Министерство образования Республики Беларусь

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра «Технология и методика преподавания»

Отладка и тестирование программного обеспечения

Учебное пособие для студентов
специальности 1-08 01 01
«Профессиональное обучение (по направлениям)»

Электронный учебный материал

Минск
БНТУ
2020

Составители:
Дробыш Алексей Анатольевич, к.т.н., доцент, Белорусский национальный технический университет
Санцевич С.Н., магистрант, Белорусский национальный технический университет

Оглавление

Раздел 1. Введение в отладку и тестирование программного обеспечения 4
Тема 1.1 История тестирования программного обеспечения 4
Тема 1.2 Тестирование: общие сведения. 6
Тема 1.3 Концепция тестирования 9
Тема 1.4 Отладка ПО 11
Раздел 2. Тестирование по принципу «белого ящика» 14
Тема 2.1 Общие сведения о принципе 14
Тема 2.2 Методы тестирования стратегии «белого ящика» 14
Раздел 3. Тестирование по принципу «черного ящика» 16
Тема 3.1 Общие сведения о принципе 16
Тема 3.2 Методы формирования тестовых наборов 16
Тема 4.1 Критерии тестирования 1
Тема 4.2 Ручное тестирование программных продуктов 3
Тема 4.3 Модульное тестирование 5
Тема 4.4 Интеграционное тестирование 7
Тема 4.5 Юзабилити-тестирование 9
Тема 4.6. Регрессионное тестирование 11
Тема 4.7. Автоматизированное тестирование 13
Тема 4.8. Тестирование производительности 15
Тема 4.9 Тестирование локализации и совместимости 17
ЛИТЕРАТУРА 18

Раздел 1. Введение в отладку и тестирование программного обеспечения

Тема 1.1 История тестирования программного обеспечения

Первые программные системы разрабатывались в рамках программ научных исследований или программ для нужд министерств обороны. Тестирование таких продуктов проводилось строго формализовано с записью всех тестовых процедур, тестовых данных, полученных результатов. Тестирование выделялось в отдельный процесс, который начинался после завершения кодирования, но при этом, как правило, выполнялось тем же персоналом.

В 1960-х много внимания уделялось «**исчерпывающему**» тестированию, которое должно проводиться с использованием всех путей в коде или всех возможных входных данных. Было отмечено, что в этих условиях полное тестирование программное обеспечение (ПО) невозможно, потому что, во-первых, количество возможных входных данных очень велико, во-вторых, существует множество путей, в-третьих, сложно найти проблемы в архитектуре и спецификациях. По этим причинам «исчерпывающее» тестирование было отклонено и признано теоретически невозможным.

В начале 1970-х тестирование ПО обозначалось как «процесс, направленный на демонстрацию корректности продукта» или как «деятельность по **подтверждению правильности** работы ПО». В зарождавшейся программной инженерии верификация ПО значилась как «доказательство правильности». Хотя концепция была теоретически перспективной, на практике она требовала много времени и была недостаточно всеобъемлющей. Было решено, что доказательство правильности – неэффективный метод тестирования ПО. Однако, в некоторых случаях демонстрация правильной работы используется и в наши дни, например, приемо-сдаточные испытания. Во второй половине 1970-х тестирование представлялось как выполнение программы с намерением найти ошибки, а не доказать, что она работает. Успешный тест – это тест, который обнаруживает ранее неизвестные проблемы. Данный подход прямо противоположен предыдущему. Указанные два определения представляют собой «парадокс тестирования», в основе которого лежат два противоположных утверждения: с одной стороны, тестирование позволяет убедиться, что продукт работает хорошо, а с другой – выявляет ошибки в ПО, показывая, что продукт не работает. Вторая цель тестирования является более продуктивной с точки зрения улучшения качества, так как не позволяет игнорировать недостатки ПО.

В 1980-х тестирование расширилось таким понятием, как **предупреждение дефектов**. Проектирование тестов – наиболее эффективный из известных

методов предупреждения ошибок. В это же время стали высказываться мысли, что необходима методология тестирования, в частности, что тестирование должно включать проверки на всем протяжении цикла разработки, и это должен быть управляемый процесс. В свою очередь, жизненный цикл разработки ПО – ряд событий, происходящих с системой в процессе ее создания и дальнейшего использования. Говоря другими словами, это время от начального момента создания какого либо программного продукта, до конца его разработки и внедрения.

В ходе тестирования надо проверить не только собранную программу, но и требования, код, архитектуру, сами тесты. «Традиционное» тестирование, существовавшее до начала 1980-х, относилось только к скомпилированной, готовой системе (сейчас это обычно называется системное тестирование), но в дальнейшем тестировщики стали вовлекаться во все аспекты жизненного цикла разработки. Это позволяло раньше находить проблемы в требованиях и архитектуре и тем самым сокращать сроки и бюджет разработки. В середине 1980-х появились первые инструменты для автоматизированного тестирования. Предполагалось, что компьютер сможет выполнить больше тестов, чем человек, и сделает это более надежно. Поначалу эти инструменты были крайне простыми и не имели возможности написания сценариев на скриптовых языках.

В начале 1990-х в понятие «тестирование» стали включать планирование, проектирование, создание, поддержку и выполнение тестов и тестовых окружений, и это означало переход от тестирования к обеспечению качества, охватывающего весь цикл разработки ПО. В это время начинают появляться различные программные инструменты для поддержки процесса тестирования: более продвинутые среды для автоматизации с возможностью создания скриптов и генерации отчетов, системы управления тестами, ПО для проведения нагрузочного тестирования. В середине 1990-х с развитием Интернета и разработкой большого количества веб-приложений особую популярность стало получать «гибкое тестирование» (по аналогии с гибкими методологиями программирования).

В 2000-х появилось еще более широкое определение тестирования, когда в него было добавлено понятие «оптимизация бизнес-технологий» (business technology optimization, BTO). BTO направляет развитие информационных технологий в соответствии с целями бизнеса. Основной подход заключается в оценке и максимизации значимости всех этапов жизненного цикла разработки ПО для достижения необходимого уровня качества, производительности, доступности.

Тема 1.2 Тестирование: общие сведения.

К базовым терминам в тестировании относятся: тестирование и отладка. Тестирование (testing) – процесс выполнения программы с целью нахождения ошибки. Отладка (debugging) – средство установления точной природы ошибок, процесс, противоположный тестированию, ведет к устранению ошибок.

На данный момент наиболее распространена и используется многоуровневая модель качества программного обеспечения, представленная в наборе стандартов ISO 9126 (рисунок 1).

Рисунок 1 – Модель качества программного обеспечения (ISO 9126)

Основой регламентирования показателей качества систем является международный стандарт ISO 9126 «Информационная технология. Оценка программного продукта. Характеристики качества и руководство по их применению». В этом стандарте описано многоуровневое распределение характеристик ПО. На верхнем уровне выделено 6 основных характеристик качества ПО, каждую из которых определяют набором атрибутов, имеющих соответствующие метрики для последующей оценки

Согласно этой модели, функциональность программного средства (functionality) – совокупность свойств ПС, определяемая наличием и конкретными особенностями набора функций, способных удовлетворять заданные или подразумеваемые потребности качества наряду с ее надежностью как технической системы. Надежность (Reliability) – способность ПО выполнять требуемые задачи в обозначенных условиях на протяжении заданного промежутка времени или указанное количество операций. Удобство использования программного средства (usability) – совокупность свойств ПС, характеризующая усилия, необходимые для его использования, и оценку результатов его использования заданным кругом пользователей ПС. Эффективность (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 именующемуся окружению. Портативность (Portability) – совокупность свойств ПС, характеризующая приспособленность для переноса из одной среды функционирования в другие.

Программная ошибка (жарг. баг) – означает ошибку в программе или в системе, из-за которой программа выдает неожиданное поведение и, как следствие, результат. Большинство программных ошибок возникают из-за ошибок, допущенных разработчиками программы в её исходном коде, либо в её дизайне.

На рисунке 2 представлен анализ трудоемкости обнаружения и исправления ошибок.

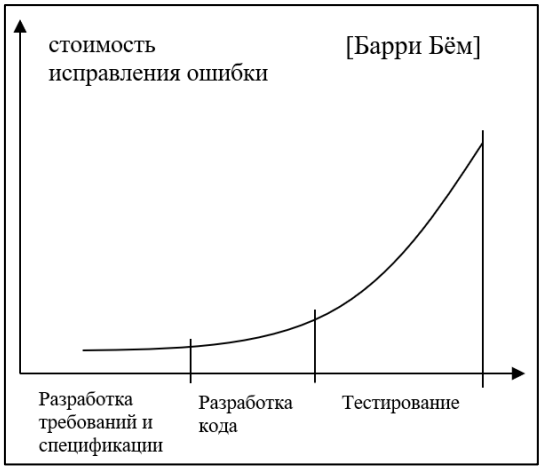

Рисунок 2 – Анализ трудоемкости обнаружения и исправления ошибок

Тестирование можно классифицировать по очень большому количеству признаков, и практически в каждой серьёзной книге о тестировании автор показывает свой взгляд на этот вопрос. В самом высоком уровне тестирование можно классифицировать, как показано на рисунке 3.

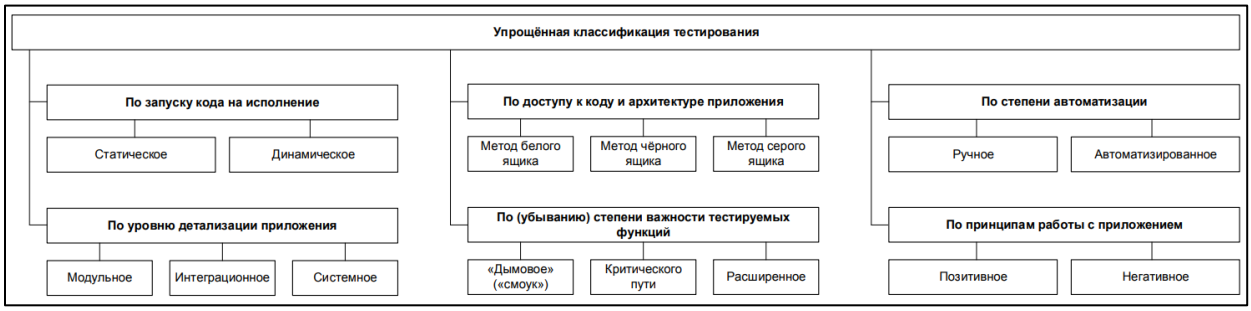

Рисунок 3 – Классификация тестирования ПО

1) По запуску кода на исполнение:

– статическое тестирование – без запуска;

– динамическое тестирование – с запуском.

2) По доступу к коду и архитектуре приложения:

– метод белого ящика – доступ к коду есть;

– метод чёрного ящика – доступа к коду нет;

– метод серого ящика – к части кода доступ есть, к части – нет.

3) По степени автоматизации:

– ручное тестирование – тест-кейсы выполняет человек;

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

4) По уровню детализации приложения (по уровню тестирования):

– модульное (компонентное) тестирование – проверяются отдельные небольшие части приложения;

– интеграционное тестирование – проверяется взаимодействие между несколькими частями приложения;

– системное тестирование – приложение проверяется как единое целое.

5) 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.

6) По принципам работы с приложением:

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

– негативное тестирование – в работе с приложением выполняются (некорректные) операции и используются данные, потенциально приводящие к ошибкам.

Тема 1.3 Концепция тестирования

Существует множество определений понятия «компьютерная программа». Ниже приведены два определения данного понятия по стандарту ISO, которые в полной мере характеризуют данный термин.

1) Компьютерная программа – комбинация компьютерных инструкций и данных, позволяющая аппаратному обеспечению вычислительной системы выполнять вычисления или функции управления (стандарт ISO/IEC/IEEE 24765:2010).

2) Компьютерная программа – синтаксическая единица, которая соответствует правилам определённого языка программирования, состоящая из определений и операторов или инструкций, необходимых для определённой функции, задачи или решения проблемы (стандарт ISO/IEC 2382-1:1993).

С точки зрения математики, компьютерная программа – это аналог формулы.

Существует два метода обоснования истинности формул.

Формальный подход или доказательство применяется, когда из исходных формул-аксиом с помощью формальных процедур (правил вывода) выводятся искомые формулы и утверждения (теоремы). Вывод осуществляется путем перехода от одних формул к другим по строгим правилам, которые позволяют свести процедуру перехода от формулы к формуле к последовательности текстовых подстановок.

Преимущество формального подхода заключается в том, что с его помощью удается избегать обращений к бесконечной области значений и на

каждом шаге доказательства оперировать только конечным множеством символов.

Интерпретационный подход применяется, когда осуществляется подстановка констант в формулы, а затем интерпретация формул как осмысленных утверждений в элементах множеств конкретных значений. Истинность интерпретируемых формул проверяется на конечных множествах возможных значений. Сложность подхода состоит в том, что на конечных множествах комбинации возможных значений для реализации исчерпывающей проверки могут оказаться достаточно велики.

Интерпретационный подход используется при экспериментальной проверке соответствия программы своей спецификации

Применение интерпретационного подхода в форме экспериментов над исполняемой программой составляет суть отладки и тестирования.

На рисунке 4 приведено сравнение одних из наиболее популярных инструментов для тестирования.

Продукт	Selenium	Katalon Studio	Unified Functional Testing	TestComplete	watir
Доступен с	2004	2015	1998	1999	2008
Тестируемое приложение	Веб-приложения	Веб (UI и API), Мобильные приложения	Веб (UI и API), Мобильные, настольные приложения	Веб (UI и API), Мобильные, настольные приложения	Веб-приложения
Стоимость	Бесплатно	Бесплатно	$$$$	$$	Бесплатно
Поддерживаемые платформы	Windows, Linux, OS X	Windows, Linux, OS X	Windows	Windows	Windows, Linux, OS X
Язык написания	Java, C#, Perl, Python, JavaScript, Ruby. PHP	Java/Groovy	VBScript	JavaScript, Python, VBScript, Jscript, Delphi, C++, C#	Ruby
Навыки программирования	Продвинутый навык, необходимый для интеграции различных инструментов	Не требуется. Рекомендуется для расширенных тестовых сценариев	Не требуется. Рекомендуется для расширенных тестовых сценариев	Не требуется. Рекомендуется для расширенных тестовых сценариев	Продвинутый навык, необходимый для интеграции различных инструментов
Простота установки и использования	Требуется продвинутый навык для установки и использования	Легок в установки и использовании	Сложен в установке. Нужна тренировка, чтобы правильно пользоваться инструментом	Легок в установке. Нужна тренировка, чтобы правильно пользоваться инструментом	Требуется продвинутый навык для установки и использования

Рисунок 4 – Сравнение популярных инструментов тестирования

В зависимости от компании, при тестирование может включать в себя составление тест-плана.

Тест план (Test Plan) – это документ, описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы

оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Существует множество стандартов, описывающих оформление планов тестирования. Тем не менее, можно выделить несколько пунктов, которые должны быть представлены в тест плане в соответствии с best practice (в переводе с английского, лучшая практика – формализация уникального успешного практического опыта):

1) Что надо тестировать (описание объекта тестирования: системы, приложения, оборудования)?
2) Что будете тестировать? (список функций и описание тестируемой системы и её компонентов отдельности)?
3) Как будете тестировать? (стратегия тестирования, а именно: виды тестирования и их применение по отношению к объекту тестирования)?
4) Когда будете тестировать (последовательность проведения работ: подготовка (Test Preparation), тестирование (Testing), анализ результатов (Test Result Analisys) в разрезе запланированных фаз разработки)?
5) Критерии начала тестирования (готовность тестовой платформы (тестового стенда), законченность разработки требуемого функционала, наличие всей необходимой документации).
6) Критерии окончания тестирования (результаты тестирования удовлетворяют критериям качества продукта).

Тема 1.4 Отладка ПО

Термин отладки может означать множество различных действий, но наиболее буквально, это этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла ошибка, приходится: узнавать текущие значения переменных; выяснять, по какому пути выполнялась программа.

В основном различают три основных типа ошибок:

– синтаксические;
– семантические;
– логические.

К **синтаксическим** ошибкам относят ошибки, вызванные нарушением правописания или пунктуации в записи выражений, операторов и т. п., иначе говоря, в нарушении грамматических правил языка, например: несогласованность между открывающими и закрывающими скобками, пропуск знаков/символов, неверное написание зарезервированных слов. Все ошибки данного типа обнаруживаются компилятором.

Семантические ошибки заключаются в нарушении порядка операторов, параметров функций и употреблении выражений. Семантические ошибки также обнаруживаются компилятором.

Логические ошибки труднее всего обнаружить. Они не вызывают сбой программы и не блокируют её запуск, они заставляют ее каким-то образом «плохо себя вести», отображая неправильный вывод. Одним из примеров логической ошибки является нулевая ссылка. Ошибка нулевой ссылки может быть вызвана тем, что свойство или поле имеет значение null, или поле локальной переменной объявлено, но не инициализировано. Компилятор может выявить только следствие логической ошибки.

Большинство ошибок можно обнаружить по косвенным признакам посредством тщательного анализа текстов программ и результатов тестирования без получения дополнительной информации. При этом используют различные методы:

– ручного тестирования;

– индукции;

– дедукции;

– обратного прослеживания.

Метод ручного тестирования – самый простой и естественный способ. При обнаружении ошибки крайне важно выполнить тестируемую программу вручную, используя тестовый набор, при работе с которым была обнаружена ошибка.

Метод очень эффективен, но не применим для больших программ, программ со сложными вычислениями и в тех случаях, когда ошибка связана с неверным представлением программиста о выполнении некоторых операций. Данный метод часто используют как составную часть других методов отладки.

Метод индукции основан на тщательном анализе симптомов ошибки, которые могут проявляться как неверные результаты вычислений или как сообщение об ошибке. В случае если компьютер просто «зависает», то фрагмент проявления ошибки вычисляют, исходя из последних полученных результатов и действий пользователя. Полученную таким образом информацию организуют и тщательно изучают, просматривая соответствующий фрагмент программы. В результате этих действий выдвигают гипотезы об ошибках, каждую из которых проверяют. В случае если гипотеза верна, то детализируют информацию об ошибке, иначе – выдвигают другую гипотезу.

В данном методе самый ответственный этап – выявление симптомов ошибки. Организуя данные об ошибке, целесообразно записать все, что известно о ее проявлениях, при этом фиксируют, как ситуации, в которых

фрагмент с ошибкой выполняется нормально, так и ситуации, в которых ошибка проявляется. В случае если в результате изучения данных никаких гипотез не появляется, то необходима дополнительная информация об ошибке. Дополнительную информацию можно получить, к примеру, в результате выполнения схожих тестов.

В процессе доказательства необходимо узнать, все ли проявления ошибки объясняет данная гипотеза, в случае если не все, то либо гипотеза не верна, либо ошибок несколько.

По методу дедукции сначала формируют множество причин, которые могли бы вызвать данное проявление ошибки. Затем, анализируя причины, исключают, те, которые противоречат имеющимся данным. В случае если все причины исключены, то следует выполнить дополнительное тестирование исследуемого фрагмента. В противном случае наиболее вероятную гипотезу пытаются доказать. В случае если гипотеза объясняет полученные признаки ошибки, то ошибка найдена, иначе – проверяют следующую причину.

Метод обратного прослеживания наиболее эффективен для небольших программ. В данном случае отладку начинают с точки вывода неправильного результата. Для этой точки строится гипотеза о значениях базовых переменных, которые могли бы привести к получению имеющегося результата. Далее, исходя из этой гипотезы, делают предположения о значениях переменных в предыдущей точке. Процесс продолжают пока не обнаружат причину ошибки.

Раздел 2. Тестирование по принципу «белого ящика»

Тема 2.1 Общие сведения о принципе

Тестирование методом белого ящика – метод тестирования программного обеспечения, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику. Мы выбираем входные значения, основываясь на знании кода, который будет их обрабатывать. Точно так же мы знаем, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование белого ящика – углубление во внутренне устройство системы, за пределы ее внешних интерфейсов.

Согласно ISTQB (International Software Qualifications Board), тестирование методом белого ящика – это:

– тестирование, основанное на анализе внутренней структуры компонента или системы;

– тест-дизайн, основанный на технике белого ящика – процедура написания или выбора тест-кейсов на основе анализа внутреннего устройства системы или компонента.

Стратегия «белого ящика» позволяет исследовать внутреннюю структуру программы. Тестирующий получает тестовые данные путем анализа логики программы.

Исчерпывающему входному тестированию может быть поставлено в соответствие исчерпывающее тестирование маршрутов – программа проверена полностью, если с помощью тестов удается осуществить выполнение программы по всем возможным маршрутам ее графа.

К недостаткам исчерпывающего тестирования маршрутов можно отнести как то, что число не повторяющих друг друга маршрутов в программе очень велико и, ввиду этого, их тяжело анализировать, так и то, что даже в случае успешной проверки каждого из маршрутов, в самой программе могут содержаться ошибки.

Тема 2.2 Методы тестирования стратегии «белого ящика»

Существуют различные виды тестирования методом «белого ящика»: модульное тестирование, статический и динамический анализ, покрытие операторов, покрытие решений, тестирование мутаций, покрытие решений/условий, комбинаторное покрытие условий.

Модульное тестирование выполняется, чтобы проверить, работает ли определенный модуль или единица кода. Модульное тестирование проходит

на самом базовом уровне, поскольку оно выполняется, когда разрабатывается блок кода, или задается определенная функциональность.

Согласно методу покрытия операторов, если отказаться полностью от тестирования всех путей, можно показать, что критерием покрытия является выполнение каждого оператора программы хотя бы один раз.

Это необходимое, но недостаточное условие для приемлемого тестирования по принципу белого ящика. Иначе говоря, в этом виде тестирования код пишется таким образом, что каждый оператор приложения выполняется хотя бы один раз. Это помогает гарантировать, что все утверждения выполняются без какого-либо побочного эффекта.

В методе покрытия решений должно быть написано достаточное число тестов, такое, что каждое направление перехода должно быть реализовано по крайней мере один раз.

Покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен.

При статическом и динамическом анализе статический включает в себя покрытие всего кода, чтобы узнать о любом возможном дефекте в коде, а динамический анализ включает в себя выполнение кода и анализ выходных данных соотственно.

Тестирование мутаций – вид тестирования, в котором приложение тестируется на код, который был изменен после исправления определенной ошибки. Это также помогает определить, какой код и какая стратегия кодирования может помочь в эффективной разработке функциональности.

Покрытие решений/условий требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия выполнялись по крайней мере один раз, все результаты каждого решения выполнялись по крайней мере один раз и, кроме того, каждой точке входа передавалось управление по крайней мере один раз.

Критерий комбинаторного покрытия условий требует, чтобы все возможные комбинации результатов условий в каждом решении, а также каждый оператор выполнились по крайней мере один раз.

Раздел 3. Тестирование по принципу «черного ящика»

Тема 3.1 Общие сведения о принципе

Согласно терминологии ISTQB, «Black-box» тестирование – это функциональное и нефункциональное тестирование без доступа к внутренней структуре компонентов системы. Метод тестирования «черного ящика» – процедура получения и выбора тестовых случаев на основе анализа спецификации (функциональной или нефункциональной), компонентов или системы без ссылки на их внутреннее устройство. Другими словами, это метод тестирования программного обеспечения, внутренняя структура, дизайн и реализация которого неизвестна тестировщику.

Целью этой техники является поиск ошибок в следующих категориях:

– неправильно реализованные или недостающие функции;

– ошибки интерфейса;

– ошибки в структурах данных или организации доступа к внешним базам данных;

– ошибки поведения или недостаточная производительности системы.

Таким образом, мы не имеем представления о структуре и внутреннем устройстве системы. Нужно концентрироваться на том, что программа делает, а не на том, как она это делает.

Очевидно, что построение исчерпывающего входного теста для большинства случаев невозможно. Поэтому, обычно выполняется «разумное» тестирование, при котором тестирование программы ограничивается прогонами на небольшом подмножестве всех возможных входных данных. Естественно при этом целесообразно выбрать наиболее подходящее подмножество (подмножество с наивысшей вероятностью обнаружения ошибок).

Правильно выбранный тест подмножества должен обладать следующими свойствами:

1) уменьшать, причем более чем на единицу число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;

2) покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Тема 3.2 Методы формирования тестовых наборов

Стратегия «черного ящика» включает в себя следующие методы формирования тестовых наборов:

– эквивалентное разбиение;

– анализ граничных значений;

– анализ причинно-следственных связей;

– предположение об ошибке.

Основу метода **эквивалентного разбиения** составляют два положения:

1. Исходные данные программы необходимо разбить на конечное число классов эквивалентности, так чтобы можно было предположить, что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если тест какого-либо класса обнаруживает ошибку, то предполагается, что все другие тесты этого класса эквивалентности тоже обнаружат эту ошибку и наоборот.

2. Каждый тест должен включать по возможности максимальное количество различных входных условий, что позволяет минимизировать общее число необходимых тестов.

Первое положение используется для разработки набора условий, которые должны быть протестированы, а второе – для разработки минимального набора тестов.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

– выделение классов эквивалентности;

– построение тестов.

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза из спецификации) и разбиением его на две или более групп. Для этого используется таблица 1.

Таблица 1. Выделение классов эквивалентности

Входное условие	Правильные классы эквивалентности	Неправильные классы эквивалентности

Правильные классы включают правильные данные, неправильные классы – неправильные данные.

Выделение классов эквивалентности является эвристическим процессом, однако при этом существует ряд правил:

– если входные условия описывают область значений (например «целое данное может принимать значения от 1 до 999»), то выделяют один правильный класс $1<X<999$ и два неправильных $X<1$ и $X>999$;

– если входное условие описывает число значений (например, «в автомобиле могут ехать от одного до шести человек»), то определяется один

правильный класс эквивалентности и два неправильных (ни одного и более шести человек);

– если входное условие описывает множество входных значений и есть основания полагать, что каждое значение программист трактует особо (например, «известные способы передвижения на автобусе, грузовике, такси, мотоцикле или пешком»), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс (например «на прицепе»);

– если входное условие описывает ситуацию «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ – буква) и один неправильный (первый символ – не буква);

– если есть любое основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс разбивается на меньшие классы эквивалентности.

Этап **построения тестов** эквивалентного разбиения заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

– назначение каждому классу эквивалентности уникального номера;

– проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых классов эквивалентности, до тех пор, пока все правильные классы не будут покрыты (только не общими) тестами;

– запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы не будут покрыты тестами.

Граничные условия – это ситуации, возникающие на, выше или ниже границ входных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения следующим:

– выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных условий осуществляется таким образом, чтобы проверить тестом каждую границу этого класса;

– при разработке тестов рассматриваются не только входные условия (пространство входов), но и пространство результатов.

Применение метода анализа граничных условий требует определенной степени творчества и специализации в рассматриваемой проблеме. Тем не менее, существует несколько общих правил этого метода:

– построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений (например, для области

входных значений от -1.0 до +1.0 необходимо написать тесты для ситуаций -1.0, +1.0, -1.001 и +1.001);

– построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих двух значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то проверить 0, 1, 255 и 256 записей;

– использовать правило 1 для каждого выходного условия. Причем, важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей. Не всегда также можно получить результат вне выходной области, но, тем не менее, стоит рассмотреть эту возможность;

– использовать правило 2 для каждого выходного условия;

– если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

Метод **анализа причинно-следственных** связей помогает системно выбирать высоко результативные тесты. Он дает полезный побочный эффект, позволяя обнаруживать неполноту и неоднозначность исходных спецификаций.

Для использования метода необходимо понимание булевской логики (логических операторов – и, или, не). Построение тестов осуществляется в несколько этапов.

1. Спецификация разбивается на «рабочие » участки, так как таблицы причинно-следственных связей становятся громоздкими при применении метода к большим спецификациям. Например, при тестировании компилятора в качестве рабочего участка можно рассматривать отдельный оператор языка.

2. В спецификации определяются множество причин и множество следствий. Причина есть отдельное входное условие или класс эквивалентности входных условий. Следствие есть выходное условие или преобразование системы. Каждым причине и следствию приписывается отдельный номер.

3. На основе анализа семантического (смыслового) содержания спецификации строится таблица истинности, в которой последовательно перебираются все возможные комбинации причин и определяются следствия каждой комбинации причин. Таблица снабжается примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, которые являются невозможными из-за синтаксических или внешних ограничений. Аналогично, при необходимости строится таблица истинности для класса эквивалентности.

Каждая строка таблицы истинности преобразуется в тест. При этом по возможности следует совмещать тесты из независимых таблиц.

Часто программист с большим опытом выискивает ошибки без методов. При этом он подсознательно использует метод **предположения об ошибке**. Процедура метода предположения об ошибке в значительной степени основана на интуиции. Основная идея метода состоит в том, чтобы перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться, а затем на основе этого списка составить тесты. Другими словами, требуется перечислить те специальные случаи, которые могут быть не учтены при проектировании.

Раздел 4. Виды тестирования

Тема 4.1 Критерии тестирования

Требования к идеальному критерию тестирования

– критерий должен быть достаточным;

– критерий должен быть полным;

– критерий должен быть надежным;

– критерий должен быть легко проверяемым.

Для нетривиальных классов программ в общем случае не существует полного и надежного критерия, зависящего от программ или спецификаций.

Поэтому обычно стремятся к идеальному общему критерию через реальные частные.

Классы критериев:

– **структурные критерии** используют информацию о структуре программы (критерии так называемого «белого ящика»);

– **функциональные критерии** формулируются в описании требований к программному изделию (критерии так называемого «черного ящика»);

– **критерии стохастического тестирования** формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы;

– **мутационные критерии** ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

Структурные критерии используют модель программы в виде «белого ящика», что предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления. Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный класс критериев часто используется на этапах модульного и интеграционного тестирования.

Структурные критерии базируются на операторах, ветвях и путях.

Условие критерия тестирования **команд** – набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.

Условие критерия тестирования **ветвей** – набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах автоматизации тестирования.

Условие критерия тестирования **путей** – набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто – 2, или числом классов выходных путей).

Функциональный критерий – важнейший для программной индустрии критерий тестирования. Он обеспечивает контроль степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При функциональном тестировании преимущественно используется модель «черного ящика». Проблема функционального тестирования – это, прежде всего, трудоемкость: дело в том, что документы, фиксирующие требования к программному изделию, как правило, достаточно объемны, тем не менее, соответствующая проверка должна быть всеобъемлющей.

Стохастическое тестирование применяется при тестировании сложных программных комплексов – когда набор детерминированных тестов (X,Y) имеет громадную мощность.

Критерии стохастического тестирования:

– статистические методы окончания тестирования – стохастические методы принятия решений о совпадении гипотез о распределении случайных величин. К ним принадлежат широко известные: метод Стьюдента, метод Хи-квадрат.

– метод оценки скорости выявления ошибок – основан на модели скорости выявления ошибок, согласно которой тестирование прекращается, если оцененный интервал времени между текущей ошибкой и следующей слишком велик для фазы тестирования приложения.

Постулируется, что профессиональные программисты пишут сразу почти правильные программы, отличающиеся от правильных мелкими ошибками или описками типа – перестановка местами максимальных значений индексов в описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы цикла на 1 и т.п. Предлагается подход, позволяющий на основе мелких ошибок оценить общее число ошибок, оставшихся в программе.

Подход базируется на следующих понятиях: **мутации** – мелкие ошибки в программе, **мутанты** – программы, отличающиеся друг от друга мутациями.

Метод мутационного тестирования – в разрабатываемую программу вносят мутации, т.е. искусственно создают программы-мутанты. Затем программа и ее мутанты тестируются на одном и том же наборе тестов.

Если на наборе подтверждается правильность программы и, кроме того, выявляются все внесенные в программы-мутанты ошибки, то набор тестов соответствует **мутационному критерию**, а тестируемая программа объявляется правильной. Если некоторые мутанты не выявили всех мутаций, то надо расширять набор тестов и продолжать тестирование.

Метод оценки **скорости выявления ошибок** – основан на модели скорости выявления ошибок, согласно которой тестирование прекращается, если оцененный интервал времени между текущей ошибкой и следующей слишком велик для фазы тестирования приложения.

Тема 4.2 Ручное тестирование программных продуктов

Ручное тестирование (manual testing) – часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно проводится тестировщиками или обычными пользователи путем моделирования возможных сценариев действия пользователя.

Задача тестировщика заключается в поиске наибольшего количества ошибок. Он должен хорошо знать наиболее часто допускаемые ошибки и уметь находить их за минимально короткий период времени. Остальные ошибки, которые не являются типовыми, обнаруживаются только тщательно созданными наборами тестов. Однако, из этого не следует, что для типовых ошибок не нужно составлять тесты.

Ручное тестирование заключается в выполнении задокументированной процедуры, где описана методика выполнения тесто. Методика задает порядок тестов и для каждого теста – список значений параметров, который подается на вход со список результатов на выходе. Так как процедура предназначена для выполнения человеком, в ее описании для краткости могут использоваться некоторые значения по умолчанию, ориентированные на здравый смысл, или ссылки на информацию, хранящуюся в другом документе.

Пример фрагмента процедуры:

– *подать на вход три разных целых числа;*

– *запустить тестовое исполнение;*

– *проверить, соответствует ли полученный результат ожиданиям с учетом поправок;*

– *убедиться в понятности и корректности выдаваемой сопроводительной информации.*

Основными методами ручного тестирования являются: инспекция исходного текста, сквозные просмотры, просмотры за столом, обзоры программ.

Инспекция исходного текста представляет собой набор процедур и приёмов обнаружения ошибок при изучении текста группой специалистов, в которую входят: автор программы, проектировщик, специалист по тестированию и координатор (компетентный программист, но не автор программы).

Общая процедура инспекции состоит из следующих этапов:

– участникам группы заранее выдаётся листинг программы и спецификация на неё;

– программист рассказывает о логике работы программы и отвечает на вопросы инспекторов;

– программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования.

Кроме нахождения ошибок результаты инспекции позволяют программисту увидеть сделанные им промахи, получить возможность оценить свой стиль программирования, выбор алгоритмов и методов тестирования. Инспекция является способом раннего выявления частей программы с большой вероятностью содержащих ошибки, что позволяет при тестировании уделить внимание этим частям.

Сквозные просмотры и инспекция представляют собой набор способов обнаружения ошибок, осуществляемый группой лиц, просматривающих текст программы. Такой просмотр имеет много общего с процессом инспектирования, но отличается процедурой и методами обнаружения ошибок.

Группа по выполнению сквозного контроля состоит из 3-5 человек (председатель или координатор, секретарь, фиксирующий все ошибки, специалист по тестированию, программист и независимый эксперт).

Этапы процедуры сквозного контроля:

– участникам группы заранее выдаётся листинг программы и спецификация на неё.

– участникам заседания предлагается несколько тестов записанных на бумаге и тестовые данные подвергаются обработке в соответствии с логикой программы (каждый тест мысленно выполняется).

Третьим методом ручного обнаружения ошибок является применявшийся ранее других методов метод **проверки за столом**. Это проверка исходного текста или сквозные просмотры, выполняемые одним человеком, который читает текст программы, проверяет его по списку и пропускает через программу тестовые данные. Исходя из принципов тестирования “проверка за столом” должна проводиться человеком, не являющимся автором программы.

Недостатки метода:

1. Проверка представляет собой полностью не упорядоченный процесс.
2. Отсутствие обмена мнениями и здоровой конкуренции.
3. Меньшая эффективность по сравнению с другими методами.

Метод обзора непосредственно не связан с тестированием. Он является методом оценки анонимной программы. Цель этого метода обеспечить сравнительно объективную оценку и самооценку программиста. Выбирается программист, который должен выполнять обязанности администратора системы. Администратор набирает группу от 6 до 20 участников, которые должны быть одного профиля. Каждому участнику предлагается представить для рассмотрения две программы, с его точки зрения наилучшую и наихудшую. Отобранные программы случайным образом распределяются среди участников, им даётся по две программы, наилучшая и наихудшая, но программист не знает какая из них наилучшая и наихудшая. Программист просматривает их и заполняет анкету, в которой предлагается оценить их качество по семи бальной шкале. Кроме того проверяющий даёт общий комментарий и рекомендации по улучшению программы.

Тема 4.3 Модульное тестирование

Модульное тестирование (Unit testing) – тестирование каждой атомарной функциональности приложения отдельно, в искусственно созданной среде. Именно потребность в создании искусственной рабочей среды для определенного модуля, требует от тестировщика знаний в автоматизации тестирования программного обеспечения, некоторых навыков программирования. Данная среда для некоторого юнита создается с помощью **драйверов** и **заглушек.**

Драйвер – определенный модуль теста, который выполняет тестируемый нами элемент.

Заглушка – часть программы, которая симулирует обмен данными с тестируемым компонентом, выполняет имитацию рабочей системы.

Заглушки нужны для:

– имитирования недостающих компонентов для работы данного элемента;

– подачи или возвращения модулю определенного значения, возможность предоставить тестеру самому ввести нужное значение;

– воссоздания определенных ситуаций (исключения или другие нестандартные условия работы элемента).

Прежде всего, нужно очертить рамки, в которых юнит-тестирование оправданно. Во-первых, архитектура проекта должна быть спроектирована в соответствии с идеями ООП (четкое деление на классы, каждый из которых

выполняет свою определенную функцию), что обеспечит систему грамотным делением на модули. Во-вторых, модульное тестирование должно быть менее затратным при поиске дефектов, чем другие виды тестов и должно снижать время отладки кода.

Планирование тестирования происходит на стадии разработки (кодирования) программного обеспечения. На стадии планирования тестирования перед тестировщиком стоит задача поиска компромисса между объемом тестирования, который возможен в теории, и объемом тестирования, который возможен на практике. На данной стадии необходимо ответить на вопрос: «как будем тестировать?». Результатом планирования тестирования является тестовая документация.

Первый вопрос, который встает перед нами: «сколько нужно тестов?». Ответ, который часто дается: тестов должно быть столько, чтобы не осталось неоттестированных участков. Некоторые вводят формальное правило: «код с не оттестированными участками не может быть опубликован».

Проблема в том, что хотя неоттестированный код почти наверняка неработоспособен, но полное покрытие не гарантирует работоспособности. Написание тестов исходя только из уже существующего кода только для того, чтобы иметь стопроцентное покрытие кода тестами – порочная практика. Такой подход со всей неизбежностью приведет к существованию оттестированного, но неработоспособного кода. Кроме того, метод белого ящика, как правило, приводит к созданию позитивных тестов. А ошибки, как правило, находятся негативными тестами. В тестировании вопрос «Как я могу сломать?» гораздо эффективней вопроса «Как я могу подтвердить правильность?».

В первую очередь тесты должны соответствовать не коду, а требованиям. Другими словами, тесты должны базироваться на спецификации.

При подготовке тестового набора рекомендуется начать с простого позитивного теста. Затраты на его создание минимальны. Также вероятность создания кода, не работающего в штатном режиме, гораздо меньше, чем отсутствие обработки исключительных ситуаций. Но исключительные условия в работе программы редки. Как правило, все работает в штатном режиме. Тесты на обработку некорректных условий, находят ошибки гораздо чаще, но если выяснится, что программа не обрабатывает штатные ситуации, то она просто никому не нужна.

Последующие тесты должны создаваться при помощи формальных методик тестирования. Таких как, классы эквивалентности, исследование граничных условий, метод ортогональных матриц и т.д.

Последнюю проверку полноты тестового набора следует проводить с помощью формальной метрики «Code Coverage». Она показывает неполноту

тестового набора. И дальнейшие тесты можно писать на основании анализа неоттестированных участков.

Тема 4.4 Интеграционное тестирование

Любая программная система тестируется как единое целое, а такой процесс как раз и называется интеграционным тестированием. Его главной задачей является проверка разных модулей системы при их системном объединении.

Интеграционное тестирование входит в состав тестирования белого и черного ящика.

Виды интеграционного тестирования:

– большой взрыв (англ. Big bang approach);
– снизу вверх (англ. Bottom-Up Approach);
– сверху вниз (англ. Top-Down Approach);
– смешанный / сэндвич (англ. Hybrid / Sandwich).

При использовании метода большого взрыва Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Рисунок 5 наглядно демонстрирует суть этого метода.

Рисунок 5 – Сущность метода «большой взрыв» при интеграционном тестировании

Такой подход очень хорош для сохранения времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования.

Преимущества:

– весьма удобен в использовании при тестировании небольших систем;

– быстрое нахождение ошибок, а значит, существенная экономия время, которое может быть потрачено на разработку и доработку уже используемого функционала.

Недостатки:

– так как модули завязаны на одной системе, порой очень трудно найти источник дефектов;

– если в системе используется много модулей, может уйти достаточно времени, чтобы пересмотреть все реализованные функциональности.

Метод «снизу вверх» подразумевает, что процесс тестирования начинается с внутреннего уровня модулей и постепенно доходит до наиболее критичных позиций.

Рисунок 6 – Схема интеграционного тестирования методом «снизу вверх»

Представленная на рисунке 6 схема красноречиво свидетельствует о том, что модули верхнего ранга не могут быть интегрированы в ПО до тех пор, пока не будет завершено тестирование модулей нижнего порядка.

При подходе «снизу вверх» может использоваться **драйвер**, который выступает в роли специального «соединителя» между модулями нижнего и верхнего уровней.

Метод «**сверху вниз**», схема работы которого представлена на рисунке 7, является полностью противоположным вышеописанному методу. Суть подхода «сверху вниз» заключается в первостепенном тестировании всех верхних модулей, и только затем QA специалист может приступать к проверке работоспособности нижестоящих модулей.

Рисунок 7 – Схема интеграционного тестирования методом «сверху вниз»

Большинство модулей нижнего уровня тестируются по отдельности, а затем выполняются проверки в совокупности реализованных модулей. По аналогии с подходом «снизу вверх», данный метод также зависим от вызова специальной связующей функции под названием «Функция-заглушка» (англ. Stubs).

Заглушки – это специальные логические операторы с коротким программным кодом, которые применяются для приема входных данных нижними модулями от модулей верхнего уровня при интеграционном тестировании.

«Смешанный» метод, принято также называть как «тестирование смешанной интеграции». Логики подходов «сверху вниз» и «снизу вверх» максимально объединены в этот подход. А значит, его запросто можно считать смешанным, своего рода гибридным методом в интеграционном тестировании.

Итак, самый верхний модуль тестируется отдельно, при этом модули нижнего уровня интегрируются и проверяются с модулями верхнего уровня.

Тема 4.5 Юзабилити-тестирование

Юзабилити-тестирование – это метод, используемый для оценки продукта, при котором продукт тестируется пользователями, принадлежащими к какой-либо целевой аудитории. В течение теста участники пытаются выполнить типичные задачи, а в это время специалисты наблюдают за ними, слушают и делают пометки.

Цель тестирования – выявить проблемы, связанные с удобством использования продукта, собрать количественные данные о том, как пользователи выполняют задания (например, время на выполнение задачи, процент ошибок), и выяснить, насколько продукт их удовлетворяет.

Для выявления проблем удобства использования, в том числе на ранних этапах планирования и разработки программных продуктов, используются два основных подхода:

1. Проверка соответствия принципам обеспечения удобства пользования и корректного визуального представления в контексте функциональных требований посредством экспертной оценки (экспертный подход).

2. Изучение опыта взаимодействия пользователя с приложением через имитацию поведения пользователей (пользовательский подход). Для юзабилити-тестирования одного программного обеспечения могут применяться оба подхода (методика двойной проверки).

При **экспертном** подходе в качестве пользователей выступают два и более экспертов (оптимальное количество для больших проектов 5-6 человек).

Эксперты проходят основные сценарии поведения пользователей и анализируют их с точки зрения:

– стандартов юзабилити для конкретного типа программного продукта (например, Android Material Design для мобильных приложений на платформе Android);

– общих принципов юзабилити (эвристики Якоба Нильсена);

– здравого смысла и опыта.

По результатам прохождения пользовательских сценариев составляется отчет о дефектах.

Преимущества экспертного подхода:

– быстрый в применении;

– эксперты гарантировано понимают общие задачи программного продукта.

Недостаток данного подхода – субъективизм (эксперты не являются реальными пользователями).

При **пользовательском** подходе пользователям (3-5 человек из каждого сегмента целевой аудитории), согласившимся участвовать в тестировании, предлагают пройти наиболее распространенные и наиболее проблемные сценарии. Эксперт протоколирует действия пользователя, фиксирует все в видео-формате, чтобы отследить реакцию (эмоции) пользователя, но никак не влияет на действия пользователя.

Преимущества пользовательского подхода:

– объективные результаты (участвуют реальные пользователи);

– процесс легко измерим.

Возможные измерения при юзабилити-тестировании:

– время выполнения задачи;

– успешность выполнения задачи;

– эффект первого впечатления.

Недостатки пользовательского подхода:

– длительный по времени;

– дорогой (если пользователей привлекают на платной основе);

– большое внимание следует уделить подбору пользователей.

Для реализации пользовательского подхода юзабилити-тестирования необходимо провести предварительную работу, которая включает следующие этапы:

1. Определение цели пользователя, цели бизнеса.

2. Исследование целевой аудитории: составление ее общего портрета, сегментация на группы, описание персонажей как ярких представителей каждой группы.

3. Выявление контекста – ситуаций, при которых пользователь обращается к программному продукту.

4. Составление пользовательских сценариев.

Целевая аудитория – группа пользователей, на которую ориентировано содержание программного продукта. При исследовании целевой аудитории на первом этапе необходимо составить ее общий портрет.

Также существует метод GOMS. Идея метода заключается в том, что все действия пользователя можно представить как набор типовых составляющих (например, нажать ту или иную кнопку на клавиатуре, передвинуть мышь, и т.п.). Для этих типовых составляющих можно провести измерения времени их выполнения (на большом числе пользователей) и получить статистические оценки времени выполнения того или иного элементарного действия.

Оценка качества интерфейса заключается в разложении выполняемой задачи на типовые составляющие и вычислении времени, которое будет в среднем затрачиваться пользователем на выполнение этой задачи.

В данном методе каждая цель или задача (Goal), которую хочет выполнить пользователь с помощью интерфейса, состоит из набора методов (Methods), которые в свою очередь построены из операторов (Operators). Если цель может быть достигнута несколькими способами, то выбор осуществляться по правилам выбора (Selection Rules).

Простым примером может служить выбор пользователем пиктограммы на рабочем столе в графическом интерфейсе ОС Windows:

Операторы:

М – передвинуть указатель мыши в нужную точку (1.1сек);

П – перенос руки между мышью на клавиатурой (0.4сек);

К – нажать клавишу на клавиатуре (0.28сек);

Л – щелчок левой кнопкой мыши (0.1сек);

А – анализ дальнейших действий (1.2сек);

Задача: выбрать пиктограмму.

Метод: Выбор с помощью мыши.

К: нажать Win-D для перехода на рабочий стол.

А: поиск пиктограммы находится на рабочем столе.

П: перенести руку на мышь.

М: переместить курсор в нужную точку.

Л: выбор пиктограммы.

Тема 4.6. Регрессионное тестирование

Регрессионное тестирование – это набор тестов, направленных на обнаружение дефектов в уже протестированных участках приложения. Делается это совсем не для того, чтобы окончательно убедиться в отсутствии

багов, а для поиска и исправления регрессионных ошибок. Регрессионные ошибки – те же баги, но появляются они не при написании программы, а при добавлении в существующий билд (сборку) нового участка программы или исправлении других багов, что и стало причиной возникновения новых дефектов в уже протестированном продукте.

Таким образом, можно сказать, что цель регрессионного тестирования – убедиться, что исправление одних багов не стало причиной возникновения других и что обновление билда не создало новых дефектов в уже проверенном коде.

Есть несколько видов регрессионных тестов:

1. Верификационные тесты. Проводится для проверки исправления обнаруженного и открытого ранее бага.

2. Тестирование верификации версии. Содержит принципы дымного тестирования и тестирование сборки: проверка работоспособности основной функциональности программы в каждой новой сборке.

3. Непосредственно само регрессионное тестирование – повторное выполнение всех тестов, которые были написаны и проведены ранее. Они выполняются по уже существующим тест-кейсам независимо от того, были в ходе их прохождения найдены баги, или нет.

4. Тестирование в новом билде уже исправленных багов в старых билдах. Это выполняется для того, чтобы проверить, не возобновило ли обновление билда старых дефектов.

Некоторые положения относительно того, как проводить регрессионное тестирование:

– данный вид тестирования проводится в каждом новом билде;

– начинать нужно с верификации версии (тестирование сборки и дымное тестирование);

– проверка исправленных багов;

– регрессионное тестирование, в основном, не покрывает все приложение, а только те участки, которые тем или иным способом «соприкасаются» с изменениями в билде.

Далее тестируются уже закрытые ранее баги.

Регрессионное тестирование рекомендуется проводить несколько раз (3-5). Поэтому, с целью экономии драгоценного времени в регрессионных тестах активно используют мощь автоматизации тестирования.

Проведение финального регрессионного тестирования, для которого отбираются тесты по приоритету, определяемому наибольшим количеством найденных ошибок.

Также регрессионное тестирование активно используется в экстремальной разработке.

Тема 4.7. Автоматизированное тестирование

Автоматизированное тестирование программного обеспечения (Software Automation Testing) – это процесс верификации программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи инструментов для автоматизированного тестирования.

Автоматизация процессов тестирования тесно связана с именем Майка Кона, автора книги «Scrum. Гибкая разработка ПО». Он представил систему автоматизации тестирования в виде пирамиды, представленной на рисунке 8.

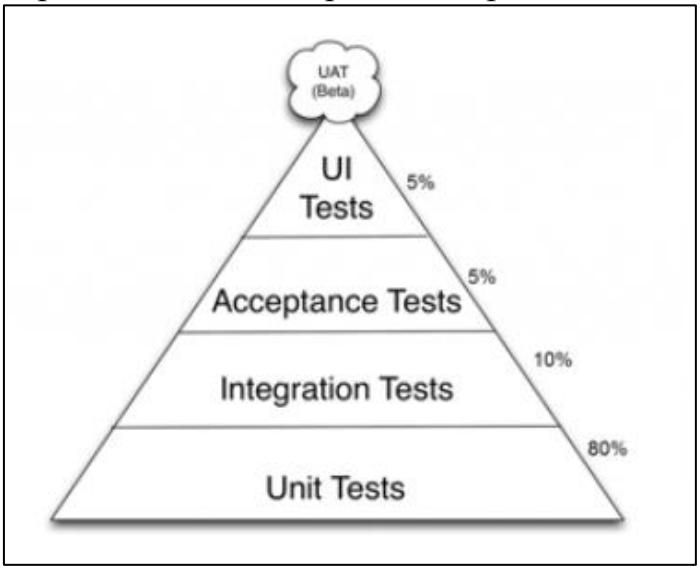

Рисунок 8 – Система автоматизации тестирования Майка Кона

Основание пирамиды составляет модульное тестирование – процесс проверки отдельных модулей исходного кода. За ним следует интеграционное тестирование – на этом этапе отдельные программные модули тестируются в группах. Далее следует приемочное тестирование, определяющее пригодность продукта к предстоящей эксплуатации. На вершине пирамиды обосновалось тестирование пользовательского интерфейса, которое может проводится как автоматизировано, так и вручную. Таким образом, автоматизированное тестирование находится на стыке тестирования и программирования.

Над вершиной часто располагают ручное тестирование в форме облака, поскольку оно не считается неотъемлемой частью пирамиды, хоть и взаимосвязано с ней. Данная пирамида считается типичной для автоматизации тестирования, хотя может быть перевернута и модифицирована в зависимости от специфики работ.

Автоматизация дает тестировщику ряд преимуществ:

1. Оперативность – автоматизированный скрипт не сверяется с инструкциями и документацией;
2. Экономия времени – автоматизация не требует вмешательства тестировщика, в это время он может переключиться на другие задачи;
3. Повторное использование – сценарий тестирования может использоваться неоднократно;
4. Отсутствие «человеческого фактора» – тестовый сценарий не допустит оплошностей в результатах и не пропустит времени тестирования;
5. Автоматическая отчетность – результаты тестирования автоматически сохраняются и рассылаются причастным специалистам.

Вместе с тем данный подход не является панацеей и не исключает ряд недостатков:

1. Затраты – хорошие инструменты автоматизированного тестирования, как и обучение автоматизированному тестированию ПО требует вложений;
2. Однообразие – написанные тесты работают всегда одинаково, что не всегда плохо, но иногда позволяет пропустить дефект, который заметил бы живой человек;
3. Затраты на поддержку и разработку – чем сложнее приложение и чем чаще оно обновляется, тем более затратна разработка и модификация автоматизированных тестов;
4. Пропуск мелких недочетов – тесты пропускают небольшие ошибки, на проверку которых не запрограммированы.

К инструментам автоматизации тестирования относят программное обеспечение, используемое для создания, наладки, выполнения и анализа результатов работы автоматизированных тестов.

Выбор инструмента зависит от объектов тестирования и требований к сценариям тестов. Естественно, что один инструмент не в состоянии поддерживать весь спектр технологий, потому остается только пробовать и искать наиболее подходящий. Достаточно часто QA-инженеры останавливают свой выбор на нескольких инструментах.

Критерии выбора:

– распознавание элементов управления в приложении;
– временные затраты на поддержку скриптов;
– удобство для написания новых скриптов.

К широкоизвестным средствам автоматизации тестирования относятся: Selenium, Coded UI, Appium/XCUITest.

Тема 4.8. Тестирование производительности

Нагрузочное тестирование или тестирование производительности – это автоматизированное тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе.

Существует несколько видов нагрузочного тестирования.

Задачей **тестирования производительности** является определение масштабируемости приложения под нагрузкой, при этом происходит:

– измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций;

– определение количества пользователей, одновременно работающих с приложением;

– определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций);

– исследование производительности на высоких, предельных, стрессовых нагрузках.

Стрессовое тестирование позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при стрессовом тестировании может быть оценка деградации производительности, таким образом цели стрессового тестирования могут пересекаться с целями тестирования производительности.

Задачей **объемного тестирования** является получение оценки производительности при увеличении объемов данных в базе данных приложения, при этом происходит:

– измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций;

– может производиться определение количества пользователей, одновременно работающих с приложением.

Задачей **тестирования стабильности** (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Время выполнения операций может играть в данном виде тестирования второстепенную роль. При этом на первое место выходит отсутствие утечек памяти, перезапусков серверов под нагрузкой и другие аспекты влияющие именно на стабильность работы.

Ниже рассмотрены некоторые экспериментальные факты, обобщённые принципы, используемые при тестировании производительности в целом и применимые к любому типу тестирования производительности (в частности и к нагрузочному тестированию).

1. Уникальность запросов

Даже сформировав реалистичный сценарий работы с системой на основе статистики её использования, необходимо понимать, что всегда найдутся исключения из этого сценария.

2. Время отклика системы

В общем случае время отклика системы подчиняется функции нормального распределения.

3. Зависимость времени отклика системы от степени распределённости этой системы.

Дисперсия (мера разброса значений случайной величины относительно её математического ожидания) нормального распределения времени отклика системы на запрос пропорциональна отношению количества узлов системы, параллельно обрабатывающих такие запросы и количеству запросов, приходящихся на каждый узел.

То есть, на разброс значений времени отклика системы влияет одновременно количество запросов приходящихся на каждый узел системы и само количество узлов, каждый из которых добавляет некоторую случайную величину задержки при обработке запросов.

4. Разброс времени отклика системы

Из утверждений 1, 2 и 3 можно также заключить, что при достаточно большом количестве измерений величины времени обработки запроса в любой системе всегда найдутся запросы, время обработки которых превышает определённые в требованиях максимумы; причем, чем больше суммарное время проведения эксперимента тем выше окажутся новые максимумы.

5. Точность воспроизведения профилей нагрузки

Необходимая точность воспроизведения профилей нагрузки тем дороже, чем больше компонент содержит система.

Часто невозможно учесть все аспекты профиля нагрузки для сложных систем, так как чем сложнее система, тем больше времени будет затрачено на проектирование, программирование и поддержку адекватного профиля нагрузки для неё, что не всегда является необходимостью. Оптимальный подход в данном случае заключается в балансировании между стоимостью разработки теста и покрытием функциональности системы, в результате которого появляются допущения о влиянии на общую производительность той или иной части тестируемой системы.

Тема 4.9 Тестирование локализации и совместимости

Тестирование локализации – это процесс тестирования локализованной версии программного продукта. Проверка правильности перевода элементов интерфейса пользователя, проверка правильности перевода системных сообщений и ошибок, проверка перевода раздела «Помощь»/»Справка» и сопроводительной документации.

С помощью тестирования локализации проверяются перевод, вспомогательные файлы, правильное обоснование и адаптация элементов интерфейса, а также правила написания текста.

Цель теста локализации – убедиться, что приложение поддерживает многоязыковой интерфейс и функции. А также проблемы связанные с локализацией (перевод на другой язык, формат дат и чисел, почтовые адреса, порядок имени и фамилии, валюты и т.д.).

Процесс локализации тестирования включает в себя:

– определение и изучение списка поддерживаемых языков;

– проверка правильности перевода согласно тематике данного сайта или приложения;

– проверка правильности перевода элементов интерфейса пользователя;

– проверка правильности перевода системных сообщений и ошибок;

– проверка перевода раздела «Помощь» и сопроводительной документации.

Тестирование совместимости (англ. compatibility testing) – вид нефункционального тестирования, основной целью которого является проверка корректной работы продукта в определенном окружении.

Окружение может включать в себя следующие элементы:

– аппаратная платформа;

– опперационная система (Unix, Windows, MacOS);

– браузеры (Internet Explorer, Firefox, Opera, Chrome, Safari);

– различное расширение экрана.

Обычно на первом этапе проверяется взаимодействие выпускаемого продукта с окружением, в которое он будет установлен, на различных аппаратных средствах.

На следующем этапе ПО проверяется с позиции его конечного пользователя и конфигурации его рабочей станции.

ЛИТЕРАТУРА

1. IT академия Stormnet [Электронный ресурс] / Тестирование программного обеспечения. – Режим доступа: https://www.it-courses.by/sqa/ – Дата доступа: 20.02.2020.
2. Сообщество IT-специалистов Хабр [Электронный ресурс] / Модели жизненного цикла программного обеспечения. – Минск, 2020. – Режим доступа: https://habr.com/ru/post/111674/ – Дата доступа: 20.02.2020.
3. Boehm B.W. The COCOMO 2.0 Software Cost Estimation Model. – American Programmer. – 2000. – 586 p.
4. ISO/IEC 9126 Software engineering – Product quality – Part 1: Quality model, 2001.
5. ISO/IEC 9126 Software engineering – Product quality – Part 2: External metrics, 2001.
6. ISO/IEC 9126 Software engineering – Product quality – Part 3: Internal metrics, 2001.
7. ISO/IEC 9126 Software engineering – Product quality – Part 4: Quality in use metrics, 2001.
8. Куликов, С.С. Тестирование программного обеспечения. Базовый курс: практ. пособие. / С.С. Куликов. – Минск: Четыре четверти, 2015. – 294 с.
9. ISTQB Стандартный глоссарий терминов, используемых в тестировании программного обеспечения. – 2014. – 73 с.
10. Вигерс, К. Разработка требований к программному обеспечению. 3-е изд., дополненное. Пер. с англ. / К. Вигерс. – М.: Издательство «Русская редакция» ; СПб.: БХВ-Петербург, 2014. – 736 стр.
11. Блэк, Р. Ключевые процессы тестирования / Р. Блэк. – М.: BHV, 2008.
12. Тамрэ, Л. Введение в тестирование программного обеспечения / Л. Тамрэ. – М.: Вильямс, 2005.
13. Бек, К. Экстремальное программирование: разработка через тестирование / К. Бек. – С-Пб.: Питер, 2005.
14. Бейзер, Б. Тестирование черного ящика. Технологии функционального тестирования ПО / Б. Бейзер. – С-Пб.: Питер, 2006.
15. Стотлемайер, Д. Тестирование Web-приложений / Д. Стотлемайер. – М.: Кудиц-образ, 2003.
16. Copeland, L.A Practitioner's Guide to Software Test Design / L. Copeland. – London: Artech House Publishers, 2004. – 274 p.
17. Котлярова, В.П. Технология программирования. Основы современного тестирования программного обеспечения, разработанного на C#: учеб. пособие / под общ. ред. В.П. Котлярова. – СПб: СПбГПУ, 2004. – 168 с.

18. Bertolino, A. Software Testing Research: Achievements, Challenges, Dreams / A. Bertolino Future of Software Engineering. Briand L. and Wolf A. (eds.). IEEE-CS Press. 2007, pp. 85–103.
19. Макгрегор, Дж. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие / Дж. Макгрегор, Д. Сайкс; пер. с англ. – К.: ООО «ТИД «ДС»», 2002. – 432 с.
20. Пивень, А.А. Тестирование программного обеспечения / А.А. Пивень, Ю.И. Скорин // Системы обработки информации. – 2011. – № 4. – С. 56-58.
21. ISO/IEC/IEEE 29119-1:2013 Software and systems engineering – Software testing – Part 1: Concepts and definitions.
22. ISO/IEC/IEEE 29119-2:2013 Software and systems engineering – Software testing – Part 2: Test processes.
23. ISO/IEC/IEEE 29119-3:2013 Software and systems engineering – Software testing – Part 3: Test documentation.