#### Термины

**Quality assurance -** Это совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации ПО, для обеспечения требуемого уровня качества выпускаемого продукта.

**Quality control -** Это подмножество QA, совокупность действий, проводимых над продуктом в процессе разработки, для получения информации о его актуальном состоянии и соответствии ожидаемым результатам.

**Что такое качество** – со стороны пользователя продукт, соответствующий ожиданиям. Со стороны QA продукт, соответствующий требованиям.

**Тестирование** – комплекс мероприятий, направленный на проведение проверок на соответствие производимого продукта требованиям, к нему предъявляемым (прямым и косвенным)

**Цель тестирования** – предоставление актуальной информации о соответствии производимого продукта требованиям.

**Баг** – несоответствие ожидаемого и фактического результата

**Use case** – документ описывающий как приложение должно вести себя при определённых сценариях, выполняемых пользователем.

**Цикл разработки ПО** – методология разработки по, включающая определённые этапы, от идеи до передачи продукта пользователю

**Верификация** – процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки предъявляемым ей требованиям

**Валидация** – проверка продукта на соответствие ожиданиям конечного пользователя или рынка в целом

**Test Plan** – вид тестовой документации регламентирующий процесс тестирования по, содержащий в себе объект, цели, время, приоритет тестирования, анализ рисков, критерии начала и конца тестирования

**Test case** – набор предусловий, входных данных, шагов и ожидаемых результатов для проверки определённой части по

**Баг репорт** - это документ, описывающий ситуацию или последовательность действий, приводящую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

**Компонентное (модульное) тестирование (Unit)** – тестирования части нового кода

**Интеграционное тестирование** – тестирование нескольких функциональностей на корректную совместную работу

**Системное тестирование** – тестирование собранного по частям продукта в качестве единой системы

**End-to-end(сквозное) тестирование** – это стратегия тестирования для выполнения тестов, которые охватывают все возможные потоки приложения от его начала до конца; проверяет программную систему вместе с ее интеграцией с внешними интерфейсами. Целью сквозного тестирования является создание полного производственного сценария, выявление программных зависимостей и утверждение, что между различными программными модулями и подсистемами передается правильный ввод.

#### Функциональное тестирование

**Ad hoc тестирование** – это вид тестирования, который выполняется без подготовки к тестированию продукта, без определения ожидаемых результатов, проектирования тестовых сценариев. Это неформальное, импровизационное тестирование. Оно не требует никакой документации, планирования, процессов, которых следует придерживаться при выполнении тестирования.

**Regression testing** – повторное тестирование продукта после добавления или обновления кода, чтобы убедится в отсутствии новых багов.

- проверить затронутый функционал

- весь основной функционал

**Exploratory testing** – подразумевает под собой одновременно изучение проекта, функционала, проектирование тест кейсов в уме и тут же их исполнение, не записывая и не создавая тестовую документацию.

**Бетта-тестирование** – тестирование по большей части законченного продукта его окончательными пользователями

**Дымовое тестирование (smoke testing)** – проверка критически важных функциональностей нового билда, для принятия решения о дальнейшем его тестировании.

**Приёмочное тестирование (Accеptance)** – тестирование продукта в конце цикла разработки для принятия решения о передаче его в релиз.

**Санити тестирование(Sanity)** – это узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Проверка функциональности, которая часто выходит из строя.

**Sanity vs Smoke -** Эти виды тестирования имеют "вектора движения", направления в разные стороны. В отличии от дымового (Smoke testing), санитарное тестирование (Sanity testing) направлено вглубь проверяемой функции, в то время как дымовое направлено вширь, для покрытия тестами как можно большего функционала в кратчайшие сроки.

#### Нефункциональное тестирование

**Performance Testing (производительности)** – вид тестирования, предполагающий проверку продукта на то, как он себя поведёт при определённой нагрузке, стрессовой нагрузке, при увеличении баз данных, его время отклика.

**Нагрузочное тестирование** – проверка продукта при определённой нагрузке (наплыва пользователей и тд)

**Стрессовое тестирование (Stress Testing)** – проверка продукта при стрессовой нагрузке, насколько быстро он восстановится

**Объемное тестирование (Volume Testing)** – тестирование продукта при увеличении баз данных, насколько ухудшается время отклика

**Тестирование совместимости (compatibility testing)** – тестирование работы продукта на различных версиях операционных систем, браузерах

**Тестирование интернационализации (internationalization testing)** — проверка работоспособности продукта в различных регионах, возможности подстраиваться под их особенности

**Тестирование локализации (localization testing)** — проверка адаптации продукта под конкретный регион

**Тестирование безопасности (Security testing)**

**Тестирование на удобство пользования (Usability testing)**

**Black Box testing** – тестирование продукта со стороны пользователя, без работы с исходным кодом продукта

**Negative testing** – тестирование по нестандартному сценарию рядового пользователя

**Gray Box Testing** – расширенное тестирование чёрного ящика с применением специальных инструментов для работы с API, лог файлами

**Test cases for Login/Password functionality** –

Valid login/valid password

Password скрыт за буллетами

Password нельзя скопировать

Password нельзя вырезать

Invalid login/valid password

Valid login/invalid password

#### Классификация видов тестирования:

1. По знанию внутренностей системы(методы)

Белый ящик

Чёрный ящик

Серый ящик

1. По объекту тестирования

##### Функциональные виды тестирования

- Функциональное тестирование (Functional testing)

- Тестирование пользовательского интерфейса (GUI Testing)

- Тестирование безопасности (Security and Access Control Testing)

##### Нефункциональные виды тестирования

- Все виды тестирования производительности:

- нагрузочное тестирование (Performance and Load Testing)

- стрессовое тестирование (Stress Testing)

- тестирование стабильности или надежности (Stability / Reliability Testing)

- объемное тестирование (Volume Testing)

- многозадачности

- Тестирование установки (Installation testing)

- Тестирование удобства пользования (Usability Testing)

- Тестирование на отказ и восстановление (Failover and Recovery Testing)

- Конфигурационное тестирование (Configuration Testing) – как будет работать приложение на определённом железа компа или мобилы

- Тестирование совместимости (кроссбраузерное тестирование)

- Тестирование документации

- UI (внешний вид: картинки, вёрстка, цвета, шрифты и т.д.)

- локализации и интернационализации

1. По субъекту тестирования

Бетта тестер

Альфа тестер

1. По времени проведения тестирования

- До релиза

Альфа

Smoke

Функциональное

Регрессионное

Приёмочное

- После релиза

Бетта тестирование

Санити тестирование

1. По критерию позитивности

Позитивное

Негативное

1. По уровню тестирования

Модульное

Интеграционное

Системное

Приемочное тестирование

1. По степени автоматизации

Автоматизированное

Ручное

1. По степени подготовки к тестированию

Monkey testing

Ad hoc

Исследовательское

1. По критерию изменений

- смоук

- регрессионное

- тестирование билда

- санити

10) По направлению

- статическое

- динамическое

#### Порядок видов тестирования

- смоук

- проверка новых фич

- интеграционное

- регресс

- нагрузочное

- приёмочное

- санити

#### Критерии начала тестирования

* готовность тестовой платформы (тестового стенда)
* законченность разработки требуемого функционала
* наличие всей необходимой документации

#### Критерии завершения тестирования

- тест кейсы пройдены

- нет critical и blocker дефектов

- определённое количество major и minor дефектов

#### Из чего состоит тест кейс

Номер

Заголовок

Предусловие

Шаги

Ожидаемый результат

#### Характеристики хорошего тест кейса

1. независимость, атомарность
2. однозначность
3. полнота
4. обезличенность
5. 1 тест кейс = 1 цель
6. Простота описания (насколько это возможно)

#### **Повысить качество тест кейса**

- матрица покрытия (таблица в одном столбце требования, в другом тестовые сценарии)

- использование техник тест дизайна

- согласование тест кейсов с аналитиком

#### Из чего состоит баг репорт

Краткое описание

Среда тестирования

Приоритет и серьёзность бага

Шаги воспроизведения бага

Ожидаемый и фактический результат

Доказательства (скриншот, видео, лог файлы)

#### Серьёзность дефекта

- **trivial** - опечатки в тексте, несоответствие шрифта и оттенка и т.п., либо плохо воспроизводимая ошибка, не касающаяся бизнес-логики, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

- **minor** - ошибки GUI, которые не влияют на функциональность, но портят юзабилити или внешний вид. Также незначительные функциональные дефекты, либо которые воспроизводятся на определенном устройстве.

- **major** - не работает часть какой-либо функции/бизнес-логики, но при выполнении специфических условий(опеределённых тестовых данных), либо не работает не очень значительная часть какой-либо функции.

- **critical** - не работает важная часть одной какой-либо функции либо не работает значительная часть, но имеется workaround (обходной путь/другие входные точки), позволяющий продолжить тестирование.

- **blocker** - тестирование значительной части функциональности вообще недоступно. Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна.

**Высокая серьёзность, низкий приоритет**:

важная фича не работает, но ей никто не пользуется

**Низкая серьёзность, высокий приоритет**

На главной странице гугл орфографическая ошибка

#### Виды требований

1. **Прямые** (т. е. формализованными в технической документации, спеках, юзер-стори и прочих формальных артефактах)
2. **Косвенные** (т. е. проистекающими из прямых, либо являющимися негласным стандартом для данной продукции или основывающиеся на опыте и здравом смысле использования данного продукта или продуктах, подобных ему)
3. **Функциональные** (описывающие какие функции должен выполнять продукт)
4. **Нефункциональные** (требования к окружению, поддерживаемости, надежности и прочим характеристикам продукта)
5. **Бизнес требования**

#### Виды тестовой документации

##### Внешняя

* **Замечание** – короткая записка, комментарий о небольшой неточности в реализации продукта.
* **Баг-репорт** – это документ, описывающий ситуацию или последовательность действий, приводящую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата
* **Запрос на изменение** (улучшение) – описание неявных/некритичных косвенных требований, которые не были учтены при планировании/реализации продукта, но несоблюдение, которых может вызвать неприятие у конечного потребителя. И пути/рекомендации по модификации продукта для соответствия им.
* **Отчет о тестировании (тест репорт)** – документ, предоставляющий сведения о соответствии/ несоответствии продукта требованиям.

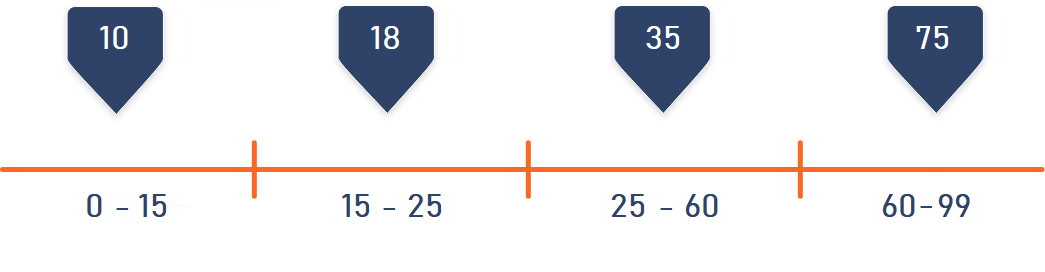
##### Внутренняя

* **Тест-план (план тестирования)** – вид тестовой документации регламентирующий процесс тестирования по, содержащий в себе объект, цели, время, скоуп работ, виды тестирования, приоритет тестирования, анализ рисков, тестовая стратегия, ресурсы и команда, критерии начала и конца тестирования
* **Тестовый сценарий** – последовательность действий над продуктом, которые связаны единым ограниченным бизнес-процессом использования, и сообразных им  проверок корректности поведения продукта в ходе этих действий.
* **Тестовый случай (тест-кейс)** – набор предусловий, входных данных, шагов и ожидаемых результатов для проверки определённой части по
* **Тестовый комплект** – набор тест кейсов объединенных между собой по общему логическому признаку.
* **Чек-лист** – список проверок
* **Матрица покрытия требований**

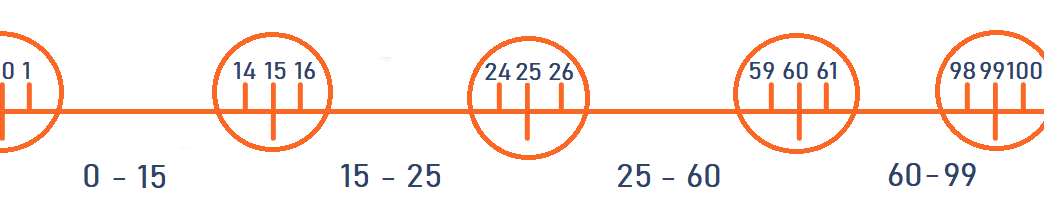
#### Техники тест дизайна:

1. **Эквивалентные классы –** это подмножество элементов из определенного набора, которые обрабатываются Системой (тестируются) одинаково.

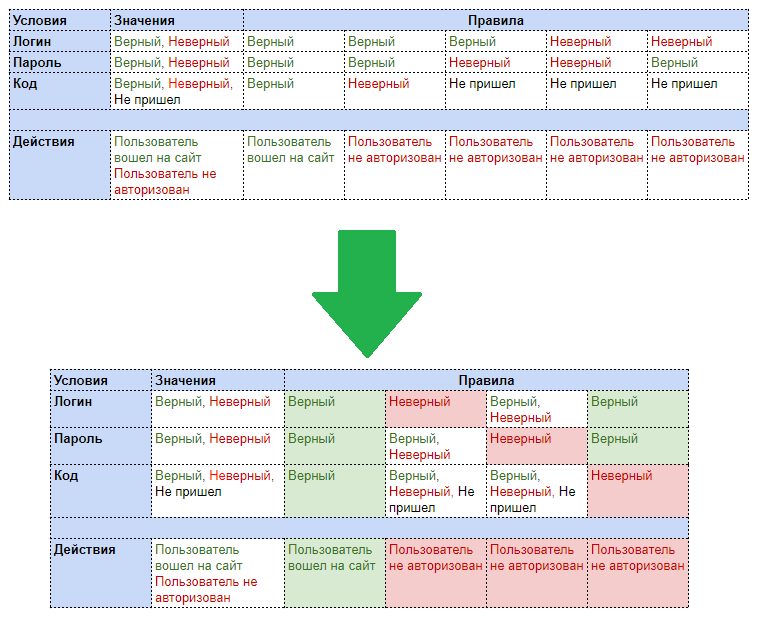
Метод эквивалентного разбиения позволяет минимизировать число тестов, не создавая сценарий для каждого возможного значения, а выбрав только одно значение из целого класса и приняв за правило, что для всех значений в данной группе результат будет аналогичным.



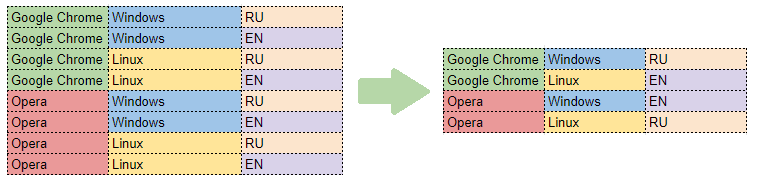
1. **Граничные значения -** техника граничных значений основана на предположении, что большинство ошибок может возникнуть на границах эквивалентных классов.



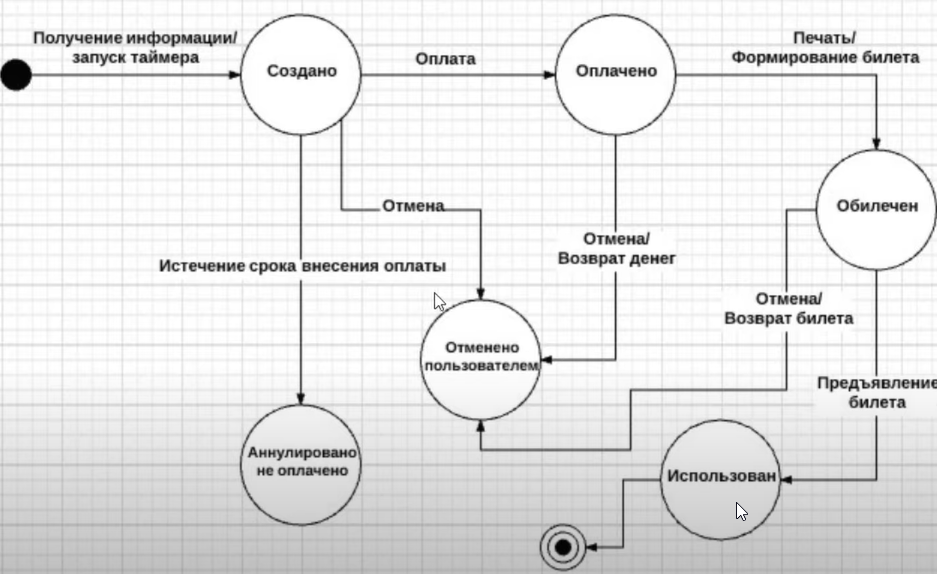
1. **Матрица принятия решений**



1. **Попарное тестирование -** формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров. Попарное тестирование позволяет обнаружить максимум ошибок без избыточных проверок.



1. **Диаграмма состояний и переходов –** составление блок схемы с переходами состояний

****

1. **Предугадывание ошибки –** тестировщик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку.
2. **Причина и следствие –** Это ввод комбинаций условий (причин), для получения ответа от системы (Следствие), то есть Простая проверка базовых действий и их результата.
3. **Сценарии, use cases**
4. **Исчерпывающее тестирование**

#### Характеристики хорошей документации:

1. Полнота

Все ли описано? Ничего не забыли? Вдруг у нас остался неописанный функционал или параметр API-метода? Не повредит составить чек лист проверок.

1. Однозначность

Требования должны трактоваться всеми одинаково.

Отчет должен загружаться быстро» → что значит «быстро»?

1. Непротиворечивость

Требования не должны противоречить сами себе

1. Необходимость. Пишите только то, что необходимо:

В ТЗ — функционал, основной сценарий и альтернативы, типы ошибок.

В пользовательской документации — то, как пользоваться системой. Но не доходя до крайностей и обучения включению компьютера.

1. Осуществимость

А можно ли реализовать то, что тут написано? Насколько это будет сложно и дорого?

1. Тестируемость

Можно ли протестировать этот функционал?

#### Принципы тестирования:

1. Тестирование демонстрирует наличие дефектов (Тестирование может показать, что дефекты присутствуют, но не может доказать, что их нет)
2. Исчерпывающее тестирование недостижимо (Вместо исчерпывающего тестирования должны использоваться анализ рисков и расстановка приоритетов)
3. Раннее тестирование
4. Скопление дефектов (большая часть дефектов в небольшом количестве модулей)
5. Парадокс пестицида
6. Тестирование зависит от контекста (программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем сайт электронной коммерции)
7. Заблуждение об отсутствии ошибок

#### Модели разработки ПО

1. **Waterfall -** разработка осуществляется поэтапно: каждая следующая стадия начинается только после того, как заканчивается предыдущая.
2. **V-образная модель** - усовершенствованная каскадная модель, в которой заказчик с командой программистов одновременно составляют требования к системе и описывают, как будут тестировать ее на каждом этапе.
3. **Incremental Model** -Это модель разработки по частям, инкремент за инкрементом разработчики совершенствуют продукт, приближаясь к описанному в ТЗ.
4. **Iterative Model** - Это модель, при которой заказчик не обязан понимать, какой продукт хочет получить в итоге, и может не прописывать сразу подробное техзадание.
5. **Spiral Model** - Используя эту модель, заказчик и команда разработчиков серьезно анализируют риски проекта и выполняют его итерациями. Последующая стадия основывается на предыдущей, а в конце каждого витка — цикла итераций — принимается решение на основе рисков, продолжать ли развивать проект. Спиральная модель похожа на инкрементную, но здесь гораздо больше времени уделяется оценке рисков. С каждым новым витком спирали процесс усложняется. Эта модель часто используется в исследовательских проектах и там, где высоки риски.
6. **Agile –** создана на основе итеративной модели. Гибкая методология, которая позволяет QA быть вовлечённым в разработку продукта на всех этапах. Agile - это гибкая методология разработки программного обеспечения, которая ставит акцент на гибкость, сотрудничество и быструю адаптацию к изменениям.

Agile подразумевает итеративный подход к разработке, разбивая проект на короткие циклы разработки, называемые спринтами.

Некоторые ключевые концепции Agile:

Спринт: Это короткий временной интервал (обычно от 1 до 4недель), в течение которого команда разрабатывает определенный набор функциональности или достигает конкретных целей.

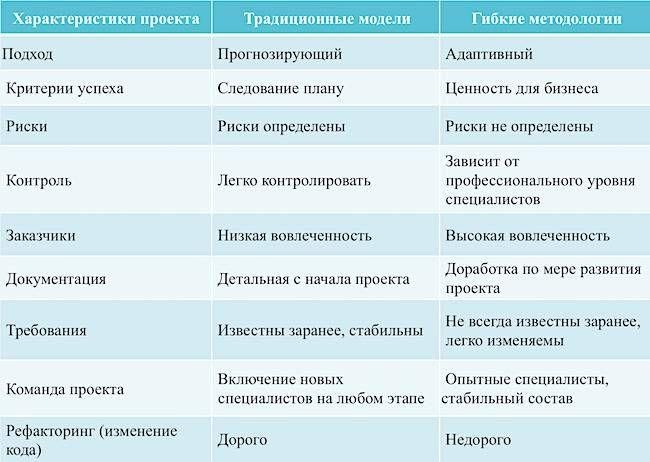
Backlog продукта: Это список требований и функциональностей, которые должны быть реализованы в проекте. Он может быть изменен и приоритизирован в течение разработки.

Планирование спринта: Команда выбирает элементы из backlog'а продукта, которые будут реализованы в следующем спринте, и планирует, как их достичь.

Daily Stand-up: Ежедневное короткое совещание, на котором члены команды обмениваются информацией о проделанной работе, планах на день и препятствиях, с которыми они сталкиваются.

Ретроспектива: После завершения спринта команда проводит ретроспективное собрание, на котором анализируются прошлый спринт, выявляются улучшения и принимаются решения для будущего развития.

Гибкость: Agile способствует быстрой адаптации к изменениям в требованиях и среде разработки, позволяя команде гибко реагировать и вносить изменения в проект.

****

#### Жизненный цикл ПО

* **Анализ.** Участники: Product owner, BA(бизнес-аналитик), QA. Артефакты: спецификация требований к ПО

Во время этого этапа BA выясняет у PO все пожелания к продукту и документирует это в процессе обсуждения требований. Необходимо убедиться в том, что все участники правильно поняли поставленные задачи и то, как именно каждое требование будет реализовано на практике. Таким образом, этот этап предполагает сбор требований к разрабатываемому ПО, их систематизацию, документирование, анализ, а также выявление и разрешение противоречий. В компаниях, где налажены процессы обеспечения качества, уже на этом этапе включается в работу QA, т.к. бывают и дефекты требований.

* **Проектирование.** Участники: Product owner, разработчики, системные архитекторы, QA. Артефакты: дизайн-спецификация, Тест-план, тест-сценарии, тест-кейсы.

На стадии проектирования программисты и системные архитекторы, руководствуясь требованиями, разрабатывают высокоуровневый дизайн системы. Разнообразные технические вопросы, возникающие в процессе проектирования, обсуждаются со всеми заинтересованными сторонами. Определяются технологии, которые будут использоваться в проекте, загрузка команды, ограничения, временные рамки и бюджет. В соответствии с уточненными требованиями выбираются наиболее подходящие проектные решения. QA/test analyst проектируют процесс тестирования в тест плане, руководствуясь также (если есть) политикой и стратегией тестирования. Тестировщики начинают писать сценарии и по ним кейсы для тестирования.

* **Разработка.** Участники: разработчики.
* **Тестирование.** Участники: QA. Артефакты: дефект-репорты, отчет о тестировании.

Тестовый администратор (если есть) настраивает тестовые среды/стенды для проведения тестирования. Тестировщики занимаются поиском дефектов в программном обеспечении и сравнивают описанное в требованиях поведение системы с реальным. В фазе тестирования обнаруживаются пропущенные при разработке баги. При обнаружении дефекта, тестировщик составляет отчет об ошибке, который передается разработчикам. Последние его исправляют, после чего тестирование повторяется – но на этот раз для того, чтобы убедиться, что проблема была исправлена, и само исправление не стало причиной появления новых дефектов в продукте. По итогам проведенного процесса тестирования составляется итоговый отчет.

* **Внедрение и сопровождение.** Участники: команда технической поддержки. Артефакты: замечания, запросы на исправление/улучшение.

#### Жизненный цикл тестирования

* Выявление и анализ требований прямых и косвенных
* Планирование испытаний (создание тест плана)
* Генерация Test case
* Проведение проверок
* Фиксация результатов
* Анализ результатов
* Передача информации о соответствии проверенного продукта требованиям.

#### интернет протоколы

1. HTTP (Hyper Text Transfer Protocol)
2. FTP (File Transfer Protocol)
3. POP3 (Post Office Protocol)
4. SMTP (Simple Mail Transfer Protocol)
5. TELNET – протокол удалённого доступа к машине
6. TCP - сетевой протокол, отвечающий за передачу данных в сети Интернет, медленнее чем UDP(важен там, где нужна уверенность, что данные получены получателем)
7. UDP - это тоже транспортный протокол передачи данных, но без подтверждения доставки, быстрее чем TCP. Используется там, где не важна уверенность получения данных получателем, например, кино онлайн, трансляции, скайп, прослушивание музыки т.к. потерянные пакеты могут быть подменены арифметическим средним.
8. Ethernet - протокол, определяющий стандарты сети на физическом и канальном уровнях.
9. http — протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.
10. https — расширение протокола HTTP для поддержки шифрования в целях повышения безопасности.

#### Компоненты HTTP

1. Request

* стартовая строка (starting line) — определяет тип сообщения, имеет вид Метод URI HTTP/Версия протокола, например GET /web-programming/index.html HTTP/1.1
* заголовки запроса (header fields) — характеризуют тело сообщения, параметры передачи и прочие сведения
* тело сообщения (body) — необязательное

1. Response

* строка состояния (status line), включающая код состояния и сообщение о причине (200 OK), время выполнения и размер
* поля заголовка ответа (header fields) user agent, content type
* дополнительное тело сообщения (body)

#### методы протокола http

HTTP определяет множество методов запроса, которые указывают, какое желаемое действие выполнится для данного ресурса.

1. Метод GET в HTTP используется для получения информации от сервера. Запросы клиентов, использующие метод GET должны получать только данные и не должны никак влиять на эти данные.
2. Метод HEAD работает точно так же, как GET, но в ответ сервер посылает только заголовки и статусную строку без тела HTTP сообщения.
3. Метод POST используется для отправки данных на сервер, например, из HTML форм, которые заполняет посетитель сайта.
4. Метод PUT заменяет все текущие представления ресурса данными запроса.
5. Метод DELETE удаляет указанный ресурс.
6. Метод CONNECT преобразует существующее соединение в тоннель.
7. Метод OPTIONS используется для получения параметров текущего HTTP соединения.
8. Метод TRACE создает петлю, благодаря которой клиент может увидеть, что происходит с сообщением на всех узлах передачи.
9. Метод PATCH используется для частичного изменения ресурса.

#### HTTP status codes

* Информационные (100-105)
* Успешные (200-226)
* Перенаправление (300-307)
* Ошибка клиента (400-499)
* Ошибка сервера (500-510)

#### well formed xml. Правила построения SOAP запроса

1. В xml только один корневой элемент
2. Все элементы должны иметь закрывающие теги
3. Названия тегов – регистрозависимы
4. Элементы не должны пересекаться (если child тэг открылся внутри parent тега, то он должен и закрыться внутри него)
5. Значения атрибутов должны быть в кавычках
6. ‘<’, ‘>’, ‘&’ нельзя использовать в текстовых блоках. Вместо них используются &lt вместо <, &gt вместо >, &amp вместо &, &apos вместо ‘, &quot вместо “
7. Xml объявляется в первой строке

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

#### XSD-схема

XSD – это язык описания структуры XML документа. Его также называют XML Schema. При использовании XML Schema XML парсер может проверить:

* какие поля будут в запросе;
* какие поля будут в ответе;
* какие типы данных у каждого поля;
* какие поля обязательны для заполнения, а какие нет;
* есть ли у поля значение по умолчанию, и какое оно;
* есть ли у поля ограничение по длине;
* есть ли у поля другие параметры;
* какая у запроса структура по вложенности элементов;

#### Чем отличается REST от SOAP протокола?

1. REST — это архитектурный стиль. SOAP — это формат обмена сообщениям, имеет веб-сервис WSDL с прописанными методами, которые можно удаленно вызывать.
2. REST работает только по HTTP/HTTPS, SOAP с любым протоколом прикладного уровня: SMPT, FTP, HTTP, HTTPS, POP3.
3. REST более простой, гибкий и быстрый, SOAP типизированный, но в некоторых случаях лучше визуализируется за счет применения им синтаксиса похожего на HTML разметку.
4. REST использует Json и XML, SOAP только XML.
5. REST может быть закэширован

#### TCP/IP модель



#### OSI модель

Прикладной (HTTP, HTTPS, FTP, POP3 и т.д.)

Представления (JPG, ASCII)

Сеансовый (RPC, PAP)

Транспортный (TCP, UDP)

Сетевой (IP)

Канальный (Ethernet, 802.11)

Физический (витая пара, медная линия, радиоканал)

#### IP

Содержит 32 bit информации, может содержать значение от 0 до 255

IPv4 192.168.0.1. -> 11000000.10101000.00000000.00000001

Между точками десятичное число, минимальное значение 00000000, а максимальное 11111111

Маска подсети имеет такой же вид (выглядит также). Слева направо идут единицы – это адрес сети, как только начинаются нули – это адрес машины в этой сети (11111111. 11111111. 11111111. 00000000)

#### URI, URL, URN

Расшифруем аббревиатуры:

* URI –  Uniform Resource Identifier (унифицированный идентификатор ресурса) - имя и адрес ресурса в сети, включает в себя URL и URN
* URL – Uniform Resource Locator (унифицированный определитель местонахождения ресурса) -  адрес ресурса в сети, определяет местонахождение и способ обращения к нему
* URN – Uniform Resource Name (унифицированное имя ресурса) - имя ресурса в сети, определяет только название ресурса, но не говорит как к нему подключиться

Рассмотрим примеры:

* URI – <https://wiki.merionet.ru/images/vse-chto-vam-nuzhno-znat-pro-devops/1.png>
* URL - [https://wiki.merionet.ru](https://wiki.merionet.ru/)
* URN - images/vse-chto-vam-nuzhno-znat-pro-devops/1.png

#### **DHCP**

DHCP – сервер, предоставляющий настройки для подключения к сети (например роутер)

#### Cookies

Файл cookie - небольшой фрагмент данных (часть http заголовка), который веб-сервер хранит в текстовом файле на жестком диске пользователя (клиента). Эта часть информации затем отправляется обратно на сервер каждый раз, когда браузер запрашивает страницу с сервера. Куки в основном используются для трех целей:

* Управление сессиями: Логины, корзины покупок, результаты игр и все, что сервер должен запомнить
* Пользовательские настройки, темы и другие настройки
* Запись и анализ поведения пользователя

**Куки состоят в основном из трех вещей:**

* Имя сервера, с которого был отправлен куки
* Время жизни (Cookies Lifetime)
* Случайно сгенерированный уникальный номер

Максимальный размер кук = 4 килобайт (4096 байт)

**Виды кук:**

* Сессионные cookie, существуют только во временной памяти, пока пользователь находится на странице веб-сайта. Браузеры обычно удаляют сессионные cookie после того, как пользователь закрывает окно браузер. В отличие от других типов cookie, сессионные cookie не имеют истечения срока действия, и поэтому браузеры понимают их как сессионные.
* Постоянные cookie-файлы удаляются в определенную дату или через определенный промежуток времени. Их иногда называются следящие cookie, поскольку они могут использоваться рекламодателями для записи о предпочтениях пользователя в течение длительного периода времени. Однако, они также могут использоваться и в «мирных» целях, например, чтобы избежать повторного ввода данных при каждом посещении сайта.
* Поскольку cookie можно очень легко удалить из браузера, программисты ищут способы идентифицировать пользователей даже после полной очистки истории браузера. Одним из таких решений являются зомби-cookie (или evercookie, или persistent cookie) — не удаляемые или трудно удаляемые cookie, которые можно восстановить в браузере с помощью JavaScript. Когда программа обнаруживает отсутствие в браузере cookie-файла, информация о котором присутствует в других хранилищах — она тут же восстанавливает его на место и, тем самым, идентифицирует пользователя для сайта.

**Примеры Test case для Cookie testing:**

* Отключение файлов cookie: отключите все файлы cookie и попытайтесь использовать основные функции сайта
* Поврежденные файлы cookie: вручную отредактируйте файл cookie в блокноте и измените параметры на несколько случайных значений
* Шифрование куки: конфиденциальная информация, такая как пароли и имена пользователей, должна быть зашифрована
* Тестирование файлов cookie в нескольких браузерах. Убедитесь, что с вашего веб-сайта правильно записываются cookie в разных браузерах
* Проверка удаления куки с веб-сайта
* Удаление файлов cookie: удалите все файлы cookie для веб-сайтов и посмотрите, как веб-сайт среагирует
* Доступ к файлам cookie: файлы cookie, написанные одним сайтом, не должны быть доступны другим
* Не допускайте чрезмерного использования файлов cookie: если тестируемое приложение является общедоступным веб-сайтом, не следует злоупотреблять файлами cookie.
* Тестирование с другими настройками. Тестирование должно выполняться правильно, чтобы убедиться, что веб-сайт работает хорошо с другими настройками файлов cookie.
* Категоризируйте куки отдельно: куки не должны храниться в той же категории вирусов, спама или шпионских программ

#### Web Storage

Интернет-хранилище представляет собой постоянное хранилище данных в браузере, похожее на куки, но со значительно расширенной емкостью и без хранения информации в заголовке запроса HTTP. Существуют два основных типа веб-хранилища: локальное хранилище (localStorage) размером 10 мб и сессионное хранилище (sessionStorage) размером 5 мб, ведущие себя аналогично постоянным и сессионным кукам соответственно.



#### Идемпотентность

Определяет несколько методов запросов, некоторые из которых являются идемпотентными, а некоторые — нет. Такие HTTP-методы гарантируют, что повторное выполнение запросов будет безопасным и не приведёт к дополнительным изменениям на сервере (кроме возможного первого изменения). **К данным методам относятся:**

- GET: Используется для запроса данных с сервера. GET должен быть безопасным и идемпотентным, что означает отсутствие изменений данных.

- PUT: Заменяет все текущие представления ресурса на загружаемый контент. Если ресурс не существует, PUT может создать его. PUT считается идемпотентным, потому что независимо от того, сколько раз запрос будет выполнен, состояние сервера будет одинаковым.

- DELETE: Удаляет указанный ресурс. Повторное выполнение DELETE на том же ресурсе не изменит состояние сервера после первого успешного удаления.

- HEAD: Такой же, как и GET, но без тела ответа. Используется для извлечения заголовков.

- OPTIONS: Описывает параметры связи для целевого ресурса.

**Методы, которые не являются идемпотентными:**

- POST: Используется для создания нового ресурса. Поскольку повторные POST-запросы могут создавать новые ресурсы или инициировать новые процессы каждый раз при выполнении, этот метод не считается идемпотентным.

#### Микросервисная архитектура

Микросервисная архитектура (MSA, Microservices Architecture) – это стиль разработки ПО, в котором приложение разбивается на мелкие независимые сервисы, взаимодействующие между собой через API.

**Каждый микросервис**

Выполняет одну конкретную бизнес-задачу (например, авторизация, платежи, управление пользователями).

Автономен – его можно разрабатывать, развертывать и масштабировать отдельно.

Общается с другими микросервисами через HTTP (REST, gRPC) или асинхронные очереди (Kafka, RabbitMQ\*.

**Ключевые принципы микросервисов**

Разделение на мелкие сервисы – нет единого монолита, каждый сервис отвечает за свою функцию.

Связь через API – микросервисы общаются по HTTP (REST, GraphQL, gRPC) или через брокеры сообщений.

Независимое развертывание – каждый сервис можно обновлять без остановки всей системы.

Собственная база данных – каждый сервис управляет своими данными (это уменьшает зависимость между сервисами).

Горизонтальное масштабирование – можно увеличить только нужные сервисы под нагрузкой.

**Плюсы**

➕Гибкость

можно разрабатывать разные сервисы разными командами и технологиями.

➕Масштабируемость

можно увеличить только нужные сервисы, а не всю систему.

➕Устойчивость

отказ одного сервиса не ломает всю систему.

➕Быстрое развертывание

можно обновлять отдельные сервисы без остановки всего приложения.

**Минусы**

➖Сложность

нужно продумывать взаимодействие сервисов, мониторинг, балансировку.

➖Задержки в коммуникации

из-за работы по сети API-запросы могут выполняться медленнее, чем в монолите.

➖Сложное тестирование

приходится тестировать не только сервисы по отдельности, но и их взаимодействие.

#### Монолитная архитектура

Монолитная архитектура – это подход, при котором вся логика приложения разрабатывается и разворачивается как единое целое. В отличие от микросервисов, в монолите все компоненты (UI, бизнес-логика, база данных) работают в одном приложении.

🚩Плюсы

➕Простота разработки и отладки

Легче настроить среду разработки – один репозиторий, одна кодовая база. Не нужно думать о сетевом взаимодействии между сервисами. Отладка и логирование проще, так как всё в одном месте.

➕Легче развертывать и управлять

Один файл (JAR, WAR) или контейнер – простое развертывание. Нет сложностей с оркестрацией сервисов, как в Kubernetes. Все компоненты запускаются вместе, не надо настраивать отдельные сервисы.

➕ Производительность выше, чем у микросервисов

Вызовы между модулями выполняются внутри приложения (методом или через классы), а не по сети. Нет оверхеда на сетевые запросы, балансировку, сериализацию данных.

➕Простое тестирование

Можно запускать юнит-тесты и интеграционные тесты без сложной инфраструктуры. Не нужно поднимать отдельные сервисы или мокать API-запросы.

➕Меньше накладных расходов

Разработчики тратят меньше времени на настройку CI/CD, мониторинг, логику взаимодействия между сервисами. Один сервер, одна база данных – проще администрировать.

➕Легче внедрять новые разработчиков

Новичку проще разобраться в коде, чем в распределенной системе с множеством сервисов. Можно быстрее делать первые коммиты.

🚩Когда монолит лучше микросервисов?

Маленькие и средние проекты (стартапы).

Если важна скорость разработки.

Когда команда небольшая.

Когда приложение редко меняется.

Если нужно простое развертывание и поддержка.