



ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ЛЕКЦИЯ № 8

ПРЕПОДАВАТЕЛЬ: ХУСТОЧКА А.В.



КАЧЕСТВО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- Каждый день в своей работе мы сталкиваемся с достаточно абстрактным понятием «качество ПО» и если задать вопрос тестировщику или программисту «что такое качество?», то у каждого найдется своё толкование. Рассмотрим определение "качества ПО" в контексте международных стандартов:
- Качество программного обеспечения - это степень, в которой ПО обладает требуемой комбинацией свойств.
- Качество программного обеспечения - это совокупность характеристик ПО, относящихся к его способности удовлетворять установленные и предполагаемые потребности.

ХАРАКТЕРИСТИКИ КАЧЕСТВА ПО

- Функциональность (Functionality) - определяется способностью ПО решать задачи, которые соответствуют зафиксированным и предполагаемым потребностям пользователя, при заданных условиях использования ПО. Т.е. эта характеристика отвечает 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.
- 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 адаптирования к имеющемуся окружению.
- Портативность (Portability) – характеризует ПО с точки зрения легкости его переноса из одного окружения (software/ hardware) в другое.

МОДЕЛЬ КАЧЕСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- На данный момент, наиболее распространена и используется многоуровневая модель качества программного обеспечения, представленная в наборе стандартов ISO 9126. На верхнем уровне выделено 6 основных характеристик качества ПО, каждую из которых определяют набором атрибутов, имеющих соответствующие метрики для последующей оценки.

Качество ПО

```
graph TD; A[Качество ПО] --> B[Функциональность:]; A --> C[Надежность:]; A --> D[Удобство использования:]; A --> E[Эффективность:]; A --> F[Удобство сопровождения:]; A --> G[Портативность:];
```

Функциональность:

- функциональная исправность;
- соответствие стандартам;
- функциональная совместимость;
- безопасность;
- точность.

Надежность:

- завершенность;
- восстанавливаемость;
- устойчивость к отказам.

Удобство использования:

- удобство изучения;
- понятность;
- удобство и простота использования.

Эффективность:

- эффективность по времени;
- эффективность использования ресурсов.

Удобство сопровождения:

- стабильность;
- анализируемость;
- контролепригодность;
- изменяемость.

Портативность:

- удобство установки;
- заменяемость;
- совместимость.

КТО ТАКОЙ ТЕСТИРОВЩИК И ЧТО ОН ДЕЛАЕТ?

Тестировщик (Испытатель) — специалист, принимающий участие в тестировании компонента или системы. В его обязанность входит поиск вероятных ошибок и сбоев в функционировании объекта тестирования (продукта, программы). Тестировщик моделирует различные ситуации, которые могут возникнуть в процессе использования предмета тестирования, чтобы разработчики смогли исправить обнаруженные ошибки.

Также отметим личностные качества, позволяющие тестировщику быстрее стать отличным специалистом:

- Повышенная ответственность и исполнительность;
- хорошие коммуникативные навыки, способность ясно, быстро, чётко выражать свои мысли;
- терпение, усидчивость, внимательность к деталям, наблюдательность;
- хорошее абстрактное и аналитическое мышление;
- способность ставить нестандартные эксперименты, склонность к исследовательской деятельности.

ОТКУДА БЕРУТСЯ ОШИБКИ В ПО?

- Почему бывает так, что программы работают неправильно? Все очень просто – они создаются и используются людьми. Если пользователь допустит ошибку, то это может привести к проблеме в работе программы – она используется неправильно, значит, может повести себя не так, как ожидалось.

ОШИБКА (ERROR)

- Ошибка – это действие человека, которое порождает неправильный результат.
- Однако программы разрабатываются и создаются людьми, которые также могут допускать (и допускают) ошибки. Это значит, что недостатки есть и в самом программном обеспечении. Они называются дефектами или багами (оба обозначения равносильны). Здесь важно помнить, что программное обеспечение – нечто большее, чем просто код.

ДЕФЕКТ, БАГ (DEFECT, BUG)

- Дефект, Баг – недостаток компонента или системы, который может привести к отказу определенной функциональности. Дефект, обнаруженный во время исполнения программы, может вызвать отказ отдельного компонента или всей системы.
- При исполнении кода программы дефекты, заложенные еще во время его написания, могут проявиться: программа может не делать того, что должна или наоборот делать то, чего не должна – происходит сбой.

СБОЙ (FAILURE)

- Сбой – несоответствие фактического результата (actual result) работы компонента или системы ожидаемому результату (expected result).
- Сбой в работе программы может являться индикатором наличия в ней дефекта.



Таким образом, баг существует при одновременном выполнении трех условий:

- известен ожидаемый результат;
- известен фактический результат;
- фактический результат отличается от ожидаемого результата.

Важно понимать, что не все баги становятся причиной сбоев – некоторые из них могут никак себя не проявлять и оставаться незамеченными (или проявляться только при очень специфических обстоятельствах).

Причиной сбоев могут быть не только дефекты, но также и условия окружающей среды: например, радиация, электромагнитные поля или загрязнение также могут влиять на работу как программного, так и аппаратного обеспечения.



Всего существует несколько источников дефектов и, соответственно, сбоев:

- ошибки в спецификации, дизайне или реализации программной системы;
- ошибки использования системы;
- неблагоприятные условия окружающей среды;
- умышленное причинение вреда;
- потенциальные последствия предыдущих ошибок, условий или умышленных действий.

Дефекты могут возникать на разных уровнях, и от того, будут ли они исправлены и когда, будет напрямую зависеть качество системы.

Условно можно выделить пять причин появления дефектов в программном коде.

- Недостаток или отсутствие общения в команде. Зачастую бизнес-требования просто не доходят до команды разработки. У заказчика есть понимание того, каким он хочет видеть готовый продукт, но, если должным образом не объяснить его идею разработчикам и тестировщикам, результат может оказаться не таким, как предполагалось. Требования должны быть доступны и понятны всем участникам процесса разработки ПО.
- Сложность программного обеспечения. Современное ПО состоит из множества компонентов, которые объединяются в сложные программные системы. Многопоточные приложения, клиент-серверная и распределенная архитектура, многоуровневые базы данных – программы становятся все сложнее в написании и поддержке, и тем труднее становится работа программистов. А чем труднее работа, тем больше ошибок может допустить исполняющий ее человек.
- Изменения требований. Даже незначительные изменения требований на поздних этапах разработки требуют большого объема работ по внесению изменений в систему. Меняется дизайн и архитектура приложения, что, в свою очередь, требует внесения изменений в исходный код и принципы взаимодействия программных модулей. Такие текущие изменения зачастую становятся источником трудноуловимых дефектов. Тем не менее, часто меняющиеся требования в современном бизнесе – скорее правило, чем исключение, поэтому непрерывное тестирование и контроль рисков в таких условиях – прямая обязанность специалистов отдела обеспечения качества.
- Плохо документированный код. Сложно поддерживать и изменять плохо написанный и слабо документированный программный код. Во многих компаниях существуют специальные правила по написанию и документированию кода программистами. Хотя на практике часто бывает так, что разработчики вынуждены писать программы в первую очередь быстро, а это сказывается на качестве продукта.
- Средства разработки ПО. Средства визуализации, библиотеки, компиляторы, генераторы скриптов и другие вспомогательные инструменты разработки – это тоже зачастую плохо работающие и слабо документированные программы, которые могут стать источником дефектов в готовом продукте.

ПРИНЦИПЫ ТЕСТИРОВАНИЯ

- Тестирование программного обеспечения – креативная и интеллектуальная работа. Разработка правильных и эффективных тестов – достаточно непростое занятие. Принципы тестирования, представленные ниже, были разработаны в последние 40 лет и являются общим руководством для тестирования в целом.

1. ТЕСТИРОВАНИЕ ПОКАЗЫВАЕТ НАЛИЧИЕ ДЕФЕКТОВ

- Тестирование может показать наличие дефектов в программе, но не доказать их отсутствие. Тем не менее, важно составлять тест-кейсы, которые будут находить как можно больше багов. Таким образом, при должном тестовом покрытии, тестирование позволяет снизить вероятность наличия дефектов в программном обеспечении. В то же время, даже если дефекты не были найдены в процессе тестирования, нельзя утверждать, что их нет.

2. ИСЧЕРПЫВАЮЩЕЕ ТЕСТИРОВАНИЕ НЕВОЗМОЖНО

- Невозможно провести исчерпывающее тестирование, которое бы покрывало все комбинации пользовательского ввода и состояний системы, за исключением совсем уж примитивных случаев. Вместо этого необходимо использовать анализ рисков и расстановку приоритетов, что позволит более эффективно распределять усилия по обеспечению качества ПО.

3. РАННЕЕ ТЕСТИРОВАНИЕ

- Тестирование должно начинаться как можно раньше в жизненном цикле разработки программного обеспечения и его усилия должны быть сконцентрированы на определенных целях.

4. СКОПЛЕНИЕ ДЕФЕКТОВ

- Разные модули системы могут содержать разное количество дефектов, то есть плотность скопления дефектов в разных элементах программы может отличаться. Усилия по тестированию должны распределяться пропорционально фактической плотности дефектов. В основном, большую часть критических дефектов находят в ограниченном количестве модулей. Это проявление принципа Парето: 80% проблем содержатся в 20% модулей.

5. ПАРАДОКС ПЕСТИЦИДА

- Прогоняя одни и те же тесты вновь и вновь, Вы столкнетесь с тем, что они находят все меньше новых ошибок. Поскольку система эволюционирует, многие из ранее найденных дефектов исправляют и старые тест-кейсы больше не срабатывают.
- Чтобы преодолеть этот парадокс, необходимо периодически вносить изменения в используемые наборы тестов, рецензировать и корректировать их с тем, чтобы они отвечали новому состоянию системы и позволяли находить как можно большее количество дефектов.

6. ТЕСТИРОВАНИЕ ЗАВИСИТ ОТ КОНТЕКСТА

- Выбор методологии, техники и типа тестирования будет напрямую зависеть от природы самой программы. Например, программное обеспечение для медицинских нужд требует гораздо более строгой и тщательной проверки, чем, скажем, компьютерная игра. Из тех же соображений сайт с большой посещаемостью должен пройти через серьезное тестирование производительности, чтобы показать возможность работы в условиях высокой нагрузки.

7. ЗАБЛУЖДЕНИЕ ОБ ОТСУТСТВИИ ОШИБОК.

- Тот факт, что тестирование не обнаружило дефектов, еще не значит, что программа готова к релизу. Нахождение и исправление дефектов будет не важным, если система окажется неудобной в использовании и не будет удовлетворять ожиданиям и потребностям пользователя.

И ЕЩЕ НЕСКОЛЬКО ВАЖНЫХ ПРИНЦИПОВ:

- тестирование должно производиться независимыми специалистами;
- привлекайте лучших профессионалов;
- тестируйте как позитивные, так и негативные сценарии;
- не допускайте изменений в программе в процессе тестирования;
- указывайте ожидаемый результат выполнения тестов.

ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ

Эти два понятия тесно связаны с процессами тестирования и обеспечения качества. К сожалению, их часто путают, хотя отличия между ними достаточно существенны.

- Верификация (verification)– это процесс оценки системы или её компонентов с целью определения того, удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа. То есть выполняются ли задачи, цели и сроки по разработке продукта.
- Валидация (validation)– это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе.
- С помощью валидации Вы можете быть уверенным в том, что создали «правильный» продукт. Продукт, который полностью удовлетворяет заказчика.
- С помощью верификации Вы можете убедиться в том, что продукт сделан «правильно»: придерживаясь необходимых методик, инструментов и стандартов.

На практике отличия верификации и валидации имеют большое значение:

- заказчика интересует, в большей степени, валидация (удовлетворение собственных требований);
- исполнителя, в свою очередь, волнует не только соблюдение всех норм качества (верификация) при реализации продукта, а и соответствие всех особенностей продукта желаниям заказчика.

ЭТАПЫ ТЕСТИРОВАНИЯ

Процесс тестирования состоит из таких этапов:

- Планирование и управление - планирование тестирования включает действия, направленные на определение основных целей тестирования и задач, выполнение которых необходимо для достижения этих целей; составление тест-стратегии, тест-плана.
- Анализ и проектирование - это процесс написания тестовых сценариев и условий на основе общих целей тестирования.
- Внедрение и реализация - написание тест-кейсов, на основе написанных ранее тестовых сценариев, собирается необходимая для проведения тестов информация, подготавливается тестовое окружение и запускаются тесты.
- Оценка критериев выхода и написание отчетов - необходимо проверить было ли проведено достаточное количество тестов, достигнута ли нужная степень обеспечения качества системы.

Действия по завершению тестирования - собираем, систематизируем и анализируем информацию о его результатах.

ЦЕЛЬ ТЕСТИРОВАНИЯ

Основные цели:

- убедиться, что вся запланированная функциональность действительно была реализована;
- проверить, что все отчеты об ошибках, поданные ранее, были, так или иначе, закрыты;
- завершение работы тестового обеспечения, тестового окружения и инфраструктуры;
- оценить общие результаты тестирования и проанализировать опыт, полученный в его процессе.

АНАЛИЗ ТРЕБОВАНИЙ. ПАРАМЕТРЫ ТЕСТИРОВАНИЯ ДОКУМЕНТАЦИИ.

1. Четкость и ясность

- Начать тестирование требований можно с поверхностного осмотра документации. Это сложно назвать именно тестированием, но нередко уже на данном этапе выявляется немало недочетов.
- Начнем с обычного сценария. Вы начали читать требования и уже с первых строк у Вас возникает масса вопросов к автору (например, «Каков ожидаемый результат после нажатия на эту кнопку?» или «Что будет, если я отменю заказ?»). Это плохо. После прочтения документации не должно быть вопросов. Совсем.
- Требования – это как свод законов для продукта, а законы не допускают двусмысленность, «воду» и неточности. Документация должна давать предельно ясную информацию о том, как должен работать каждый отдельный модуль и весь продукт в целом. К сожалению, после прочтения большинства требований остается целый ряд вопросов.

- Пример. В требованиях было записано: «В поле «Имя пользователя» могут быть введены буквы и цифры». Разработчик начал выяснять у аналитика, какие именно буквы (кириллица, латиница или арабские) и какие цифры (целые, дробные, римские) имеются в виду. После уточнения требований разработчик реализовал функционал согласно комментариям аналитика. Задача перешла в тестирование. Тестировщик не понимал, по каким критериям проверять данное поле, и тоже начал расспрашивать аналитика.

Последствия:

- Затраченное время нескольких членов команды.
- Несовпадение итогового и изначально планируемого функционалов.

Как тестировать:

- Если у Вас после прочтения требований остались вопросы – необходима доработка.
- Если разработчики часто уточняют детали в чатах – это плохой знак.

Дальнейшее («более глубокое») исследование требует гораздо больших временных затрат.



2. Актуальность

- Необходимость поддержания актуальности требований кажется очевидной. Однако, на некоторых проектах требования не обновляются месяцами, а то и годами. Это может быть связано с тем, что в штате нет аналитика, а у исполняющего его обязанности сотрудника просто не хватает времени. Случается и другое: требования обновляют только при наличии действительно значимых изменений, при этом различные «мелочи», в виде изменения кнопок или текстов, игнорируются.

- Пример. Было решено изменить положение кнопок на странице авторизации. Аналитик не стал править документацию, а написал разработчику личное сообщение с просьбой поправить расположение кнопок. Разработчик внес правки и закрыл задачу. Во время очередного регрессионного тестирования тестировщик решил, что это дефект, и завел на него баг. Другой разработчик вернул кнопки на прежние позиции согласно документации.

Последствия:

- Время нескольких членов команды потрачено впустую.
- Итоговая позиция кнопок не соответствует ожидаемому результату.

Как тестировать:

- При наличии подобных сообщений в командном чате нужно убедиться, что обновленные требования задокументированы.
- Необходимо сравнить даты обновления технического задания и пояснительной записки с датой последнего обновления требований.

3. Логика

- Как следует из названия, работа системы должна быть логичной. Пользователь не может изменить настройки своего профиля или написать письмо до того, как пройдет авторизацию в системе. Звучит, опять же, элементарно, но в проектах со множеством клиентов или со сложной логикой подобные ошибки часто допускаются.

- Пример. В мобильном приложении появилась необходимость реализовать функционал электронной подписи документа. Пользователю предлагалось ввести свои данные, после чего они автоматически подставлялись в шаблон документа. Приложение открывало документ и предлагало его подписать. Если пользователь понимал, что в документе есть ошибки, то исправить он их уже не мог: у него была возможность только подписать этот документ. Заккрытие приложения или его переустановка не помогали – при входе пользователя в аккаунт сразу отображался тот же документ на подпись.

Последствия:


- Пользователь в бешенстве.
- Дальнейшая работа с аккаунтом без обращения в техподдержку невозможна.

Как тестировать:

- Нарисовать примерную блок-схему работы системы в соответствии с требованиями и убедиться, что в ней нет логических пробелов.
- Убедиться, что в требованиях описан необходимый основной функционал.
- Убедиться, что взаимодействие между модулями системы изложено корректно.

4. Возможные сценарии

- В документации должны быть подробно описаны как очевидные, так и неочевидные варианты использования системы. К очевидным (позитивным) вариантам, например, можно отнести ввод корректной пары логин/пароль. К неочевидным (негативным) – ввод некорректной пары логин/ пароль или отсутствие этих данных вовсе.

- 
- Пример. Часто из виду упускаются такие моменты, как тексты ошибок, поведение системы при потере связи, а также обработка ошибок, связанных со сторонними сервисами (например, с оплатой).

Последствия:

- При потере связи система ведет себя некорректно (отсутствие ошибок, зависание).
- Сообщения об ошибках не очевидны.
- В худшем случае возможны репутационные или финансовые потери.

Как тестировать:

- Нарисовать блок-схему отдельного модуля системы, в рамках которой обозначить все возможные условия и действия пользователя.
- Убедиться, что в требованиях есть описание каждого возможного случая.



5. Интеграция

- Имеет смысл выделить интеграцию со сторонними сервисами, так как здесь приходится выходить за рамки проверки документации. Перед началом разработки аналитики, как правило, изучают работу сторонней системы, а затем описывают схему взаимодействия этой системы с разрабатываемым продуктом. В данном случае, вероятность ошибки очень велика, так как ошибиться могут как аналитики, так и представители стороннего сервиса, которые консультировали или писали документацию.

- Пример. На проекте необходимо было реализовать возможность авторизации через сторонний сервис. Аналитик по ошибке изучил устаревшую документацию стороннего сервиса и описал заведомо нерабочую схему взаимодействия. Разработчики начали работу, в соответствии с готовой схемой, но постоянно получали ошибки. Они «допрашивали» аналитика, а тот в спешке звонил в техподдержку стороннего сервиса и выяснял причины ошибок.

Последствия:

- Задержка разработки функционала на неделю.

Как тестировать:

- Необходимо вручную проверить, что сторонний сервис обрабатывает все необходимые запросы, в соответствии с описанной схемой.
- Проверить, указал ли аналитик корректно и в полном объеме всю необходимую для разработки информацию.

СВОЙСТВА КАЧЕСТВЕННЫХ ТРЕБОВАНИЙ

В процессе тестирования требований проверяется их соответствие определённому набору свойств:

- Атомарность, единичность (atomicity). Требование является атомарным, если его нельзя разбить на отдельные требования без потери завершённости и оно описывает одну и только одну ситуацию.
- Непротиворечивость, последовательность (consistency). Требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам.
- Недвусмысленность (unambiguousness, clearness). Требование должно быть описано без использования жаргона, неочевидных аббревиатур и расплывчатых формулировок, а также должно допускать только однозначное объективное понимание и быть атомарным в плане невозможности различной трактовки сочетания отдельных фраз.
- Обязательность, нужность (obligatoriness) актуальность (up-to-date). Если требование не является обязательным к реализации, то оно должно быть просто исключено из набора требований. Если требование нужное, но «не очень важное», для указания этого факта используется указание приоритета (см. «проранжированность по...»). Также должны быть исключены (или переработаны) требования, утратившие актуальность.

- Прослеживаемость (traceability). Прослеживаемость бывает вертикальной (vertical traceability) и горизонтальной (horizontal traceability). Вертикальная прослеживаемость позволяет соотносить между собой требования на различных уровнях требований, горизонтальная - 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).
- Модифицируемость (modifiability) - это свойство характеризует 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 её изменение не приводит к нарушению иных описанных в этом перечне свойств.
- 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 ближайшем будущем в требование не будет внесено никаких изменений. Срочность определяет распределение по времени усилий проектной команды по реализации того или иного требования.
- Корректность (correctness) и проверяемость (verifiability). Фактически, эти свойства вытекают из соблюдения всех вышеперечисленных (или, можно сказать, 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.

ТИПЫ ТЕСТИРОВАНИЯ

- Для того, чтобы лучше понимать подходы к тестированию программного обеспечения, нужно, конечно же, знать, какие виды и типы тестирования в принципе бывают. Давайте начнем с рассмотрения основных типов тестирования, которые определяют высокоуровневую классификацию тестов.
- Самым высоким уровнем в иерархии подходов к тестированию будет понятие типа, которое может охватывать сразу несколько смежных техник тестирования. То есть, одному типу тестирования может соответствовать несколько его видов. Рассмотрим, для начала, несколько типов тестирования, которые отличаются знанием внутреннего устройства объекта тестирования.

BLACK BOX

Summary: Мы не знаем, как устроена тестируемая система.

- Тестирование методом «черного ящика», также известное как тестирование, основанное на спецификации или тестирование поведения – техника тестирования, основанная на работе исключительно с внешними интерфейсами тестируемой системы.

Согласно ISTQB, тестирование черного ящика – это:

- тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы;
- тест-дизайн, основанный на технике черного ящика – процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.

Почему именно «черный ящик»? Тестируемая программа для тестировщика – как черный непрозрачный ящик, содержания которого он не видит. Целью этой техники является поиск ошибок в таких 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.



Пример:

- Тестировщик проводит тестирование веб-сайта, не зная особенностей его реализации, используя только предусмотренные разработчиком поля ввода и кнопки. Источник ожидаемого результата – спецификация.
- Поскольку это тип тестирования, то он может включать и другие его виды. Тестирование черного ящика может быть как функциональным, так и нефункциональным. Функциональное тестирование предполагает проверку работы функций системы, а нефункциональное – общие характеристики нашей программы.
- Техника черного ящика применима на всех уровнях тестирования (от модульного до приемочного), для которых существует спецификация. Например, при осуществлении системного или интеграционного тестирования, требования или функциональная спецификация будут основой для написания тест-кейсов.

Техники тест-дизайна, основанные на использовании черного ящика, включают:

- классы эквивалентности;
- анализ граничных значений;
- таблицы решений;
- диаграммы изменения состояния;
- тестирование всех пар.

Преимущества:

- тестирование производится с позиции конечного пользователя и может помочь обнаружить неточности и противоречия в спецификации;
- тестировщику нет необходимости знать языки программирования и углубляться в особенности реализации программы;
- тестирование может производиться специалистами, независимыми от отдела разработки, что помогает избежать предвзятого отношения;
- можно начинать писать тест-кейсы, как только готова спецификация.

Недостатки:

- тестируется только очень ограниченное количество путей выполнения программы;
- без четкой спецификации (а это скорее реальность на многих проектах) достаточно трудно составить эффективные тест-кейсы;
- некоторые тесты могут оказаться избыточными, если они уже были проведены разработчиком на уровне модульного тестирования.
- Противоположностью техники черного ящика является тестирование методом белого ящика, речь о котором пойдет ниже.

WHITE BOX

Summary: Нам известны все детали реализации тестируемой программы.

- Тестирование методом белого ящика (также прозрачного, открытого, стеклянного ящика или же основанное на коде или структурное тестирование) – 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 внутреннее устройство системы за пределы ее внешних интерфейсов.

Согласно ISTQB: тестирование 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.

Почему «белый ящик»? Тестируемая программа для тестировщика – прозрачный ящик, содержимое которого он прекрасно видит.



Пример:

- Тестировщик уточняет у программиста реализацию кода поля ввода на веб-странице, определяет все предусмотренные (как правильные, так и неправильные) и не предусмотренные пользовательские вводы и сравнивает фактический результат выполнения программы с ожидаемым. При этом ожидаемый результат определяется именно тем, как должен работать код программы.
- Тестирование методом белого ящика похоже на работу механика, который изучает двигатель машины, чтобы понять, почему она не заводится.
- Техника белого ящика применима на разных уровнях тестирования: от модульного до системного, но, главным образом, применяется именно для реализации модульного тестирования компонента его автором.



Преимущества:

- тестирование может производиться на ранних этапах: нет необходимости ждать создания пользовательского интерфейса;
- можно провести более тщательное тестирование с покрытием большого количества путей выполнения программы.

Недостатки:

- для выполнения тестирования белого ящика необходимо большое количество специальных знаний;
- при использовании автоматизации тестирования на этом уровне поддержка тестовых скриптов может оказаться достаточно накладной, если программа часто изменяется.

GREY BOX

Summary: Нам известны только некоторые особенности реализации тестируемой системы.

- Тестирование методом серого ящика – метод тестирования программного обеспечения, который предполагает комбинацию White Box и Black Box подходов. То есть внутреннее устройство программы нам известно лишь частично. Предполагается, например, доступ ко внутренней структуре и алгоритмам работы ПО для написания максимально эффективных тест-кейсов, но само тестирование проводится с помощью техники черного ящика, то есть с позиции пользователя.
- Эту технику тестирования также называют методом полупрозрачного ящика: что-то мы видим, а что-то – нет.



Пример:

- Тестировщик изучает код программы с тем, чтобы лучше понимать принципы ее работы и изучить возможные пути ее выполнения. Такое знание поможет написать тест-кейс, который наверняка будет проверять определенную функциональность.
- Техника серого ящика применима на разных уровнях тестирования: от модульного до системного, но, главным образом, применяется на интеграционном уровне для проверки взаимодействия разных модулей программы.

СТАТИЧЕСКОЕ И ДИНАМИЧЕСКОЕ ТЕСТИРОВАНИЕ

- По критерию запуска программы (исполняется ли программный код) выделяют еще два типа тестирования: статическое и динамическое.

СТАТИЧЕСКОЕ ТЕСТИРОВАНИЕ

- Статистическое тестирование – тип тестирования, который предполагает, что программный код во время тестирования не будет выполняться. При этом, само тестирование может быть как ручным, так и автоматизированным.
- Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации. Для этого типа тестирования в некоторых случаях даже не нужен компьютер, например, при проверке требований.
- Большинство статических техник могут быть использованы для «тестирования» любых форм документации, включая вычитку кода, инспекцию проектной документации, функциональной спецификации и требований.
- Даже статическое тестирование может быть автоматизировано, например, можно использовать автоматические средства проверки синтаксиса программного кода.

Виды статического тестирования:

- вычитка исходного кода программы;
- проверка требований.

ДИНАМИЧЕСКОЕ ТЕСТИРОВАНИЕ

- Динамическое тестирование – тип тестирования, который предполагает запуск программного кода. Таким образом, анализируется поведение программы во время ее работы.
- Для выполнения динамического тестирования необходимо, чтобы тестируемый программный код был написан, скомпилирован и запущен. При этом, может выполняться проверка внешних параметров работы программы: загрузка процессора, использование памяти, время отклика и т.д., то есть ее производительность.
- Динамическое тестирование является частью процесса валидации программного обеспечения.

Кроме того, динамическое тестирование может включать разные подвиды, каждый из которых зависит от:

- Доступа к коду (тестирование черным, белым и серым ящиками).
- Уровня тестирования (модульное интеграционное, системное и приемочное тестирование).
- Сферы использования приложения (функциональное, нагрузочное, тестирование безопасности и пр.).

РУЧНОЕ И АВТОМАТИЗИРОВАННОЕ

- По критерию способа тестирования программы выделяют следующие типа тестирования: ручное и автоматизированное.

РУЧНОЕ ТЕСТИРОВАНИЕ

- При ручном тестировании (manual testing) тестировщики вручную выполняют тесты, не используя никаких средств автоматизации. Ручное тестирование – самый низкоуровневый и простой тип тестирования, не требующий большого количества дополнительных знаний.
- Тем не менее, перед тем, как автоматизировать тестирование любого приложения, необходимо сначала выполнить серию тестов вручную. Мануальное тестирование требует значительных усилий, но без него мы не сможем убедиться в том, возможна ли автоматизация в принципе. Один из фундаментальных принципов тестирования гласит: 100% автоматизация невозможна. Поэтому, ручное тестирование – необходимость.

Мифы о ручном тестировании:

- Кто угодно может провести ручное тестирование. Нет, выполнение любого вида тестирования требует специальных знаний и профессиональной подготовки.
- Автоматизированное тестирование мощнее ручного.
- Полная автоматизация невозможна. Необходимо использовать также и ручное тестирование.
- Ручное тестирование – это просто.

Тестирование может быть очень непростым занятием. Проведение тестирования для проверки максимально возможного количества путей выполнения, с использованием минимального числа тест-кейсов, требует серьезных аналитических навыков.

АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ

- Автоматизированное тестирование (automation testing) предполагает использование специального программного обеспечения (помимо тестируемого) для контроля выполнения тестов и сравнения ожидаемого фактического результата работы программы. Этот тип тестирования помогает автоматизировать часто повторяющиеся, но необходимые для максимизации тестового покрытия, задачи.
- Некоторые задачи тестирования, такие как низкоуровневое регрессионное тестирование, могут быть трудозатратными и требующими много времени, если выполнять их вручную. Кроме того, мануальное тестирование может недостаточно эффективно находить некоторые классы ошибок. В таких случаях автоматизация может помочь сэкономить время и усилия проектной команды.
- После создания автоматизированных тестов, их можно в любой момент запустить снова, причем, запускаются и выполняются они быстро и точно. Таким образом, если есть необходимость частого повторного прогона тестов, значение автоматизации для упрощения сопровождения проекта и снижения его стоимости трудно переоценить. Ведь даже минимальные патчи и изменения кода могут стать причиной появления новых багов.



Существует несколько основных видов автоматизированного тестирования:

- Автоматизация тестирования кода (Code-driven testing) – тестирование на уровне программных модулей, классов и библиотек (фактически, автоматические юнит-тесты).
- Автоматизация тестирования графического пользовательского интерфейса (Graphical user interface testing) – специальная программа (фреймворк автоматизации тестирования) позволяет генерировать пользовательские события– нажатия клавиш, клики мышкой, и отслеживание реакции программы на эти действия: соответствует ли она спецификации.
- Автоматизация тестирования API (Application Programming Interface) – тестирование программного интерфейса программы. Тестируются интерфейсы, предназначенные для взаимодействия, например, с другими программами или с пользователем. Здесь, опять же, как правило, используются специальные фреймворки.

Когда, что и как автоматизировать и автоматизировать ли вообще – очень важные вопросы, ответы на которые должна дать команда разработки. Выбор правильных элементов программы для автоматизации в большой степени будет определять успех автоматизации тестирования в принципе. Нужно избегать автоматизации тестирования участков кода, которые могут часто меняться.

СРАВНЕНИЕ РУЧНОГО И АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ

- Как ручное, так и автоматизированное тестирования могут использоваться на разных уровнях тестирования, а также быть частью других типов и видов тестирования.
- Автоматизация сохраняет время, силы и деньги. Автоматизированный тест можно запускать снова и снова, прилагая минимум усилий.
- Вручную можно протестировать практически любое приложение, в то время как автоматизировать стоит только стабильные системы. Автоматизированное тестирование используется, главным образом, для регрессии. Кроме того, некоторые виды тестирования, например, исследовательское тестирование может быть выполнено только вручную.
- Мануальное тестирование может быть повторяющимся и скучным. В то же время, автоматизация может помочь этого избежать – за вас все сделает компьютер.

Таким образом, на реальных проектах зачастую используется комбинация ручного и автоматизированного тестирования, причем уровень автоматизации будет зависеть как от типа проекта, так и от особенностей постановки производственных процессов в компании.

ВИДЫ ТЕСТИРОВАНИЯ

Все виды тестирования программного обеспечения, в зависимости от преследуемых целей, можно условно разделить на следующие группы:

- Функциональные.
- Нефункциональные.
- Связанные с изменениями.

ФУНКЦИОНАЛЬНЫЕ ВИДЫ ТЕСТИРОВАНИЯ

- Функциональные тесты базируются на функциях и особенностях, а также на взаимодействии с другими системами и могут быть представлены на всех уровнях тестирования: компонентном или модульном (Component/Unit testing), интеграционном (Integration testing), системном (System testing), приемочном (Acceptance testing). Функциональные виды тестирования рассматривают внешнее поведение системы. Далее перечислены одни из самых распространенных видов функциональных тестов.
- Функциональное тестирование рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.

1. ФУНКЦИОНАЛЬНЫЕ ТЕСТЫ

Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех уровнях тестирования (компонентном, интеграционном, системном, приемочном). Как правило, эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (use cases).

Тестирование функциональности может проводиться в двух аспектах:

- Требования.
- Бизнес-процессы.

Тестирование в аспекте «требования» использует спецификацию функциональных требований к системе, как основу для дизайна тестовых случаев (Test Cases). В этом случае необходимо сделать список того, что будет тестироваться, а что нет, приоритезировать требования на основе рисков (если это не сделано в документе с требованиями), а на основе этого приоритезировать тестовые сценарии (test cases). Это позволит сфокусироваться и не упустить при тестировании наиболее важный функционал.

Тестирование в аспекте «бизнес-процессы» использует знание бизнес-процессов, которые описывают сценарии ежедневного использования системы. В этом аспекте тестовые сценарии (test scripts), как правило, основываются на случаях использования системы (use cases).

Преимущества функционального тестирования:

- имитирует фактическое использование системы.

Недостатки функционального тестирования:

- возможность упущения логических ошибок в программном обеспечении;
- вероятность избыточного тестирования.

Достаточно распространенной является автоматизация функционального тестирования.

2. ТЕСТИРОВАНИЕ БЕЗОПАСНОСТИ (SECURITY AND ACCESS CONTROL TESTING)

Тестирование безопасности - это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

Общая стратегия безопасности основывается на трех основных принципах:

- Конфиденциальность.
- Целостность.
- Доступность.

Конфиденциальность

- Конфиденциальность - это сокрытие определенных ресурсов или информации. Под конфиденциальностью можно понимать ограничение доступа к ресурсу некоторой категории пользователей или, другими словами, при каких условиях пользователь авторизован получить доступ к данному ресурсу.

Целостность

- Существует два основных критерия при определении понятия целостности:
- Доверие. Ожидается, что ресурс будет изменен только соответствующим способом определенной группой пользователей.
- Повреждение и восстановление. В случае, когда данные повреждаются или неправильно меняются авторизованным или не авторизованным пользователем, Вы должны определить, на сколько важной является процедура восстановления данных.

Доступность

- Доступность представляет собой требования о том, что ресурсы должны быть доступны авторизованному пользователю, внутреннему объекту или устройству. Как правило, чем более критичен ресурс, тем выше уровень доступности должен быть.

3. ТЕСТИРОВАНИЕ ВЗАИМОДЕЙСТВИЯ ИЛИ INTEROPERABILITY TESTING

- Тестирование взаимодействия (Interoperability Testing) – это функциональное тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в себя тестирование совместимости (compatibility testing) и интеграционное тестирование (integration testing).
- Программное обеспечение с хорошими характеристиками взаимодействия может быть легко интегрировано с другими системами, не требуя каких-либо серьезных модификаций. В этом случае, количество изменений и время, требуемое на их выполнение, могут быть использованы для измерения возможности взаимодействия.

НЕФУНКЦИОНАЛЬНЫЕ ВИДЫ ТЕСТИРОВАНИЯ

- Нефункциональное тестирование описывает тесты, необходимые для определения характеристик программного обеспечения, которые могут быть измерены различными величинами. В целом, это тестирование того, как система работает.

1. ВСЕ ВИДЫ ТЕСТИРОВАНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

Тестирование производительности (Performance testing).

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 (Stress Testing)

- Стрессовое тестирование позволяет проверить, насколько приложение и система в целом работоспособны в условиях стресса, а 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.

Объемное тестирование (Volume Testing)

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.

Тестирование стабильности или надежности (Stability / Reliability Testing)


- 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.
- В англоязычной терминологии вы можете так же найти еще один вид тестирования - Load Testing - тестирование реакции системы на изменение нагрузки (в пределах допустимого). Нам показалось, что Load и Performance преследуют все же одну и ту же цель: проверка производительности (времени отклика) на разных нагрузках. Собственно поэтому мы и не стали разделять их. В то же время кто то может разделить. Главное все таки понимать цели того или иного вида тестирования и постараться их достигнуть.

2. ТЕСТИРОВАНИЕ УСТАНОВКИ (INSTALLATION TESTING)

- Тестирование установки направленно на проверку успешной инсталляции и настройки, а также на обновление или удаление программного обеспечения.
- В настоящий момент, наиболее распространена установка ПО при помощи инсталляторов (специальных программ, которые сами по себе так же требуют надлежащего тестирования, описание которого рассмотрено в разделе "Особенности тестирования инсталляторов").
- В реальных условиях инсталляторов может не быть. В этом случае придется самостоятельно выполнять установку программного обеспечения, используя документацию в виде инструкций или "read me" файлов, шаг за шагом описывающих все необходимые действия и проверки.
- В распределенных системах, где приложение разворачивается на уже работающем окружении, простого набора инструкций может быть мало. Для этого часто пишется план установки (Deployment Plan), включающий не только шаги по инсталляции приложения, но и шаги отката (roll-back) к предыдущей версии (в случае неудачи). Сам по себе план установки также должен пройти процедуру тестирования для избежания проблем при выдаче в реальную эксплуатацию. Особенно это актуально, если установка выполняется на системы, где каждая минута простоя - это потеря репутации и большого количества средств, например: банки, финансовые компании или даже баннерные сети. Поэтому тестирование установки можно назвать одной из важнейших задач по обеспечению качества программного обеспечения.

3. ТЕСТИРОВАНИЕ УДОБСТВА ПОЛЬЗОВАНИЯ (USABILITY TESTING)

- Иногда мы сталкиваемся с непонятными или нелогичными приложениями, многие функции и способы использования которых часто не очевидны. После такой работы редко возникает желание использовать приложение снова, и мы ищем более удобные аналоги. Для того, чтобы приложение было популярным, ему мало быть функциональным – оно должно быть еще и удобным. Если задуматься, интуитивно понятные приложения экономят нервы пользователям и затраты работодателя на обучение. Значит, они более конкурентоспособны! Поэтому тестирование удобства использования, о котором пойдет речь далее, является неотъемлемой частью тестирования любых массовых продуктов.




Тестирование удобства пользования - это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

Тестирование удобства пользования дает оценку уровня удобства использования приложения по следующим пунктам:

- Производительность, эффективность (efficiency) - сколько времени и шагов понадобится пользователю для завершения основных задач приложения, например, размещение новости, регистрации, покупка и т.д. (меньше - лучше).
- Правильность (accuracy) - сколько ошибок сделал пользователь во время работы с приложением (меньше - лучше).
- Активизация в памяти (recall) – как много пользователь помнит о работе приложения после приостановки работы с ним на длительный период времени (повторное выполнение операций после перерыва должно проходить быстрее, чем у нового пользователя).
- Эмоциональная реакция (emotional response) – как пользователь себя чувствует после завершения задачи - растерян, испытал стресс? Порекомендует ли пользователь систему своим друзьям? (положительная реакция - лучше).

4. ТЕСТИРОВАНИЕ НА ОТКАЗ И ВОССТАНОВЛЕНИЕ (FAILOVER AND RECOVERY TESTING)


- Тестирование на отказ и восстановление (Failover and Recovery Testing) проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети).
- Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.
- Тестирование на отказ и восстановление очень важно для систем, работающих по принципу “24x7”. Если Вы создаете продукт, который будет работать, например, в интернете, то без проведения данного вида тестирования Вам просто не обойтись, т.к. каждая минута простоя или потеря данных, в случае отказа оборудования, может стоить вам денег, потери клиентов и репутации на рынке.



Методика подобного тестирования заключается в симулировании различных условий сбоя и последующем изучении и оценке реакции защитных систем. В процессе подобных проверок выясняется, была ли достигнута требуемая степень восстановления системы после возникновения сбоя.

Для наглядности, рассмотрим некоторые варианты подобного тестирования и общие методы их проведения. Объектом тестирования, в большинстве случаев, являются весьма вероятные эксплуатационные проблемы, такие как:

- Отказ электричества на компьютере-сервере.
- Отказ электричества на компьютере-клиенте.
- Незавершенные циклы обработки данных (прерывание работы фильтров данных, прерывание синхронизации).
- Объявление или внесение в массивы данных невозможных или ошибочных элементов.
- Отказ носителей данных.



Данные ситуации могут быть воспроизведены, как только достигнута некоторая точка в разработке, когда все системы восстановления или дублирования готовы выполнять свои функции. Технически реализовать тесты можно следующими путями:

- Симулировать внезапный отказ электричества на компьютере (обесточить компьютер).
- Симулировать потерю связи с сетью (выключить сетевой кабель, обесточить сетевое устройство).
- Симулировать отказ носителей (обесточить внешний носитель данных).
- Симулировать ситуацию наличия в системе неверных данных (специальный тестовый набор или база данных).

При достижении соответствующих условий сбоя и по результатам работы систем восстановления, можно оценить продукт с точки зрения тестирования на отказ. Во всех вышеперечисленных случаях, по завершении процедур восстановления, должно быть достигнуто определенное требуемое состояние данных продукта:

- Потеря или порча данных в допустимых пределах.
- Отчет или система отчетов, с указанием процессов или транзакций, которые не были завершены в результате сбоя.

Стоит заметить, что тестирование на отказ и восстановление – это весьма специфичное тестирование. Разработка тестовых сценариев должна производиться с учетом всех особенностей тестируемой системы. Принимая во внимание довольно жесткие методы воздействия, стоит также оценить целесообразность проведения данного вида тестирования для конкретного программного продукта.

5. КОНФИГУРАЦИОННОЕ ТЕСТИРОВАНИЕ (CONFIGURATION TESTING)

Конфигурационное тестирование(Configuration Testing) — специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.)

В зависимости от типа проекта конфигурационное тестирование может иметь разные цели:

- Проект по профилированию работы системы.
Цель Тестирования: определить оптимальную конфигурацию оборудования, обеспечивающую требуемые характеристики производительности и времени реакции тестируемой системы.
- Проект по миграции системы с одной платформы на другую.
Цель Тестирования: Проверить объект тестирования на совместимость с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм.

Примечание: В ISTQB Syllabus (классификация тестировщиков) вообще не говорится о таком виде тестирования, как конфигурационное. Согласно глоссарию, данный вид тестирования рассматривается там как тестирование портируемости (portability testing: The process of testing to determine the portability of a software product.).

СВЯЗАННЫЕ С ИЗМЕНЕНИЯМИ ВИДЫ ТЕСТИРОВАНИЯ

- После проведения необходимых изменений, таких как исправление багов/дефектов, программное обеспечение должно быть перетестировано для подтверждения того факта, что проблема была действительно решена.
- Ниже перечислены виды тестирования, которые необходимо проводить после установки программного обеспечения, для подтверждения работоспособности приложения или правильности осуществленного исправления дефекта:
 - Дымовое тестирование (Smoke Testing)
 - Регрессионное тестирование (Regression Testing)
 - Тестирование сборки (Build Verification Test)
 - Санитарное тестирование или проверка согласованности/исправности (Sanity Testing)

1. ДЫМОВОЕ ТЕСТИРОВАНИЕ (SMOKE TESTING)

- Понятие дымовое тестирование пошло из инженерной среды:
- "При вводе в эксплуатацию нового оборудования ("железа") считалось, что тестирование прошло удачно, если из установки не пошел дым."
- В области же программного обеспечения дымовое тестирование рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что, после сборки кода (нового или исправленного), устанавливаемое приложение стартует и выполняет основные функции.
- Вывод о работоспособности основных функций делается на основании результатов поверхностного тестирования наиболее важных модулей приложения на предмет возможности выполнения требуемых задач и наличия быстронаходимых критических и блокирующих дефектов. В случае отсутствия таковых дефектов дымовое тестирование объявляется пройденным и приложение передается для проведения полного цикла тестирования, в противном случае, дымовое тестирование объявляется проваленным и приложение уходит на доработку.
- Аналогами дымового тестирования являются Build Verification Testing и Acceptance Testing, выполняемые на функциональном уровне командой тестирования, по результатам которых делается вывод о том, принимается или нет установленная версия программного обеспечения в тестирование, эксплуатацию или на поставку заказчику.
- Для облегчения работы, экономии времени и людских ресурсов рекомендуется внедрить автоматизацию тестовых сценариев для дымового тестирования.

2. РЕГРЕССИОННОЕ ТЕСТИРОВАНИЕ (REGRESSION TESTING)

- Регрессионное тестирование - это вид тестирования, направленный на проверку изменений, сделанных в приложении или окружающей среде (починка дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб-сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде. Регрессионными могут быть как функциональные, так и нефункциональные тесты.
- Как правило, для регрессионного тестирования используются тест-кейсы, написанные на ранних стадиях разработки и тестирования. Это дает гарантию того, что изменения в новой версии приложения не повредили уже существующую функциональность. Рекомендуется делать автоматизацию регрессионных тестов для ускорения последующего процесса тестирования и обнаружения дефектов на ранних стадиях разработки программного обеспечения.
- Сам по себе термин "регрессионное тестирование", в зависимости от контекста использования, может иметь разный смысл. Сэм Канер, к примеру, описал 3 основных типа регрессионного тестирования:
 - Регрессия багов (Bug regression) - попытка доказать, что исправленная ошибка на самом деле не исправлена.
 - Регрессия старых багов (Old bugs regression) - попытка доказать, что недавнее изменение кода или данных сломало исправление старых ошибок, т.е. старые баги стали снова воспроизводиться.
 - Регрессия побочного эффекта (Side effect regression) - попытка доказать, что недавнее изменение кода или данных сломало другие части разрабатываемого приложения.

3. ТЕСТИРОВАНИЕ СБОРКИ (BUILD VERIFICATION TEST)

- Тестирование, направленное на определение соответствия выпущенной версии критериям качества для начала тестирования. По своим целям является аналогом дымового тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию. Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии.

4. САНИТАРНОЕ ТЕСТИРОВАНИЕ ИЛИ ПРОВЕРКА СОГЛАСОВАННОСТИ/ИСПРАВНОСТИ (SANITY TESTING)

- Санитарное тестирование - это узконаправленное тестирование, достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений произведенных в ней или окружающей среде. Обычно выполняется вручную.

ОТЛИЧИЕ САНИТАРНОГО ТЕСТИРОВАНИЯ ОТ ДЫМОВОГО (SANITY VS SMOKE TESTING)

- В некоторых источниках ошибочно полагают, что санитарное и дымовое тестирование - это одно и то же. Мы же полагаем, что эти виды тестирования имеют "векторы движения"- направления в разные стороны. В отличии от дымового (Smoke testing), санитарное тестирование (Sanity testing) направлено вглубь проверяемой функции, в то время как дымовое - направлено вширь, для покрытия тестами как можно большего функционала в кратчайшие сроки.

УРОВНИ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

- Тестирование на разных уровнях производится на протяжении всего жизненного цикла разработки и сопровождения программного обеспечения. Уровень тестирования определяет то, над чем производятся тесты: над отдельным модулем, группой модулей или системой, в целом. Проведение тестирования на всех уровнях системы - это залог успешной реализации и сдачи проекта.

1. КОМПОНЕНТНОЕ ИЛИ МОДУЛЬНОЕ ТЕСТИРОВАНИЕ (COMPONENT OR UNIT TESTING)

- Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.). Обычно компонентное (модульное) тестирование проводится вызывая код, который необходимо проверить и при поддержке сред разработки, таких как фреймворки (frameworks - каркасы) для модульного тестирования или инструменты для отладки. Все найденные дефекты, как правило исправляются в коде без формального их описания в системе менеджмента багов (Bug Tracking System).
- Один из наиболее эффективных подходов к компонентному (модульному) тестированию - это подготовка автоматизированных тестов до начала основного кодирования (разработки) программного обеспечения. Это называется разработка от тестирования (test-driven development) или подход тестирования вначале (test first approach). При этом подходе создаются и интегрируются небольшие куски кода, напротив которых запускаются тесты, написанные до начала кодирования. Разработка ведется до тех пор, пока все тесты не будут успешно пройдены.

Есть ли какое-то отличие, между компонентным и модульным тестированием?

По-существу, эти уровни тестирования представляют одно и тоже и разница лишь в том, что в компонентном тестировании, в качестве параметров функций, используют реальные объекты и драйверы, а в модульном тестировании - конкретные значения.

2. ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ (INTEGRATION TESTING)

- Интеграционное тестирование предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами).
- Уровни интеграционного тестирования:
 - Компонентный интеграционный уровень (Component Integration testing) проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.
 - Системный интеграционный уровень (System Integration Testing) - проверяется взаимодействие между разными системами после проведения системного тестирования.

Подходы к интеграционному тестированию:

- **Снизу вверх (Bottom Up Integration):**
Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения.
- **Сверху вниз (Top Down Integration):**
Сначала тестируются все высокоуровневые модули, затем постепенно, один за другим, добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем, по мере готовности, они заменяются реальными активными компонентами. Таким образом, мы проводим тестирование сверху вниз.
- **Большой взрыв ("Big Bang" Integration):**
Все (или практически все) разработанные модули собираются вместе в виде законченной системы или ее основной части, а затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако, если тест-кейсы и их результаты записаны неверно, то сам процесс интеграции будет осложнен, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования.

3. СИСТЕМНОЕ ТЕСТИРОВАНИЕ (SYSTEM TESTING)

Основной задачей системного тестирования является проверка как функциональных, так и нефункциональных требований, дефекты в системе в целом. При этом выявляется неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения в системы в той или иной среде, во время тестирования рекомендуется использовать окружение, максимально приближенное к тому, на которое будет установлен продукт после выдачи.

Можно выделить два подхода к системному тестированию:

- на базе требований (requirements based) - для каждого требования пишутся тестовые случаи (test cases), проверяющие выполнение данного требования.
- на базе случаев использования (use case based) - на основе представления о способах использования продукта создаются случаи использования системы (Use Cases). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся тест-кейсы (test cases), которые должны быть протестированы.

4. ПРИЕМОЧНОЕ ТЕСТИРОВАНИЕ ИЛИ ПРИЕМО-СДАТОЧНОЕ ИСПЫТАНИЕ (ACCEPTANCE TESTING)

Приемочное тестирование или приемо-сдаточное испытание (Acceptance Testing) - формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:

- определения удовлетворения системой приемочным критериям;
- вынесения решения заказчиком или другим уполномоченным лицом принятия приложения.

Приемочное тестирование выполняется на основании набора типичных тестовых случаев и сценариев, разработанных на основании требований к данному приложению.

Решение о проведении приемочного тестирования принимается, когда:

- Продукт достиг необходимого уровня качества.
- Заказчик ознакомлен с Планом Приемочных Работ (Product Acceptance Plan) или иным документом, где описан набор действий, связанных с проведением приемочного тестирования, дата проведения, ответственные и т.д.

Фаза приемочного тестирования длится до тех пор, пока заказчик не выносит решение об отправлении приложения на доработку или выдаче приложения.