АВТОНОМНАЯ НЕКОММЕРЧЕСКАЯ ОРГАНИЗАЦИЯ
ПРОФЕССИОНАЛЬНАЯ ОБРАЗОВАТЕЛЬНАЯ ОРГАНИЗАЦИЯ
«Международный Колледж Бизнеса и Дизайна»
(АНО ПОО «Международный Колледж Бизнеса и Дизайна»)

УТВЕРЖДАЮ
Директор АНО ПОО «МКБиД»
_____ Н.Н.Репин
«18» августа 2023 г.

КОНСПЕКТ ЛЕКЦИЙ

по учебной дисциплине МДК 05.03 Тестирование информационных систем

программы подготовки специалистов среднего звена
по специальности: 09.02.07 «Информационные системы и программирование»

2023 год

Лекция

Тема: Тестирование ПО: суть профессии, ЗП. Общие сведения и понятия.

Разработка программного обеспечения — сфера, которая будет в ближайшее время только расти, несмотря ни на эпидемию, ни на экономический кризис. Соответственно, будет увеличиваться дефицит технических специальностей, связанных с разработкой.

Одна из них — тестирование ПО. Забегая наперед, скажем, что в тестировщиках нуждаются практически все компании, которые занимаются созданием программного обеспечения и сервисов. Что касается порога входа, требований, которые предъявляются к разработке ПО и размере заработной платы тестировщиков рассмотрим на этой паре.

Тестирование (testing) программного обеспечения – это процесс исследования ПО с целью выявления ошибок и определения соответствия между реальным и ожидаемым поведением ПО, осуществляемый на основе набора тестов, выбранных определённым образом.

Тестирование - очень важный и трудоемкий этап процесса разработки программного обеспечения, так как он позволяет выявить подавляющее большинство ошибок, допущенных при составлении программ.

В более широком смысле тестирование ПО – это техника контроля качества программного продукта, включающая в себя проектирование тестов, выполнение тестирования и анализ полученных результатов.

Тестирование – важный этап процесса разработки ПО, обеспечивается безопасность, надёжность и удобство создаваемого продукта.

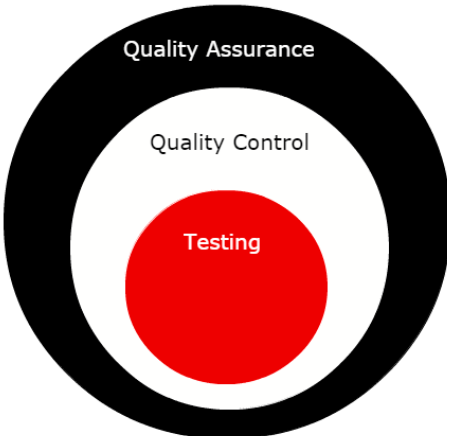
В настоящее время существует множество подходов и методик к решению задачи тестирования ПО, но эффективное тестирование сложных программных систем — процесс творческий, не сводящийся к следованию строгим и чётким правилам.

основные этапы тестирования ПО с точки зрения STLC (software testing life cycle):

1. Анализ требований.
2. Планирование тестирования.
3. Создание тест-кейсов.
4. Настройка тестового окружения.
5. Выполнение тестирования
6. Завершение цикла тестирования

QA, QC и тестирование_чем они отличаются?

Тестирование программного обеспечения — обширное понятие, которое включает планирование, проектирование и, собственно, выполнение тестов.



Из кого состоит сфера тестирования ПО:

1.QA (Quality Assurance кволити ешуренс**)** — обеспечение качества продукта. ***QA-специалист*** контролирует и обеспечивает качество работы продукта компании. Он *отвечает и за отдельные этапы разработки софта*. В частности, за выбор инструментов для разработки, предотвращение возможных проблем. Еще он участвует в процессе совершенствования продукта.

QA охватывает все этапы разработки, включая описание проекта, собственно, тестирование, релиз и, зачастую, пост-релизный этап.

Данные специалисты зачастую являются и разработчиками ПО

2.QC (Quality Control) — контроль качества продукта. Задача ***QC-специалиста*** — проверка конкретного продукта, что включает *анализ кода продукта, дизайна, плюс тестирование*. QC-инженер *разрабатывает стратегию тестирование вполне определенного тестирования*, взаимодействует с разработчиками и организует само тестирование.

3.Специалист по тестированию занимается выполнением тестов. Тестированием называют проверку соответствия результатов работы программного продукта на соответствие заданным критериям. Тестировщики занимаются тестированием всего продукта в целом или же отдельных компонентов. Тестирование играет важнейшую роль в обеспечении качества продукта.

Кстати, есть внешнее ответвление — современное направление тестирования Developer in test. Специалисты этого направления — вроде как и разработчики, но занимаются они обеспечением качества разрабатываемого продукта.

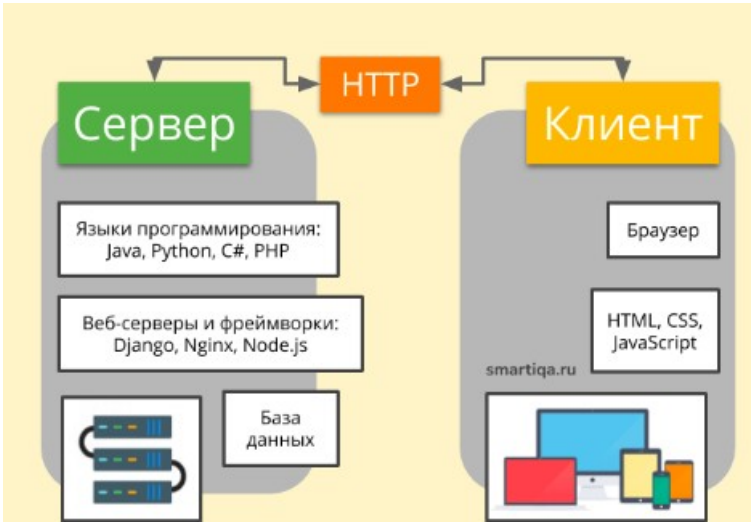
Что должен знать и уметь хороший тестировщик.

Исходя из всего, что сказано выше, сложно выделить конкретные знания или умения. Все сильно зависит от проекта, на котором работает специалист, соответственно, и от стека технологий, которые на этом проекте используются.

общие навыки QA:

1. знания в клиент-серверной архитектуре.

2. понимать механизм взаимодействия веб-приложений



3. иметь базовые навыки использования специализированного софта,

Крайне важны soft-скиллы:

1. Умение общаться с коллегами.
2. Умение ясно излагать мысли.
3. Способность четко описать проблему разработчику.
4. Умение работать с документацией.
5. Понимание стандартов разработки ПО.
6. Внимательность.
7. Готовность доказать и отстоять свою позицию, основываясь на документации или здравом смысле.

Существует мнение, что профессионалом в сфере тестирования можно стать через 3 года, при условии наличия технического бэкграунда.

В первый год молодой специалист начинает понимать, что от него требуют,

во второй год — понимает, как нужно выполнять то, что от него требуют,

на третий — пытается улучшить выстроенный процесс, добавляя свое видение.

Что касается тестировщиков с большим опытом и обширными знаниями, то им крайне необходимо постоянно расширять навыки, следить за тенденциями в мире IT, искать новые подходы к решению вчерашних задач и всегда быть «на волне».

В разных компаниях требования к тестировщиком отличаются. Кому-то нужны Developer in test, а для кого-то важнейшую роль играют софт-скиллы специалистов.

Мифы о тестировании ПО и тестировщиках

1.тестировщики занимаются тем, что просто нажимают на кнопки и вводят рандомную информацию в разные поля программы.

На самом деле это не так, если бы тестировщики хаотично нажимали на кнопки и вводили случайные данные, то результаты тестирования никакой ценности для разработчика не принесли бы.

У тестировщиков всегда есть стратегия работы, план, который позволяет получить объективное описание актуального состояния продукта.

2. заключается в утверждении, что тестировщики ответственны за качество ПО.

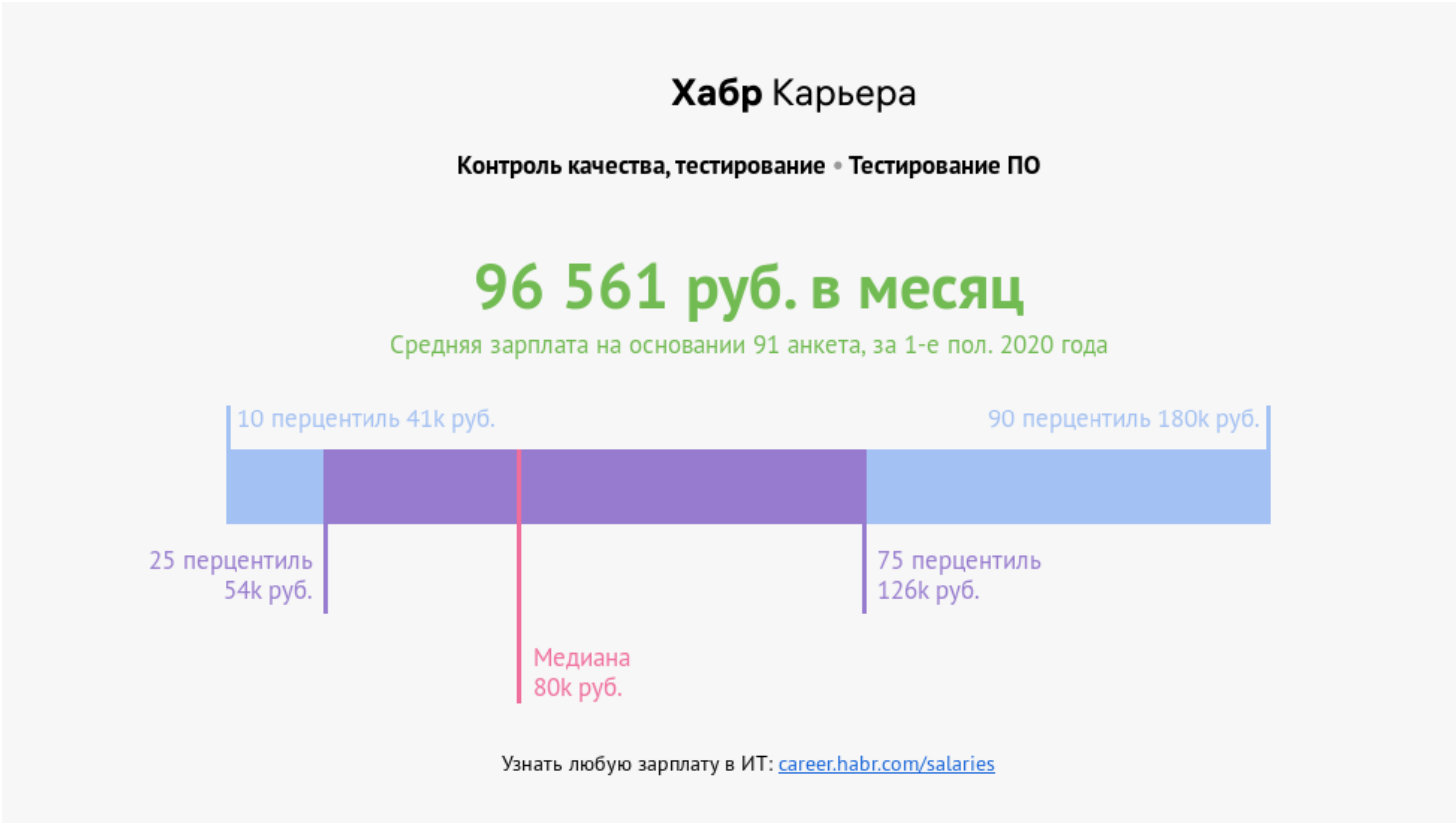
На самом деле, ответственность за качество разработки продукта несет вся команда. Тестировщики же помогают улучшать качество разработки, а также выявляют проблемы на ранних стадиях.

4. тестировщиков очень много.

На самом деле хороших специалистов на рынке мало. Много тех, кто выкладывает резюме с пометкой «тестировщик», не понимая сути тестирования ПО.

Востребованность профессии и доходы тестировщиков ПО

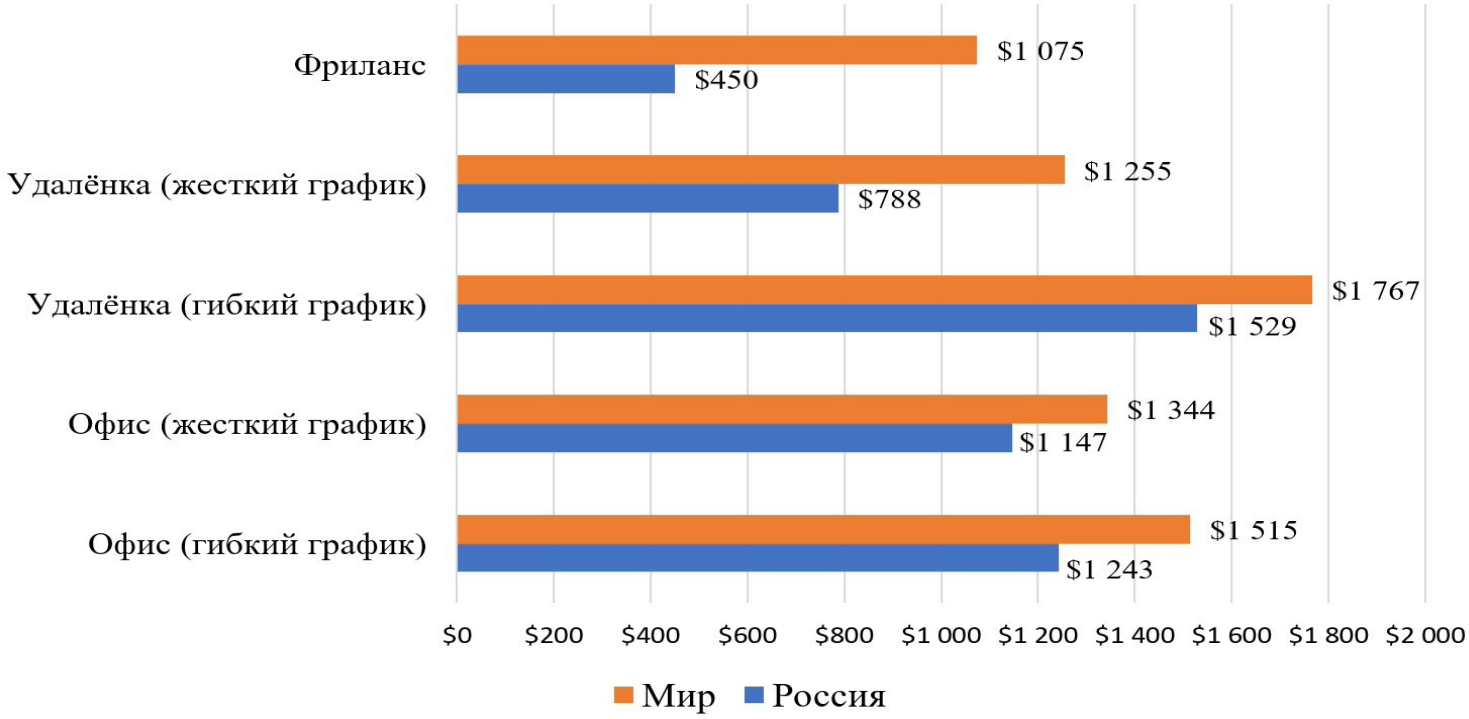
По данным зарплатного калькулятора Хабр Карьеры, средний размер заработной платы тестировщика составляет чуть больше 80-96 тысяч рублей в месяц. Конечно, это среднее значение. Есть те, кто зарабатывает значительно меньше, скажем, тысяч 30, а есть и те, кто получает в 10 раз больше — около 300 тысяч рублей.



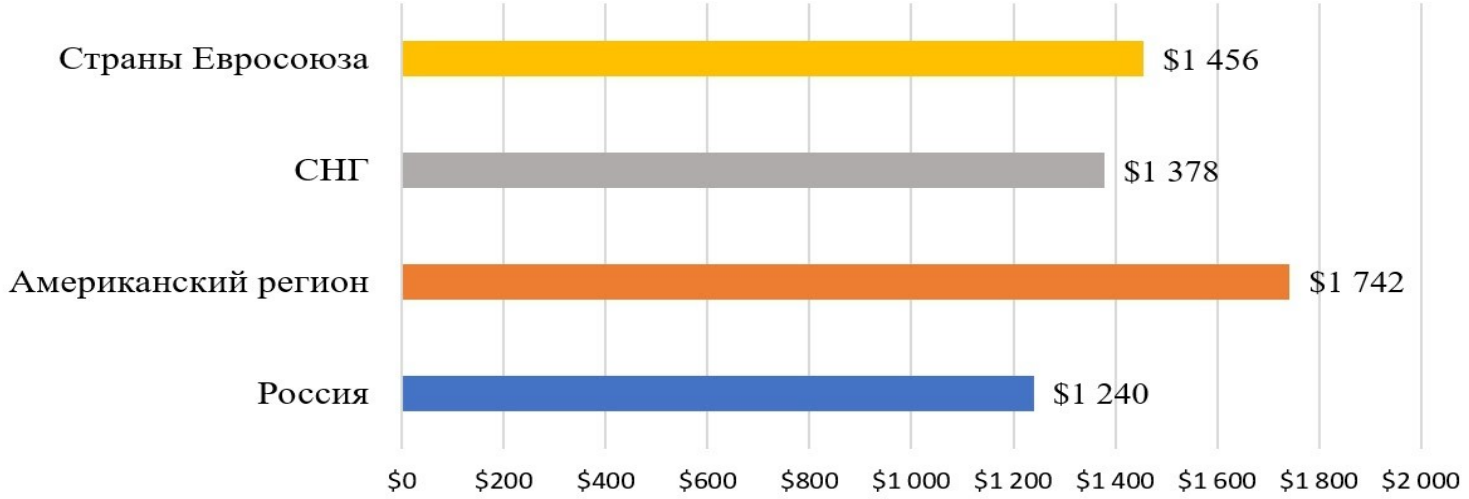
Профессионалы примерно одного уровня, выполняющие один и тот же пул задач в столице и в регионе могут получать сильно отличающуюся зарплату.

В Москве это 100+ тысяч рублей, в регионах — 40-50 тысяч рублей, а в некоторых случаях и вовсе 20-30 тысяч.

Если сравнивать уровень зарплаты тестировщиков в РФ и за рубежом, то разница в среднем 30-50%.



Плюс можно сравнить еще разброс уровня заработной платы в зависимости от региона — Евросоюз, СНГ, США и Канада, РФ.



От каких навыков зависит уровень ЗП:

- Тестирование ПО;
- Ручное тестирование;
- Функциональное тестирование;
- Автоматизация тестирования;
- Python;
- Selenium webdriver;

- Разработка тест-кейсов;
- Тестирование мобильных приложений;
- Контроль качества;
- Black box testing.

Лекция 2

Тема: Принципы тестирования ПО. Цели тестирования ПО. Подходы к формированию тестовых наборов.

Тестирование ПО принципы:

Это основополагающие идеи, которые помогают тестировщикам достигать наилучших результатов при тестировании ПО.

1. Корректность тестирования. Тестирование должно проверять то, что должно быть проверено, и только то.

Это означает, что каждый тест должен быть разработан для тестирования определенной функциональности, и не должен проверять ничего, что не относится к данной функциональности.

2. Извлечение дефектов. Цель тестирования состоит не только в том, чтобы показать, что программа работает правильно, но и в том, чтобы найти все дефекты.

Тестирование должно проводиться таким образом, чтобы максимально извлекать дефекты из программы.

3. Раннее тестирование. Должно начинаться как можно раньше в жизненном цикле разработки ПО.

Чем раньше найдены дефекты, тем меньше затрат на их исправление.

4. Комплексное тестирование. Включает в себя все возможные случаи использования программы, чтобы гарантировать ее правильную работу в любых условиях.

5. Тестирование без пристрастия. Тестирование должно проводиться без пристрастия к разработчикам или продукту.

Тестировщик должен относиться к программе как к независимому наблюдателю.

6. Повторяемость тестирования.

Это означает, что тестирование должно быть проведено таким образом, чтобы результаты могли быть воспроизведены в любое время.

7. Постоянное улучшение. Тестирование должно быть непрерывным процессом улучшения.

Тестировщики должны постоянно анализировать свой подход к тестированию и искать способы улучшения его эффективности и результативности.

Соблюдение этих принципов поможет тестировщикам достигать более качественных и эффективных результатов в своей работе.

Цели тестирования ПО:

Вместо итого, поговорим о целях тестирования ПО, они могут варьироваться в зависимости от фазы жизненного цикла проекта, конечных целей бизнеса и требований заказчика. Но в целом можно выделить следующие общие цели:

1. ***Выявление дефектов и ошибок в ПО***. Один из основных и очевидных целей тестирования — обнаружение дефектов в ПО. Чем раньше дефекты будут обнаружены, тем дешевле и проще будет их исправить.

2. ***Улучшение качества ПО.*** Цель тестирования не только в выявлении дефектов, но и в обеспечении высокого качества продукта. Тестирование помогает убедиться, что ПО работает корректно, соответствует требованиям и ожиданиям пользователей, а также соответствует стандартам и правилам.

3. ***Оценка рисков***. Тестирование помогает оценить риски, связанные с продуктом и его функциональностью. Это может помочь в принятии решений о дальнейшей разработке и релизе ПО.

4. ***Проверка соответствия требованиям***. Цель тестирования — убедиться, что ПО соответствует требованиям и спецификациям заказчика. Тестирование также помогает убедиться, что ПО соответствует правилам и стандартам в отрасли.

5. ***Улучшение процесса разработки***. Тестирование может помочь улучшить процесс разработки ПО. Результаты тестирования могут помочь выявить слабые места в процессе, которые можно улучшить для повышения эффективности и качества разработки.

Целью проектирования тестовых вариантов является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

Тестирование обеспечивает:

1. Обнаружение ошибок.
2. Демонстрацию соответствия функций программы ее назначению.
3. Демонстрацию реализации требований к характеристикам программы.

4. Отображение надежности как индикатора качества программы.

Тестирование не может показать отсутствие дефектов (оно может показывать только присутствие дефектов). Важно помнить это утверждение при проведении тестирования.

Процесс разработки программного обеспечения предполагает три стадии тестирования:

- автономное тестирование компонентов программного обеспечения;
- комплексное тестирование разрабатываемого программного обеспечения;
- системное или оценочное тестирование на соответствие основным критериям качества.

Для повышения качества тестирования рекомендуется соблюдать следующие основные принципы:

1. предполагаемые результаты должны быть известны до тестирования;
2. следует избегать тестирования программы автором;
3. досконально изучать результаты каждого теста;
4. необходимо проверять работу программы на неверных данных;
5. вероятность наличия необнаруженных ошибок в части программы пропорциональна числу ошибок, уже обнаруженных в этой части.

Формирование набора тестов имеет большое значение, поскольку тестирование является одним из наиболее трудоемких этапов (от 30 до 60 % общей трудоемкости) создания программного продукта.

Существуют два принципиально разных подхода к формированию тестовых наборов – структурный и функциональный.

Подходы к формированию тестовых наборов:

1.*Структурный подход* базируется на том, что известны алгоритмы работы программы. В основе структурного тестирования лежит концепция максимально полного тестирования всех маршрутов программы.

Здесь тестируется весе ПО и его элементы, знаем логику, как работает, какие данные должны быть на входе и какие на выходе.

Проверяется на соответствие не функциональным требованиям:

1.1 **Удобство** (В основном производиться оценка удобства для пользователей)

1.2 **Маштабируемость** (проверяется как вертикальная так и горизонтальная маштабируемость тестируемого приложения)

1.3 **Производительность** (Способность работы приложения при различных нагрузках)

1.4 **Безопасность** (Защита пользовательских данных, защита данных приложения, стойкость на взлом)

1.5 **Портируемость** (Совместимость и переносимость приложения для и под различные окружения, платформы и т.д.)

1.6 **Надежность** (Поведение системы при различных непредвиденных ситуациях, способность обработки нестандартных действий пользователя)

2.*Функциональный подход* основывается на том, что алгоритм работы программного обеспечения не известен. Тесты строят, опираясь на функциональные спецификации. Программа рассматривается как «черный ящик», и целью тестирования является выяснение обстоятельств, в которых поведение программы не соответствует требованиям

Тестируем как обычный пользователь, не зная логики и последовательностей ищем ошибки в последовательностях, вводим различные тест кейсы для выявления дефектов **главная цель** которого это проверка реализуемости функциональных требований приложения, т.е. способность приложения в заданных критериях решать возложенные на него (на приложение) задачи. ***Требования включают в себя:***

1. защищенность
2. соответствие стандартам
3. способность к взаимодействию
4. функциональная пригодность
5. точность

Опытные тестировщики обнаруживают ошибки путём сравнения шаблонов тестовых выходных данных с выходными данными тестируемых систем.

Чтобы определить местоположение ошибки, необходимы знания о типах ошибок, шаблонах выходных данных, языке и процессе программирования.

Лекция 3

Тема: коммуникация и взаимодействие в процессе тестирования

В зависимости от потребностей проекта команда может включать участников с различными ролями:

При этом на практике часто бывает, что один человек одновременно выполняет несколько ролей, если имеет необходимые навыки и знания для исполнения обязанностей.

1 **Аккаунт-менеджер** - менеджер по работе с клиентами, специалист, который работает с клиентами компании и обеспечивает их лояльность. Аккаунт-менеджер обеспечивает выполнение всех необходимых клиенту задач, находит к каждому заказчику индивидуальный подход, поддерживает с ним хорошие отношения (даже после того, как все заказы уже выполнены), предлагает ему новые услуги и продукты.

Иными словами, профессия аккаунт-менеджера обязывает представителя сделать все возможное, чтобы клиент был счастлив, работал с компанией всегда и всем ее рекомендовал.

Человек который общается с клиентом и узнает его желания.

2 **Менеджер проекта** (Project manager, руководитель проекта, проект-менеджер; сокращенно - PM, ПМ, РП) - лицо, ответственное за управление проектом. Менеджер проекта несет ответственность за достижение целей проекта в рамках бюджета, в срок и с заданным уровнем качества.

3 **Системный аналитик** (аналитик) является “мостиком” между заказчиком и членами команды. Переводит пожелания заказчика в формат точно описанных технических заданий.

4 **Системный архитектор** (архитектор) проектирует разрабатываемую систему на самом верхнем уровне и принимает ключевые решения по поводу технологий и методологий разработки. Активно занимается исследованиями, экспериментами, рисует многочисленные диаграммы и документирует архитектурные решения.

5 **Программист** (разработчик) пишет код на языках программирования, т.е. непосредственно кодирует логику работы программы. Также является ее первым пользователем и тестировщиком. Непосредственно отвечает за то, что программа работает и работает правильно (в соответствии с техническим заданием).

6 **Ведущий программист** (технический лидер, техлид) - программист, который с технической точки зрения принимает решения о формате реализации функционала и координирует работу команды разработчиков.

7 **QA-специалист** - специалист, который обеспечивает качество продукта (тестирует, контролирует и управляет качеством продукта).

8 **QC специалисты** (контроль качества, автоматизация тестирования) - специалист, который проверяет и отвечает за качество продукта. Пишет код для автоматизации процесса тестирования на разных языках программирования. Помогает команде разработки с точки зрения технических вопросов, вопросов архитектуры и построения приложения

9 **Бизнес-аналитик** собирает, обрабатывает и анализирует информацию, связанную с работой бизнеса: доходы, расходы, продажи, количество задействованных в процессах сотрудников. Бизнес-аналитики нужны не только в IT, но и в компаниях, которые занимаются металлургией, финансами, медициной, строительством. Везде бизнес-аналитик старается сделать так, чтобы область функционировала максимально эффективно

10 **Тимлид** — лидер команды, обеспечивающий достижение проектных целей посредством организации работы команды, состоящей из сотрудников различных направлений компании, а также отвечающий за развитие участников команды, построение коммуникаций (как внутри, так и извне), дисциплину и управление составом команды.

Лекция 4

Тема: процесс тестирования ПО с помощью моделей ЖЦ: Водопадная, V-образная модель, инкрементная модель, спиральная

Чтобы лучше разобраться в том, как тестирование соотносится с программированием и иными видами проектной деятельности, для начала рассмотрим самые основы — модели разработки ПО (как часть жизненного цикла ПО.

При этом сразу подчеркну, что разработка ПО является лишь частью жизненного цикла ПО, и здесь мы говорим именно о разработке.

Материал расскажу сжато: пожалуйста, не воспринимайте его как исчерпывающее руководство — здесь едва ли рассмотрена и сотая доля процента соответствующей предметной области.

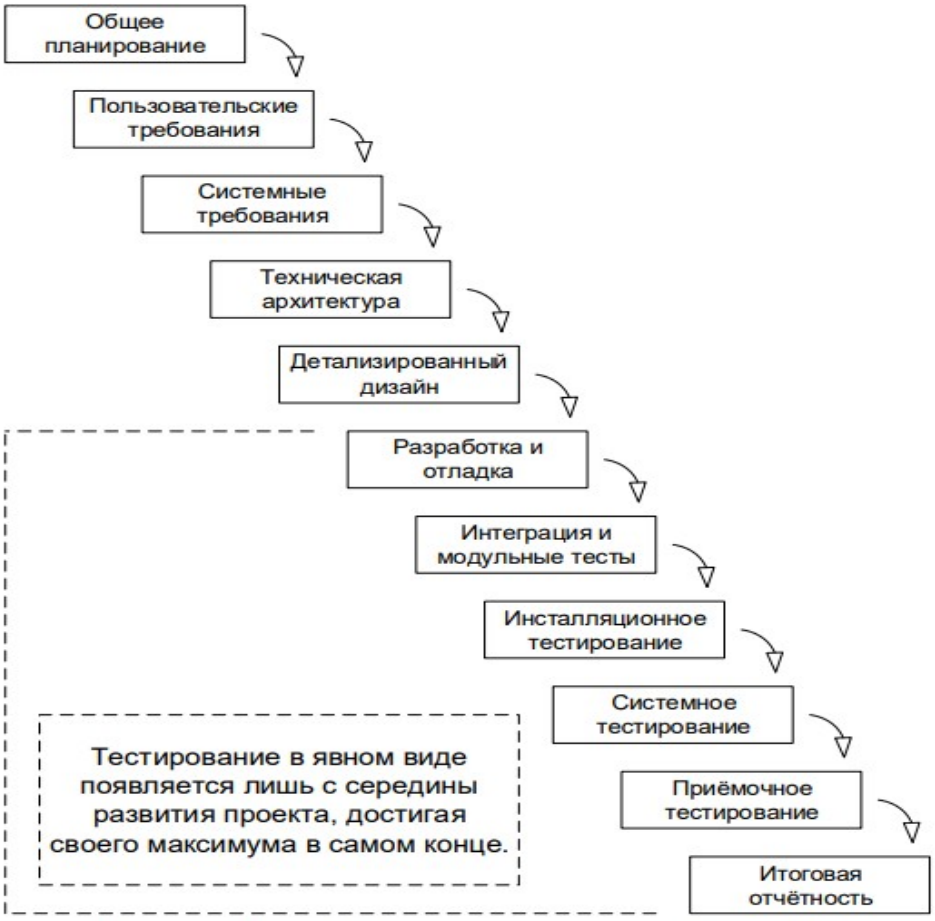


Рисунок 2.1.а — Водопадная модель разработки ПО

Водопадная модель (waterfall model20) сейчас представляет скорее исторический интерес, т.к. в современных проектах практически неприменима. Она предполагает однократное выполнение каждой из фаз проекта, которые, в свою очередь, строго следуют друг за другом (рисунок 2.1.а).

Тестированию подвергается только часть этапов. На середине ЖЦ.

Очень упрощённо можно сказать, что в рамках этой модели в любой момент времени команде «видна» лишь предыдущая и следующая фаза. В реальной же разработке ПО приходится «видеть весь проект целиком» и возвращаться к предыдущим фазам, чтобы исправить недоработки или что-то уточнить.

К недостаткам водопадной модели принято относить

1. тот факт, что участие пользователей ПО в ней либо не предусмотрено вообще, либо предусмотрено лишь косвенно на стадии однократного сбора требований.

2. С точки зрения же тестирования эта модель плоха тем, что тестирование в явном виде появляется здесь лишь с середины развития проекта, достигая своего максимума в самом конце.

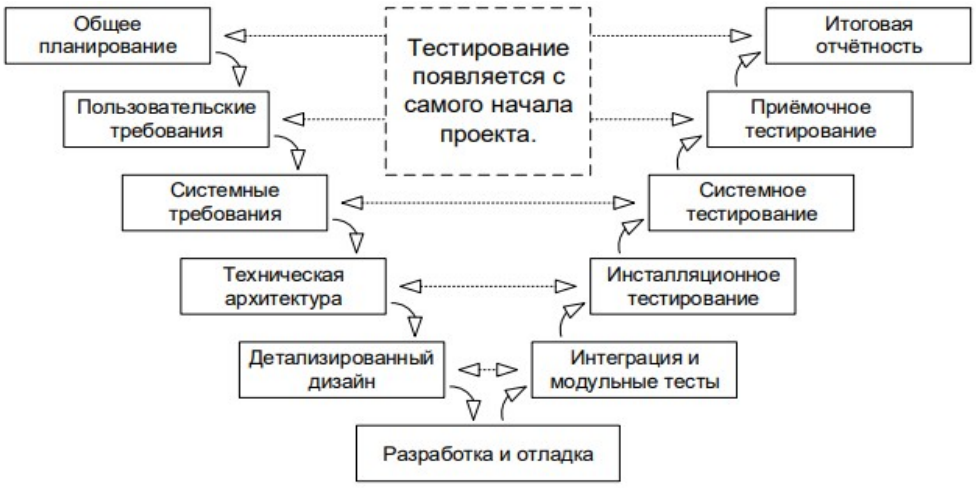


Рисунок 2.1.b — V-образная модель разработки ПО

V-образная модель является логическим развитием водопадной.

Можно заметить, что в общем случае как водопадная, так и v-образная модели жизненного цикла ПО могут содержать один и тот же набор стадий, но *принципиальное отличие заключается в том, как эта информация о всех стадиях используется в процессе реализации проекта*.

Тестирование применяется На переходах между стадиями на всех этапах.

Очень упрощённо можно сказать, что при использовании v-образной модели на каждой стадии «на спуске» нужно думать о том, что и как будет происходить на соответствующей стадии «на подъёме».

На каждой стадии думаем, что будет результатом этой стадии.

Тестирование здесь появляется уже на самых ранних стадиях развития проекта, что позволяет минимизировать риски, а также обнаружить и устранить множество потенциальных проблем до того, как они станут проблемами реальными.

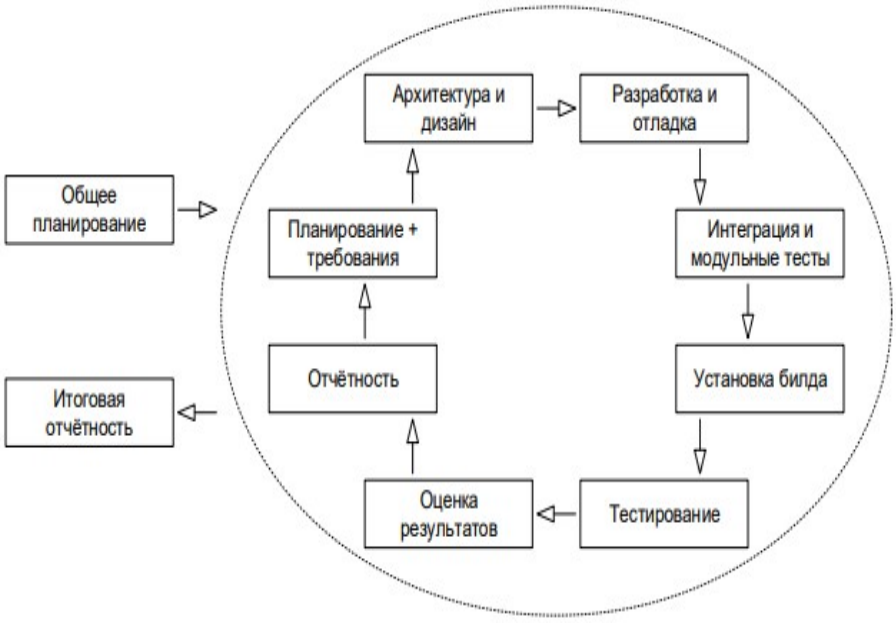


Рисунок 2.1.c — Итерационная инкрементальная модель разработки ПО

Итерационная(инкрементальная) модель является фундаментальной основой современного подхода к разработке ПО.

Тестирование идет с самых первых этапов ЖЦ. Все что включено в круг.

Как следует из названия модели, ей свойственна определённая двойственность:

• с точки зрения жизненного цикла модель является итерационной, т.к. подразумевает многократное повторение одних и тех же стадий;

• с точки зрения развития продукта (приращения его полезных функций) модель является инкрементальной.

Ключевой особенностью данной модели является разбиение проекта на относительно небольшие промежутки (итерации), каждый из которых в общем случае может включать в себя все классические стадии, присущие водопадной и v-образной моделям.

Итогом итерации является приращение (инкремент) функциональности продукта, выраженное в промежуточном билде .

Длина итераций может меняться в зависимости от множества факторов, однако сам принцип многократного повторения позволяет гарантировать, что и *тестирование, и демонстрация продукта конечному заказчику (с получением обратной связи) будет активно применяться с самого начала и на протяжении всего времени разработки проекта.*

сточки зрения тестирования:

-Применяется только в определённые моменты итераций.

- возможно повторное тестирование (после доработки) уже проверенного ранее.

Спиральная модель частный случай итерационной инкрементальной модели, в котором особое внимание уделяется управлению рисками, в особенности влияющими на организацию процесса разработки проекта и контрольные точки.

Обратите внимание на то, что здесь явно выделены **четыре ключевые фазы для тестирования:**

1. проработка целей, альтернатив и ограничений;
2. анализ рисков и прототипирование;
3. разработка (промежуточной версии) продукта;
4. планирование следующего цикла.

С точки зрения тестирования *и управления качеством* ***повышенное внимание рискам является ощутимым преимуществом при использовании спиральной модели*** для разработки концептуальных проектов, в которых требования естественным образом являются сложными и нестабильными (могут многократно меняться по ходу выполнения проекта).

В данной модели тестируется ПО и риски следующего инкремента, стоит его добавлять или нет, что позволяет значительно снизить расходы на разработки и находить слабые места в программе.

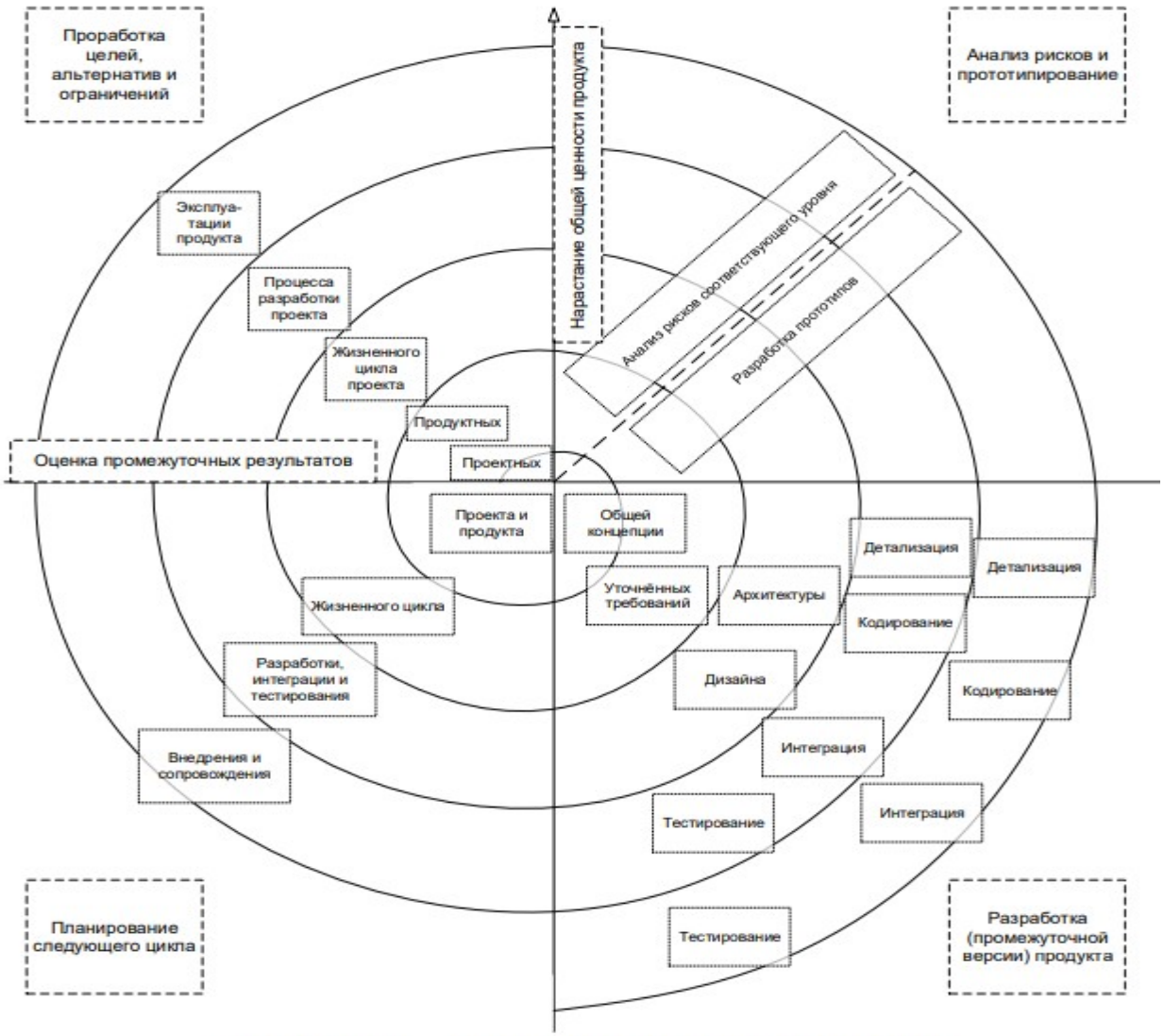


Рисунок 2.1.d — Спиральная модель разработки ПО

сточки зрения тестирования:

-Применяется только в определённые моменты итераций.

- возможно повторное тестирование (после доработки) уже проверенного ранее.

Лекция 5

Тема: Процесс тестирования ПО по гибкой методологии Agile и итерационному подходу в рамках модели и Scrum

Что такое Agile Testing?

Agile Testing – это практика тестирования программного обеспечения, которая следует принципам гибкой разработки программного обеспечения.

Agile Testing включает в себя всех членов команды проекта, со специальным опытом, предоставленным тестировщиками. Тестирование не является отдельной фазой и тесно связано со всеми фазами разработки, такими как требования, проектирование, кодирование и генерация тестовых наборов.

Тестирование происходит одновременно через жизненный цикл разработки.

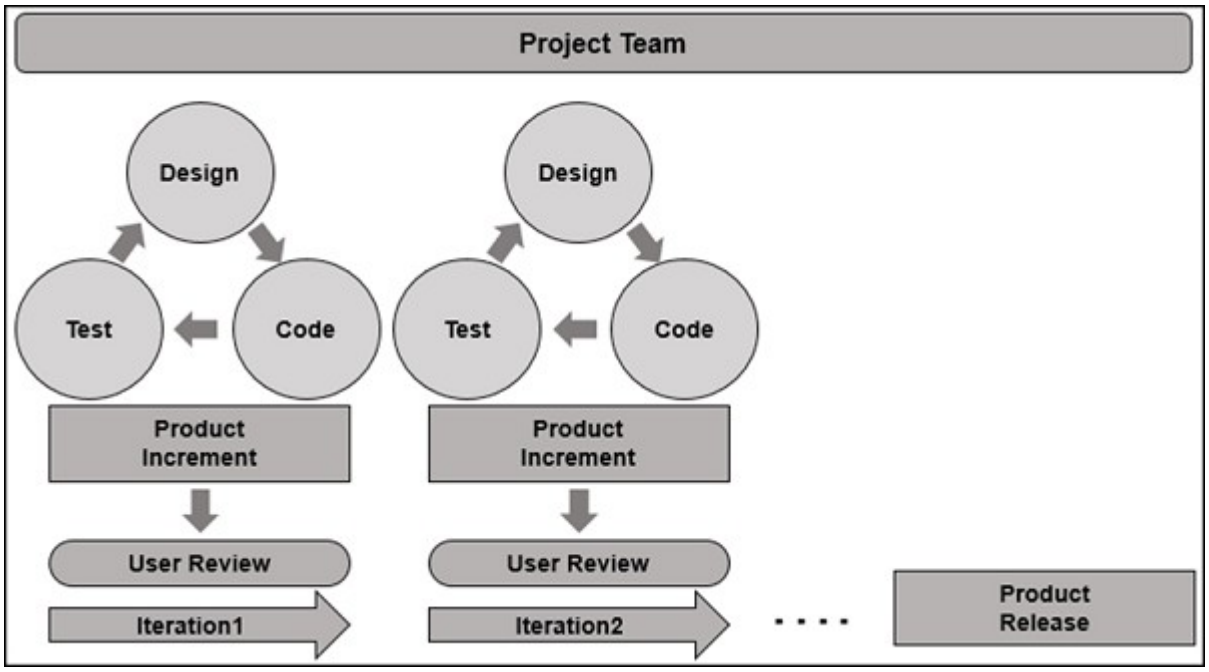
Кроме того, с участием тестировщиков, участвующих во всем жизненном цикле разработки совместно с кросс-функциональными членами команды, станет возможен вклад тестеров в создание программного обеспечения в соответствии с требованиями заказчика, с улучшенным дизайном и кодом.

Agile Testing охватывает все уровни тестирования и все виды тестирования.

Agile Testing – Методологии

Agile – это методология итеративной разработки, в которой вся команда проекта участвует во всех видах деятельности. Требования развиваются по мере продвижения итераций благодаря сотрудничеству между заказчиком и самоорганизующимися командами. Поскольку кодирование и тестирование выполняются в интерактивном режиме и постепенно, в ходе разработки конечный продукт будет иметь качество и обеспечивать требования клиентов.

Каждая итерация приводит к интегрированному приращению рабочего продукта и доставляется для приемочного тестирования. Полученная таким образом обратная связь с клиентом будет входом для следующих / последующих итераций.



в основу гибкой модели подходы являются логическим развитием водопадной, v-образной, итерационной инкрементальной, спиральной и иных моделях. Причём здесь впервые был достигнут ощутимый результат в ***снижении бюрократической составляющей и максимальной адаптации процесса разработки ПО*** к мгновенным изменениям рынка и требований заказчика.***Тестирование применяется В определённые моменты итераций и в любой необходимый момент.***

Agile Testing	Водопад Тестирование
Тестирование не является отдельной фазой и происходит одновременно с разработкой.	Тестирование – это отдельная фаза. Все уровни и виды тестирования могут начинаться только после завершения разработки.
Тестеры и разработчики работают вместе.	Тестеры работают отдельно от разработчиков.
Тестеры участвуют в разработке требований. Это помогает в сопоставлении требований к поведению в сценарии реального мира, а также в формировании критериев приемлемости. Приемочные тестовые случаи будут готовы вместе с требованиями.	Тестеры могут быть не вовлечены в фазу требований.
Приемочное тестирование проводится после каждой итерации и запрашивается обратная связь с клиентом.	Приемочные испытания проводятся только в конце проекта.
Каждая итерация завершает свое собственное тестирование, что позволяет проводить	Регрессионное тестирование может быть реализовано только после завершения

регрессионное тестирование каждый раз, когда выпускаются новые функции или логика.	разработки.
Никаких задержек между кодированием и тестированием.	Обычные задержки между кодированием и тестированием.
Непрерывное тестирование с перекрывающимися уровнями тестирования.	Тестирование – это заданное по времени действие, и уровни тестирования не могут перекрываться.
Тестирование – это лучшая практика.	Тестирование часто пропускается.

Принципы гибкого тестирования:

- **Тестирование продвигает проект вперед.** Непрерывное тестирование – единственный способ обеспечить непрерывный прогресс.

Agile Testing обеспечивает обратную связь на постоянной основе, и конечный продукт отвечает требованиям бизнеса.

- **Тестирование не является этапом** – Agile команда проводит тестирование вместе с командой разработчиков, чтобы убедиться, что функции, реализованные во время данной итерации, действительно выполнены. Тестирование не ведется для более поздней фазы.
- **Вся команда тестирует.** В гибком тестировании приложение тестирует вся команда, включая аналитиков, разработчиков и тестировщиков. После каждой итерации даже заказчик проводит приемочное тестирование.
- **Сокращение циклов обратной связи** – В Agile Testing бизнес-команда узнает о разработке продукта для каждой итерации. Они участвуют в каждой итерации. Непрерывная обратная связь сокращает время отклика обратной связи, и, следовательно, затраты на его устранение уменьшаются.
- **Содержите код в чистоте** . Дефекты исправляются по мере их появления в одной и той же итерации. Это гарантирует чистый код на любом этапе разработки.
- **Облегченная документация** – вместо исчерпывающей документации по тестированию Agile Testers –
    - Используйте многоразовые контрольные списки, чтобы предложить тесты.
    - Сосредоточьтесь на сути теста, а не на случайных деталях.
    - Используйте легкие стили документации / инструменты.
    - Запишите идеи испытаний в чартеры для пробных испытаний.

- Используйте документы для нескольких целей.

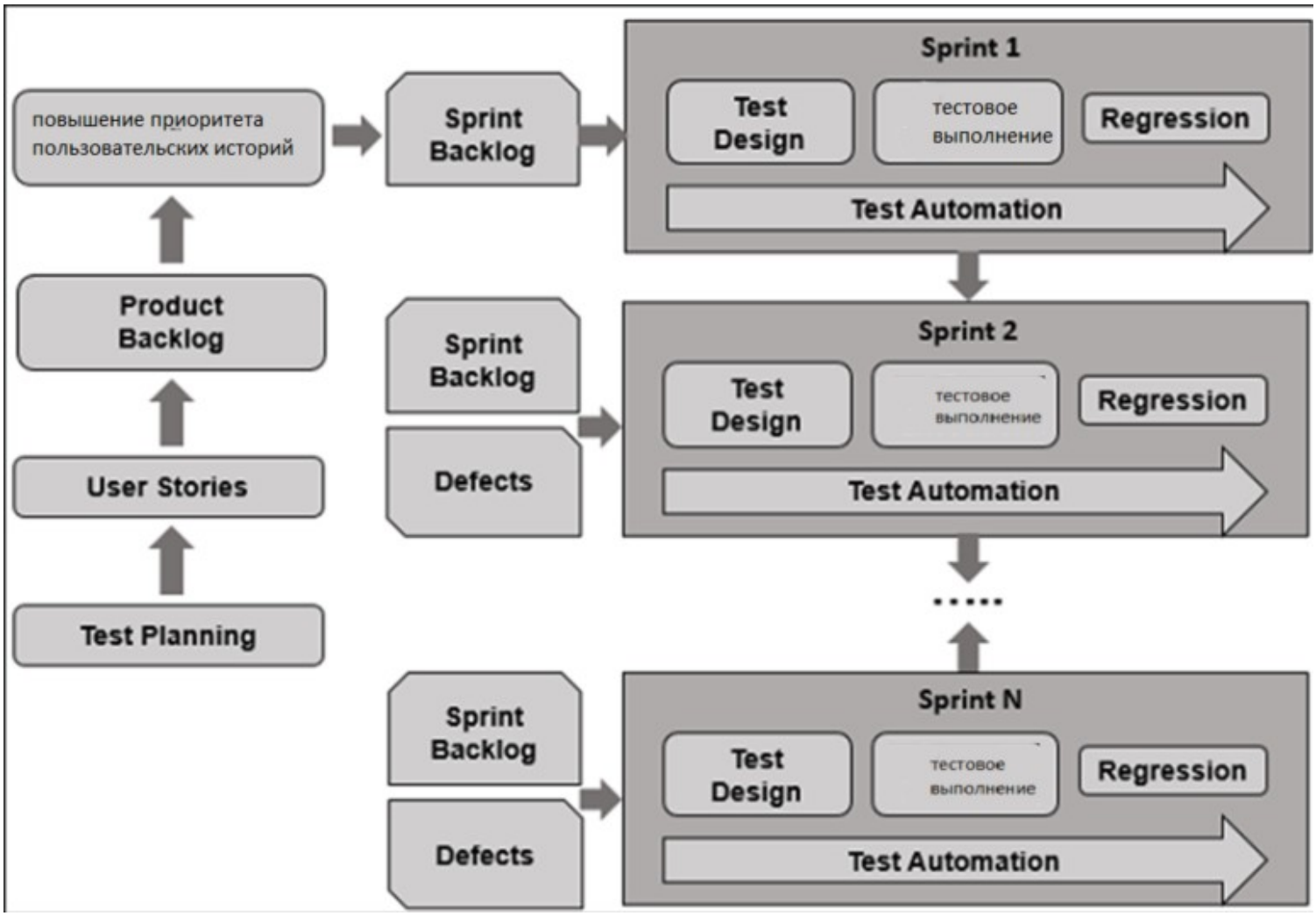
- **Использование одного тестового артефакта для ручных и автоматических тестов. Один и** тот же артефакт тестового скрипта можно использовать для ручного тестирования и в качестве входных данных для автоматических тестов. Это устраняет необходимость в документации по тестированию вручную, а затем эквивалентный сценарий теста автоматизации.

Scrum тестирование включает в себя:

- пользовательские истории, тестовые случаи
- Планирование выпуска, основанное на усилиях по тестированию и дефектах
- Планирование спринта на основе пользовательских историй и дефектов
- Выполнение спринта с непрерывным тестированием
- Регрессионное тестирование после завершения спринта
- Отчет о результатах теста
- Автоматизация тестирования

Вклад в пользовательские истории, основанный на ожидаемом поведении системы, обозначенной как тестовые случаи

Тестирование является итеративным и основано на спринтах, как показано на диаграмме, приведенной ниже –



<u>В определённые моменты итераций. На каждом витке и внутри витка</u>

Лекция 6

Тема: Жизненный цикл тестирования. STLC.

Следуя общей логике итеративности, превалирующей во всех современных моделях разработки ПО, жизненный цикл тестирования также выражается замкнутой последовательностью действий (рисунок 2.1.g).

Жизненный цикл тестирования ПО (**STLC** - Software Testing Lifecycle лайфсайкл)

STLC - это процесс тестирования, который включает в себя определенную последовательность шагов, чтобы гарантировать достижение целей в области качества. В процессе STLC каждое действие выполняется планомерно и систематически. Каждый этап имеет разные цели и результаты. У разных организаций разные этапы STLC, однако основа остается прежней.

Каждая фаза STLC имеет критерии начала и окончания:

- **Критерии входа** (entry criteria): Набор общих и специфичных условий для продолжения процесса с определенной задачей,

например, фаза тестирования.

Цель критериев входа - предотвращение начала задачи, которое может потребовать больше (бесполезных) усилий, чем на устранение не пройденных критериев входа.

- **Критерии выхода** (exit criteria): Набор общих и специфичных условий, согласованных заранее с заинтересованными сторонами, для того, чтобы процесс мог официально считаться завершенным.

Цель критериев выхода - предотвращение возможности, когда задание считается завершенным, однако еще существуют отдельные незавершенные части задания.

Критерии выхода используются для отчетности, а также планирования того, когда остановить тестирование.

Важно понимать, что длина такой итерации (и, соответственно, степень подробности каждой стадии) может варьироваться в широчайшем диапазоне — от единиц часов до десятков месяцев.

Как правило, если речь идёт о длительном промежутке времени, он разбивается на множество относительно коротких итераций, но сам при этом «тяготеет» к той или

иной стадии в каждый момент времени (например, в начале проекта больше планирования, в конце — больше отчётности).

Также ещё раз подчеркнём, что приведённая схема — не правило, и вы легко можете найти альтернативы, но общая суть и ключевые принципы остаются неизменными. Их и рассмотрим.

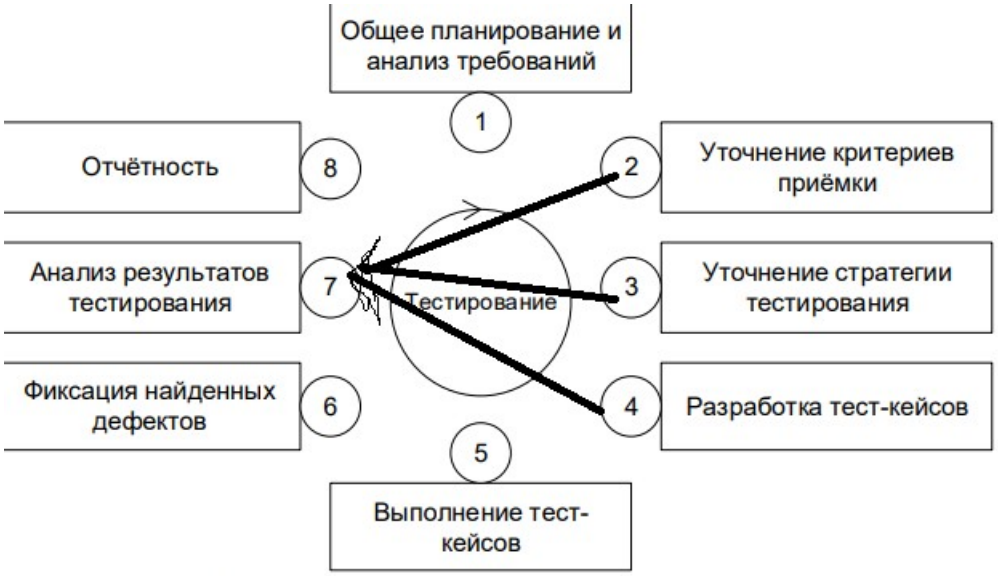


Рисунок 2.1.g — Жизненный цикл тестирования

Стадия 1 (общее планирование и анализ требований) объективно необходима как минимум для того, чтобы иметь ответ на такие вопросы, как: что нам предстоит тестировать; как много будет работы; какие есть сложности; всё ли необходимое у нас есть

один из важнейших этапов, потому что именно на нем можно почти бесплатно исправить недостатки проекта

Как правило, получить ответы на эти вопросы невозможно без анализа требований, т.к. именно требования являются первичным источником ответов.

Стадия 2 (уточнение критериев приёмки) позволяет сформулировать или уточнить метрики и признаки возможности или необходимости начала тестирования, приостановки и возобновления тестирования, завершения или прекращения тестирования.

Как мы должны понять, когда прекращать тестирования, по каким параметрам и критериям смотреть

Стадия 3 (уточнение стратегии тестирования) представляет собой ещё одно обращение к планированию, но уже на локальном уровне: рассматриваются и уточняются те части стратегии тестирования, которые актуальны для текущей итерации.

Тестируются элементы в конкретном витке или спринте, зависит от цели спринта.

Если цель рефакторинг, мы используем тесты для определенных модулей,

если цель внедрение нового функционала используется функциональное и не функциональное тестирование.

Стадия 4 (разработка тест-кейсов) подразумевает использование ручного и автоматизированного тестирования для достижения полного охвата функциональности программного обеспечения, при этом процесс основан на заранее установленных требованиях. Чаще всего тест-кейсы для автоматического тестирования пишутся отдельно, так как кейсы для ручного тестирования описаны в виде шпаргалок

посвящена разработке, пересмотру, уточнению, доработке, переработке и прочим действиям с тест-кейсами, наборами тесткейсов, тестовыми сценариями и иными артефактами, которые будут использоваться при непосредственном выполнении тестирования.

Стадия 5 (выполнение тест-кейсов) и стадия 6 (фиксация найденных дефектов) тесно связаны между собой и фактически выполняются параллельно: дефекты фиксируются сразу по факту их обнаружения в процессе выполнения тест-кейсов.

Однако зачастую после выполнения всех тест-кейсов и написания всех отчётов о найденных дефектах проводится явно выделенная ***стадия уточнения****, на которой все отчёты о дефектах рассматриваются повторно* с *целью формирования единого понимания проблемы и уточнения таких характеристик дефекта, как важность и срочность.*

Стадия 7 (анализ результатов тестирования) и стадия 8 (отчётность) также тесно связаны между собой и выполняются практически параллельно. Формулируемые на стадии анализа результатов выводы напрямую зависят от плана тестирования, критериев приёмки и уточнённой стратегии, полученных на стадиях 1, 2 и 3.

Полученные выводы оформляются на стадии 8 и служат основой для стадий 1, 2 и 3 следующей итерации тестирования. Таким образом, цикл замыкается.

Выводы должны включать затраченное время, процент обнаруженных ошибок и положительных результатов тестирования, общее количество обнаруженных и исправленных ошибок. Что касается отдела тестирования, то это момент для анализа его работы, подведения итогов, анализа его продуктивности и возможности внести предложения по улучшению качества тестирования.

Фазы тестирования.

1. **Создание тестового набора** (test suite) путем ручной разработки или автоматической генерации для конкретной среды тестирования (testing environment).
2. **Прогон программы на тестах**, управляемый тестовым монитором (test monitor, test driver) с получением протокола тестирования (test log).

3. **Оценка результатов выполнения программы** на наборе тестов с целью принятия решения о продолжении или остановке тестирования.

Полный цикл тестирования и его задачи

Рассмотрим более подробно существующие активности/задачи связанные с тестированием:

1) планирование тестов:

- определение требований к тестам;
- оценка рисков;
- выбор стратегии тестирования;
- определение ресурсов;
- создание расписания/последовательностей;
- разработка Плана тестирования;

2) дизайн тестов:

- анализ объёма работ;
- определение и описание тестовых случаев;
- определение и структурирование тестовых процедур;
- обзор и оценка тестового покрытия;

3) разработка тестов:

- запись или программирование тестовых скриптов;
- определение тесто-критичной функциональности в Дизайне и Модели реализации;
- создание/подготовка внешних наборов данных;

4) выполнение тестов:

- выполнение тестовых процедур;
- оценка выполнения тестов;
- восстановление после сбойных тестов;
- проверка результатов;
- исследование неожиданных результатов;
- запись ошибок;

5) оценка тестов:

- оценка покрытия тестовыми случаями;
- оценка покрытия кода;
- анализ дефектов;
- определение критериев завершения и успешности тестирования

Лекция 7

Тема: Тест кейс. Чек-лист. Правила создания чек листов. Функции чек-листа.

Любая программа должна соответствовать всем требованиям. В сложных проектах список требований может быть огромными — а на каждое требование нужны какие-то тесты.

Чтобы тестировщику было удобнее работать с таким количеством требований, ему нужны инструменты — чеклисты и тест-кейсы.

По своей сути, это разные форматы документации.

Но цель обоих документов освободить тестировщиков от того , чтобы им не пришлось все держать в голове.

Чек-листы и тест-кейсы снижают вероятность ошибок, связанных с человеческим фактором — причем это работает даже в проектах с неидеальной документацией.

Чек- лист — это просто список того, что нужно проверить.

Тест-кейс — подборный план того, как именно мы будем это проверять.

Чек-лист включают в себя тест-кейсы

Тест-кейсы и чек-лист составляются **до того**, как мы сели тестировать. Это наш план проверки.

То есть мы не тестируем наш продукт, а просто сели думать, как мы будем его тестировать.

Чек-лист не имеет ожидаемого результата(хотя не запрещено его там указывать)

Фактический результат мы узнаем после тестирования. И *когда этот фактический результат будет отличаться от ожидаемого* (который мы прописали в тест-кейсе), то *мы запишем это в баг-репорт*.

чек-лист – это документация, в которой тестировщик описывает процесс по шагам. В нем отражается набор идей по тестированию, разработке, планированию и управлению — любых идей, а их детализация в виде шагов и ожидаемых результатов будет в тест-кейсах. Чек-лист не имеет ожидаемого результата

Самое важное, что отличает чек-лист от тест-кейсов – здесь нет подробной детализации. Чеклист не описывает подробно все шаги, а просто перечисляет их. В нем не уточняется, какие тестовые данные нужно использовать, как проводить проверки.

В простом случае чек-лист выглядит примерно так:

- Проверить фильтр «Поиск» в Программах
- Проверить фильтры в Программах
- Проверить «Программы»

Чек-лист чаще всего представляет собой список:

1. в котором последовательность пунктов не имеет значения

список значений некоего поля, перечень значений;

2. в котором последовательность пунктов важна

шаги в краткой инструкции

3. структурированный (многоуровневый) список, который позволяет отразить иерархию идей.

Чек листы могут быть графические, наброски, идеи, план, списки

Случаи добавления, ожидаемые результатов в чек-лист:

1. в некоем *пункте чек-листа рассматривается особое, нетривиальное поведение* приложения или сложная проверка, результат которой важно отметить уже сейчас, чтобы не забыть;

2. в силу *сжатых сроков и/или нехватки иных ресурсов* тестирование проводится напрямую по чек-листам без тест-кейсов.

Свойства важные для составления верного чек-листа:

1. **Логичность.** Умение правильно изложить собственные мысли и построить между ними взаимосвязь

Чек-лист пишется не «просто так», а на основе целей и чтобы достичь их нужно установить порядок мыслей и взаимосвязь между ними

Основная ошибка при составлении чек-листа является превращение его в свалку мыслей, которые никак не связаны друг с другом.

2. **Последовательность и структурированность**. структурированностью достигается за счёт оформления чек-листа в виде многоуровневого списка. Что до последовательности, то даже в том случае, когда пункты чек-листа не описывают цепочку действий, человеку всё равно удобнее воспринимать информацию в виде неких небольших групп идей, переход между которыми является понятным и очевидным

Структорность-многоуровневый список

Последовательность- можно прописать идеи простых позитивных тест-кейсов, потом идеи простых негативных тест-кейсов, потом постепенно повышать сложность тест-кейсов, но не стоит писать эти идеи вперемешку).

3. **Полнота и неизбыточность**. Чек-лист должен представлять собой аккуратную «сухую выжимку» идей, в которых нет дублирования (часто появляется из-за разных формулировок одной и той же идеи), и в то же время ничто важное не упущено.

Типичными вариантами такой логики является создание отдельных чек-листов для:

1. ***типичных пользовательских сценариев*** ;

регистрация, авторизация, покупка

2. <u>различных уровней функционального тестирования ;</u>

- модульное тестирование(сами модули),
- интеграционное тестирование (связь между модулями),
- системное тестирование(всю систему сквозное) ,
- приемочное(на этапе приема-сдачи продукта),
- *операционное тестирование(бизнес-процессы)*

3. <u>отдельных частей (модулей и подмодулей) приложения;</u>

модуль работы с заказом, модуль работы с покупка,бронь

4. <u>отдельных требований, групп требований, уровней и типов требований ;</u>

пользовательский(какой должен быть макет), системные(сколько памяти), общие и.т.д

5. <u>частей или функций приложения, наиболее подверженных рискам.</u>

Работа с кэш,

Этот список можно расширять и дополнять, можно комбинировать его пункты, получая, например, чек-листы для проверки наиболее типичных сценариев, затрагивающих некую часть приложения.

Чтобы проиллюстрировать принципы построения чек-листов, мы воспользуемся логикой разбиения функций приложения по степени их важности на три категории

<u>Разбиение функций по степени важности:</u>

1. *<u>Базовые функции</u>*, <u>без которых существование приложения теряет смысл</u>

самые важные — то, ради чего приложение вообще создавалось, или нарушение работы которых создаёт объективные серьёзные проблемы для среды исполнения. «<u>Дымовое тестирование</u>».

- ✓ ***<u>Конфигурирование и запуск.</u>***

Если приложение невозможно настроить для работы в пользовательской среде. Если приложение не запускается. Если на стадии запуска возникают проблемы

- ✓ ***<u>Обработка файлов</u>***

здесь на стадии создания чек-листа мы создаем матрицу, отражающую все возможные комбинации допустимых форматов и допустимых кодировок входных и выходных файлов

- ✓ ***<u>Остановка</u>***

С точки зрения пользователя эта функция не важна, но остановка (и запуск) любого приложения связаны с большим количеством системных операций, проблемы с которыми могут привести к множеству серьёзных последствий (вплоть до невозможности повторного запуска приложения или нарушения работы операционной системы

2. ***Функции, востребованные большинством пользователей*** в их повседневной работе. «Тестирование критического пути» .

3. ***Остальные функции*** разнообразные «мелочи», проблемы с которыми не сильно повлияют на ценность приложения для конечного пользователя. «Расширенное тестирование»

Лекция 8

Тема: Чек-лист для Web приложений и сайтов. Разделы чек-листа: Функциональное тестирование. Интеграционное тестирование. Тестирование безопасности. Тестирование локализации и глобализации. Тестирование удобства использования. Кросс-платформенное тестирование

Чек-лист для тестирования WEB приложений состоит из шести разделов:

1. Функциональное тестирование
2. Интеграционное тестирование
3. Тестирование безопасности
4. Тестирование локализации и глобализации
5. Тестирование удобства использования
6. Кросс-платформенное тестирование

ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

В данном пункте нам важно убедиться, что наш продукт соответствует нужной функциональной спецификации, упомянутой в документации по разработке.

Что проверяем?

1. Тестирование форм
    1. Регистрация
        1. Пользователь с данными существует в системе.
        2. Пользователь с данными не существует в системе.
        3. Пользователь, заблокированный в системе, не может пройти повторную регистрацию.
    2. Авторизация
        1. Пользователь существует в системе с введенным логином и паролем.
        2. Пользователь с введенным логином не существует в системе.
        3. Пользователь с введенным логином существует в системе, но пароль неверный.
        4. Пользователь с введенным логином и паролем существует в системе, но заблокирован модерацией (страница заморожена).
        5. Валидация полей ввода.
    3. Протестируйте валидацию всех обязательных полей
        1. Максимальная и минимальная длина.
        2. Диапазон допустимых символов, спецсимволы.
        3. Обязательность к заполнению.
        4. Убедитесь, что астериск (знак звездочки) отображается у всех обязательных полей.
        5. Убедитесь, что система не отображает окно ошибки при незаполненных необязательных полях.
    4. Формы обратной связи
    5. Ссылки на пользовательские соглашения
2. Поиск
    1. Результаты существуют/не существуют.
    2. Корректное сообщение о пустом результате.
    3. Пустой поисковой запрос.

    4. Поиск по эмодзи.
3. Поля
    1. Числовые поля: они не должны принимать буквы, в этом случае должно отображаться соответствующее сообщение об ошибке.
    2. Дробные значения, например, как система валидирует 1.1 и 1,1.
    3. Отрицательные значения в числовых полях, если они разрешены.
    4. Деление на ноль корректно обрабатывается.
    5. Протестируйте максимальную длину каждого поля, чтобы убедиться, что данные не обрезаются или скрываются под многоточие.
    6. Протестируйте все поля ввода на спецсимволы.
    7. Проверить что текст не выезжает за границы поля.
4. Всплывающие сообщения
    1. Протестируйте всплывающие сообщения («Это поле ограничено N знаками»).
    2. Подтверждающие сообщения отображается для операций обновления и удаления.
    3. Сообщения об ошибках ввода.
5. Фильтры
    1. Протестируйте функциональность сортировки (по возрастанию, по убыванию, по новизне).
    2. Задать фильтры с выдачей.
    3. Задать фильтры, по которым нет выдачи.
    4. Фильтры по категориям/подкатегориям.
    5. Фильтры с радиусом поиска.
    6. Данные в выпадающих списках.
6. Протестируйте функциональность доступных кнопок.
7. Наличие favicon.
8. Проверка обработки различных ошибок (страница не найдена, тайм-аут, ошибка сервера и т.д.).
9. Протестируйте, что все загруженные документы правильно открываются.
10. Пользователь может скачать/прикрепить/загрузить файлы/медиа (картинки, видео и т.д.). А также удалить эти файлы из вложений. Убедитесь, что файлы уходят на сервер только после нажатия соответствующей кнопки
11. Протестируйте почтовую функциональность системы.
12. Кеш, cookie и сессии
    1. Пользователь очистил кэш браузера
    2. Посмотрите, что будет, если пользователь удалит куки, находясь на сайте.
    3. Посмотрите, что будет, если пользователь удалит куки после посещения сайта.
13. DevTools
    1. Ошибки в Console.
    2. Все стили загружаются.
    3. Картинки загружаются.

ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ

Интеграционное тестирование проводится для того, чтобы убедиться, что ваше приложение совместимо со сторонними сервисами.

Что проверяем?

1. Проверяем работу сторонних модулей: оплата, шаринг, карты.
2. Реклама (просмотр, переходы по рекламе, аналитика).

3. Метрики (переходы по страницам, показы элементов, клики).

ТЕСТИРОВАНИЕ БЕЗОПАСНОСТИ

Данная проверка нацелена на поиск недостатков и пробелов с точки зрения безопасности нашего приложения.

Что проверяем?

1. Пользователь не может авторизоваться: под старым паролем, заблочен в сервисе, достиг лимита авторизаций, ввел чужой код верификации.
2. Страницы, содержащие важные данные (пароль, номер кредитной карты и CVC, ответы на секретные вопросы и т. п.) открываются через HTTPS (SSL).
3. Пароль скрыт астерисками на страницах: регистрация, «забыли пароль», «смена пароля».
4. Корректное отображение сообщений об ошибках.
5. Завершение сессии после разлогина.
6. Доступ к закрытым разделам сайта.
7. SQL-инъекции.
8. Cross-Site Scripting (XSS) уязвимости.
9. HTML-инъекции.
10. Cookie должны храниться в зашифрованном виде.
11. Роли пользователей и доступ к контенту.

ТЕСТИРОВАНИЕ ЛОКАЛИЗАЦИИ И ГЛОБАЛИЗАЦИИ

Тестирование интернационализации/глобализации WEB приложения включает тестирование приложения для различных местоположений, форматов дат, чисел и валют. Тестирование локализации включает тестирование WEB приложения с локализованными строками, изображениями и рабочими процессами для определенного региона.

Что проверяем?

1. Дата и время. Например отображение времени, даты в соответствии с часовым поясом пользователя.
2. Смена языка и проверка перевода всех элементов WEB приложения исходя из выбранного языка.
3. Выбор номера телефона с разными кодами стран.
4. Определение местоположения пользователя и отображение соответствующего пермишена ГЕО.
5. Отображение соответствующих символов валюты.

ТЕСТИРОВАНИЕ УДОБСТВА ИСПОЛЬЗОВАНИЯ

Тестирование удобства использования подразумевает проверку навигации, контента, другая информация для пользователя.

Что проверяем?

1. Отсутствие орфографических и грамматических ошибок, все страницы имеют корректные заголовки.
2. Выравнивание картинок, шрифтов, текстов.
3. Информативные ошибки, подсказки.
4. Подсказки существуют для всех полей.
5. Отступы между полями, колонками, рядами и сообщениями об ошибках.
6. Кнопки имеют стандартный размер, цвет.
7. На сайте нет битых ссылок и изображений.
8. Неактивные поля отображаются серым цветом.
9. Проверьте сайт при разных разрешениях экрана.

10. Скролл должен появляться только тогда, когда он требуется.
11. Отображение чекбоксов и радио-кнопок, кнопки должны быть доступны с клавиатуры, и пользователь должен быть в состоянии пользоваться сайтом, используя только клавиатуру.
12. Отображение выпадающих списков.
13. Длинный текст скрывается под многоточие.
14. Корректный выбор даты.
15. Наличие плейсхолдеров в полях.
16. Логотип ведет на главную страницу сайта.
17. Переходы и навигация между страницами и разделами меню.

КРОСС-ПЛАТФОРМЕННОЕ ТЕСТИРОВАНИЕ

Кросс-платформенное тестирование проводится, чтобы убедиться, что ваше приложение совместимо с другими браузерами, различными оболочками, аппаратным обеспечением устройства.

Что проверяем?

1. Тестирование в различных браузерах (Firefox, Chrome, Safari — это минимальный набор): анимация, верстка, шрифты, уведомления и т.д.
2. Тестирование в различных версиях ОС: Windows, Mac, Linux.
3. Java Script код работает в разных браузерах.
4. Просмотр на мобильных устройствах.

Лекция 9

Тема: тест-кейс. Баг репорт. Жизненный цикл тест кейса. Виды тест-кейсов

Для начала определимся с терминологией.

Во главе всего лежит термин «тест». Официальное определение звучит так.

Тест (test) — набор из одного или нескольких тест-кейсов

Теперь рассмотрим самый главный для нас термин — «тест-кейс».

Тест-кейс (test case) — набор входных данных, условий выполнения и ожидаемых результатов, разработанный с целью проверки того или иного свойства или поведения программного средства. Под тест-кейсом также может пониматься соответствующий документ, представляющий формальную запись тест-кейса.

Тест-кейс — подборный план того, как именно мы будем это проверять по чек листу.

Тест-кейс — это более подробная документация, в которой описано все необходимое:

- ✓ -набор входных значений,
- ✓ -предусловия выполнения,
- ✓ -ожидаемые результаты.
- ✓ Посту условия

У каждого тест-кейса есть определенная цель или тестовое условие — например, проверить выполнение определенного пути программы или соответствие продукта определенному требованию.

В отличие от чек-листа, тест-кейсы не так просты в составлении. Чтобы составить кейс, нужно четко описать необходимые действия, поля для ввода, кнопки и так далее.

Обычно тест-кейс содержит:

1. **Предусловия**. Они описывают, что нужно сделать до начала проверки
2. **Шаги**. Это действия, которые надо выполнить во время проверки
3. **Ожидаемый результат**. Здесь тестировщик описывает, что должно произойти после выполнения действий для проверки

```
* Предусловие:
    * В браузере открыта страница https://ru.hexlet.io/programs
* Шаги:
    * Ввести в поле «Поиск» значение «фронтенд»
* Ожидаемый результат:
    * В списке программ отображаются только программы, которые содержат слово «фронтенд» в названии
```

Тестировщик выполняет тест-кейс последовательно, шаг за шагом:

– Если фактический результат соответствует ожидаемому — всё хорошо.

– Если нет, тестировщик анализирует, что произошло. Это может быть ошибка в программе, в тест-кейсе из-за его неактуальности или в тестовом стенде.

– Если дело в программе, инженер составляет отчёт об ошибке и отправляет его разработчикам.

– Если в тест-кейсе — исправляет сам.

– Если в постановке — обращается к техническим специалистам.

Как правило, один тест-кейс описывает небольшую последовательность действий с одним конкретным результатом. Например, успешную авторизацию на сайте для конкретного пользователя или добавление одного конкретного товара в корзину. Обычно пишут к задачам, которые нужно периодически повторять

Баг-репорт же вообще относится не к проверке, а к её результату. Тестировщик во время проверки находит ошибку — и пишет по ней баг-репорт, то есть отчёт об этой ошибке. Получается, что тест-кейс — это описание процесса проверки, а баг-репорт — описание процесса воспроизведения ошибки и материалы, относящиеся к ошибке.

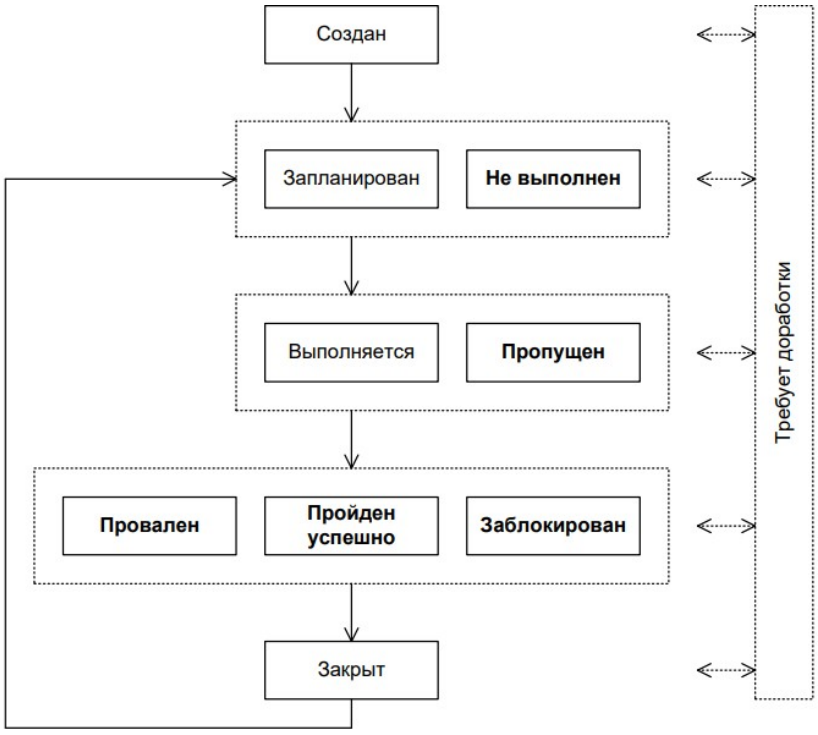


Рисунок 2.4.а — Жизненный цикл (набор состояний) тест-кейса

Жизненный цикл тест-кейса В отличие от отчёта о дефекте, у которого есть полноценный развитый жизненный цикл, для тест-кейса речь скорее идёт о наборе состояний , в которых он может находиться (*жирным шрифтом отмечены наиболее важные состояния*).

Создан (new) — типичное начальное состояние практически любого артефакта. Тест-кейс автоматически переходит в это состояние после создания.

Запланирован (planned, ready for testing) — в этом состоянии тест-кейс находится, когда он или явно включён в план ближайшей итерации тестирования, или как минимум готов для выполнения.

НЕ ВЫПОЛНЕН (not tested) — в некоторых системах управления тест-кейсами это состояние заменяет собой предыдущее («запланирован»). Нахождение тест-кейса в данном состоянии означает, что он готов к выполнению, но ещё не был выполнен.

Выполняется (work in progress) — если тест-кейс требует длительного времени на выполнение, он может быть переведён в это состояние для подчёркивания того факта, что работа идёт, и скоро можно ожидать её результатов.

Если выполнение тест-кейса занимает мало времени, это состояние, как правило, пропускается, а тест-кейс сразу переводится в одно из трёх следующих состояний — «провален», «пройден успешно» или «заблокирован».

ПРОПУЩЕН (skipped) — бывают ситуации, когда выполнение тест-кейса отменяется по соображениям нехватки времени или изменения логики тестирования.

ПРОВАЛЕН (failed) — данное состояние означает, что в процессе выполнения тест-кейса был обнаружен дефект, заключающийся в том, что ожидаемый результат по как минимум одному шагу тест-кейса не совпадает с фактическим результатом.

Если в процессе выполнения тест-кейса был «случайно» обнаружен дефект, никак не связанный с шагами тест-кейса и их ожидаемыми результатами, тест-кейс считается пройденным успешно (при этом, естественно, по обнаруженному дефекту создаётся отчёт о дефекте).

ПРОЙДЕН УСПЕШНО (passed) — данное состояние означает, что в процессе выполнения тест-кейса не было обнаружено дефектов, связанных с расхождением ожидаемых и фактических результатов его шагов.

ЗАБЛОКИРОВАН (blocked) — данное состояние означает, что по какой-то причине выполнение тест-кейса невозможно (как правило, такой причиной является наличие дефекта, не позволяющего реализовать некий пользовательский сценарий).

Закрыт (closed) — очень редкий случай, т.к. тест-кейс, как правило, оставляют в состояниях «провален / пройден успешно / заблокирован / пропущен». В данное состояние в некоторых системах управления тест-кейс переводят, чтобы подчеркнуть тот факт, что на данной итерации тестирования все действия с ним завершены.

Требует доработки (not ready) — как видно из схемы, в это состояние (и из него) тест-кейс может быть переведён в любой момент времени, если в нём будет обнаружена ошибка, если изменятся требования, по которым он был написан, или наступит иная ситуация, не позволяющая считать тест-кейс пригодным для выполнения и перевода в иные состояния.

Виды тест-кейсов:

Существует три вида тест-кейсов:

● **Позитивные, или положительные.** Проверяют, что система адекватно реагирует на корректные данные.

Например, если при регистрации ввести в поле логина существующий, корректно написанный email, ещё не зарегистрированный в системе, сайт поймёт это правильно и допустит регистрацию.

● **Негативные, или отрицательные.** Показывают, что система умеет работать с некорректными данными.

Например, если не написать в email значок @ или пропустить точку, сайт сообщит об ошибке и не допустит регистрацию.

● **Деструктивные.** Служат для проверки прочности системы.

Например, позволяют убедиться, что в поле для email нельзя ввести команду, которая удалит базу данных зарегистрированных пользователей.

Лекция 10

Тема: Атрибуты тест-кейса. Правила составления тест кейсов. Сравнительный анализ тест-кейсов и чек-листов.

Атрибуты тест-кейса:

Тест-кейсы принято оформлять по определённому стандарту. Поэтому каждый тест-кейс состоит из нескольких чётких элементов — атрибутов:

1. Идентификатор. Это может быть любая нумерация, принятая в проекте. Он позволит ссылаться на определённые тесты по номеру.

2. Приоритет показывает важность тест-кейса. Он может быть выражен буквами (A, B, C, D, E), цифрами (1, 2, 3, 4, 5), словами («крайне высокий», «высокий», «средний», «низкий», «крайне низкий»)

Количество градаций также не фиксировано, но чаще всего лежит в диапазоне от трёх до пяти.

Приоритет тест-кейса может корректировать с:

1. *важностью требования*, пользовательского сценария или функции, с которыми связан тест-кейс;

2. *потенциальной важностью дефекта*, на поиск которого направлен тест-кейс;

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

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

4. Заголовок. Кратко, но ёмко описывает конкретную цель тест-кейса **—** что именно нужно проверить.

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

6. Исходные данные. Условия, которые нужно соблюсти перед началом тест-кейса. Как правило, нужно авторизоваться или находиться в определённом разделе программы.

7. Шаги — последовательность шагов, которую нужно проделать для проверки.

8. Ожидаемый результат тест-кейса. То, что тестировщик должен получить от системы после или во время прохождения шагов.

9. Статус. Passed/Failed, то есть Успех/Провал или другой. Его заполняет тестировщик из заранее определённых вариантов, принятых в команде.

10. Окружение. Где именно работал тестировщик: на каком устройстве, в каком браузере. Иногда его заполняют до тестирования, чтобы указать, на каком именно оборудовании и ПО его проходить. Иногда — после, и тогда тестировщик сам указывает, в каком окружении работал.

11. Постусловия. Действия, которые нужно проделать после проведения проверки. Этот пункт встречается редко, но иногда он необходим.

Например, может понадобиться удалить внесённые данные, чтобы они не скапливались в базе.

12. Фактический результат тест-кейса. То, что получилось после выполнения шагов тест-кейса. Часто этого поля нет, и фактический результат описывают в баг-репорте в случае статуса failed.

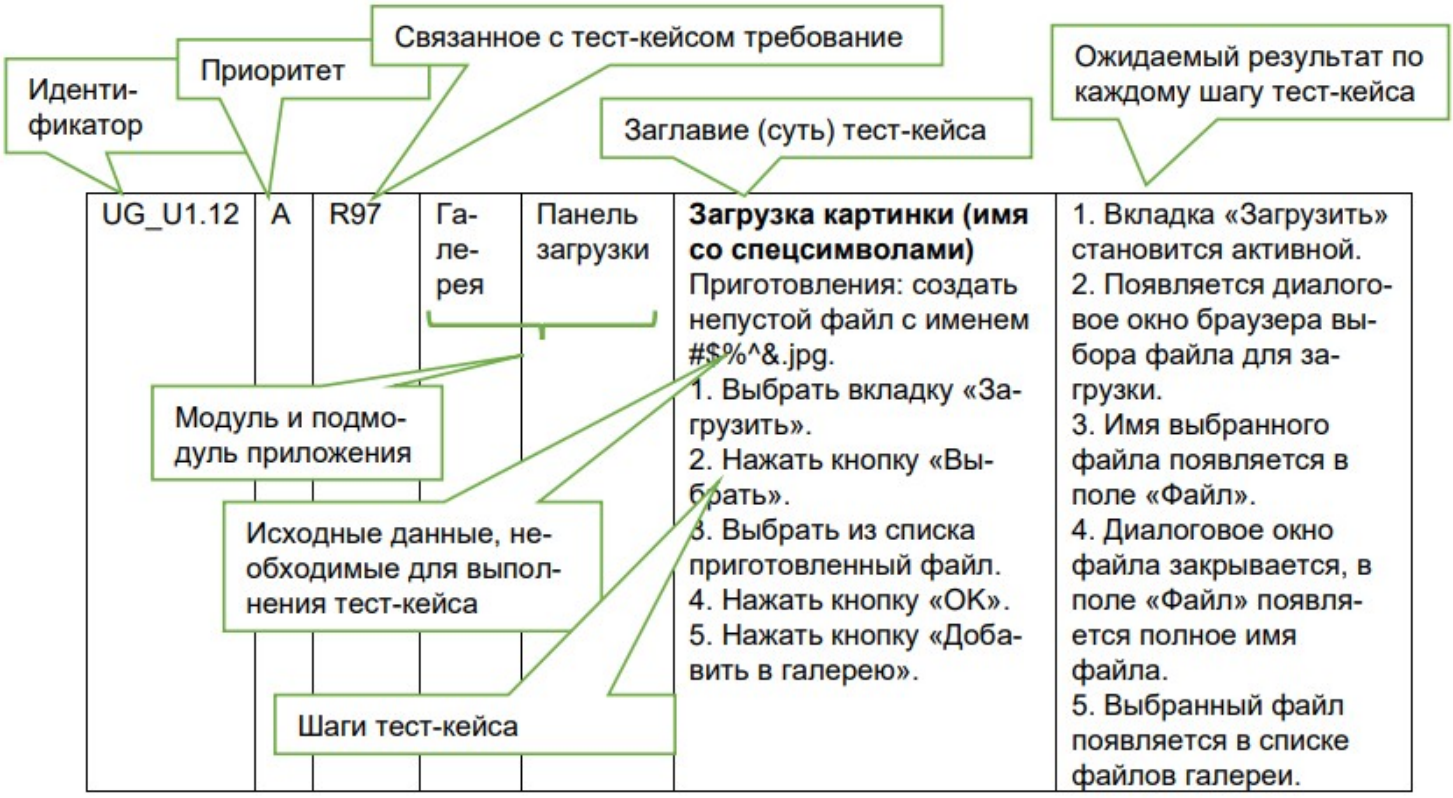


Рисунок 2.4.b — Общий вид тест-кейса

Правила составления тест-кейсов:

Тестировщики сами составляют тест-кейсы на основе требований к разрабатываемому продукту. Поэтому для них важен навык правильного написания тест-кейсов.

При создании тест-кейса важно учитывать следующие моменты:

1. Заголовок должен быть чётким, лаконичным и выражающим суть проверки. В него не нужно добавлять шаги тест-кейса.

2. В исходных данных важно описать состояние системы, которое нужно для выполнения шагов тест-кейса. Например, там могут быть конкретные ссылки на среды, где проводятся тестирования. Или на документы, которые понадобятся для прохождения шагов.

3. Шаги тест-кейса не нужно описывать слишком подробно. Например, следует писать «Введите email» вместо «Введите email, нажимая клавиши на клавиатуре».

4. *Шаги не должны быть размытыми или абстрактными*. Нельзя сказать «Зайдите в раздел «Магазин» — лучше указать путь к нему, если он не слишком очевиден.

5. Скриншоты лучше использовать только как дополнение к словесному описанию, но не в качестве его замены.

Лекция 11

Тема: Метрики и покрытие тестами ПО. Метрики остановки и продолжения тестирования. Правила покрытия тестами ПО.

Метрики (metrics). Числовые характеристики показателей качества, способы их оценки, формулы и т.д.

На этот раздел, как правило, формируется множество ссылок из других разделов тест-плана.

Метрика (metric) — числовая характеристика показателя качества. Может включать описание способов оценки и анализа результата.

Сначала поясним важность метрик на тривиальном примере.

Представьте, что заказчик интересуется текущим положением дел и просит вас кратко охарактеризовать ситуацию с тестированием на проекте.

Общие слова в стиле «всё хорошо», «всё плохо», «нормально» и тому подобное его, конечно, не устроят, т.к. они предельно субъективны и могут быть крайне далеки от реальности.

И совсем иначе выглядит ответ наподобие такого: «Реализовано 79 % требований (в т.ч. 94 % важных), за последние три спринта тестовое покрытие выросло с 63 % до 71 %, а общий показатель прохождения тест-кейсов вырос с 85 % до 89 %.

Иными словами, мы полностью укладываемся в план по всем ключевым показателям, а по разработке даже идём с небольшим опережением».

Чтобы оперировать всеми этими числами (а они нужны не только для отчётности, но и для организации работы проектной команды), их нужно как-то вычислить.

Именно это и позволяют сделать метрики. Затем **вычисленные значения можно использовать для**:

1. принятия решений о начале, приостановке, возобновлении или прекращении тестирования
2. определения степени соответствия продукта заявленным критериям качества;
3. определения степени отклонения фактического развития проекта от плана;
4. выявления «узких мест», потенциальных рисков и иных проблем;
5. оценки результативности принятых управленческих решений;
6. подготовки объективной информативной отчётности;

Метрики могут быть

1. **прямыми** (не требуют вычислений) количество разработанных тест-кейсов, количество найденных дефектов
2. **расчётными** (вычисляются по формуле). В расчётных метриках могут использоваться как совершенно тривиальные, так и довольно сложные формулы

расчетные метрики:

МЕТРИКИ:

1) УСПЕШНОЕ ПРОХОЖДЕНИЕ ТЕСТ-КЕЙСОВ:

$T^{SP} = \frac{T^{Success}}{T^{Total}} \cdot 100\%$, где

T^{SP} — процентный показатель успешного прохождения тест-кейсов,
$T^{Success}$ — количество успешно выполненных тест-кейсов,
T^{Total} — общее количество выполненных тест-кейсов.

Минимальные границы значений:

- Начальная фаза проекта: 10%.
- Основная фаза проекта: 40%.
- Финальная фаза проекта: 80%.

2) ОБЩЕЕ УСТРАНЕНИЕ ДЕФЕКТОВ:

$D_{Level}^{FTP} = \frac{D_{Level}^{Closed}}{D_{Level}^{Found}} \cdot 100\%$, где

D_{Level}^{FTP} — процентный показатель устранения дефектов уровня важности $Level$ за время существования проекта,
D_{Level}^{Closed} — количество устранённых за время существования проекта дефектов уровня важности $Level$,
D_{Level}^{Found} — количество обнаруженных за время существования проекта дефектов уровня важности $Level$.

Минимальные границы значений:

		Важность дефекта			
		Низкая	Средняя	Высокая	Критическая
Фаза проекта	Начальная	10%	40%	50%	80%
	Основная	15%	50%	75%	90%
	Финальная	20%	60%	100%	100%

3) ТЕКУЩЕЕ УСТРАНЕНИЕ ДЕФЕКТОВ:

$D_{Level}^{FCP} = \frac{D_{Level}^{Closed}}{D_{Level}^{Found}} \cdot 100\%$, где

D_{Level}^{FCP} — процентный показатель устранения в текущем билде дефектов уровня важности $Level$, обнаруженных в предыдущем билде,
D_{Level}^{Closed} — количество устранённых в текущем билде дефектов уровня важности $Level$,
D_{Level}^{Found} — количество обнаруженных в предыдущем билде дефектов уровня важности $Level$.
Минимальные границы значений:

		Важность дефекта			
		Низкая	Средняя	Высокая	Критическая
Фаза проекта	Начальная	60%	60%	60%	60%
	Основная	65%	70%	85%	90%
	Финальная	70%	80%	95%	100%

4) **СТОП-ФАКТОР:**

$$S = \begin{cases} Yes, T^E \geq 25\% \;\&\&\; T^{SP} < 50\% \\ No, T^E < 25\% \;||\; T^{SP} \geq 50\% \end{cases}, \text{ где}$$

S — решение о приостановке тестирования,
T^E — текущее значение метрики T^E,
T^{SP} — текущее значение метрики T^{SP}.

T^{SP} — процентный показатель успешного прохождения тест-кейсов,

5) **ВЫПОЛНЕНИЕ ТЕСТ-КЕЙСОВ:**

$$T^E = \frac{T^{Executed}}{T^{Planned}} \cdot 100\%, \text{ где}$$

T^E — процентный показатель выполнения тест-кейсов,
$T^{Executed}$ — количество выполненных тест-кейсов,
$T^{Planned}$ — количество тест-кейсов, запланированных к выполнению.
Уровни (границы):

- Минимальный уровень: 80 %.
- Желаемый уровень: 95–100 %.

6) **ПОКРЫТИЕ ТРЕБОВАНИЙ ТЕСТ-КЕЙСАМИ:**

$$R^C = \frac{R^{Covered}}{R^{Total}} \cdot 100\%, \text{ где}$$

R^C — процентный показатель покрытия требования тест-кейсами,
$R^{Covered}$ — количество покрытых тест-кейсами требований,
R^{Total} — общее количество требований.
Минимальные границы значений:

- Начальная фаза проекта: 40 %.
- Основная фаза проекта: 60 %.
- Финальная фаза проекта: 80 % (рекомендуется 90 % и более).

И, наконец, стоит упомянуть про так называемые «метрики покрытия», т.к. они очень часто упоминаются в различной литературе.

Покрытие (coverage) — процентное выражение степени, в которой исследуемый элемент (coverage item) затронут соответствующим набором тест-кейсов

Самыми простыми представителями **метрик покрытия можно считать**:

1. ***Метрику покрытия требований*** (требование считается «покрытым», если на него ссылается хотя бы один тест-кейс):

$$R^{SimpleCoverage} = \frac{R^{Covered}}{R^{Total}} \cdot 100\%, \text{ где}$$

$R^{SimpleCoverage}$ — метрика покрытия требований,
$R^{Covered}$ — количество требований, покрытых хотя бы одним тест-кейсом,
R^{Total} — общее количество требований.

2. ***Метрику плотности покрытия требований*** (учитывается, сколько тест-кейсов ссылается на несколько требований):

$$R^{DensityCoverage} = \frac{\sum T_i}{T^{Total}.R^{Total}} \cdot 100\%, \text{ где}$$

$R^{DensityCoverage}$ — плотность покрытия требований,
T_i — количество тест-кейсов, покрывающих *i*-е требование,
T^{Total} — общее количество тест-кейсов,
R^{Total} — общее количество требований.

3. ***Метрику покрытия классов эквивалентности*** (анализируется, сколько классов эквивалентности затронуто тест-кейсами):

$E^{Coverage} = \frac{E^{Covered}}{E^{Total}} \cdot 100\%$, где

$E^{Coverage}$ — метрика покрытия классов эквивалентности,

$E^{Covered}$ — количество классов эквивалентности, покрытых хотя бы одним тест-кейсом,

E^{Total} — общее количество классов эквивалентности.

4. ***Метрику покрытия граничных условий*** (анализируется, сколько значений из группы граничных условий затронуто тест-кейсами):

$B^{Coverage} = \frac{B^{Covered}}{B^{Total}} \cdot 100\%$, где

$B^{Coverage}$ — метрика покрытия граничных условий,

$B^{Covered}$ — количество граничных условий, покрытых хотя бы одним тест-кейсом,

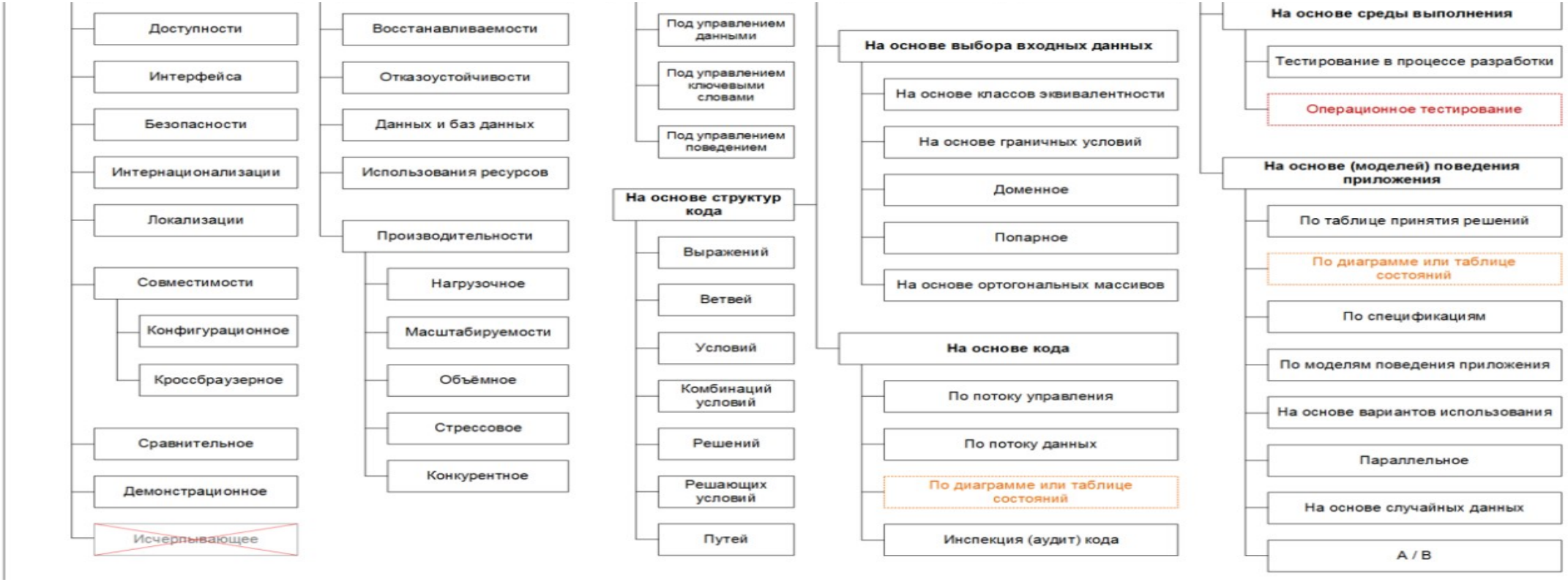
B^{Total} — общее количество граничных условий.

5. ***Метрики покрытия кода модульными тест-кейсами***. Таких метрик очень много, но вся их суть сводится к выявлению некоей характеристики кода (количество строк, ветвей, путей, условий и т.д.) и определению, какой процент представителей этой характеристики покрыт тест-кейсами

Лекция 12

Тема : Подробная классификация тестирования. Есть тестирование , которое изложено только в теории, а есть, которое применяется на практике


Классификация тестирования
По запуску кода на исполнение
Статическое тестирование
Динамическое тестирование
По уровню детализации приложения (по уровню тестирования)
Модульное тестирование
Интеграционное тестирование
Системное тестирование
По фокусировке на уровне архитектуры приложения
Уровень представления
Уровень бизнес-логики
Уровень данных
По доступу к коду и архитектуре приложения
Метод белого ящика
Метод чёрного ящика
Метод серого ящика
По (убыванию) степени важности тестируемых функций (по уровню функционального тестирования)
«Дымовое» («смоук»)
Критического пути
Расширенное
По привлечению конечных пользователей
Альфа-тестирование
Бета-тестирование
Гамма-тестирование
По степени автоматизации
Ручное
Автоматизированное (и «автоматическое»)
По природе приложения
Веб
Мобильное
Настольное
...
По степени формализации
На основе тест-кейсов
Исследова-тельское
Свободное (интуитивное)


Доступности
Интерфейса
Безопасности
Интернационализации
Локализации
Совместимости
Конфигурационное
Кроссбраузерное
Сравнительное
Демонстрационное
Исчерпывающее
Восстанавливаемости
Отказоустойчивости
Данных и баз данных
Использования ресурсов
Производительности
Нагрузочное
Масштабируемости
Объёмное
Стрессовое
Конкурентное
Под управлением данными
Под управлением ключевыми словами
Под управлением поведением
На основе структур кода
Выражений
Ветвей
Условий
Комбинаций условий
Решений
Решающих условий
Путей
На основе выбора входных данных
На основе классов эквивалентности
На основе граничных условий
Доменное
Попарное
На основе ортогональных массивов
На основе кода
По потоку управления
По потоку данных
По диаграмме или таблице состояний
Инспекция (аудит) кода
На основе среды выполнения
Тестирование в процессе разработки
Операционное тестирование
На основе (моделей) поведения приложения
По таблице принятия решений
По диаграмме или таблице состояний
По спецификациям
По моделям поведения приложения
На основе вариантов использования
Параллельное
На основе случайных данных
A / B

Лекция 13-14

Тема: Основные принципы тестирования. Тестирование показывает наличие дефектов. Исчерпывающее тестирование невозможно. Раннее тестирование. Кластеризация дефектов. Парадокс пестицида. Тестирование зависит от контекста. Заблуждение об отсутствии ошибок

Если вы решили быт тестировщиком, вы должны понимать, что есть:

1. -***Принципы тестирования*** (это то что чаще всего спрашивают на собеседованиях и просят объяснить. Несет в себе более широкий смыл тестирования, является более теоретической основой , лежащей в основе любого тестирования, которое будет далее выбираться)
2. ***Уровни тестирования*** (что мы тестируем: модуль, все ПО или связи)
3. ***Типы тестирования (***как мы будем проводить тестирования, в ручную с помощью ПО или доп. Софта, с известными данными или как пользователь***)***
4. ***Виды тестирования (***конкретные инструменты, которые имеют правила и последовательности***)***

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.

Основные принципы тестирования ПО:

1. Тестирование показывает наличие дефектов Тестирование может показать, что дефекты присутствуют, но не может доказать, что дефектов больше нет. Сколько бы

успешных тестов вы не провели, вы не можете утверждать, что нет таких тестов, которые не нашли бы ошибку.

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.

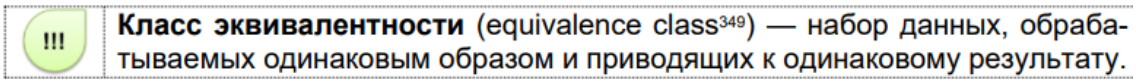
Таким образом, при должном тестовом покрытии, тестирование позволяет снизить вероятность наличия дефектов в программном обеспечении. В то же время, даже если дефекты не были найдены в процессе тестирования, нельзя утверждать, что их нет.

2. Исчерпывающее тестирование невозможно Исчерпывающее тестирование в теории призвано проверить приложение «со всеми возможными комбинациями всех возможных входных данных во всех возможных условиях выполнения». Но— это невозможно физически.

Невозможно провести исчерпывающее тестирование, которое бы покрывало все комбинации пользовательского ввода и состояний системы, за исключениям совсем уж примитивных случаев.

Вместо этого необходимо использовать анализ рисков и расстановку приоритетов, что позволит более эффективно распределять усилия по обеспечению качества ПО.

«Классы эквивалентности и граничные условия») даже для одного простого поля для ввода имени пользователя может существовать порядка 2.4^32 позитивных проверок и бесконечное количество негативных проверок. Потому в силу законов физики нет ни малейшего шанса протестировать программный продукт полностью, «исчерпывающе».



Однако, из этого не следует, что тестирование как таковое не является эффективным. Вдумчивый анализ требований, учёт рисков, расстановка приоритетов, анализ предметной области, моделирование, работа с конечными пользователями, применение специальных техник тестирования позволяют выявить те области или условия эксплуатации продукта, которые требуют особенно тщательной проверки. И поскольку здесь объём работы несоизмеримо меньше — такое тестирование уже не просто возможно, но и выполняется на каждодневной основе.

3. Раннее тестирование должно начинаться как можно раньше в жизненном цикле разработки программного обеспечения, и его усилия должны быть сконцентрированы на определенных целях.

Этот принцип связан с понятием «цена дефекта»/ Цена дефекта существенно растет на протяжении жизненного цикла разработки ПО. Чем раньше обнаружен дефект, тем быстрее, проще и дешевле его исправить. Дефект, найденный в требованиях, обходится дешевле всего.

Еще одно важное преимущество раннего тестирования - экономия времени. Тестовые активности могут начинаться еще до того, как написана первая строчка кода. По мере того, как готовятся требования и спецификации, тестировщики могут приступать к разработке и ревью тест-кейсов. И когда появится первая тестовая версия, можно будет сразу приступать к выполнению тестов.

чрезмерно раннее тестирование тоже плохо и даже может привести к необходимости повторно выполнять большой объём работы, но начатое вовремя (без промедления) тестирование даёт наибольший эффект.

Пример: Представьте, что вы собираетесь в поездку и продумываете список вещей, которые необходимо взять с собой.

На стадии обдумывания добавить, изменить, удалить любой пункт в этом списке не стоит ничего.

На стадии поездки по магазинам для закупки необходимого недоработки в списке уже могут привести к необходимости повторной поездки в магазин.

На стадии отправки на место назначения недоработки в списке вещей явно приведут к ощутимой потере нервов, времени и денег.

А если фатальный недостаток списка вещей выяснится только по прибытии, может так оказаться, что вся поездка потеряла смысл.

4. **кластеризация 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*, дефекты «кучкуются». Это может происходить потому, что определенная область кода особенно сложна и запутана, или потому, что внесение изменений производит «эффект домино» и просто по тому, что в разработке учувствовал не грамотный специалист.

Принцип Парето используется для оценки рисков при планировании тестов - тестировщики фокусируются на известных «проблемных зонах», после чего ищут первопричины и проводят анализ, чтобы предотвратить повторное появление дефектов, обнаружить причины возникновения скоплений дефектов и спрогнозировать потенциальные скопления дефектов в будущем.

Если проблемный кластер выявлен-это позволит минимизировать усилия и повысить качество продукта при разработке.

Дефекты не возникают «просто так». И уже тем более «просто так» не появляется много дефектов в какой-то «проблемной» области приложения.

Возможно используется какая-то новая или сложная технология или приложению приходится работать в неблагоприятных условиях или взаимодействовать с внешними ненадёжными компонентами.

Или так получилось, что соответствующая часть требований не была проработана должным образом. Или вовсе за реализацию данной части приложения отвечали недостаточно ответственные или недостаточно компетентные люди.

В любом случае «группировка» дефектов по какому-то явному признаку покажет, что именно здесь будет обнаружено ещё больше дефектов.

если проблемный «кластер» выявлен — это позволяет ощутимо минимизировать усилия и повысить качество приложения.

5. Парадокс пестицида чем больше вы тестируете ПО, тем более невосприимчивым оно становится к имеющимся тестам, т.е.

1. каждый метод и набор тестов, который используется для предотвращения или поиска ошибок, может оставлять часть не найденных ошибок, против которых эти методы и тесты неэффективны;
2. имеющиеся тесты устаревают после исправления дефекта и не могут обнаружить новые;

Из чего следует, что набор тестов, тестовых данных и подходов нужно постоянно пересматривать и улучшать для выявления не найденных ошибок, а также необходимо обновлять тесты и тестовые данные после исправления уже найденных дефектов.

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 вносить изменения в используемые наборы тестов, корректировать их, 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.

Этот принцип тесно связан с понятием риска. Что такое риск?

Риск - это потенциальная проблема.

У риска есть вероятность- она всегда выше 0 и ниже 100% - и есть влияние - те негативные последствия, которых мы опасаемся.

Анализируя риски, мы всегда взвешиваем эти два аспекта: вероятность и влияние. Разные системы связаны с различными уровнями риска, влияние того или иного дефекта также сильно варьируется. Одни проблемы довольно тривиальны, другие могут дорого обойтись и привести к большим потерям денег, времени, деловой репутации, а в некоторых случаях даже привести к травмам и смерти.

Уровень риска влияет на выбор методологий, техник и типов тестирования.

7. Заблуждение об отсутствии ошибок. Тот факт, что тестирование не обнаружило дефектов, еще не значит, что программа готова к релизу. Нахождение и исправление дефектов будут не важны, 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.

Заказчики ПО - на самом деле совершенно не интересуются дефектами и их количеством, кроме тех случаев, когда они непосредственно сталкиваются с нестабильностью продукта.

Им также неинтересно, насколько ПО соответствует формальным требованиям, которые были задокументированы. Пользователи ПО более заинтересованы в том, чтобы оно помогало им эффективно выполнять задачи. ПО должно отвечать их потребностям, и именно с этой точки зрения они его оценивают.

Даже если вы выполнили все тесты и ошибок не обнаружили, это еще не гарантия того, что ПО будет соответствовать нуждам и ожиданиям пользователей.

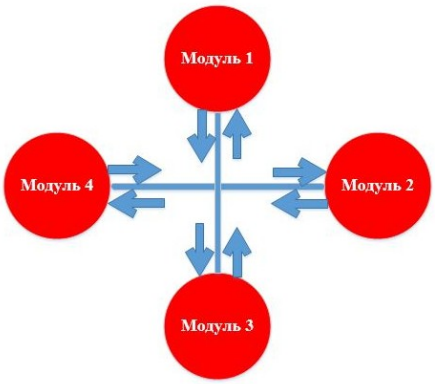
Иначе говоря, верификация не равна валидации.

Лекция 15

Тема: Уровни тестирования. Модульное тестирование. Интеграционное тестирование. Системное тестирование. Приемочное тестирование.

Уровни тестирования: модульное, интеграционное, системное, приемочное, операционное

1. **МОДУЛЬНОЕ ТЕСТИРОВАНИЕ** – это процесс исследования ПО, при котором тестируется минимально возможный компонент, например отдельный класс или функция. Часто модульное тестирование осуществляется разработчиками ПО.



Объект тестирования выделен красным цветом.

проверяются отдельные небольшие части приложения.

В общем это написание юнит-тестов, для конкретных модулей или функций. Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода.

Идея состоит в том, чтобы изолировать отдельные части программы и показать, что по отдельности эти части работоспособны

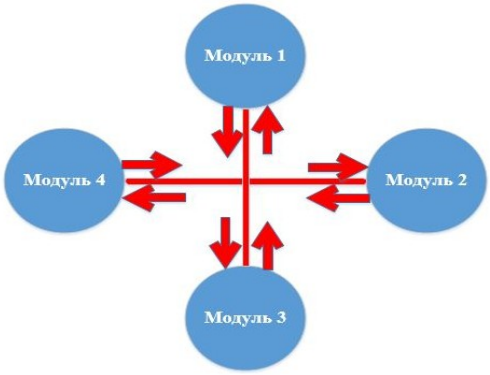
Драйвер – определенный модуль теста, который выполняют тестируемый нами элемент.

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).

2. **ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ** – это процесс исследования ПО, при котором тестируются интерфейсы между компонентами или подсистемами.



Объект тестирования выделен красным цветом.

Проводится после модульного тестирования.

проверяется взаимодействие между несколькими частями приложения.

Основная функция или цель-это протестировать интерфейсы связывающие основные модули и проверить общую структуру их работы.

Сначала каждый модуль тестируется изолированно друг от друга, после чего последовательно объединяются логикой.

Есть два вида тестирования нисходящее и восходящее тестирование.

Существует несколько подходов к интеграционному тестированию:

• **Снизу вверх**. Сначала собираются и тестируются модули самих нижних уровней, а затем по возрастанию к вершине иерархии. Данный подход требует готовности всех собираемых модулей на всех уровнях системы.

• **Сверху вниз**. Данный подход предусматривает движение с высокоуровневых модулей, а затем направляется вниз. При этом используются заглушки для тех модулей, которые находятся ниже по уровню, но включение которых в тест еще не произошло.

Заглушка - это имитация вызываемой функции, возвращающая те же данные, но ничего больше не делающая.

• **Большой взрыв**. Все модули всех уровней собираются воедино, а затем тестируется. Данный метод экономит время, но требует тщательной проработки тест кейсов.

тестируется один раз и в целом. Есть недостаток, есть шанс не найти основную проблему ошибки системы, за то есть экономия трудозатрат.

При автоматизации тестирования используется Система непрерывной интеграции. Принцип ее действия заключается в следующем:

1) Система непрерывной интеграции производит мониторинг системы контроля версий.

2) При изменении исходных кодов в репозитории производится обновление локального хранилища.

3) Выполняются необходимые проверки и модульные тесты.

4) Исходные коды компилируются в готовые выполняемые модули.

5) Выполняются тесты интеграционного уровня.

6) Генерируется отчет о тестировании.

Это позволяет тестировать систему сразу после внесения изменений, что существенно сокращает время обнаружения и исправления ошибок.

3. СИСТЕМНОЕ ТЕСТИРОВАНИЕ – это процесс исследования ПО, при котором тестируется интегрированная система на её соответствие требованиям заказчика. Альфа- и бета-тестирование относятся к подкатегориям системного тестирования.

приложение проверяется как единое целое

Основной задачей системного тестирования является **проверка как функциональных, так и не функциональных требований в системе в целом**.

Функциональные требования определяют, что система должна делать; нефункциональные требования определяют, какой система должна быть.

Выполняя системное тестирование, можно обнаружить следующие типы дефектов:

1. Неправильное использование системных ресурсов.
2. Непредусмотренные комбинации пользовательских данных.
3. Проблемы с совместимостью окружения.
4. Непредусмотренные сценарии использования.
5. Несоответствие с функциональными требованиями.
6. Плохое удобство использования.

Системное тестирование выполняется методом «Черного ящика», т.к. проверяемое множество является «внешними» сущностями, которые не требуют взаимодействия с внутренним устройством программы. Также выполнять его рекомендуется в окружении, максимально приближенном к окружению конечного пользователя.

Можно выделить 2 подхода к системному тестированию:

• На базе требований. Тестирование проводится в соответствии с функциональными или нефункциональными требованиями, для каждого из которых пишется testcase (тестовые прецеденты).

• На базе случаев использования. Тестирование происходит в соответствии с вариантами использования продукта, на основе которых создаются usercases (пользовательские прецеденты). Для каждого из данных пользовательских прецедентов создаются свои тестовые прецеденты.

4. ПРИЕМОЧНОЕ (acceptance) – вид тестирования, проводимый на этапе сдачи готового продукта (или готовой части продукта) заказчику. Целью приемочного тестирования является определение готовности продукта, что достигается путем прохода тестовых сценариев и случаев, которые построены на основе спецификации требований к разрабатываемому ПО.

Результатом приемочного тестирования может стать:

• Отправка проекта на доработку.

• Принятие его заказчиком, в качестве выполненной задачи.

Это финальный этап тестирования продукта перед его релизом. При этом, он не является сверх тщательным, всеохватывающим и полным – тестируется, в основном, только основной функционал.

Приемочное тестирование проводиться либо самим заказчиком, либо группой тестировщиков, представляющих интересы заказчика, либо тестировщиками компании-разработчика. Зависит от предпочтений компании-заказчика.

5. *ОПЕРАЦИОННОЕ ТЕСТИРОВАНИЕ*- Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес-модели системы.

- Входные требования — Бизнес модель
- Объект тестирования — Разработанная система

Тестируются все бизнес процессы затребованные заказчиком.

Лекция 16-17

Тема: Типы тестирования**.** Тестирование по доступу к коду и архитектуре. Метод черного ящика. Парадигмы тестирования.

White/Black/Grey Box-тестирование

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 Мы не знаем, как устроена тестируемая система.

у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования

тестировщик работают с данным типом тестирования так, как будто они обычные пользователи и работают с реальной системой в идентичной среде для пользователя.

тестирование черного ящика это стратегия, в которой тестирование основано исключительно на требованиях и спецификациях, при этом мы не знаем, как устроена внутри тестируемая система и работаем исключительно с внешними интерфейсами тестируемой системы или компонента. Тестирование черного ящика может быть применено на всех уровнях - модульном, интеграционном, системном и приемочном. Мы тестируем систему, как пользователи.

Тестировщики оказывают на приложение воздействия (и проверяет реакцию) тем же способом, каким при реальной эксплуатации приложения на него воздействовали бы пользователи или другие приложения.

В рамках тестирования по методу чёрного ящика ***основной информацией для создания тест-кейсов выступает документация*** (особенно — требования и общий здравый смысл (для случаев, когда поведение приложения в некоторой ситуации не регламентировано явно; иногда это называют «тестированием на основе неявных требований

тестирование черного ящика можно разделить на две составляющие– это:

1) **тестирование, как функциональное, так и нефункциональное**, не предполагающее знания внутреннего устройства компонента или системы.

Функциональное тестирование- это тип касается функциональных требований или спецификаций приложения. Здесь различные действия или функции системы *тестируются путем предоставления входных данных и сравнения фактического выхода с ожидаемым выходом*.

Например, когда мы тестируем раскрывающийся список, мы нажимаем на него и проверяем, что он раскрывается и все ожидаемые значения отображаются

основных типов функционального тестирования:

- Дымовое тестирование;(проверка на стабильность и наличие явных ошибок)
- Тестирование на вменяемость;
- Интеграционное тестирование; (модули объединяются группами и тестируются все вместе)
- Тестирование системы;
- Регрессионное тестирование; (тестирование основывается на проверке изменений внесенных в систему, с шагом назад)
- Приемочное тестирование пользователей

Не функциональное тестирование- проверяет не функциональные аспекты ПО. Стремиться улучшить качество и производительность приложения.

типов нефункционального тестирования включают:

- Юзабилити-тестирование;
- Нагрузочное тестирование;
- Тестирование производительности;
- Тестирование совместимости;
- Стрессовое тестирование;
- Тестирование масштабируемости;

2) **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.

Техники тест-дизайна, включают:

- классы эквивалентности;
- анализ граничных значений;
- таблицы решений;

- диаграммы изменения состояния;
- тестирование всех пар.

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:

1. неправильно реализованные или недостающие функции;
2. ошибки интерфейса;
3. ошибки в структурах данных или организации доступа к внешним базам данных;
4. ошибки поведения или недостаточная производительность 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.

Тестирование черного ящика может быть как функциональным, так и нефункциональным.

Функциональное тестирование предполагает *проверку работы функций системы*, а нефункциональное – *соответственно, общие характеристики нашей программы*.

Преимущества:

1. *тестирование производится с позиции конечного пользователя* и может помочь обнаружить неточности и противоречия в спецификации;
2. Тестирование можно начинать после завершения разработки проекта.
3. И тестировщики, и разработчики работают независимо, не мешая друг другу;
4. эффективно для больших и сложных приложений;
5. Дефекты и несоответствия можно выявить на ранней стадии тестирования;
6. тестировщику нет необходимости знать языки программирования и углубляться в особенности реализации программы;
7. тестирование может производиться специалистами, независимыми от отдела разработки, что помогает избежать предвзятого отношения;
8. можно начинать писать тест-кейсы, как только готова спецификация.

Недостатки:

1. –тестируется только очень ограниченное количество путей выполнения программы;

2. –без четкой спецификации (а это скорее реальность на многих проектах) достаточно трудно составить эффективные тест-кейсы;
3. некоторые тесты могут оказаться избыточными, если они уже были проведены разработчиком на уровне модульного тестирования;

Парадигмы тестирования методом черного ящика- Парадигма создает основное направление мышления - она дает понимание и направление для дальнейших исследований или работы.

Парадигма тестирования будет определять типы тестов, которые (для человека, работающего в рамках этой парадигмы) актуальны и интересны.

Используются следующие техники тестирования

ДОМЕННОЕ ТЕСТИРОВАНИЕ

- Ключевые идеи:
  - «Попробуйте диапазоны и варианты»;
  - «Разделите мир на классы»;
- Основной вопрос или цель:
  - Стратегия стратифицированной выборки. Разделите большое пространство возможных тестов на подмножества. Выберите лучших представителей из каждого набора;
- Примеры кейсов:
  - Анализ эквивалентности простого числового поля;
  - Тестирование совместимости принтеров;
- Сильные стороны:
  - Нахождение ошибок с наибольшей вероятностью с помощью относительно небольшого набора тестов;
  - Интуитивно понятный подход, хорошо обобщает;
- Слепые зоны:
  - Ошибки, выходящие за рамки границ, или в очевидных особых случаях;
  - Кроме того, фактические домены часто остаются неизвестными;

СТРЕССОВОЕ ТЕСТИРОВАНИЕ

- Ключевые идеи:
  - «Сокруши продукт»;
  - «Проведи его через отказы;
- Основной вопрос или цель:
  - Узнать о возможностях и слабых сторонах продукта, проведя его через отказ и за его пределами. Что сбои в экстремальных случаях говорят нам об изменениях, необходимых в работе программы в нормальных случаях?
- Примеры кейсов:
  - Большие объемы данных, подключения устройств, длинные цепочки транзакций;
  - Условия нехватки памяти, сбои устройств, вирусы и другие проблемы;
- Сильные стороны:
  - Выявление слабых мест, в т.ч. дыр в безопасности;
- Слепые зоны:

- Слабости, которые не становятся более заметными из-за стресса;

ПО СПЕЦИФИКАЦИЯМ ТЕСТИРОВАНИЕ

- Ключевые идеи:
  - «Проверяйте каждое требование»;
- Основной вопрос или цель:
  - Проверяйте соответствие (conformance) продукта каждому заявлению в каждой спецификации, документе с требованиями и т. д.;
- Примеры кейсов:
  - Матрица прослеживаемости, отслеживает тестовые случаи, связанные с каждым элементом спецификации;
- Сильные стороны:
  - Критическая защита от гарантийных претензий, обвинений в мошенничестве, потери доверия со стороны клиентов;
- Слепые зоны:
  - Любые проблемы, не указанные в спецификациях или плохо решенные в спецификациях;

НА ОСНОВЕ СЛУЧАЙНЫХ ДАННЫХ ТЕСТИРОВАНИЕ

- Ключевые идеи:
  - «Объемное тестирование с новыми кейсами»;
- Основной вопрос или цель:
  - Пусть компьютер создает, выполняет и оценивает огромное количество тестов;
- Примеры кейсов:
  - Валидация функции или подсистемы (например, тестирование эквивалентности функций) на основе оракулов (Oracle-driven);
  - Стохастическое (переход между состояниями) тестирование для выявления конкретных сбоев (ассерты, утечки и т. д.);
  - Оценка статистической надежности;
  - Частичный или эвристический оракул, чтобы найти некоторые типы ошибок без общей проверки;
- Сильные стороны:
  - Регрессия не зависит каждый раз от одного и того же старого теста;
  - Частичные оракулы могут быстро и дешево находить ошибки в молодом коде;
  - Меньше вероятность пропустить невидимые извне внутренние оптимизации;
  - Может обнаруживать сбои, возникающие из-за длинных сложных цепочек, которые было бы трудно создать в соответствии с запланированными испытаниями;
- Слепые зоны:
  - Нужно уметь отличать pass от failure. Слишком много людей думают: «Not crash = not fail»;
  - Кроме того, эти методы часто охватывают многие типы рисков, но затемняют необходимость в других тестах, которые не поддаются автоматизации;

ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

- Ключевые идеи:
  - «Модульное тестирование черного ящика»;
- Основной вопрос или цель:
  - Тщательно проверяйте каждую функцию по очереди;
- Примеры кейсов:
  - Таблица, тестируйте каждый элемент по отдельности;
  - База данных, тестируйте каждый отчет по отдельности;

- Сильные стороны:
  - Тщательный анализ каждого протестированного элемента;
- Слепые зоны:
  - Упускает взаимодействия, пропускает исследование преимуществ предлагаемые программой;

РЕГРЕССИОННОЕ ТЕСТИРОВАНИЕ

- Ключевые идеи:
  - «Повторить тестирование после изменений»;
- Основной вопрос или цель:
  - Управляйте рисками, связанными с тем, что (а) исправление ошибки не устраняет ошибку или (б) исправление (или другое изменение) имело побочный эффект (side effect);
- Примеры кейсов:
  - Регрессия ошибок, регрессия старых исправлений, общая функциональная регрессия;
  - Наборы автоматизированной регрессии графического интерфейса;
- Сильные стороны:
  - Обнадеживает, укрепляет доверие, удобен для регуляторов;
- Слепые зоны:
  - Все, что не вошло в регрессионную серию. Кроме того, поддержание этого стандартного списка может быть очень дорогостоящим;

НА ОСНОВЕ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ (USE-CASE) ТЕСТИРОВАНИЕ

- Ключевые идеи:
  - «Делай что-нибудь полезное и интересное»;
  - «Делайте одно за другим»;
- Основной вопрос или цель:
  - Сложные случаи, отражающие реальное использование;
- Примеры кейсов:
  - Оценивайте продукт на предмет соответствия бизнес-правилам, данным о клиентах и продукции конкурентов;
  - Тестирование жизненного цикла
  - Варианты использования (Use cases) - это более простая форма, часто основанная на возможностях продукта и пользовательской модели, а не на естественном наблюдении за системами такого типа;
- Сильные стороны:
  - Сложные, реалистичные события. Может помогать справляться в ситуациях, которые слишком сложны для моделирования;
  - Выявляет сбои, которые происходят (развиваются) с течением времени;
- Слепые зоны:
  - Отказ одной функции может сделать этот тест неэффективным;
  - Необходимо хорошо подумать, чтобы добиться хорошего покрытия;

ИССЛЕДОВАТЕЛЬСКОЕ ТЕСТИРОВАНИЕ

- Ключевые идеи:
  - «Интерактивное, одновременное исследование, разработка тестов и тестирование»;
- Основной вопрос или цель:
  - ПО поступает тестировщику без документации. Тестировщик должен одновременно узнавать о продукте и о тестовых примерах / стратегиях, которые позволят выявить продукт и его дефекты;
- Примеры кейсов:
  - Полное тестирование test-it-today;
  - Сторонние компоненты;

- Горилла-тестинг;

- Сильные стороны:
  - Продуманная стратегия получения результата в неизвестности;
  - Стратегия выявления несоответствия ожиданиям клиентов;
- Слепые зоны:
  - Чем меньше мы знаем, тем больше рискуем упустить.

Лекция 18

Тема: Типы тестирования**.** Тестирование по доступу к коду и архитектуре. Метод белого ящика.

Противоположностью техники черного ящика является тестирование методом белого ящика, речь о котором пойдет ниже.

WHITE BOX Нам известны все детали реализации тестируемой программы.

тестирование 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.

у тестировщика есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного.

Выделяют глобальную технику — тестирование на основе тест-дизайна

тестирование белого ящика можно разделить на две составляющие– это:

1. –тестирование, *основанное на анализе внутренней структуры* компонента или системы.
2. –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, который, является программистом, изучает реализацию кода поля ввода на веб-странице, определяет все предусмотренные 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.

Техника белого ящика применима *на разных уровнях тестирования – от модульного до системного*, но главным образом применяется именно для реализации модульного тестирования компонента его автором.

Процесс тестирования методом белого ящика:

1. Анализируется реализация программы;
2. В программе определяются возможные маршруты;
3. Выбираются такие входные данные, чтобы программа выполнила выбранные пути. Это называется сенсибилизацией путей. Заранее определяются ожидаемые результаты для входных данных;
4. Тесты выполняются;
5. Результаты анализируются;

Покрытие тестами требований в коде:

Покрытие— процентное выражение степени, в которой исследуемый элемент затронут соответствующим набором тест-кейсов

1. ***Покрытие сегмента кода***

(каждый оператор выполняется 1 раз)

2. ***Покрытие ветвей***

покрытие каждой ветки кода из всех возможных было выполнено;

3. ***Покрытие множественных условий***

Для нескольких условий проверяется каждое условие с несколькими путями и комбинацией разных путей для достижения этого условия;

4. Покрытие базового пути:

каждый независимый путь в коде взят на тестирование;

5. ***Покрытие потоков***

Диаграмма потоков имеет тенденцию отражать зависимости, но в основном это происходит через последовательности манипуляций с данными.

Короче говоря, каждая переменная данных отслеживается, и ее использование проверяется. Этот подход имеет тенденцию обнаруживать ошибки, такие как переменные, которые используются, но не инициализируются, или объявлены, но не используются, и т.д. (компиляторы/линтеры/IDE уже вполне способны на это сами);

6. ***Покрытие критического пути***:

это определение и покрытие всех возможных путей прохождения через код;

7. *Покрытие циклов*:

эти стратегии относятся к тестированию одиночных циклов, составных циклов и вложенных циклов;

Преимущества:

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

Недостатки:

1. *Тестировщик должен обладать навыками программирования* для того, чтобы понять и оценить тестируемое программное обеспечение
2. поток управления правильный. Поскольку эти тесты основаны на существующих путях, с их помощью нельзя обнаружить несуществующие пути;

когда мы составляем тест кейсы по диаграмме потоков мы смотрим на картинку и в точности отражаем тест, но может случится, что диаграмма потоков не все отобразила.

Часто программисты условно отображают часть функционала и основных потоков

3. при использовании автоматизации тестирования на этом уровне, *поддержка тестовых скриптов может оказаться достаточно накладной*, если программа часто изменяется.
4. *Количество выполняемых путей* может быть настолько большим, что не удастся проверить их все.

Как правило, попытка протестировать все пути выполнения с помощью тестирования белого ящика так же невозможна, как и тестирование всех комбинаций всех входных данных при тестировании черного ящика;

Лекция 19

Тема: Типы тестирования. Тестирование по доступу к коду и архитектуре. Метод серого ящика. Сравнительный анализ техник тестирования.

Сравнение Black Box и White Box

	белый ящик	**черный ящик**
Определение	дизайн и реализация программы известны тестировщику.	дизайн и реализация программы не известны тестировщику.
Кем выполняется	Это делается разработчиками программного обеспечения.	Это сделано профессиональной командой тестирования.
Знание кодирования	Это требует знания внутреннего кодирования.	Это не требует знания внутреннего кодирования.
Что затрагивает	Это связано с тестированием реализации программы.	Это не касается структуры программы.
Тестирование	Это в основном применимо для более низкого уровня тестирования, таких как: ▪ Модульное тестирование ▪ Интеграционное тестирование	Это в основном применимо к более высокому уровню тестирования, такому как: ▪ Приемочное тестирование ▪ Тестирование системы
Знания	Внедрение знаний требуется для тестирования.	Внедрение знаний не требуется для тестирования.
Тестовая основа	Тестовые случаи основаны на детальном проектировании.	Контрольные примеры основаны на требуемых спецификациях.
Время	Это трудоемкий и исчерпывающий.	Это менее трудоемкий и исчерпывающий.
Тест алгоритма	Подходит для тестирования алгоритмов.	Он не подходит для тестирования алгоритмов.
Метод тестирования	Область данных и внутренние границы лучше проверены.	Это может быть сделано только методом проб и ошибок.

GREY BOX Нам известны только некоторые особенности реализации тестируемой системы.

Тестирование «серого ящика» - хороший способ поиска уязвимостей в программах. Это может помочь в обнаружении ошибок или эксплойтов(дырки в системе) из-за неправильной структуры кода или неправильного использования приложений.

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.

например, доступ к внутренней структуре и алгоритмам работы ПО для написания максимально эффективных тест-кейсов предоставлен, 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.

Цель тестирования: является:

1. Улучшение качества продукта, для которого используется функциональное и не функциональное тестирование
2. Тестирует с точки зрения пользователя, а не сточки зрения дизайнера

Как проводится тестирование серого ящика Шаги:

Здесь тестовые примеры разработаны с учетом архитектуры приложения и понимания его поведения в различных ситуациях. Ниже приводится краткое описание шагов, которые необходимо выполнить:

1. Соберите входные данные из подходов к тестированию «белого ящика» и «черного ящика».
2. Определите выходы входов на шаге 1.
3. Определите все пути пользователя, которые преобразуют эти входные данные в выходные.
4. Определите подфункции для прохождения тестирования на один уровень глубже.
5. Определите входы для подфункций.
6. Определите ожидаемый результат для входов на шаге 5.
7. Выполните тестовые примеры для подфункций.
8. Проверить правильность результата выполнения шага 7.
9. Повторите шаги 4-8 для каждой подфункции.

Плюсы тестирования серого ящика

1. *Тестирование проводится как с точки зрения пользователя, так и с точки зрения разработчика*, поскольку оно сочетает в себе тестирование черного и белого ящиков.

2. Поскольку он является производным от методов «черного ящика» и «белого ящика», он *добавляет больше преимуществ в обоих методах тестирования*.

3. Тестирование проводится *в большей степени с точки зрения пользователя, чем с точки зрения разработчиков.*

4. Качество ПО улучшается.

5. Этот метод больше *ориентирован на восприятие пользователем*.

6. Разработчики получают выгоду, поскольку у них есть достаточно времени для исправления ошибок.

У разраба больше времени на разработку

7. *Мгновенные исправления* могут быть сделаны, так как доступен частичный код.

8. *Поток данных управляется* и поддерживается правильно.

9. Сделан честный обзор программного обеспечения, и *между разработчиками и тестировщиками не возникает конфликтов*

10. Этот метод тестирования *эффективен при интеграционном тестировании*.

11. *эффективно тестировать сложные приложения и сценарии.*

Недостатки тестирования серого ящика

1. Полное знание кода не достигается, поэтому *трудно достичь полного покрытия кода*.

2. Поскольку доступен только ограниченный доступ к коду / логике, *полные исправления иногда не могут быть сделаны*, а значит, иногда программное обеспечение может оставаться таким, как есть.

3. Другие типы тестирования белого ящика, такие как *тестирование алгоритма, не могут быть выполнены*, так как полная логика недоступна.

4. *Трудно выполнить этот тип тестирования в распределенных архитектурных* программных системах.

КРАТКИЕ ИТОГИ

1. Итак, тестирование методом серого ящика наиболее востребовано в ситуации, когда *QA-инженеры могут получить полноценный доступ к проектной документации и у них есть достаточно времени на подготовку тест-кейсов* со сценариями тестирования.
2. *Максимальная польза* от применения подобного вида тестирования наблюдается *во время проверки веб-приложений, веб-сервисов, графического интерфейса пользователя*

3. Эффективно при выполнении разнообразных *функциональных тестов* ПО,
4. Тесты *ориентированы на системную и клиентскую безопасность*.

Лекция 20

Тема: Типы тестирования. Классификация по запуску кода на исполнение. Статистическое тестирование

СТАТИЧЕСКОЕ

Статическое 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.

Можем проверять синтаксис кода (phpstan) с помощью стороннего ПО или смотреть глазами.

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации.

Метод статического тестирования – это тип тестирования ПО, где программное обеспечение проверяется без запуска кода; является процессом или инструментом, направленным на обнаружение возможных багов в ПО. Кроме этого, он находит и устраняет ошибки в разного рода сопроводительных документах, например, специфики требований к ПО.

Тестированию подвергаются:

1. ***Документы***

(требования, тест-кейсы, описания архитектуры приложения, схемы баз данных и т.д.).

2. ***Графические прототипы***

(например, эскизы пользовательского интерфейса).

3. ***Код приложения***

(что часто выполняется самими программистами (code review), взаимный просмотр кода).
Код приложения также можно проверять с использованием техник тестирования на основе структур кода

4. ***Параметры (настройки) среды исполнения приложения***
5. ***Подготовленные тестовые данные***

Статическое тестирование бывает двух типов:

1. **проверка требований**;
2. **статистический анализ**.

1) **проверка требований**– тестирование, направленное на обнаружение дефектов в документации (обнаружение дефектов в: требованиях, дизайнерском оформлении, тестовые случаи).

Проверки классифицируются на:

1. ***Инспектирование программного обеспечения***.

Подразумевает, в большинстве случаев, проверку документации высшими органами, к примеру, изучение требований к ПО;

2. ***Неформальные***.

Рассматривая созданный документ неофициально, он демонстрируется публике, где каждый высказывает свое мнение на счет него. Это помогает обнаружить недочеты на ранних стадиях создания продукта;

3. ***Оценка экспертов***.

Команда специалистов проверяет документацию, с целью выявить и устранить ошибки.

4. ***Сквозные просмотры***.

Выполняет опытный специалист (эксперт), проверяет наличие дефектов. Основная цель – предупредить возникновение проблем в процессе разработки или тестирования.

2)Статический анализ – созданный программистами код, который поддается анализу на наличие недоработок в структуре, способных привести к багам.

Анализируется и проверяется программный код, ищутся баги

В «состав» статического анализа входит оценка качества написанного разработчиками кода. Для анализа кодовой комбинации и сравнения его со стандартами соответствия, применяют различные инструменты.

С помощью статического анализа можно обнаружить следующие дефекты:

1. Мертвый код;
2. Переменные, которые не используются;
3. Неправильный синтаксис;
4. Переменные со значениями, которые невозможно определить;
5. Нескончаемые циклы.

Статический анализ имеет три составляющих:

1. ***Поток информации*** (данные, связанные с обработкой потока);
2. ***Контроль потока*** (степень выполнения требований);

3. ***Цикломатическая сложность*** (исчисление сложности приложения, связанной, в большей степени, с количеством независимых путей в колонке потоков управления приложения).

Статический анализ выполняется как вручную, так и при помощи специального оборудования.

Плюсы метода статического тестирования:

1. *Находит ошибки на первичных этапах* разработки ПО, что способствует снижению стоимости исправления обнаруженных дефектов;
2. *Отзывы, оставленные в процессе данного тестирования*, помогают усовершенствовать функциональную сторону процесса, что тоже направлено на избегание возникновения похожих багов;
3. Дает высокий уровень *информативности о проблемах качества* ПО;
4. Способствует улучшению *обмена важной информацией между сотрудниками*;
5. *Исправление багов требует минимальных усилий*, что помогает сделать разработку еще более продуктивной.

Минусы метода статического тестирования:

- Поскольку статическое тестирование, чаще всего, выполняется вручную, *процесс получается длительным;*
- *Мешает находить уязвимости*, находящиеся в среде выполнения.

Лекция 21

Тема: Типы тестирования. Классификация по запуску кода на исполнение. Динамическое тестирование. Сравнительный анализ.

динамическое тестирование – тип тестирования, который предполагает запуск программного кода и анализирует *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 – это методика, направленная на проверку и анализ функционала программы, во время выполнения кода.

То есть, данный тип тестирования подразумевает фактическую эксплуатацию определяет как работает ее функционал, соответствии ожиданиям или нет.

Основная идея данного тестирования проверить реальное поведение системы или ее части.

Как она себя ведет, отрабатывает основные и побочные сценарии или нет, где выдает ошибки.

проверяет возможные параметры для работы основных сценариев, т.е. при выводе ошибок система не падает, а требует определенных действий.

Динамический тип тестирования состоит:

1. ***из непосредственного тестирования программного обеспечения*** в реальное время, способом предоставления входной информации
2. ***исследования полученного результата*** поведения приложения, При этом, может выполняться проверка внешних параметров работы программы: загрузка процессора, использование памяти, время отклика

Пример: Регистрируя новую учетную запись и придумывая к ней пароль, нужно придерживаться определенных правил создания надежного кода.

1. пароль должен состоять с не менее, чем из 7 знаков
2. иметь как заглавные, так и прописные буквы

3. в его составе должна быть хоть одна цифра и т. д.

Это и есть параметры или условия, которых должен придерживаться пользователь при регистрации.

Если же он будет вписывать другие данные, программа должна их отклонить.

Во время тестирования данной функции необходимо:

1.ввести пароль по обозначенным параметрам

2.проверить результат.

Динамическое тестирование выполнения на протяжении всего ЖЦ ПО.

Данный метод тестирования помогает команде проверить разные критические моменты программного обеспечения.

Если закрыть глаза на существование ошибок и никак не отреагировать на них, это может определенным образом сказаться на производительности, функциональной стороне и надежности приложения.

Динамическое тестирование является частью процесса валидации программного обеспечения.

Валидация — это процесс оценки того, соответствует ли система потребностям и ожиданиям пользователя.

динамическое тестирование включать подвиды, каждый из которых зависит от:

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

Плюсы методики динамического тестирования:

1. В процессе тестирования проводится *тщательное изучение всего функционала* программы, в результате чего получаем *высокое качество проверки*;

Чем больше мы знаем о ПО тем меньше логических ошибок при разработке функционала будет

2. *хорошо структурированный процесс тестирования*, осуществляющий проверку программы со стороны пользователя, что, в свою очередь, значительно повышает качество программного обеспечения;

есть план тестирования со стороны пользователя-это позволит допустить меньше ошибок и будет выше качество ПО

3. *Фиксация сложных дефектов*, которые могли остаться незамеченными на этапе проверки кода;

Правило Парето, что если мы обнаружили сложные дефекты в некоторых модулях, то стоит проверить больше связанных с ним модулей, что уменьшит количество ошибок

4. *Динамический тип тестирования*, при помощи специальных инструментов, можно сделать автоматизированным.

Пишутся авто тесты и запускаются после каждого изменения

Минусы методики динамического тестирования:

1. достаточно сложный механизм тестирования, выполнение которого требует много времени;
2. дорогостоящий процесс;
3. В основном данный метод тестирования *выполняется по завершению кодирования*, и баги находятся уже в процессе реального жизненного цикла разработки.

Верификация —проверки того, соответствует ли система заданным спецификациям и требованиям.

Валидация —оценки того, соответствует ли система потребностям и ожиданиям пользователя.

Сравнительная характеристика статического и динамического тестирования

Статическое тестирование	**Динамическое тестирование**
Верификация ПО	Валидация ПО
Нет необходимости в выполнении программного кода	Необходимо выполнение программного кода
Направлено на предотвращение багов	Отвечает за функционал программы
Применяется на первичных этапах разработки ПО	Применяется под конец разработки ПО
Стоимость исправления дефектов ниже	Высокая стоимость исправления ошибок
За более короткое время охватывает более широкий круг, нежели при динамическом тестировании	Покрывает определенную часть кода, требует меньшего охвата
Состоит из разных методик оценивания,	Включает в себя и функциональное, и

проверки, сквозного просмотра и прочее	нефункциональное тестирование
Цель: предотвращение ошибок в ПО	Цель: найти и устранить дефекты
Код тестируется комплексно, что помогает обнаружить больше багов	Находит меньше багов, чем во время статического тестирования
Выполняется до развертывания кода	Выполняется по окончанию развертывания кода

Лекция 22

Тема: Типы тестирования. Классификация по степени автоматизации. Ручное тестирование.

Классификация по степени автоматизации:

-Ручное тестирование -Автоматизированное

Ручное тестирование (manual testing) — тестирование, в котором тесткейсы выполняются человеком вручную без использования средств автоматизации.

Несмотря на то что это звучит очень просто, от тестировщика требуются такие качества, как терпеливость, наблюдательность, умение ставить нестандартные эксперименты, а также умение видеть и понимать, что происходит «внутри системы»,

т.е. как внешние воздействия на приложение трансформируются в его внутренние процессы.

Все баги фиксируются в баг-репорте

баг-репорте — отчете о тестировании, по которому разработчики будут исправлять недочеты

Ручное тестирование — это процесс поиска ошибок в программе без использования специальных ПО, силами человека. Тестировщик имитирует реальные действия пользователя и старается охватить максимум функций продукта и найти ошибки (на языке QA — «баги»). Специалист по QA ищет недоработки в визуале, функционале, логике ПО, проверяет его надежность и удобство. Все найденные ошибки QA фиксирует в баг-репорте — отчете о тестировании, по которому разработчики будут исправлять недочеты.

В целом, ручные тестировщики проверяют качество разрабатываемого ПО и обеспечивают это качество конечному пользователям.

Т.е. чтобы качество ПО не было хорошим только для разработчика, а для пользователя было сложны и не понятным- такое ПО считается плохим.

Принцип ручного тестирования 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.

Принцип мануального тестирования говорит о том, что мы не можем заменить ручное тестирование на 100% -автоматизированным, так базовый принцип гласит, что полная автоматизация невозможна. *Поэтому ручное тестирование неизбежно в любом проекте.*

Цель ручного тестирования- убедиться, что в приложении нет ошибок и что оно работает в полном соответствии с требованиями

Шаги ручного тестирование:

1. Читаем документацию и работаем с требованиями. Тестировщики узнают, как должно работать ПО, чего от него ждут разработчики и бизнес. На этом этапе QA-инженер может добавить требования, если они неполные, и сократить, если они невыполнимы.

Вот у клиента появляется некая идея, бизнес-аналитики собирают из этого требования — а тестировщики уже в этот момент приступают к работе, проверяя эти требования.

Как это происходит? Они задают вопросы по предполагаемому функционалу. Пытаются представить, как будет выглядеть приложение, когда оно будет реализовано.

Если речь идёт о новой фиче в уже существующем продукте — пытаются понять, как она будет взаимодействовать с существующим функционалом.

2. Планируем тестирование. Определяем объем работы, бюджет, выбираем методы, типы , виды и инструменты.

- Исследовательское тестирование

После того, как разработчики провели оценку трудозатратности идей клиента и определили сколько потребуется времени на их реализацию.

QA выбирает для данной идеи метод, вид и тип тестирования, не забывая давать часовую оценку

3. Разрабатываем тестовые сценарии. Специалисты создают тест-кейсы — алгоритм проверки ПО, а также чек-листы и готовят среду для выполнения тестов.

появляется необходимость понять, как будут осуществляться переходы между экранами, как будет проходить проверка полей, как приложение или его отдельная функция вообще будет взаимодействовать с конечным пользователем.

И на основе этого строятся тест кейсы и составляются чек-листы.

На разных этапах разработки они могут обладать различным уровнем детализации, но к её окончанию желательно обеспечить максимальное покрытие кейсами всего продукта

4. ***Анализируем дизайн*** когда создаётся карта переходов между экранами, тестировщик уточняет поведение отдельных элементов из которых они состоят и то, какими визуальными эффектами будут сопровождаться те или иные действия пользователя.

тестировщик проверяет макеты на полноту, подтверждая, что они отображают всё, что нужно для реализации задуманного функционала. Уточняются анимационные переходы

Создаются дополнительные тест кейсы и чек листы по дизайну

- Тестирование локализации
- Интернационализации
- Удобства пользователя
- интерфейса

5. ***Проводим первое тестирование.*** Команда выполняет тесты и сообщает разработчикам об ошибках.
    - выполняется смоук-тест:

оценка того, насколько приложение или его новая часть вообще готовы к проверке.

- веб тестирование
- мобильное
- настольное
- кросс браузерное
- негативное
- позитивное

Выполняются тест-кейсы и дается фид-бэк разрабам.

Проверяется запускается ли приложение или конкретная функция в принципе и на конкретных экранах и браузерах, вводятся положительные результаты и отрицательные.

6. ***Делаем повторное тестирование.*** Когда программисты исправили ошибки, тестирование повторяют, чтобы проверить, что после изменений все работает.
    - *проведение регрессионное-тестирование.*

это оценка функционала на стандартный набор кейсов при внедрении каждого нового модуля и каждого изменения приложения.

Проводится после правки разрабами предыдущих ошибок.

Регрессионное тестирование хорошо тем, что оно позволяет найти ошибки даже в тех местах, где раньше всё было в порядке.

Ведь, когда разработчики добавляют новый функционал может быть повреждена текущая версия или новая фича может вступать в конфликт с уже существующими

Например, добавление новых экранов, а значит и изменение навигации может нарушить функционирование меню или, как минимум — его отображение. С другой стороны неприятные сюрпризы может принести и глобальный рефакторинг кода приложения — после него тоже необходимо проводить регресс-тесты.

7. ***Готовим отчет о результатах***. И проводим ***тестирование финальной сборки***, *которое состоит:*

- бета-тестирование

внутренними тестировщиками в реальной среде,

- гамма-тестирование

оценка получившегося продукта самим клиентом и бизнес-аналитиками + получение от них обратной связи

- альфа-тестирование

приложения или его новых возможностей. После этого приложение готово к тому, чтобы его выкатили в продакшн.

Параллельно описывается. в итоговом документе все тесты, выполненные во время разработки программы

Плюсы ручного тестирования

1. **Отчет тестировщика** – это первый отзыв потенциального клиента, который позволит понять, насколько продукт удобен для конечного пользователя.
2. **Обратная связь по UI.** Протестировать общий дизайн приложения и выявить его недостатки представляется возможным только при ручном тестировании.
3. **Стоимость.** Когда речь идет о небольшом проекте, внедрять ручное тестирование всегда менее затратно, чем автоматизацию.
4. **Гибкость.** Тестирование несущественных изменений происходит сразу, без затрат на написание кода. Это особенно важно при быстром внедрении новой функциональности, когда нужно быть уверенным в ее корректной работе.

5. **Исследовательское тестирование и возможность импровизации** позволяет проверить потенциал приложения в нетипичных сценариях и обнаружить существенные дефекты в короткие сроки.

Минусы ручного тестирования

1. **Человеческий фактор**. Часть ошибок продукта может быть пропущена, а некоторые результаты проверки могут оказаться субъективными.
2. **Трудозатраты и продолжительность**. Серия автоматизированных тестов позволяет протестировать программное обеспечение значительно быстрее.
3. **Отсутствие возможности моделирования большой нагрузки.** При ручном тестировании невозможно смоделировать большое количество пользователей.

Лекция 23

Тема: Типы тестирования. Классификация по степени автоматизации. Автоматизированное тестирование.

Автоматизированное тестирование (авто-тесты) — набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения некоторых задач в процессе тестирования. Тест-кейсы частично или полностью выполняет специальное инструментальное средство, однако разработка тест-кейсов, подготовка данных, оценка результатов выполнения, написания отчётов об обнаруженных дефектах — всё это и многое другое по-прежнему делает человек.

техника тестирования, в которой для выполнения тест кейсов используются специальные программы и запускаются они по множеству раз без учета рисков пользовательских ошибок

Цель автоматизации — уменьшить количество тестов, которые нужно выполнять вручную.

Основные области применения автоматизированного тестирования

1. **При выполнении повторяющихся тестов.**

когда один и тот же набор тест-кейсов выполняется каждый день или несколько раз в день, имеет смысл автоматизировать его и вносить незначительные изменения только по мере необходимости.

2. **При использовании тестирования производительности или при нагрузочном тестировании.**

Эти два вида тестирования требуют много времени и усилий от команды QA, поскольку найти уязвимости в производительности продукта может быть непросто.

Автоматизация тестирования - это разумный способ проверить производительность продукта с разных сторон.

3. **Когда имеется большое количество тест-кейсов.**

команда QA проработала над продуктом некоторое время, количество тест-кейсов может достигать нескольких тысяч и более.

Следовательно, команда, работающая вручную, рискует потратить недели на выполнение набора тестов, в этом случае автоматизация.

4. **Когда необходимо исключить человеческий фактор.**

Нужно убедиться, что человеческая ошибка не исказит результаты тестирования.

При правильной реализации автоматизация тестирования устраняет этот риск.

5. **При работе с большими объемами данных.**

тестирование баз данных. Хорошо написанный набор тест-кейсов может обработать миллионы записей за гораздо меньшее время, чем потребовалось бы ручному QA для выполнения даже малой доли этой задачи.

Когда автоматизированное тестирование - не используется

1. **Когда вы планируете выполнять тесты только один или два раза.**

Когда повторное выполнение одних и тех же тестов не входит в планы, автоматизированное тестирование не имеет большого смысла.

2. **Когда нет предсказуемых результатов.**

автоматизированное тестирование обычно включает тесты, которые имеют либо положительный либо отрицательный результат,

т.е. есть четкое представление результатов тестирования, если мы не знаем что на выходе, то тогда ручное тестирование

3. **Когда время является фактором риска.**

Хотя известно, что автоматизация экономит время команды в долгосрочной перспективе, настройка автоматизированных тестов требует времени, которого у вас может не быть на конкретный проект.

Более того, специалисты по автоматизированному тестированию и разработчики могут иметь разные представления о времени, необходимом для выполнения задачи.

4. **Новые тест кейсы, которые еще не были выполнены вручную**
5. **Тест кейсы для функциональности, требования к которой часто меняются**
6. **Тест кейсы, которые выполняются редко**

Плюсы автоматизированного тестирования

1. **Нагрузка на приложение**. Когда используется автоматизированное тестирование, становится возможным моделирование большой нагрузки, которая приближена к реальной ситуации.
2. **Временной фактор.** Ручное тестирование – это долгий и ресурсоемкий процесс, в то время как код для сценария пишется один раз.
3. **Повторяемость.** Код автотестов может быть использован неоднократно

Минусы автоматизированного тестирования

1. **Отсутствие обратной связи.** Автоматизированное тестирование не способно предоставить обратную связь относительно качества продукта – оно лишь выполняет запрограммированные сценарии.

2. **Отсутствие тестирования глазами пользователя.** Иногда в приложении остаются ошибки, которые могут быть не покрыты автотестами.

3. **Отсутствие возможности тестирования цвета, дизайна и эргономики.** Этот пункт не является первостепенным, но может значительно повлиять на качество продукта.

4. **Надежность.** Автоматизированные тесты могут упасть по многим причинам, например, при большой загруженности тестовой машины или при проблемах с сетью.

5. **Стоимость.** Для небольших проектов инструменты автоматизированного тестирования могут оказаться достаточно затратными, поэтому более рационально их использовать для долгосрочных проектов.

Шаги автоматизированного тестирования:

Шаг 1: Выбор инструмента для автоматизации

Шаг 2: Определение функциональности, которую нужно автоматизировать

- Функциональность, которая важна для бизнеса
- Сценарии, для тестирования которых нужны большие объемы входных данных
- Функциональность, использующаяся в нескольких частях приложения
- Целесообразность с технической точки зрения
- Сложность написания тест кейсов
- Возможность использования одних и тех же тест кейсов для кроссбраузерного тестирования

Шаг 3: Планирование, тест дизайн и разработка тестов

На этом этапе создается тест стратегия и тест-план, которые содержат следующие детали:

- Выбранный инструмент автоматизации
- Фреймворк с описанием его особенностей
- Описание функциональности, тестирование которой будет автоматизировано
- Подготовка стендов для выполнения тестов
- Расписание выполнение автотестов
- Результаты автоматизированного тестирования

Шаг 4: Выполнение тестов

подробный тест репорт.
Выполнение тестов может быть запущено как из инструмента автоматизации напрямую, так и с помощью системы управления тестированием (Test Management Tool), который запустит инструмент автоматизации.
Пример: HP Quality Center — система управления тестированием, которая управляет QTP для выполнения автотестов.

Шаг 5: Поддержка написанных тестов

Какое тестирование можно использовать:

- ❖ Тестирование дыма
- ❖ Модульное тестирование
- ❖ Интеграционное тестирование

- Функциональное тестирование
- Регрессионное тестирование
- Тестирование на основе выбора входных данных
- Тестирование черного ящика

Лекция 24

Тема: Техника тестирования требований. Функциональное тестирование. Комплексное тестирование

Комплексное тестирование — процесс поиска несоответствия системы Ее исходным целям. В процессе тестирования участвует система, описание целей продукта и вся документация, поставляемая вместе с системой.

Следующие 15 пунктов дают некоторое представление о том, какие виды тестов могут понадобиться.

Виды тестов:

1. **Тестирование стрессов**. Как правило, системы функционируют нормально при слабой или умеренной нагрузке, но выходят из строя при большой нагрузке и в стрессовых ситуациях реальной среды. Тестирование стрессов представляет собой попытки подвергнуть систему крайнему

«давлению», например, попытку одновременно подключить к системе большое количество терминалов, насытить банковскую систему мощным потоком входных сообщений.

2. Тестирование объема представляет собой попытку предъявить системе большие объемы данных в течение более длительного времени.. На вход компилятора следует подать огромную программу (например,программу обработки текстов). Очередь заданий операционной системы следует заполнить до предела.

Цель — показать, что система не может обрабатывать данные в количествах, указанных в их спецификациях.

3. Тестирование конфигурации. Система должна быть проверена со всяким аппаратным устройством, которое она обслуживает, или со всякой программой, с которой она должна взаимодействовать. Если сама программная система допускает несколько конфигураций, должна быть тестирована каждая из них.

4. Тестирование совместимости. Как правило, разрабатываемые системы не являются совершенно новыми; они представляют собой улучшение прежних версий или замену устаревших. На систему накладывается дополнительное требование совместимости, в соответствии с которым взаимодействие пользователя с прежней версией должно полностью сохраниться и в новой системе.

5. Тестирование защиты. К большинству систем предъявляются определенные требования по обеспечению защиты от несанкционированного доступа. Цель тестирования защиты — нарушить секретность в системе.

Один из методов — нанять профессиональную группу «взломщиков», т. е. людей с опытом разрушения средств обеспечения защиты в системах.

Одним из путей разработки подобных тестов является изучение известных проблем защиты в этих системах и генерация тестов, которые позволяют проверить, как решаются аналогичные проблемы в тестируемой системе.

6. Тестирование требований к памяти. При проектировании многих систем ставятся цели, определяющие объем основной и вторичной памяти, которую системе разрешено использовать при различных условиях. С помощью специальных тестов нужно попытаться показать, что система этих целей не достигает.

7. Тестирование производительности. Определяются такие характеристики, как *время отклика и уровень пропускной способности при определенной нагрузке* и конфигурации оборудования. Проверка системы в этих случаях сводится к демонстрации того, что данная программа не удовлетворяет поставленным целям.

8. Тестирование настройки. Тестирование процесса настройки системы очень важно, поскольку зачастую покупатель оказывается не в состоянии даже настроить новую систему. Должна быть четкая инструкция и описание последовательности ,для сборки и конфигурации системы для пользователя, чтобы пользователь смог сам запустить приложение

9. Тестирование надежности/готовности заключается в попытке доказать, что система не удовлетворяет исходным требованиям к надежности(среднее время между отказами, количество ошибок, способность к обнаружению, исправлению ошибок и/или устойчивость к ошибкам и т. д.).

Например, в систему можно намеренно внести ошибки (как аппаратные, так и программные), чтобы тестировать средства обнаружения, исправления и обеспечения устойчивости.

10. Тестирование средств восстановления. Можно намеренно ввести в операционную систему программные ошибки,чтобы проверить,восстановится ли она после их устранения. Неисправности аппаратуры, ошибки в данных (помехи в линиях связи и

неправильные значения указателей в базе данных) можно намеренно создать или промоделировать для анализа реакции на них системы.

11. Тестирование удобства обслуживания. Все документы, описывающие внутреннюю логику, следует проанализировать глазами обслуживающего персонала, чтобы понять, как быстро и точно можно указать причину ошибки, если известны только некоторые се симптомы.

12. Тестирование публикаций. Все комплексные тесты следует строить только на основе документации для пользователя.

Любые примеры, приведенные в документации, следует оформить как тест и подать на вход программы.

13. Тестирование психологических факторов. Эта сторона не так важна, как другие, однако мелкие недостатки могут быть обнаружены и устранены при тестировании системы. Например, может оказаться, что ответы или сообщения системы плохо сформулированы или ввод команды пользователя требует постоянных переключений регистров.

14. Тестирование удобства установки.

15. Тестирование удобства эксплуатации.

Не все из перечисленных 15 пунктов применимы к тестированию всякой системы (например, когда тестируется отдельная прикладная программа), но тем не менее это перечень вопросов, которые разумно иметь в виду.

По своей природе комплексные тесты никогда не сводятся к проверке отдельных функций системы. Они часто пишутся в форме сценариев, представляющих ряд последовательных действий пользователя.