

График «ежемесячный доход» на оси *x* и «количество жилищ» на оси *y* показывает «местоположения» точек тестовых данных (рис. 4.7).

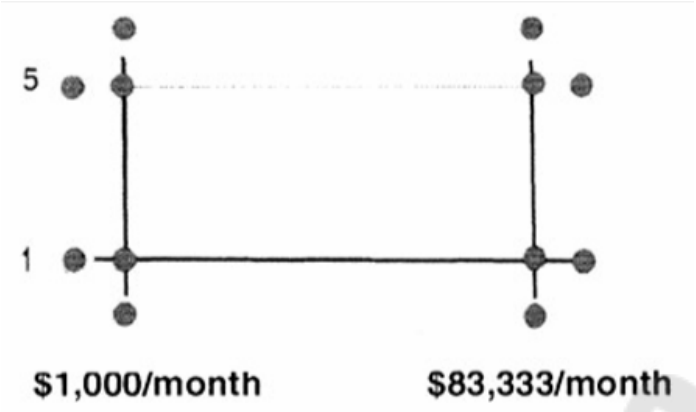

Рис. 4.7. Точки данных на границах и точки данных за пределами границ

Обратите внимание, что четыре из входных комбинаций находятся на границах, а восемь – только снаружи. Также обратите внимание, что точки снаружи всегда объединяют одно действительное значение с одним недопустимым значением (только на одну единицу ниже или на одну единицу выше).

Применимость и ограничения

Тестирование граничных значений может значительно сократить количество тест-кейсов, которые должны быть созданы и выполнены. Оно наиболее подходит для систем, в которых большая часть входных данных принимает значения в пределах диапазонов или наборов. Граничное тестирование в равной степени применимо на уровне единиц, интеграции, системы и приемочных испытаний. Все, что для этого требуется, – это входные данные, которые можно разделить, и границы, которые можно определить на основе требований системы.

Выводы

Несмотря на полезность тестирования классов эквивалентности, его основной вклад заключается в том, чтобы привести нас к проверке граничных значений.

Тестирование граничных значений – это метод, используемый для уменьшения количества тестовых случаев до приемлемого размера при сохранении разумного охвата.

Тестирование граничных значений фокусируется на границах, потому что именно здесь скрываются многие дефекты. Опытные тестеры сталкивались с такой ситуацией много раз. Неопытные тестировщики могут интуитивно чувствовать, что ошибки будут чаще всего встречаться на границах.

Создайте контрольные примеры для каждого значения границы, выбрав одну точку на границе, одну точку чуть ниже границы и одну

точку чуть выше границы. «Ниже» и «выше» являются относительными терминами и зависят от единиц значения данных.

4.3. Аттестация программного обеспечения на основе тестирования таблицы решений

Крис Эско

Я наткнулся на раскрытие своего первого дела об убийстве, обнаружив себя единственным свидетелем, но как бы я отчаянно ни умолял Джона Лоу, что преступник был прямо перед ними – та самая дама, которую они допрашивали – знойная, но изворотливая мисс Китвинкль, которая играла скорбную простушку, как пианистка, играющая на концерте. Полицейские продолжая улыбаться тщательно проводили допрос.

Введение

Таблицы решений являются отличным инструментом для сбора системных требований и для документирования внутреннего дизайна системы. Они используются для записи сложных бизнес-правил, которые должна использовать система. Кроме того, они могут служить руководством для создания тест-кейсов. Таблицы решений являются жизненно важным инструментом в личном наборе инструментов тестера. К сожалению, многие аналитики, разработчики и тестировщики не знакомы с этой техникой.

Технология тестирования

Таблицы решений представляют сложные бизнес-правила, основанные на наборе условий. Общая форма представлена в виде табл. 4.6.

Таблица 4.6

Общая форма таблицы решений

Исходные данные	**Правило 1**	**Правило 2**	**…**	**Правило *p***
Условия				
Условие 1				
Условие 2				
…				
Условие *m*				
Действия				
Действие 1				
Действие 2				
…				
Действие *n*				

Условия с 1 по m представляют различные исходные условия. Действия с 1 по *n* – это действия, которые необходимо выполнить в зависимости от различных комбинаций условий ввода. Каждое из правил определяет уникальную комбинацию условий, которые приводят к выполнению («запуску») действий, связанных с этим правилом. Обратите внимание, что действия не зависят от порядка, в котором оцениваются условия, а только от их значений. (Предполагается, что все значения доступны одновременно.) Кроме того, действия зависят только от указанных условий, а не от каких-либо предыдущих условий ввода или состояния системы. Возможно, конкретный пример прояснит понятия. Компания автострахования предоставляет скидки водителям, которые состоят в браке и (или) хорошо учатся. Начнем с условий. В следующей таблице решений есть два условия, каждое из которых принимает значения «Да» или «Нет».

Таблица 4.7

Таблица решений с двумя двоичными условиями

Исходные данные	Правило 1	Правило 2	Правило 3	Правило 4
Условия				
Женат?	Да	Да	Нет	Нет
Хорошо учится?	Да	Нет	Да	Нет

Обратите внимание, что таблица содержит все комбинации условий. Учитывая два бинарных условия («Да» или «Нет»), возможные комбинации: {Да, Да}, {Да, Нет}, {Нет, Да} и {Нет, Нет}. Каждое правило представляет одну из этих комбинаций. В качестве тестировщика мы проверим, что все комбинации условий определены. Пропуск комбинации может привести к разработке системы, которая может не обрабатывать определенный набор входов должным образом. Теперь о действиях. Каждое правило вызывает действие. Каждое правило может указывать действие, уникальное для этого правила, или правила могут совместно использовать действия.

Таблица 4.8

Добавление одного действия в таблицу решений

Исходные данные	Правило 1	Правило 2	Правило 3	Правило 4
Условия				
Женат?	Да	Да	Нет	Нет
Хорошо учится?	Да	Нет	Да	Нет
Действия				
Скидка, долл. США	60	25	50	0

Таблицы решений могут указывать более одного действия для каждого правила. Эти правила могут быть уникальными или могут быть общими.

Таблица 4.9

Таблица решений с несколькими действиями

Исходын данные	**Правило 1**	**Правило 2**	**Правило 3**	**Правило 4**
Условия				
Женат?	Да	Да	Нет	Нет
Хорошо учится?	Да	Нет	Да	Нет
Действия				
Действие 1	Делать *X*	Делать *Y*	Делать *X*	Делать *Z*
Действие 2	Делать *A*	Делать *B*	Делать *B*	Делать *B*

В этой ситуации выбор тест-кейса прост – каждое правило (вертикальный столбец) становится тестовым. Условия определяют входные данные, а действия определяют ожидаемые результаты. Условия могут быть более сложными, хотя предыдущий пример использует простые двоичные условия.

Таблица 4.10

Таблица решений с недвоичными условиями

Исходные данные	**Правило 1**	**Правило 2**	**Правило 3**	**Правило 4**
Условия				
Условие 1	0–1	1–10	10–100	100–1000
Условие 2	< 5	5	6 или 7	> 7
Действия				
Действие 1	Делать *X*	Делать *Y*	Делать *X*	Делать *Z*
Действие 2	Делать *A*	Делать *B*	Делать *B*	Делать *B*

В этой ситуации выбор тест-кейсов немного сложнее – каждое правило (вертикальный столбец) становится тестовым примером, но должны быть выбраны значения, удовлетворяющие условиям. Выбирая подходящие значения, мы создаем следующие тест-кейсы.

Таблица 4.11

Примеры тест-кейсов

Тест кейс ID	Условие 1	Условие 2	Ожидаемый результат
ТК1	0	3	Делать *X* / Делать *A*
ТК2	5	5	Делать *Y* / Делать *B*
ТК3	50	7	Делать *X* / Делать *B*
ТК4	500	10	Делать *Z* / Делать *B*

Если тестируемая система имеет сложные бизнес-правила и если бизнес-аналитики или разработчики не задокументировали эти правила в этой форме, тестировщики должны собрать эту информацию и представить ее в форме таблицы решений. Причина проста. Учитывая поведение системы, представленное в этой полной и компактной форме, тест-кейсы могут быть созданы непосредственно из таблицы решений. При тестировании необходимо создать хотя бы один тестовый пример для каждого правила. Если условия правила являются двоичными, то, вероятно, достаточно одного теста для каждой комбинации. С другой стороны, если условие представляет собой диапазон значений, рассмотрите возможность тестирования как в нижней, так и в верхней части диапазона. Таким образом, мы объединяем идеи тестирования граничных значений с тестированием таблиц решений.

Создайте хотя бы один тест-кейс для каждого правила.

Чтобы создать таблицу тест-кейсов, необходимо изменить заголовки строк и столбцов.

Таблица 4.12

Таблица решений, преобразованная в таблицу контрольных примеров

Исходные данные	Тест-кейс 1	Тест-кейс 2	Тест-кейс 3	Тест-кейс 4
Условия				
Условие 1	Да	Да	Нет	Нет
Условие 2	Да	Нет	Да	Нет
Действия				
Действие 1	Делать *X*	Делать *Y*	Делать *X*	Делать *Z*
Действие 2	Делать *A*	Делать *B*	Делать *B*	Делать *B*

Применимость и ограничения

Таблицу решений можно использовать всякий раз, когда система должна реализовать сложные бизнес-правила, когда эти правила могут быть представлены в виде комбинации условий и когда с этими условиями связаны дискретные действия.

Выводы

Таблицы решений используются для документирования сложных бизнес-правил, которые должна использовать система. Кроме того, они служат руководством для создания контрольных примеров.

Условия представляют различные условия ввода. Действия – это процессы, которые должны выполняться в зависимости от различных комбинаций условий ввода. Каждое правило определяет уникальную комбинацию условий, которые приводят к выполнению («срабатыванию») действий, связанных с этим правилом.

Создавайте хотя бы один контрольный пример для каждого правила. Если условия правила являются двоичными, то, вероятно, достаточно одного теста для каждой комбинации. С другой стороны, если условие представляет собой диапазон значений, рассмотрите возможность тестирования как в нижней, так и в верхней части диапазона.

Глава 5. Методики обеспечения качества программного обеспечения

5.1. Место инспектирования в процессе разработки программного обеспечения

Инспектирование программ – это просмотр и проверка программ с целью обнаружения в них ошибок. Идея формализованного процесса проверки программ была сформулирована корпорацией IBM в 1970-х гг. В настоящее время данный метод верификации получил широкое распространение. На его базе разработано множество других методов, но все они основываются на базовой идее метода инспектирования, согласно которому группа специалистов выполняет тщательный построчный просмотр и анализ исходного кода программы. Главное отличие инспектирования от других методов оценивания качества программ состоит в том, что его цель – обнаружение дефектов, а не исследование общих проблем проекта. Дефектами являются либо ошибки в исходном коде, либо несоответствия программы стандартам.

Процесс инспектирования – формализованный. В нем принимает участие небольшая группа людей. У каждого в группе есть своя роль. Обязательно должны присутствовать: автор, рецензент, инспектор, координатор. Рецензент «озвучивает» программный код, инспектор проверяет его, координатор отвечает за организацию процесса. По мере накопления опыта инспектирования в организациях могут появляться другие предложения по распределению ролей в группе.

Для начала процесса инспектирования программы необходимы следующие условия:

– наличие точной спецификации кода;

– члены инспекционной группы должны хорошо знать стандарты разработки;

– в распоряжении группы должна быть синтаксически корректная последняя версия программы.

На рис. 5.1 показан общий процесс инспектирования. Он адаптирован к требованиям организаций, использующих инспектирование программ.

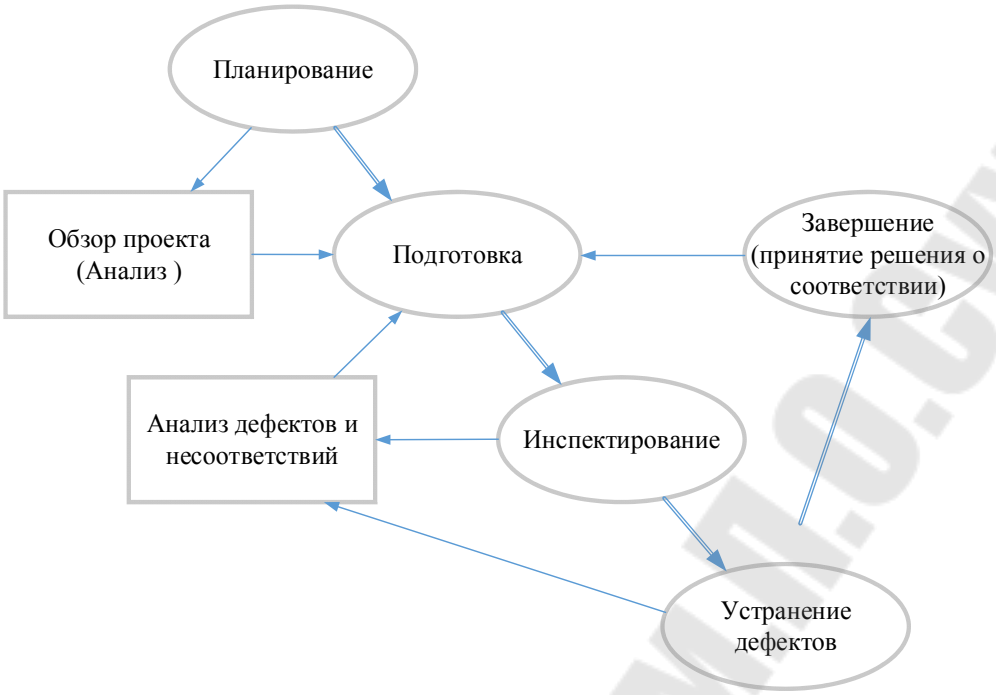

Рис. 5.1. Процесс инспектирования

Сам процесс инспектирования должен быть относительно коротким и сосредоточенным только на выявлении дефектов, аномалий и несоответствий стандартам. Инспекционная группа не должна предлагать способы исправления дефектов или рекомендовать какие-либо изменения в других программных компонентах.

После инспектирования автор изменяет программу, исправляя обнаруженные ошибки. На этапе доработки координатор принимает решение о том, необходимо ли повторное инспектирование. Если повторное инспектирование не требуется, все обнаруженные дефекты фиксируются документально.

В процессе инспектирования организация накапливает определенный опыт, поэтому результаты инспектирования можно использовать для улучшения всего процесса разработки ПО. В ходе инспектирования выполняется анализ обнаруженных дефектов. Группа инспектирования и авторы инспектируемого кода определяют причины возникновения дефектов. Чтобы подобные дефекты не возникали в будущих системах, необходимо по возможности устранить причины возникновения дефектов, что означает внесение изменений в процесс разработки программных систем.

Обеспечение инспектирования ПО требует квалифицированного управления и правильного отношения к результатам его проведения. Инспектирование – открытый процесс обнаружения ошибок, когда

ошибки, допущенные отдельным программистом, становятся известны всей группе программистов. Менеджеры должны четко разграничивать инспектирование программного кода и оценку кадров. При оценке профессиональных качеств ни в коем случае нельзя учитывать ошибки, обнаруженные в процессе инспектирования. Руководителям инспекционных групп необходимо пройти тщательную подготовку, чтобы грамотно управлять процессом и совершенствовать культуру отношений.

5.2. Методы инспектирования программного обеспечения

Верификацией и аттестацией называются процессы проверки и анализа, в ходе которых проверяется соответствие программного обеспечения своей спецификации и требованиям заказчиков. Верификация и аттестация охватывают весь цикл жизни ПО – они начинаются на этапе анализа требований и завершаются проверкой программного кода на этапе тестирования программной системы.

Верификация и аттестация – абсолютно разные понятия, однако часто их путают. Для того чтобы различать их, выведем главное различие между этими терминами. Верификация отвечает на вопрос, правильно ли создана система, а аттестация отвечает на вопрос, правильно ли работает система. Из этого следует, что верификация проверяет соответствие ПО системной спецификации, в частности, функциональным и нефункциональным требованиям. Аттестация – это более общий процесс. Во время аттестации цель инженера – доказать заказчику, что продукт оправдывает ожидания последнего. Аттестация проводится после верификации.

На ранних этапах разработки ПО очень важна аттестация системных требований. В требованиях очень часто встречаются ошибки, недочеты, упущения, что может привести к несоответствию продукта замыслу заказчика. Инженер должен справляться с этой проблемой. Однако, как известно, сложно искоренить все погрешности в требованиях. Отдельные ошибки могут обнаружиться лишь тогда, когда программный продукт реализован.

В процессах верификации и аттестации используются две основные методики проверки и анализа систем: инспектирование ПО и тестирование ПО. Инспектирование ПО подразумевает анализ и проверку различных представлений системы, например, документации.

Инспектирование происходит на всех этапах разработки программной системы. Параллельно с инспектированием может проводиться автоматический анализ исходного кода программ и соответствующих документов. Инспектирование и автоматический анализ – это статические методы верификации и аттестации, поскольку им не требуется исполняемая система. Тестирование ПО есть анализ выходных данных и рабочих характеристик программного продукта для проверки правильности работы системы. Тестирование – динамический метод верификации и аттестации, так как применяется к исполняемой системе.

Согласно этой схеме, инспектирование можно выполнять на всех этапах разработки системы, а тестирование – в тех случаях, когда создан прототип или исполняемая программа (рис. 5.2). Стрелки указывают на те этапы процесса разработки, на которых можно применять данные методы.

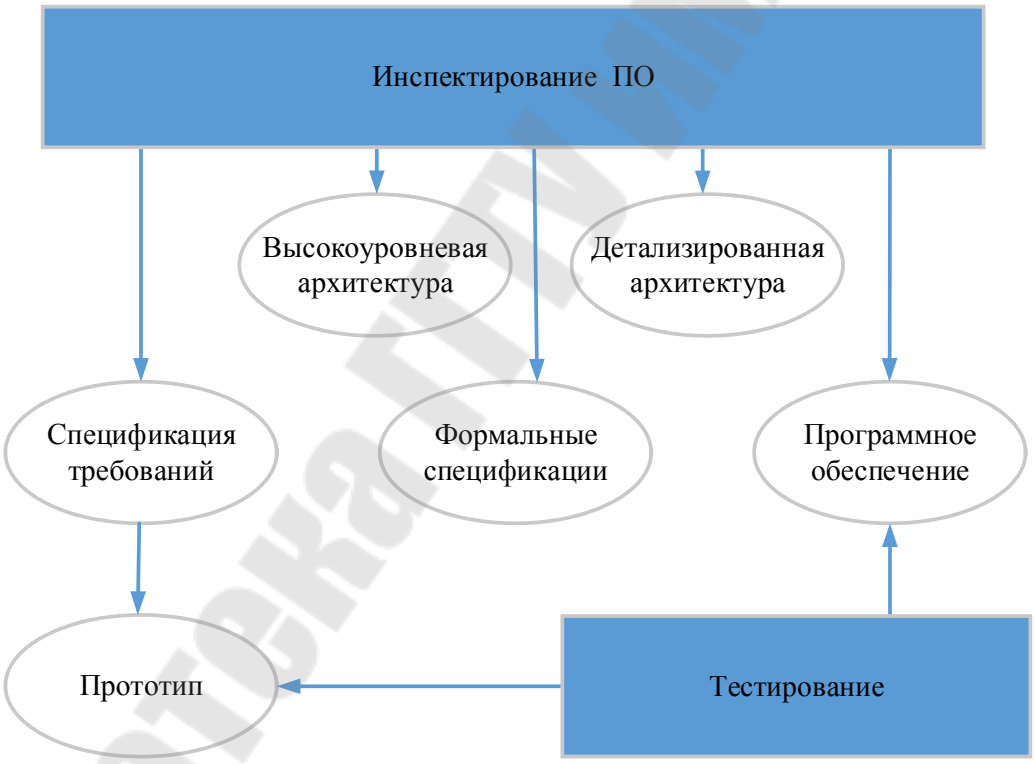

Рис. 5.2. Варианты использования инспектирования и тестирования ПО

К методам инспектирования относятся:
– инспектирование программ;
– автоматический анализ исходного кода;
– формальная верификация.

Однако статическими методами возможно осуществить проверку только на соответствие программ спецификации, с их помощью невозможно выяснить правильность функционирования системы. Кроме того, статическими методами нельзя проверить такие нефункциональные характеристики, как производительность и надежность. Следовательно, для анализа нефункциональных характеристик следует проводить тестирование системы. В настоящее время, несмотря на широкое применение инспектирования ПО, преобладающим методом верификации и аттестации все еще остается тестирование. Тестирование – это проверка работы программ с данными, подобными реальным, которые будут обрабатываться в процессе эксплуатации системы. Неполадки в работе ПО обнаруживаются при анализе выходных данных, среди которых выделяются и исследуются аномальные.

На разных этапах процесса разработки ПО применяют различные виды тестирования. Тестирование дефектов проводится для выявления несоответствий между программным продуктом и его спецификацией, которые обусловлены ошибками в программном коде. Такие тесты разрабатываются для выявления ошибок в системе, а не для имитации ее работы. Статистическое тестирование оценивает производительность и надежность программ, а также работу программы при использовании различных режимов ее эксплуатации. Тесты разрабатываются с целью имитирования, причем имитируется реальная работа системы с реальными выходными данными. Надежность функционирования системы определяется по количеству сбоев, отмеченных в работе программ. Производительность оценивается по результатам измерений полного времени выполнения операций и времени отклика системы при обработке тестовых данных. Конечно, между этими двумя методами тестирования нет четких границ. Во время тестирования испытатель может получить интуитивное представление о надежности ПО, а во время статистического тестирования есть возможность выявления программных дефектов.

Главная цель верификации и аттестации – удостовериться в том, что система «соответствует своему назначению». Соответствие программной системы своему назначению отнюдь не предполагает, что в ней совершенно не должно быть ошибок. Скорее, система должна хорошо соответствовать тем целям, для которых она планировалась. Уровень необходимой достоверности соответствия определяется:

– назначением системы,

– ожиданиями пользователей,

– условиями на рынке программных продуктов.

Назначение ПО. Уровень достоверности соответствия зависит от важности разрабатываемого программного продукта по каким-либо критериям. Можно привести пример систем малой критичности. Это, в частности, опытные образцы программных систем, разрабатываемые для демонстрации некоторых новых идей.

Ожидания пользователей. Здесь следует отметить, что у большинства пользователей в настоящее время очень низкие требования к ПО. Это связано с тем, что пользователи привыкли к разнообразным сбоям, которые происходят при работе с ПО, а значит, они не удивляются этому. Однако, все же, в последнее время терпимость пользователей по отношению к ПО снижается. Создание недоброкачественного ПО становится недопустимым. Теперь фирмы, изготовляющие программные продукты, стремятся уделять больше внимания верификации и аттестации ПО.

При проведении верификации и аттестации в системе обнаруживаются ошибки, которые должны исправляться. После исправления ошибок необходимо снова проверить программу. Для этого можно еще раз выполнить инспектирование программы или повторить тестирование. Разработчик должен знать, что простых методов исправления ошибок в программах не существует. Повторное тестирование необходимо проводить для того, чтобы убедиться, что сделанные в программе изменения не внесли в систему новых ошибок, поскольку на практике высокий процент «исправления ошибок» либо не завершается полностью, либо вносит новые ошибки в программу. При разработке крупных систем каждое повторное тестирование всей системы обходится очень дорого; при этом для экономии средств определяют связи и зависимости между частями системы и проводят тестирование именно этих отдельных частей.

5.3. Планирование верификации и аттестации

Верификация и аттестация – дорогостоящие процессы. Для сложных систем, например, характерно такое соотношение: половина всего бюджета, выделенного на реализацию системы, тратится на верификацию и аттестацию. Планирование верификации и аттестации должно начинаться как можно раньше. На рис. 5.3 показана модель разработки ПО, учитывающая процесс планирования испытаний. Из рисунка видно, что процесс верификации и аттестации разделяется на несколько этапов, причем на каждом этапе проводится определенный тест.

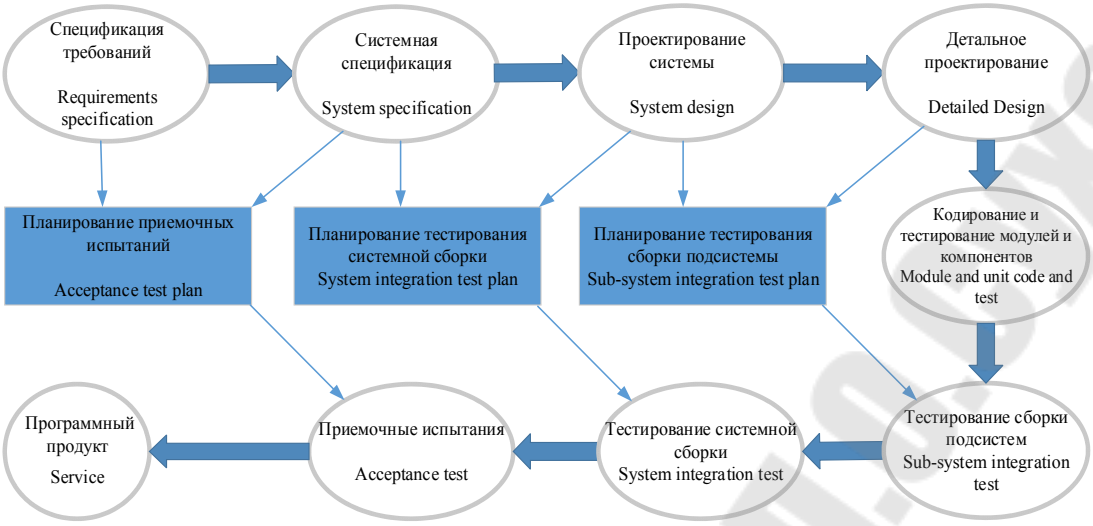

Рис. 5.3. Планирование испытаний в процессе разработки и тестирования

В процессе планирования верификации и аттестации необходимо определить соотношение между статическими и динамическими методами проверки системы, определить стандарты и процедуры инспектирования, составить план тестирования программ. От типа разрабатываемой системы зависит то, чему следует уделить больше внимания – инспектированию или тестированию. Чем более критична система, тем больше внимания следует уделять статическим методам верификации.

План испытаний ПО обязательно должен включать в себя:

– описание основных этапов процесса тестирования;

– возможность отслеживания требований (тестирование следует спланировать так, чтобы протестировать все требования в отдельности);

– тестируемые элементы (следует определить все «выходные» продукты процесса разработки ПО, которые необходимо тестировать);

– график тестирования (составляется временной график тестирования и распределение ресурсов проводится согласно этому графику, причем график тестирования привязан к более общему графику разработки проекта);

– процедуры записи тестов тестов (для проверки правильности выполнения тестов);

– аппаратные и программные требования;

– ограничения (попытаться предвидеть все неблагоприятные факторы, влияющие на процессы тестирования, например, нехватку средств, персонала и т. п.).

Подобно другим планам, план испытаний не является неизменным документом. Его следует регулярно пересматривать, так как тестирование зависит от процесса реализации системы.

Системное тестирование программ требует разработки огромного количества тестов, их выполнения и проверки. Это значит, что данный процесс достаточно трудоемкий и дорогостоящий. Каждый тест способен обнаружить в программе одну, реже несколько ошибок. Причина такого положения заключается в том, что сбои в работе, происходящие из-за ошибок в системе, часто приводят к разрушению данных. Поэтому сложно сказать, какое количество ошибок «ответственно» за сбой в системе.

Инспектирование программ не требует от последних быть завершенными, поэтому инспектировать можно даже на начальных стадиях разработки. Во время инспектирования проверяется исходное представление системы. Это может быть модель системы, спецификация или программа, написанная на языке высокого уровня. Обнаружение ошибок достигается путем использования знаний рассматриваемой системы и семантики ее исходного представления. Каждую ошибку можно рассмотреть отдельно, не обращая внимания на то, как она влияет на поведение системы.

Доказано, что инспектирование является эффективным методом обнаружения ошибок, причем оно значительно дешевле экстенсивного тестирования. Инспектированием можно обнаружить более 60 % всех ошибок, а при более формальном подходе – более 90 %. Процесс инспектирования также может оценить другие качественные характеристики систем: соответствие стандартам, переносимость и удобства сопровождения.

В системных компонентах выявление ошибок путем инспектирования более эффективно, чем путем тестирования. Во-первых, за один сеанс инспектирования можно обнаружить очень многие дефекты программного кода; при применении тестирования за один сеанс обнаруживается обычно лишь одна ошибка, поскольку ошибки могут привести к полному останову системы, а эффекты ошибок могут накладываться друг на друга. Во-вторых, инспектирование использует знание о предметной области и языке программирования. Специалист, проводящий инспектирование, должен знать типы ошибок, что дает возможность сосредоточиться на конкретных видах дефектов.

Понятно, что инспектирование не может заменить тестирование. Инспектирование лучше применять на начальных стадиях для выявления наибольшего количества ошибок. Инспектированием проверяют соответствие ПО его спецификации, но таким способом, например, не-

возможно оценить динамическое поведение системы. Более того, нерационально инспектировать законченные системы, собранные из нескольких подсистем. На этом уровне возможно только тестирование.

Ошибочно полагать, что тестирование и инспектирование являются конкурирующими методами верификации и аттестации. У каждого из них есть свои преимущества и недостатки. Следовательно, в процессе верификации и аттестации инспектирование и тестирование следует использовать совместно.

Иногда при инспектировании в организации возникают трудности. Эксперты, имеющие большой опыт в тестировании программ, неохотно соглашаются с тем, что инспектирование является более эффективным методом устранения дефектов системы, чем тестирование. Менеджеры относятся к этим технологиям с недоверием, потому что внедрение инспектирования требует дополнительных расходов. Конечная экономия средств при применении инспектирования достигается только благодаря опыту проводящих его специалистов.

5.4. Отличие верификации от аттестации

Несмотря на кажущуюся схожесть, термины «верификация» и «аттестация» означают разные уровни проверки корректности работы программной системы. Дабы избежать дальнейшей путаницы, четко определим эти понятия.

Верификация – это проверка соответствия ПО проектной спецификации и стандартам, технической документации, представленной техзаданием, архитектурой или моделью предметной области. Также целью верификации является достижение гарантии того, что верифицируемый объект (требование или программный код) реализован без непредусмотренных функций.

Аттестация программной системы – более общий процесс, целью которого является доказательство того, что в результате разработки системы мы достигли тех целей, которые планировали достичь благодаря ее использованию. Иными словами, аттестация – это проверка соответствия системы ожиданиям заказчика, поэтому она проводится после верификации. Если посмотреть на эти процессы с точки зрения вопроса, на который они дают ответ, верификация отвечает на вопрос «Выполнено ли программное обеспечение правильно?» или «Соответствует ли разработанная система требованиям?», а аттестация – «Правильно ли работает система?» или «Соответствует ли разработанная система ожиданиям заказчика?»

Поскольку основной задачей верификации, как и аттестации, является контроль качества программного обеспечения, необходимо обратиться к этому понятию. Стандарт ISO 9126 [6] предлагает учитывать три разных точки зрения при рассмотрении качества ПО:

– точку зрения разработчиков, которые воспринимают внутреннее качество ПО;

– точку зрения руководства и аттестации ПО на соответствие сформулированным к нему требованиям, в ходе которой определяется внешнее качество ПО;

– точку зрения пользователей, ощущающих качество ПО при использовании.

Во всех трех случаях для описания качества используется многоуровневая модель, состоящая из целей или факторов, атрибутов или критериев и метрик качества. Цели (факторы) позволяют на верхнем уровне определять основные характеристики, которые ПО должно иметь или уже имеет. Каждый фактор состоит из набора атрибутов (критериев), позволяющих качественно описать желаемые или полученные характеристики более детально. Каждый атрибут поддерживается набором метрик, которые позволяют количественно оценивать наличие соответствующей характеристики. Для двух точек зрения – внешнего качества и внутреннего качества – в рамках ISO 9126 предложена модель качества, схематически представленная на рис. 5.4.

Рис. 5.4. Факторы и атрибуты внешнего и внутреннего качества ПО ГОСТ ИСО/МЭК 9126–2001

Рассмотрим характеристики качества ПО более подробно.

Функциональность (Functionality) – определяется способностью ПО решать задачи, которые соответствуют зафиксированным и предполагаемым потребностям пользователя, при заданных условиях использования ПО. То есть эта характеристика отвечает за то, что ПО работает исправно и точно, функционально совместимо, соответствует стандартам отрасли и защищено от несанкционированного доступа.

Надежность (Reliability) – способность ПО выполнять требуемые задачи в обозначенных условиях на протяжении заданного промежутка времени или указанное количество операций. Атрибуты данной характеристики – это завершенность и целостность всей системы, способность самостоятельно и корректно восстанавливаться после сбоев в работе, отказоустойчивость.

Удобство использования (Usability) – возможность легкого понимания, изучения, использования и привлекательности ПО для пользователя.

Производительность (Efficiency) – способность ПО обеспечивать требуемый уровень производительности в соответствии с выделенными ресурсами, временем и другими обозначенными условиями.

Удобство сопровождения (Maintainability) – легкость, с которой ПО может анализироваться, тестироваться, изменяться для исправления дефектов, для реализации новых требований, для облегчения дальнейшего обслуживания и адаптироваться к имеющемуся окружению.

Переносимость или мобильность (Portability) – характеризует ПО с точки зрения легкости его переноса из одного окружения (software/hardware) в другое.

5.5. Характеристики качества программного обеспечения – основные понятия и определения

В многочисленой литературе [6]–[11], посвященной аттестации и верификации ПО, используются различные определения используемых терминов. Например, качество ПО (Software Quality) определяется следующим образом:

- Качество программного обеспечения – это степень, в которой программное обеспечение обладает требуемой комбинацией свойств [IEEE 1061–1998 IEEE Standard for Software Quality Metrics Methodology].

• Качество программного обеспечения – это совокупность характеристик ПО, относящихся к его способности удовлетворять установленные и предполагаемые потребности [ISO 8402:1994 Quality management and quality assurance].

Рассмотрим термины, используемые в данной области.

Обеспечение качества (Quality Assurance – QA) – это совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации ПО информационных систем, предпринимаемых на разных стадиях жизненного цикла ПО, для обеспечения требуемого уровня качества выпускаемого продукта.

Контроль качества (Quality Control – QC) – это совокупность действий, проводимых над продуктом в процессе разработки, для получения информации о его актуальном состоянии в разрезах: «готовность продукта к выпуску», «соответствие зафиксированным требованиям», «соответствие заявленному уровню качества продукта».

Тестирование программного обеспечения (Software Testing) – это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

Верификация (verification) – это процесс оценки системы или ее компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа. То есть выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.

Валидация (validation) – это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе.

В параграфе 5.4 (рис. 5.4) были определены шесть характеристик качества и описана модель процесса оценки программного продукта на основе ИСО/МЭК 9126 «Программная инженерия – Качество продукта». В настоящее время используется более современный стандарт ГОСТ Р ИСО/МЭК 25010–2015 «Информационные технологии. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов», в котором определены те же характеристики качества программного обеспечения с некоторыми поправками:

– область применения моделей качества была расширена, с тем чтобы включить в себя вычислительные системы и качество при использовании с системной точки зрения;

– в качестве характеристики качества при использовании было добавлено «Покрытие контекста» с подхарактеристиками «Полнота контекста» и «Гибкость»;

– как характеристика, а не подхарактеристика функциональности была добавлена «Безопасность», с подхарактеристиками «Конфиденциальность», «Целостность», «Безотказность», «Отслеживаемость» и «Подлинность»;

– была добавлена как характеристика «Совместимость» (включая функциональную совместимость и сосуществование);

– были добавлены следующие подхарактеристики: «Функциональная полнота», «Емкость», «Защищенность от ошибки пользователя», «Доступность», «Готовность», «Модульность» и «Возможность многократного использования»;

– подхарактеристики соответствия были удалены, поскольку они являются в соответствии с законами и правилами частью общих требований к системе, а не частью характеристики качества;

– модели внутреннего и внешнего качества были объединены в составе модели качества продукта;

– там, где это представилось возможным, специфичные для программного обеспечения определения были заменены на универсальные;

– нескольким характеристикам и подхарактеристикам были даны более точные названия.

Данный международный стандарт предназначен для применения в сочетании с другими частями международных стандартов серии SQuaRE (ИСО/МЭК 25000 – ИСО/МЭК 25099) и ИСО/МЭК 14598 до тех пор, пока он не заменен серией международных стандартов ИСО/МЭК 2504n.

На рис. 5.5 показана организация серии международных стандартов SQuaRE, которая представлена семействами стандартов, называемых также разделами.

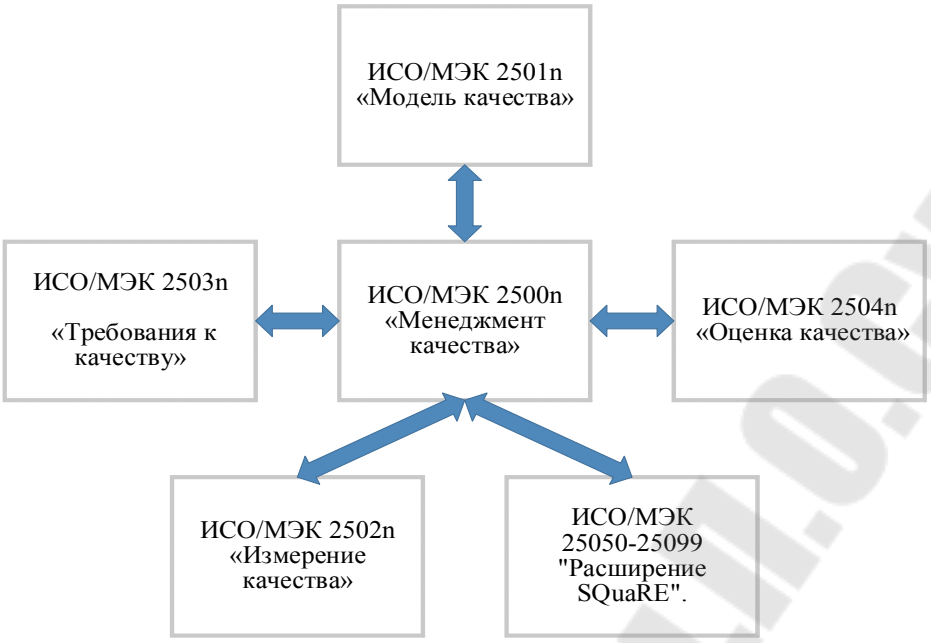

Рис. 5.5. Организация серии международных стандартов SQuaRE

Серия стандартов SQuaRE состоит из следующих разделов стандартов:

– ИСО/МЭК 2500n – раздел «Менеджмент качества». Международные стандарты, входящие в этот раздел, определяют общие модели, термины и определения, используемые далее во всех других международных стандартах серии SQuaRE. В разделе также представлены требования и методические материалы, касающиеся функций поддержки, которые отвечают за управление требованиями к программному продукту, его спецификацией и оценкой;

– ИСО/МЭК 2501n – раздел «Модель качества». Международные стандарты, которые входят в этот раздел, представляют детализированные модели качества вычислительных систем и программного обеспечения, качества при использовании и качества данных. Кроме того, представлено практическое руководство по использованию модели качества;

– ИСО/МЭК 2502n – раздел «Измерение качества». Международные стандарты, входящие в этот раздел, включают в себя эталонную модель измерения качества программного продукта, математические определения показателей качества и практическое руководство по их использованию. В этом разделе представлены показатели внутреннего качества программного обеспечения, показатели внешнего качества программного обеспечения и показатели качества при использовании. Кроме того, определены и представлены элементы по-

казателей качества, формирующие основу для вышеперечисленных показателей;

– ИСО/МЭК 2503n – раздел «Требования к качеству». Международные стандарты, которые входят в этот раздел, определяют требования к качеству на основе моделей качества и показателей качества. Такие требования к качеству могут использоваться в процессе формирования требований к качеству программного продукта перед разработкой или как входные данные для процесса оценки;

– ИСО/МЭК 2504n – раздел «Оценка качества». Международные стандарты, которые входят в этот раздел, формулируют требования, рекомендации и методические материалы для оценки программного продукта, выполняемой как оценщиками, так и заказчиками или разработчиками. Кроме того, в них представлена поддержка документирования показателя измерения как модуля оценки;

– ИСО/МЭК 25050-25099 – раздел «Расширение SQuaRE». Международные стандарты этого раздела в настоящее время включают в себя требования к качеству готового коммерческого (коробочного) программного обеспечения и общему промышленному формату для отчетов по удобству использования.

Модели качества международного стандарта ИСО/МЭК 25010–2015 в сочетании с ИСО/МЭК 12207 и ИСО/МЭК 15288 могут использоваться, в частности, для процессов, связанных с определением требований, для верификации и валидации с особым акцентом на спецификации и оценки требований к качеству. В ИСО/МЭК 25030 определено, каким образом модели качества можно использовать для требований к качеству программного обеспечения, а ИСО/МЭК 25040 описывает применение модели качества в процессе оценки качества программного обеспечения.

В сочетании с ИСО/МЭК 15504, который относится к оценке процессов программного обеспечения, настоящий международный стандарт обеспечивает:

– основы определения качества программного продукта в процессах «поставщик–потребитель»;

– поддержку анализа, верификации и валидации и основы количественной оценки качества в процессах поддержки;

– поддержку настройки целей качества в процессе управления организацией.

Стандарт ИСО/МЭК 25010–2015 может быть использован в сочетании с ИСО 9001, который посвящен процессам обеспечения качества, для обеспечения:

– поддержки определения цели качества;
– поддержки анализа, верификации и валидации проекта.

5.6. Модели качества программного обеспечения в соответствии с ИСО/МЭК 25010–2015

К настоящему времени в серии SQuaRE имеются три модели качества: модель качества при использовании, модель качества продукта и модель качества данных, определенная в ИСО/МЭК 25012. Совместное использование моделей качества дает основание считать, что учтены все характеристики качества. Данные модели обеспечивают множество характеристик качества, в которых заинтересован широкий круг лиц, таких как: разработчики программного обеспечения, системные интеграторы, приобретатели, владельцы, специалисты по обслуживанию, подрядчики, профессионалы обеспечения и управления качеством и пользователи.

5.6.1. Модель качества при использовании

Модель качества при использовании определяет пять характеристик, связанных с результатами взаимодействия с системой (рис. 5.6):
– результативность;
– производительность;
– удовлетворенность;
– свободу от риска;
– покрытие контекста.

Каждая характеристика применима для различных видов деятельности заинтересованных лиц, например, для взаимодействия оператора или поддержки разработчика.

Качество при использовании системы характеризует воздействие продукции (система или программный продукт) на заинтересованную сторону. Оно определяется качествами программного обеспечения, аппаратных средств, операционной среды, а также характеристиками пользователей, задач и социальной среды. Все эти факторы вносят свой вклад в качество системы при использовании.

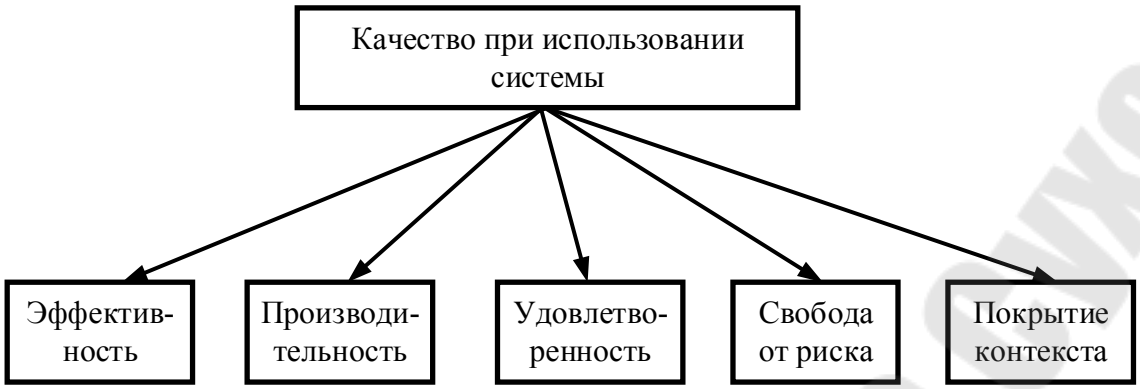

Рис. 5.6. Модель качества при использовании

Во избежание противоречий термин «удобство использования» определен как подмножество качества при использовании, в состав которого входят эффективность, производительность и удовлетворенность. Таким образом, эффективность определяется как результативность (effectiveness), т. е. точность и полнота, с которой пользователи достигают определенных целей и как производительность (efficiency), т. е. как связь точности и полноты достижения пользователями целей с израсходованными ресурсами, которые могут включать в себя время выполнения задачи (человеческие ресурсы), материалы или финансовые затраты на использование.

Удовлетворенность (satisfaction) подразумевает способность продукта или системы удовлетворить требованиям пользователя в заданном контексте использования и включает в себя следующие подхарактеристики:

– *полноценность* (usefulness): степень удовлетворенности пользователя достижением прагматических целей, включая результаты использования и последствия использования;

– *доверие* (trust): степень уверенности пользователя или другого заинтересованного лица в том, что продукт или система будут выполнять свои функции так, как это предполагалось;

– *удовольствие* (pleasure): степень удовольствия пользователя от удовлетворения персональных требований;

– *комфорт* (comfort): степень удовлетворенности пользователя физическим комфортом.

Свобода от риска (freedom from risk): способность продукта или системы смягчать потенциальный риск для экономического положения, жизни, здоровья или окружающей среды, включает в себя следующие подхарактеристики:

– *смягчение отрицательных последствий экономического риска* (economic risk mitigation): способность продукта или системы смягчать потенциальный риск для финансового положения и эффективной работы, коммерческой недвижимости, репутации или других ресурсов в предполагаемых условиях использования;

– *смягчение отрицательных последствий риска для здоровья и безопасности* (health and safety risk mitigation): способность продукта или системы смягчать потенциальный риск для людей в предполагаемых условиях использования;

– *смягчение отрицательных последствий экологического риска* (environmental risk mitigation): способность продукта или системы смягчать потенциальный риск для имущества или окружающей среды в предполагаемых условиях использования.

Покрытие контекста (context coverage): степень, в которой продукт или система могут быть использованы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями как в первоначально определенных условиях использования, так и в условиях, выходящих за спецификации:

– *полнота контекста* (context completeness): степень, в которой продукт или система могут быть использованы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями при всех указанных условиях использования;

– *гибкость* (flexibility): степень, в которой продукт или система могут быть использованы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями в условиях, выходящих за рамки первоначально определенных в требованиях.

5.6.2. Модель качества продукта

Модель качества продукта сводит свойства качества системы/программного продукта к восьми характеристикам, которыми являются (рис. 5.7): функциональная пригодность, уровень производительности, совместимость, удобство пользования, надежность, защищенность, сопровождаемость и переносимость (мобильность). Каждая характеристика, в свою очередь, состоит из ряда соответствующих подхарактеристик.

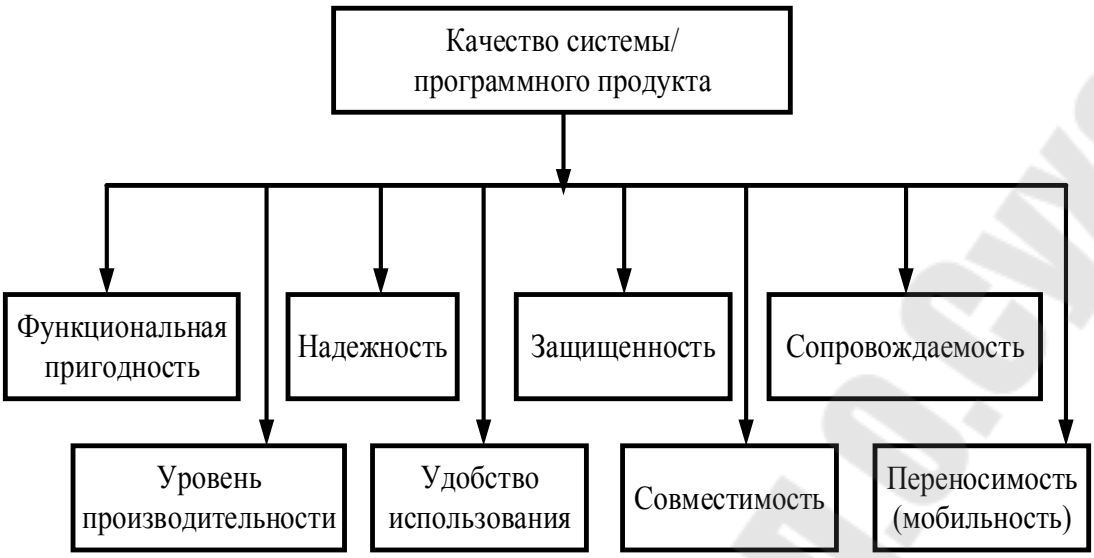

Рис. 5.7. Модель качества продукта

Модель качества продукта можно применять как для программного продукта, так и для компьютерной системы, в состав которой входит ПО, поскольку большинство подхарактеристик применимо и к ПО, и к системам.

Функциональная пригодность (functional suitability): степень, в которой продукт или система обеспечивают выполнение функции в соответствии с заявленными и подразумеваемыми потребностями при использовании в указанных условиях. Включает в себя следующие подхарактеристики:

– *функциональная полнота* (functional completeness): степень покрытия совокупностью функций всех определенных задач и целей пользователя;

– *функциональная корректность* (functional correctness): степень обеспечения продуктом или системой необходимой степени точности корректных результатов;

– *функциональная целесообразность* (functional appropriateness): степень функционального упрощения выполнения определенных задач и достижения целей.

Уровень производительности (performance efficiency): производительность относительно суммы использованных при определенных условиях ресурсов. Включает в себя следующие подхарактеристики:

– *временные характеристики* (time behaviour): степень соответствия требованиям по времени отклика, времени обработки и показателей пропускной способности продукта или системы;

– *использование ресурсов* (resource utilization): степень удовлетворения требований по потреблению объемов и видов ресурсов продуктом или системой при выполнении их функций;

– *потенциальные возможности* (capacity): степень соответствия требованиям предельных значений параметров продукта или системы.

Совместимость (compatibility): способность продукта, системы или компонента обмениваться информацией с другими продуктами, системами или компонентами и(или) выполнять требуемые функции при совместном использовании одних и тех же аппаратных средств или программной среды. Включает в себя следующие подхарактеристики:

– *сосуществование* (совместимость) (co-existence): способность продукта совместно функционировать с другими независимыми продуктами в общей среде с разделением общих ресурсов и без отрицательного влияния на любой другой продукт;

– *функциональная совместимость* (интероперабельность) (inte-roperability): способность двух или более систем, продуктов или компонент обмениваться информацией и использовать такую информацию.

Удобство использования (usability): степень, в которой продукт или система могут быть использованы определенными пользователями для достижения конкретных целей с эффективностью, результативностью и удовлетворенностью в заданном контексте использования. Включает в себя следующие подхарактеристики:

– *определимость пригодности* (appropriateness recognizability): возможность пользователей понять, подходит ли продукт или система для их потребностей, сравним ли с функциональной целесообразностью (functional appropriateness);

– *изучаемость* (learnability): возможность использования продукта или системы определенными пользователями для достижения конкретных целей обучения для эксплуатации продукта или системы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями в указанном контексте использования;

– *управляемость* (operability): наличие в продукте или системе атрибутов, обеспечивающих простое управление и контроль;

– *защищенность от ошибки пользователя* (user error protection): уровень системной защиты пользователей от ошибок;

– *эстетика пользовательского интерфейса* (user interface aesthe-tics): степень «приятности» и «удовлетворенности» пользователя интерфейсом взаимодействия с пользователем;

– *доступность* (accessibility): возможность использования продукта или системы для достижения определенной цели в указанном контексте использования широким кругом людей с самыми разными возможностями.

Надежность (reliability): степень выполнения системой, продуктом или компонентом определенных функций при указанных условиях в течение установленного периода времени. Включает в себя следующие подхарактеристики:

– *завершенность* (maturity): степень соответствия системы, продукта или компонента при нормальной работе требованиям надежности;

– *готовность* (availability): степень работоспособности и доступности системы, продукта или компонента;

– *отказоустойчивость* (fault tolerance): способность системы, продукта или компонента работать как предназначено, несмотря на наличие дефектов программного обеспечения или аппаратных средств;

– *восстанавливаемость* (recoverability): способность продукта или системы восстановить данные и требуемое состояние системы в случае прерывания или сбоя.

Защищенность (security): степень защищенности информации и данных, обеспечиваемая продуктом или системой путем ограничения доступа людей, других продуктов или систем к данным в соответствии с типами и уровнями авторизации. Включает в себя следующие подхарактеристики:

– *конфиденциальность* (confidentiality): обеспечение продуктом или системой ограничения доступа к данным только для тех, кому доступ разрешен;

– *целостность* (integrity): степень предотвращения системой, продуктом или компонентом несанкционированного доступа или модификации компьютерных программ или данных;

– *неподдельность* (non-repudiation): степень, с которой может быть доказан факт события или действия таким образом, что этот факт не может быть отвергнут когда-либо позже;

– *отслеживаемость* (accountability): степень, до которой действия объекта могут быть прослежены однозначно;

– *подлинность* (authenticity): степень достоверности тождественности объекта или ресурса требуемому объекту или ресурсу.

Сопровождаемость, модифицируемость (maintainability): результативность и эффективность, с которыми продукт или система

могут быть модифицированы предполагаемыми специалистами по обслуживанию. Включает в себя следующие подхарактеристики:

– *модульность* (modularity): степень представления системы или компьютерной программы в виде отдельных блоков таким образом, чтобы изменение одного компонента оказывало минимальное воздействие на другие компоненты;

– *возможность многократного использования* (reusability): степень, в которой актив может быть использован в нескольких системах или в создании других активов;

– *анализируемость* (analysability): степень простоты оценки влияния изменений одной или более частей на продукт или систему или простоты диагностики продукта для выявления недостатков и причин отказов, или простоты идентификации частей, подлежащих изменению;

– *модифицируемость* (modifiabiIity): степень простоты эффективного и рационального изменения продукта или системы без добавления дефектов и снижения качества продукта;

– *тестируемость* (testability): степень простоты эффективного и рационального определения для системы, продукта или компонента критериев тестирования, а также простоты выполнения тестирования с целью определения соответствия этим критериям.

Переносимость, ***мобильность*** (portability): степень простоты эффективного и рационального переноса системы, продукта или компонента из одной среды (аппаратных средств, ПО, операционных условий или условий использования) в другую. Включает в себя следующие подхарактеристики:

– *адаптируемость* (adaptability): степень простоты эффективной и рациональной адаптации для отличающихся или усовершенствованных аппаратных средств, ПО, других операционных сред или условий использования;

– *устанавливаемость* (installability): степень простоты эффективной и рациональной, успешной установки и (или) удаления продукта или системы в заданной среде;

– *взаимозаменяемость* (replaceability): способность продукта заменить другой конкретный программный продукт для достижения тех же целей в тех же условиях.

На качество программного продукта и системы влияют процессы жизненного цикла ПО, такие как процесс обработки требований к качеству, процесс проектирования и процесс тестирования. Качество

ресурсов, таких как человеческие ресурсы, используемые в процессе программные инструменты и методы, влияет на качество процесса, а следовательно, и на качество продукции.

5.7. Общие сведения о стандартах в области оценки качества, действующих на территории Республики Беларусь

В настоящее время в области оценки качества ПС на территории Республики Беларусь действуют следующие основные стандарты:

– межгосударственный стандарт стран СНГ ГОСТ 28806–90. Качество программных средств. Термины и определения [12];

– межгосударственный стандарт стран СНГ ГОСТ 28195–99. Оценка качества программных средств. Общие положения [13];

– национальный стандарт Беларуси СТБ ИСО/МЭК 9126-2003. Информационные технологии. Оценка программной продукции. Характеристики качества и руководства по их применению [14];

– национальный стандарт Беларуси СТБ ISO/IEC 25000-2009. Разработка программного обеспечения. Требования к качеству и оценка программного продукта (SQuaRE). Руководство по SQuaRE [15];

– национальный стандарт Беларуси СТБ ISO/IEC 25001-2009. Разработка программного обеспечения. Требования к качеству и оценка программного продукта (SQuaRE). Планирование и управление [16].

– СТБ ИСО/МЭК 9126–2003 представляет собой аутентичный перевод международного стандарта ISO/IEC 9126:1991 [17].

В ГОСТ 28806–90 приведены основные термины и определения, принятые в области обеспечения качества ПС, определена модель качества ПС.

В ГОСТ 28195–99 представлены модель качества ПС, метод оценки качества ПС, классификация методов измерений свойств ПС.

СТБ ISO/IEC 25000–2009 и СТБ ISO/IEC 25001–2009 представляют собой аутентичные переводы международных стандартов ISO/IEC 25000:2005 и ISO/IEC 25001:2007. Данные стандарты относятся к серии стандартов SQuaRE.

5.8. Иерархическая модель оценки качества программных средств, регламентированная в СТБ ИСО/МЭК 9126–2003

ГОСТ 28806–90, ГОСТ 28195–99 и СТБ ИСО/МЭК 9126–2003 [13], [14] регламентируют выполнение оценки качества ПС и систем на основе иерархической модели качества. В соответствии с данной моделью совокупность свойств, отражающих качество программного средства, представляется в виде многоуровневой структуры.

ГОСТ 28806–90 и СТБ ИСО/МЭК 9126–2003 определяют первые два уровня иерархической модели качества. На первом (верхнем) уровне модели находятся характеристики. Характеристики соответствуют основным свойствам ПС. Характеристики оцениваются посредством подхарактеристик, находящихся на втором уровне модели. При этом номенклатура характеристик является обязательной, а номенклатура подхарактеристик – рекомендуемой.

ГОСТ 28195–99 определяет четырехуровневую иерархическую модель оценки качества ПС. На первом уровне модели находятся факторы качества (соответствуют характеристикам качества), на втором уровне – критерии качества (соответствуют подхарактеристикам качества), на третьем уровне – метрики, на четвертом – оценочные элементы. Номенклатура факторов и критериев является обязательной, а номенклатура метрик и оценочных элементов – рекомендуемой.

В ГОСТ 28806–90 и СТБ ИСО/МЭК 9126–2003 определены шесть характеристик качества ПС:

1. *Функциональность* (Functionality) – совокупность свойств ПС, определяемая наличием и конкретными особенностями набора функций, способных удовлетворять заданные или подразумеваемые потребности.

2. *Надежность* (Reliability) – совокупность свойств, характеризующая способность ПС сохранять заданный уровень пригодности в заданных условиях в течение заданного интервала времени.

3. *Удобство использования* (практичность, Usability) – совокупность свойств программного средства, характеризующая усилия, необходимые для его использования, и индивидуальную оценку результатов его использования заданным или подразумеваемым кругом пользователей.

4. *Эффективность* (Efficiency) – совокупность свойств ПС, характеризующая те аспекты его уровня пригодности, которые связаны с характером и временем использования ресурсов, необходимых при заданных условиях функционирования.

5. *Сопровождаемость* (Maintainability) – совокупность свойств ПС, характеризующая усилия, которые необходимы для его модификации.

6. *Мобильность* (Portability) – совокупность свойств ПС, характеризующая приспособленность для переноса из одной среды функционирования в другие.

На рис. 5.8 приведены два верхних уровня иерархической модели качества, определенной в ГОСТ 28806–90 и СТБ ИСО/МЭК 9126–2003.

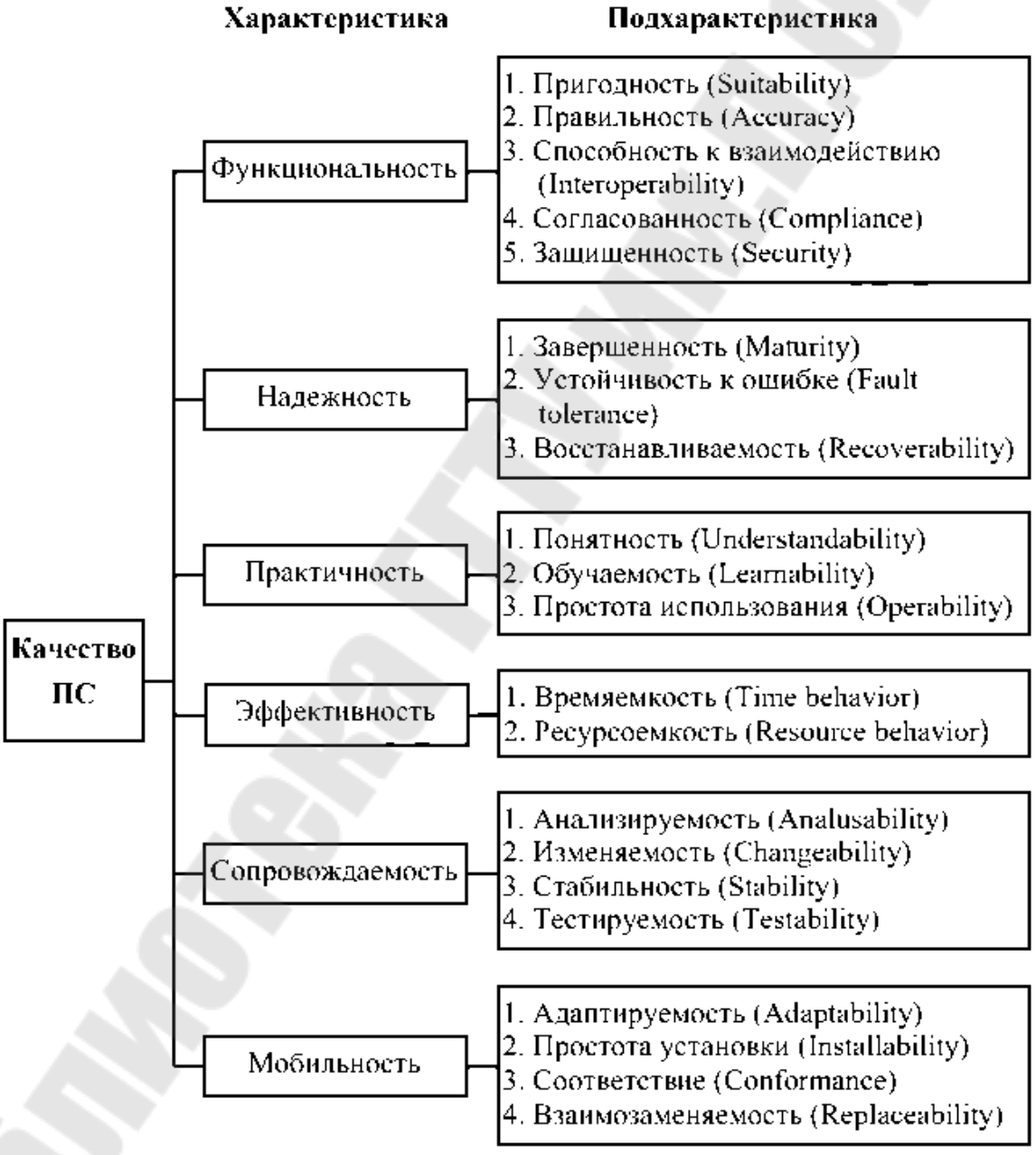

Рис. 5.8. Модель качества по СТБ ИСО/МЭК 9126–2003 (ISO/IEC 9126:1991) и ГОСТ 28806–90

5.9. Развитие стандартизации качества программных средств за рубежом

В течение десяти лет (1991–2001 гг.) основой регламентирования характеристик качества ПС за рубежом являлся международный стандарт ISO/IEC 9126:1991. Информационная технология. Оценка программного продукта. Характеристики качества и руководства по их применению [17]. В параграфе 5.7 описана модель качества ПС, приведенная в стандарте СТБ ИСО/МЭК 9126–2003 [14], который является аутентичным переводом вышеназванного стандарта.

В 2001–2004 гг. стандарт ISO/IEC 9126:1991 был заменен на две взаимосвязанные серии стандартов: ISO/IEC 9126–1–4:2001–2004 и ISO/IEC 14598–1–6:1998–2001 [18]–[21], [22]–[27].

Как и в ISO/IEC 9126:1991, в серии стандартов ISO/IEC 9126–1–4:2001–2004 регламентирована иерархическая модель качества программных средств. На верхнем уровне модели находятся характеристики. Характеристики разделяются на подхарактеристики. Подхарактеристики определяются метриками. Метрики измеряют атрибуты (свойства) ПС.

Данная серия стандартов состоит из четырех частей под общим названием «Программная инженерия. Качество продукта» [18]–[21]:

ISO/IEC 9126–1:2001. – Часть 1: Модель качества;

ISO/IEC TR 9126–2:2003. – Часть 2: Внешние метрики;

ISO/IEC TR 9126–3:2003. – Часть 3: Внутренние метрики;

ISO/IEC TR 9126–4:2004. – Часть 4: Метрики качества в использовании.

Первая часть ISO/IEC 9126–1:2001 является пересмотренной редакцией ISO/IEC 9126:1991. В данной части определены два верхних уровня (характеристики и подхарактеристики) иерархической модели качества, приведены общие требования к метрикам качества. В отличие от ISO/IEC 9126:1991 подхарактеристики второго уровня стали нормативными, а не рекомендуемыми, определены две части модели качества (модель внутреннего и внешнего качества и модель качества в использовании) и исключен процесс оценки качества (он определен в стандарте ISO/IEC 14598–1:1999).

Модель внутреннего и внешнего качества близка к модели качества, определенной в ISO/IEC 9126:1991. В ней сохранена та же номенклатура из шести базовых характеристик качества ПС (рис. 5.8), к подхарактеристикам качества добавлено несколько новых

подхарактеристик. Модель внутреннего и внешнего качества из стандарта ISO/IEC 9126–1:2001 приведена на рис. 5.9. Курсивом на данном рисунке выделены новые подхарактеристики.

Модель качества в использовании, определенная в ISO/IEC 9126–1:2001, представляет собой двухуровневую модель, на верхнем уровне которой находятся характеристики, на втором – метрики. В соответствии с данной моделью качество в использовании разделяется на четыре характеристики: результативность, продуктивность, безопасность, удовлетворенность.

Рис. 5.9. Модель внешнего и внутреннего качества по ISO/IEC 9126–1:2001

В 2011 г. ISO/IEC 9126–1:2001 заменен на стандарт ISO/IEC 25010:2011 серии SQuaRE.

Вторая–четвертая части ISO/IEC TR 9126–2–4:2003–2004 опубликованы в виде технических отчетов (TR). Совокупности метрик, перечисленные в данных частях, являются рекомендуемыми, их набор не является исчерпывающим. Метрики могут модифицироваться. Возможно применение метрик, не включенных в данные части. В этих частях стандарта содержатся примеры метрик для каждой подхарактеристики, примеры применения метрик на протяжении жизненного цикла ПС.

Части ISO/IEC TR 9126–2–4:2003–2004 в настоящее время являются действующими. На их основе организациями ISO и IEC разрабатываются стандарты ISO/IEC 25022 и ISO/IEC 25023 серии SQuaRE.

Во второй части ISO/IEC TR 9126–2:2003 определены метрики внешнего качества ПС. Внешние метрики – метрики, предназначенные для измерения качества программного продукта путем измерения поведения системы, частью которой является данный продукт. Внешние метрики могут использоваться в процессе эксплуатации и на стадиях тестирования или испытаний в процессах разработки и сопровождения ПС, когда уже созданы исполнимые коды программного продукта.

В третьей части ISO/IEC TR 9126–3:2003 определяются метрики внутреннего качества ПС. Внутренние метрики – метрики, измеряющие собственные свойства ПС. Они измеряются в процессе разработки ПС на основе спецификации требований, результатов проектирования, исходного кода или другой документации ПС. Внутренние метрики дают возможность оценить качество промежуточных программных продуктов разработки, предсказывая качество конечного ПС.

В четвертой части ISO/IEC TR 9126–4:2004 определяются метрики качества в использовании. Метрики качества в использовании – метрики, измеряющие соответствие продукта потребностям заданных пользователей в достижении заданных целей с результативностью, продуктивностью, безопасностью и удовлетворением в заданных контекстах использования. Данные метрики могут использоваться только в процессе эксплуатации ПС в реальной среде окружения. Метрики качества в использовании основаны на измерении поведения типичных пользователей и системы, содержащей данное ПС.

В серии стандартов ISO/IEC 14598–1–6:1998–2001 определены процессы оценки качества программного продукта, содержатся руко-

водство и требования к оценке. Данная серия предназначена для применения при разработке, приобретении и независимой оценке программного средства. Серия состоит из шести частей [22–27]:

– ISO/IEC 14598–1:1999. Информационная технология. Оценка программного продукта. – Часть 1: Общий обзор;

– ISO/IEC 14598–2:2000. Программная инженерия. Оценка продукта. – Часть 2: Планирование и управление;

– ISO/IEC 14598–3:2000. Программная инженерия. Оценка продукта. – Часть 3: Процесс для разработчиков;

– ISO/IEC 14598–4:1999. Программная инженерия. Оценка продукта. – Часть 4: Процесс для заказчиков;

– ISO/IEC 14598–5:1998. Информационная технология. Оценка программного продукта. – Часть 5: Процесс для оценщиков;

– ISO/IEC 14598–6:2001. Программная инженерия. Оценка продукта. – Часть 6: Документация модулей оценки.

В первой части ISO/IEC 14598–1:1999 приведен обзор остальных частей, определена связь серии ISO/IEC 14598 с серией ISO/IEC 9126 и стандартом ISO/IEC 12207:1995. В данной части представлены общие требования к спецификации и оценке качества, разъяснены концепции оценки. Установлены требования к методам измерений и оценки программных продуктов. Определен общий процесс оценки качества программного продукта. Основой для данного процесса оценки явился процесс оценки качества ПС из ISO/IEC 9126:1991.

В 2011 г. ISO/IEC 14598–1:1999 заменен на стандарт ISO/IEC 25040:2011 серии SQuaRE.

Во второй части ISO/IEC 14598–2:2000 приводятся концепции планирования и управления процессом оценки качества программного продукта, рассматривается содержание плана количественной оценки качества.

Третья–пятая части ISO/IEC 14598–3–5:1998–2000 предназначены для организаций-разработчиков, организаций-заказчиков и оценщиков ПС соответственно. В данных частях рассмотрены особенности процесса оценки качества ПС при его выполнении вышеприведенными сторонами.

Шестая часть стандарта ISO/IEC 14598–6:2001 содержит руководство по документированию модулей оценки. Модуль оценки представляет собой полностью укомплектованную информацию, необходимую для проведения процесса оценки некоторой характеристики или подхарактеристики качества. Модуль содержит спецификацию

соответствующей модели качества (характеристика, подхарактеристики, метрики качества), процедуры оценки, входные данные для оценки, структуру типового отчета о результатах выполненной оценки. Рассмотрены примеры модулей оценки.

5.10. Метрики качества программных средств

5.10.1. Свойства и критерии обоснованности метрик

Метрики качества ПС классифицируются на внутренние, внешние и метрики качества в использовании. Применение конкретного вида метрик определяется стадией жизненного цикла программного средства.

Вторая–четвертая части стандарта ISO/IEC TR 9126–2–4:2003–2004 [19]–[21] посвящены детальному рассмотрению соответственно внешних и внутренних метрик качества ПС и метрик качества в использовании.

В данных частях стандарта определены следующие желательные свойства метрик:

1) *надежность*. Свойство метрики связано со случайной ошибкой; метрика свободна от случайной ошибки, если случайные изменения не влияют на результаты метрики;

2) *повторяемость*. Повторное использование метрики для того же продукта теми же специалистами по оценке, используя ту же спецификацию оценки (включая ту же окружающую среду) и тот же тип пользователей, должно привести к тем же результатам с соответствующими допусками;

3) *воспроизводимость*. Применение метрики для того же продукта различными специалистами по оценке, используя ту же спецификацию оценки (включая ту же окружающую среду) и тот же тип пользователей, должно привести к тем же результатам с соответствующими допусками;

4) *доступность*. Метрика должна четко указывать условия (например, наличие определенных атрибутов), которые ограничивают ее употребление;

5) *показательность*. Данное свойство определяет способность метрики идентифицировать части или элементы программы, которые должны быть улучшены, на основании сравнения измеренных и ожидаемых результатов;

6) *корректность*. Метрика должна обладать следующими свойствами:

– *объективность*. Результаты метрики и ее входные данные должны быть основаны на фактах и не подвластны чувствам или мнениям специалистов по оценке или тестированию (исключая метрики удовлетворенности или привлекательности, измеряющие чувства и мнения пользователя);

– *беспристрастность*. Измерение не должно быть направлено на получение какого-либо специфического результата;

– *адекватность точности*. Точность определяется при проектировании метрики и при выборе описаний фактов, используемых как основа для метрики; разработчик метрики должен описать точность и чувствительность метрики;

7) *значимость*. Измерение должно давать значимые результаты, касающиеся поведения программы или характеристик качества.

Метрика должна также быть эффективной по отношению к стоимости. Это значит, что более дорогие метрики должны обеспечивать лучшие результаты оценки.

Разработчик метрики должен доказать ее обоснованность. Метрика должна удовлетворять хотя бы одному из следующих критериев обоснованности метрики:

1) *корреляция*. Изменение в значениях характеристик качества (оперативно определенных по результатам измерения основных метрик), обусловленное изменением в значениях метрики, должно определяться линейной зависимостью;

2) *трассировка*. Если метрика М непосредственно связана с величиной характеристики качества Q (оперативно определенной по результатам измерения основных метрик), то изменение величины $Q(T1)$, имеющейся в момент времени $T1$, к величине $Q(T2)$, полученной в момент времени $T2$, должно сопровождаться изменением значения метрики от $M(T1)$ до $M(T2)$ в том же направлении (например, если увеличивается Q, то М тоже увеличивается);

3) *непротиворечивость*. Если значения характеристик качества (оперативно полученные по результатам измерения основных метрик) $Q1, Q2,\ldots, Qn$, связанные с продуктами или процессами 1, 2,..., n, определяются соотношением $Q1 > Q2 > \ldots > Qn$, то соответствующие значения метрики должны удовлетворять соотношению $M1 > M2 > \ldots > Mn$;

4) *предсказуемость*. Если метрика используется в момент времени Т1 для прогноза значения (оперативно полученного по результатам

измерения основных метрик) характеристики качества Q в момент времени $T2$, то ошибка прогнозирования, определяемая выражением

(прогнозное Q ($T2$) – фактическое Q ($T2$)) / (фактическое Q($T2$)),

должна попадать в допустимый диапазон ошибок прогнозирования;

5) *селективность*. Метрика должна быть способной различать высокое и низкое качество ПС.

В стандартах ISO/IEC 9126–2–4:2003–2004 для каждой подхарактеристики внешнего и внутреннего качества и характеристики качества в использовании приведены таблицы, в которых даны примеры метрик качества.

Таблицы имеют следующую структуру:

1) название метрики;

2) назначение метрики;

3) метод применения;

4) способ измерения, формула, исходные и вычисляемые данные;

5) интерпретация измеренного значения (диапазон и предпочтительные значения);

6) тип шкалы, используемой при измерении метрики (номинальная, порядковая, интервальная, шкала отношений или абсолютная);

7) тип измеренного значения; используются следующие типы измеренных значений: тип размера (например, функциональный размер, размер исходного текста); тип времени (например, затраченное время, необходимое пользователю время); тип количества (например, количество изменений, количество отказов);

8) источники входных данных для измерения;

9) ссылка на ISO/IEC 12207:1995 (процессы жизненного цикла программных средств, при выполнении которых применима метрика);

10) целевая аудитория.

Для обеспечения возможности совместного использования различных метрик (независимо от их физического смысла, единиц измерения и диапазонов значений) при интегральной оценке качества программных продуктов метрики по возможности должны быть представлены в относительных единицах в виде

$$X = A / B \qquad (5.1)$$

или

$$X = 1 - A / B, \qquad (5.2)$$

где X – значение метрики; A и B – значения некоторых свойств оцениваемого продукта или документации.

В параграфах 5.9.2–5.9.4 приведены конкретные примеры свойств A и B. В стандартах серии SQuaRE элементы A и B, участвующие в вычислении метрики, называются элементами меры качества.

Из формул (5.1) и (5.2) для конкретной метрики выбирается та, которая соответствует критериям трассировки и непротиворечивости: с увеличением относительного значения метрики значение подхарактеристики и характеристики качества должно увеличиваться.

Вычисление метрик по формуле (5.1) или (5.2) при использовании для A и B одинаковых единиц измерения позволяет привести их относительные значения в диапазон

$$0 \le X \le 1, \tag{5.3}$$

что упрощает их совместное использование при интегральной оценке качества ПС.

В параграфах 5.9.2–5.9.4 приведены примеры метрик (по одной на каждую подхарактеристику или характеристику качества) из рекомендуемых в стандартах ISO/IEC TR 9126–2–4:2003–2004 [19]–[21].

Следует отметить, что не все метрики, приведенные в данных стандартах, удовлетворяют вышеприведенным свойствам, критериям, оцениваются с помощью выражений (5.1), (5.2) или попадают в диапазон (5.3).

5.10.2. Внутренние метрики качества программных средств

Примеры внутренних метрик качества ПС, содержащихся в стандарте ISO/IEC TR 9126–3:2003 [20], представлены в табл. 5.1. Во втором столбце таблицы по каждой подхарактеристике приведено название одной метрики, уникальная формула или номер формулы (5.1) или (5.2) из параграфа 5.9.1 для оценки данной метрики. Исходные данные в третьем столбце – это данные, используемые в соответствующей формуле для вычисления значения представленной метрики.

Таблица 5.1

Внутренние метрики качества программных средств

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Функциональность		
Пригодность	Полнота функциональной реализации (5.2)	*A* – число нереализованных функций, обнаруженных при оценке; *B* – число функций, описанных в спецификации требований
Правильность	Точность (5.1)	*A* – количество элементов данных, реализованных с заданными уровнями точности, подтвержденное при оценке; *B* – количество элементов данных, для которых в спецификации заданы уровни точности
Способность к взаимодействию	Соответствие интерфейсов (протоколов) (5.1)	*A* – количество интерфейсных протоколов, реализующих заданный в спецификации формат, подтвержденных при проверке; *B* – количество интерфейсных протоколов, которые должны быть реализованы в соответствии со спецификацией
Защищенность	Предотвращение разрушения данных (5.1)	*A* – количество реализованных случаев предотвращения разрушения данных из заданных в спецификации, подтвержденное при проверке; *B* – количество случаев доступа, которые определены в спецификации как способные разрушить данные
Соответствие функциональности	Соответствие функциональности (5.1)	*A* – количество корректно реализованных элементов, связанных с соответствием функциональности, подтвержденное при оценке; *B* – общее количество элементов соответствия
Надежность		
Завершенность	Полнота тестирования (5.1)	*A* – количество тестовых комбинаций, спроектированных в плане тестирования и подтвержденных при проверке; *B* – количество требуемых тестовых комбинаций

Продолжение табл. 5.1

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Устойчивость к ошибке	Предотвращение некорректных действий (5.1)	*A* – количество функций, реализованных с предотвращением некорректных действий; *B* – количество типичных некорректных действий, которое должно быть учтено
Восстанавливаемость	Способность к восстановлению (5.1)	*A* – количество реализованных требований к восстановлению, подтвержденное при проверке; *B* – общее количество требований к восстановлению, определенных в спецификации
Соответствие надежности	Соответствие надежности (5.1)	*A* – количество корректно реализованных элементов, связанных с соответствием надежности, подтвержденное при оценке; *B* – общее количество элементов соответствия
Практичность		
Понятность	Способность к демонстрации (5.1)	*A* – количество демонстрируемых функций, подтвержденное при проверке; *B* – общее количество функций, которые должны обладать способностью к демонстрации
Обучаемость	Полнота документации пользователя и (или) возможности электронной справки help (5.1)	*A* – количество описанных функций; *B* – общее количество предоставляемых функций
Простота использования	Отменяемость действий пользователя (5.1)	*A* – количество реализованных функций, которые могут быть отменены пользователем с восстановлением предыдущих данных; *B* – общее количество функций
Привлекательность	Настраиваемость вида интерфейса пользователя (5.1)	*A* – количество типов элементов интерфейса, которые могут быть настроены; *B* – общее количество типов элементов интерфейса

Продолжение табл. 5.1

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Соответствие практичности	Соответствие практичности (5.1)	*A* – количество корректно реализованных элементов, связанных с соответствием практичности, подтвержденное при оценке; *B* – общее количество элементов соответствия
Эффективность		
Поведение во времени	Пропускная способность $X = A$	*A* – число задач в единицу времени, подтвержденное при проверке
Использование ресурсов	Использование памяти $X = A$	*A* – размер памяти в байтах (вычисленный или смоделированный)
Соответствие эффективности	Соответствие эффективности (5.1)	*A* – количество корректно реализованных элементов, связанных с соответствием эффективности, подтвержденное при проверке; *B* – общее количество элементов соответствия
Сопровождаемость		
Анализируемость	Готовность диагностических функций (5.1)	*A* – количество реализованных диагностических функций из заданных в спецификации, подтвержденное при проверке; *B* – требуемое количество диагностических функций
Изменяемость	Регистрируемость изменений (5.1)	*A* – количество изменений в функциях / модулях, отраженных в комментариях, подтвержденное при проверке; *B* – общее количество изменений в функциях / модулях относительно оригинального кода
Стабильность	Влияние изменений (5.2)	*A* – количество обнаруженных вредных влияний после модификаций; *B* – количество сделанных модификаций
Тестируемость	Полнота встроенных функций тестирования (5.1)	*A* – количество реализованных встроенных функций тестирования из заданных в спецификации, подтвержденное при проверке; *B* – требуемое количество встроенных функций тестирования

Окончание табл. 5.1

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Соответствие сопровождаемости	Соответствие сопровождаемости (5.1)	*A* – количество корректно реализованных элементов, связанных с соответствием сопровожаемости, подтвержденное при оценке; *B* – общее количество элементов соответствия
Мобильность		
Адаптируемость	Адаптируемость структур данных (5.1)	*A* – количество структур данных, работоспособность которых не нарушена после адаптации, подтвержденное при проверке; *B* – общее количество структур данных, которые должны обладать способностью к адаптации
Настраиваемость	Объем работ по установке (5.1)	*A* – количество автоматических шагов инсталляции, подтвержденное при проверке; *B* – требуемое количество шагов инсталляции
Совместимость	Доступная совместимость (5.1)	*A* – количество объектов, с которыми продукт может сосуществовать, из заданных в спецификации; *B* – количество объектов в окружающей среде, с которыми продукт должен сосуществовать
Взаимозаменяемость	Преемственность данных (5.1)	*A* – количество элементов данных ПС, которые продолжают использоваться после замещения (из заданных в спецификации), подтвержденное при проверке; *B* – количество элементов старых данных, которые должны использоваться из старого ПС
Соответствие мобильности	Соответствие мобильности (5.1)	*A* – количество корректно реализованных элементов, связанных с соответствием мобильности, подтвержденное при проверке; *B* – общее количество элементов соответствия

5.10.3. Внешние метрики качества программных средств

Примеры внешних метрик качества программных средств из стандарта ISO/IEC TR 9126–2:2003 [19] содержит табл. 5.2. Во втором столбце таблицы по каждой подхарактеристике приведено название одной метрики, уникальная формула или номер формулы (5.1) или (5.2) из параграфа 5.9.1 для оценки данной метрики. Исходные данные в третьем столбце – это данные, используемые в соответствующей формуле для вычисления значения представленной метрики.

Таблица 5.2

Внешние метрики качества программных средств

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Функциональность		
Пригодность	Полнота функциональной реализации (5.2)	A – количество отсутствующих функций, обнаруженное при проверке; B – количество функций, описанных в спецификации
Правильность	Точность $X = A / T$	A – количество результатов, подсчитанное пользователями, с уровнем точности, отличающимся от требуемого; T – продолжительность работы
Способность к взаимодействию	Способность к обмену данными (основанная на успешных попытках пользователя) (5.2)	A – количество случаев, в которых пользователю не удалось обменяться данными с другими ПС или системами; B – количество случаев, в которых пользователь пытался обмениваться данными
Защищенность	Предотвращение разрушения данных (5.2)	A – количество произошедших случаев разрушения важных данных; B – количество тестовых случаев, направленных на разрушение данных
Соответствие функциональности	Соответствие функциональности (5.2)	A – количество заданных элементов соответствия функциональности, не подтвержденных при тестировании; B – общее количество заданных элементов соответствия функциональности

Продолжение табл. 5.2

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Надежность		
Завершенность	Плотность ошибок $X = A / Z$	A – количество ошибок, обнаруженных в течение определенного испытательного срока; Z – размер продукта
Устойчивость к ошибке	Предотвращение некорректных действий (5.1)	A – количество предотвращенных критических и серьезных отказов; B – количество выполненных при тестировании тестовых случаев, направленных на проверку типовых некорректных действий, которые могут привести к отказу
Восстанавливаемость	Способность к восстановлению (5.1)	A – количество случаев успешного восстановления; B – количество случаев восстановления, протестированных согласно требованиям
Соответствие надежности	Соответствие надежности (5.2)	A – количество заданных элементов соответствия надежности, не подтвержденных при тестировании; B – общее количество заданных элементов соответствия надежности
Практичность		
Понятность	Полнота описания (5.1)	A – количество функций (или классов функций), понятных после прочтения документации на программный продукт; B – общее количество функций (или классов функций), реализуемых программным продуктом
Обучаемость	Эффективность документации пользователя и(или) справочной системы (help) (5.1)	A – количество задач, успешно выполненных после получения оперативной справки и(или) чтения документации; B – общее количество протестированных задач

Продолжение табл. 5.2

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Простота использования	Исправление ошибок при использовании (5.1)	*A* – число экранов или форм, где входные данные были успешно модифицированы или изменены (восстановлены) перед очередной обработкой; *B* – число экранов или форм, где пользователь пытался модифицировать или изменить (восстановить) входные данные в течение испытательного срока использования
Привлекательность	Изменяемость вида интерфейса (5.1)	*A* – количество элементов интерфейса, измененных внешне для удовлетворения пользователя; *B* – количество элементов интерфейса, которые пользователь хотел изменить
Соответствие практичности	Соответствие практичности (5.2)	*A* – количество заданных элементов соответствия практичности, не подтвержденных при тестировании; *B* – общее количество заданных элементов соответствия практичности
Эффективность		
Поведение во времени	Время отклика $X = A - B$	*A* – момент времени получения результата; *B* – момент времени завершения ввода команды
Использование ресурсов	Использование устройств ввода/вывода (5.1)	*A* – время занятости устройств ввода / вывода; *B* – заданное время, предназначенное для использования устройств ввода / вывода
Соответствие эффективности	Соответствие эффективности (5.2)	*A* – количество заданных элементов соответствия эффективности, не подтвержденных при тестировании; *B* – общее количество заданных элементов соответствия эффективности
Сопровождаемость		
Анализируемость	Поддержка диагностическими функциями (5.1)	*A* – количество отказов, при которых персонал сопровождения с помощью диагностических функций может диагностировать причину; *B* – общее число зарегистрированных отказов

Продолжение табл. 5.2

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Изменяемость	Возможность управления изменением ПС (5.1)	A – количество фактически записанных данных регистрации изменений; B – запланированное количество данных регистрации изменений, достаточное для отслеживания изменений ПС
Стабильность	Возникновение отказа после изменения $X = A / N$	A – количество отказов, возникших в течение заданного периода после устранения отказа; N – количество отказов, устраненных путем изменения ПС
Тестируемость	Доступность встроенных функций тестирования (5.1)	A – количество случаев, в которых персонал сопровождения может использовать встроенные функции тестирования; B – количество подходящих случаев, в которых можно было бы использовать встроенные тесты
Соответствие сопровождаемости	Соответствие сопровождаемости (5.2)	A – количество заданных элементов соответствия сопровождаемости, не подтвержденных при тестировании; B – общее количество заданных элементов соответствия сопровождаемости
Мобильность		
Адаптируемость	Адаптируемость структур данных (5.1)	A – количество работоспособных данных, которые не требуют сопровождения при адаптации; B – ожидаемое число работоспособных данных в окружающей среде, к которой ПС адаптировано
Настраиваемость	Простота установки (5.1)	A – число успешных случаев приспосабливания пользователем операции инсталляции к среде эксплуатации; B – общее число попыток пользователя приспособить операцию инсталляции к среде окружения
Совместимость	Доступная совместимость $X = A / T$	A – число любых ограничений или непредусмотренных отказов, с которыми пользователь сталкивается во время одновременной работы с другими ПС; T – продолжительность одновременной работы с другими ПС

Окончание табл. 5.2

Название подхарактеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Взаимозаменяемость	Преемственность данных (5.1)	*A* – число данных замещаемого ПС, которые могут продолжать использоваться после его замещения; *B* – число данных замещаемого ПС, которые по плану должны продолжать использоваться после его замещения
Соответствие мобильности	Соответствие мобильности (5.2)	*A* – количество заданных элементов соответствия мобильности, не подтвержденных при тестировании; *B* – общее количество заданных элементов соответствия мобильности

5.10.4. Метрики качества программных средств в использовании

Примеры метрик качества в использовании из стандарта ISO/IEC TR 9126–4:2004 [21] содержит табл. 5.3. Во втором столбце таблицы по каждой подхарактеристике приведено название одной метрики, номер формулы (5.1) или (5.2) из параграфа 5.9.1 для оценки данной метрики. Исходные данные в третьем столбце – данные, используемые в соответствующей формуле для вычисления значения представленной метрики.

Таблица 5.3

Метрики качества ПС в использовании

Название под характеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Результативность	Завершение задачи (5.1)	*A* – количество завершенных задач; *B* – общее количество попыток выполнения задач
Продуктивность	Коэффициент продуктивности (5.1)	*A* – продуктивное время; $A = B – (B1 + B2 + + B3)$, где *B* – продолжительность выполнения задачи; *B*1 – продолжительность помощи; *B*2 – продолжительность обработки ошибок; *B*3 – продолжительность поиска

Окончание табл. 5.3

Название под характеристики	**Название метрики, формула для ее оценки**	**Исходные данные для вычисления метрики по соответствующей формуле**
Безопасность	Экономический ущерб (5.2)	*A* – число случаев экономического ущерба; *B* – общее число случаев использования
Удовлетворенность	Использование по собственному усмотрению (5.1)	*A* – количество случаев использования заданных в спецификации функций программного средства/приложений/систем; *B* – количество случаев их запланированного использования

Литература

1. Куликов, С. С. Тестирование программного обеспечения. Базовый курс: практ. пособие / С. С. Куликов. – Минск : Четыре четверти, 2015. – 294 с.

2. Савин, Р. Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах / Р. Савин – М. : Дело, 2007. – 312 с.

3. Зельдман, Д. Web-дизайн по стандартам / Д. Зельдман. – М. : НТ Пресс, 2005.

4. Романец, Ю. В. Защита информации в компьютерных системах и сетях / Ю. В. Романец, П. А. Тимофеев, В. Ф. Шаньгин. – 2-е изд., перераб., доп. – М. : Радио и связь, 2001. – 376 с.

5. ISO/IEC 14598–4:1999. Программная инженерия. Оценка продукта. – Ч. 2: Процесс для заказчиков. – Введ. 1999–10–01. – Женева : ISO/IEC, 1999.

6. ISO/IEC 25001:2007. Программная инженерия. Требования к качеству и оценка программного продукта (SQuaRE). Планирование и управление. – Введ. 2007–02–01. – Женева : ISO/IEC, 2007.

7. ISO/IEC 14598–4:1999. Программная инженерия. Оценка продукта. – Ч. 4: Процесс для заказчиков. – Введ. 1999–10–01. – Женева : ISO/IEC, 1999.

8. ISO/IEC 14598–5:1998. Информационная технология. Оценка программного продукта. – Ч. 5 : Процесс для оценщиков. – Введ. 1998–07–01. – Женева : ISO/IEC, 1998.

9. ISO/IEC 14598–6:2001. Программная инженерия. Оценка продукта. – Ч. 6: Документация модулей оценки. – Введ. 2001–06–01. – Женева : ISO/IEC, 2001.

10. ISO/IEC 15939:2007. Программная инженерия. Процесс измерения программных средств. – Введ. 2008–10–01. – Женева : ISO/IEC, 2007.

11. ISO/IEC 25000:2005. Программная инженерия. Требования к качеству и оценка программного продукта (SQuaRE). Руководство по SQuaRE. – Введ. 2005-08-01. – Женева : ISO/IEC, 2005.

12. ГОСТ 28806–90. Качество программных средств. Термины и определения. – Введ. 1992–01–01. – М. : Изд-во стандартов, 1991.

13. ГОСТ 28195–99. Оценка качества программных средств. Общие положения. – Введ. 2000–03–01. – Минск : Межгосударственный совет по стандартизации, метрологии и сертификации, 2001.

14. СТБ ИСО/МЭК 9126-2003. Информационные технологии.

Оценка программной продукции. Характеристики качества и руководства по их применению. – Введ. 2003–11–01. – Минск : Госстандарт Респ. Беларусь, 2003.

15. СТБ ISO/IEC 25000-2009. Разработка программного обеспечения. Требования к качеству и оценка программного продукта (SQuaRE). Руководство по SQuaRE. – Введ. 2010–01–01. – Минск : Госстандарт Респ. Беларусь, 2009.

16. СТБ ISO/IEC 25001-2009. Разработка программного обеспечения. Требования к качеству и оценка программного продукта (SQuaRE). Планирование и управление. – Введ. 2010–01–01. – Минск : Госстандарт Респ. Беларусь, 2009.

17. ISO/IEC 9126:1991. Информационная технология. Оценка программного продукта. Характеристики качества и руководства по их применению.

18. ISO/IEC 9126–1:2001. Программная инженерия. Качество продукта. – Ч. 1: Модель качества. – Введ. 2001–06–15. – Женева : ISO/IEC, 2001.

19. ISO/IEC TR 9126–2:2003. Программная инженерия. Качество продукта. – Ч. 2: Внешние метрики. – Введ. 2003–07–01. – Женева : ISO/IEC, 2003.

20. ISO/IEC TR 9126–3:2003. Программная инженерия. Качество продукта. – Ч. 3: Внутренние метрики. – Введ. 2003–07–01. – Женева : ISO/IEC, 2003.

21. ISO/IEC TR 9126–4:2004. Программная инженерия. Качество продукта. – Ч. 4: Метрики качества в использовании. – Введ. 2004–04–01. – Женева : ISO/IEC, 2004.

22. ISO/IEC 14598–1:1999. Информационная технология. Оценка программного продукта. – Ч. 1: Общий обзор. – Введ. 1999–04–15. – Женева : ISO/IEC, 2001.

23. ISO/IEC 14598–2:2000. Программная инженерия. Оценка продукта. – Ч. 2: Планирование и управление. – Введ. 2000–02–01. – Женева : ISO/IEC, 2000.

24. ISO/IEC 14598–3:2000. Программная инженерия. Оценка продукта. – Ч. 3: Процесс для разработчиков. – Введ. 2000–02–01. – Женева : ISO/IEC, 2000.

25. ISO/IEC 14598–4:1999. Программная инженерия. Оценка продукта. – Ч. 4: Процесс для заказчиков. – Введ. 1999–10–01. – Женева : ISO/IEC, 1999.

26. ISO/IEC 14598–5:1998. Информационная технология. Оценка программного продукта. – Ч. 5: Процесс для оценщиков. – Введ. 1998–07–01. – Женева : ISO/IEC, 1998.

27. ISO/IEC 14598–6:2001. Программная инженерия. Оценка продукта. – Ч. 6: Документация модулей оценки. – Введ. 2001–06–01. – Женева : ISO/IEC, 2001.

Учебное электронное издание комбинированного распространения

Учебное издание

Мурашко Игорь Александрович
Комраков Владимир Викторович

ВЕРИФИКАЦИЯ И АТТЕСТАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» дневной и заочной форм обучения

Электронный аналог печатного издания

Редактор *Н. Г. Мансурова*
Компьютерная верстка *И. П. Минина*

Подписано в печать 20.09.21.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Ризография. Усл. печ. л. 11,39. Уч.-изд. л. 11,91.
Изд. № 2.
http://www.gstu.by

Издатель и полиграфическое исполнение
Гомельский государственный
технический университет имени П. О. Сухого.
Свидетельство о гос. регистрации в качестве издателя
печатных изданий за № 1/273 от 04.04.2014 г.
пр. Октября, 48, 246746, г. Гомель