Оглавление

[**Цели тестирования:** 3](#_Toc182598346)

[**ВЕРИФИКАЦИЯ и ВАЛИДАЦИЯ** 4](#_Toc182598347)

[**ПРИНЦИПЫ ТЕСТИРОВАНИЯ** 6](#_Toc182598348)

[**СИСТЕМНОЕ ТЕСТИРОВАНИЕ** 12](#_Toc182598349)

[**ТИПЫ ТЕСТИРОВАНИЯ** 15](#_Toc182598350)

[**КЛАССИФИКАЦМЯ ПО ПРИНЦИПАМ РАБОТЫ С ПРИЛОЖЕНИЯМИ** 20](#_Toc182598351)

[**ОТЧЕТЫ ПО ТЕСТРИРОВАНИЮ** 48](#_Toc182598352)

[**ТЕСТИРОВАНИЕ WEB-ПРИЛОЖЕНИЙ** 64](#_Toc182598353)

[**Кэш и куки. Разница в определениях веб-сайта, приложения, сервиса и сервера.** 75](#_Toc182598354)

[**Веб-сайты выполняют информационную функцию, веб-приложение – это уже программное обеспечение.** 75](#_Toc182598355)

**Управление качеством**

**\*QUALITY MANAGEMENT\***

***Обеспечение качества (QA – Quality Assurance)*** – это часть Quality Management - совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и поддержки ПО, предпринимаемых на разных стадиях разработки ПО, для ***обеспечения требуемого уровня качества*** выпускаемого продукта. QA – обеспечивает создание правильных процессов для получения в результате качественного продукта.

***Контроль качества (QC – Quality Control)***, свою очередь гарантируют, что процессы, установленные QA, соблюдаются. Т.е. QC – это часть QA – процесса установления стандартов и проверки, что ПО сделано правильно. Цель контроля качества – проверить, соблюдается ли предписанная модель. Это может быть достигнуто путем аудита и определения того, следовала ли команда определенной модели достижения качества.

Активности QA проходят на всем протяжении SDLC: на этапе построения, анализа и улучшения процессов, формирования различных политик, риск менеджмента.

QC – подключается на этапе составления критериев качества, метрик и способов оценки.

Тестировщик вступает уже после этапа разработки (с shift left на этапе получения тз и превращения его в спецификацию).

Основные цели тестирования как части QC являются:

* Предоставления информации о качестве элемента тестирования и любых остаточных рисках относительного того, до какой степени элемент тестирования был проверен.
* Обнаружение дефектов в элементе тестирования до его передачи в эксплуатацию
* Смягчение рисков получения продукта низкого качества заинтересованными сторонами.

***Как понять, что тетстировщик хорошо выполнил работу?***

Бытует мнение, что основная задача тестировщика – сломать продукт, однако это не так. Основная задача – выявить дефект.

Понять, что тестировщик выполнил свою работу хорошо, можно по факту выполнения следующих задач:

* Продукт проверен на соответствие требованиям
* Сведено к минимуму количество дефектов, которые обнаружит конечный пользователь
* Предоставлена отчетность по актуальному качеству продукта и любых остаточных рисках заинтересованным лицам.

***STLC – жизненный цикл тестирования.***

* ***Общее планирование и анализ требований***. Необходимо определиться, что мы будет тестировать, как мы будем тестировать. Для этого пишется тест – план и у нас появляются, какие-то требования. Существуют отдельные люди, ***бизнес-аналитики***, которые собирают требования от заказчика и приводят их в удобный вид для нас, людей, которые в будущем будут с этим работать. После этого, мы как тестировщики проводим ***анализ этих требований***, всё ли там хорошо написано (*в дальнейшем мы будем рассматривать различные техники, как можно проводить анализ требования*)
* ***Уточнение критериев приемки.*** Это показатели, по которым можно определить, что всё, наше ПО готово, мы согласились на то, что есть баги некритические, мы согласились с тем, что все требования реализованы.
* ***Уточнение стратегии тестирования.*** Тут расписываются все виды тестирований, которые мы будем проводить в рамках нашей работы (*в будущем мы будем рассматривать каждый вид тестирования*)
* ***Разработка тест-кейсов.*** Это небольшие сценарии, в которых расписано по шагам, что мы будем делать, что бы проверить ту, или иную функциональность (*тест-кейсы, чек-листы будем рассматривать далее*).
* ***Выполнение тест-кейсов.*** Запускаем наши тест-кейсы.
* ***Фиксация найденных дефектов.*** Найденные дефекты необходимо фиксировать и отправить разработчикам.
* ***Анализ результатов тестирования.*** Есть определенные метрики, по которым мы решаем, всё ли прошло хорошо.
* ***Отчетность.*** Завершающий этап, на котором оформляется отчет по результатам тестирования

|  |  |
| --- | --- |
| Тестирование | процесс в рамках жизненного цикла разработки программного обеспечения, который оценивает качество компонента или системы, а также связанных с ними рабочих продуктов. |
| Тестирование | процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта. |
| Тестирование | проверка соответствия между реальным и ожидаемым поведением программы |

# **Цели тестирования:**

1. Оценка рабочих продуктов (требования, пользовательские истории, проектирование и код)
2. Все ли указанные требования выполнены
3. Завершен ли объект тестирования и работает, как ожидают пользователи и заинтересованные лица
4. Создание уверенности в уровне качества объекта тестирования
5. Предотвращение обнаружение отказов и дефектов.
6. Предоставление заинтересованным лицам достаточной информации, позволяющей им принято обоснованные решения
7. Снижение уровня риска ненадлежащего качества программного обеспечения
8. Соблюдение договорных, правовых или нормативных требований или стандартов.

# **ВЕРИФИКАЦИЯ и ВАЛИДАЦИЯ**

**Верификация**- доказанное объективными результатами исследования подтверждение того, что определенные требования были выполнены.

**Валидация** - подтверждение путем экспертизы того, что рабочий продукт соответствует потребностям заинтересованной стороны.

Верификация – это всегда нужды заказчика. Он хочет, что бы у нас телевизор был с определенной диагональю, имел определённые каналы, можно было подключить ноутбук и т п.

Валидация – это уже нужды конечного пользователя. И тут мы говорим о том, что мы делаем правильный продукт для конкретного пользователя.

В первом случае мы проверяем, что подукт работает правильно, по требованиям которые дал нам заказчик. А во втором, мы убеждаемся в том, что конечный пользователь когда продукт получает, он думает, что всё классно в продукте, он работает, он правильный продукт для меня, для конечного пользователя.

Пример Велосипед. Велосипед есть и соответствует определнным требованиям заказчика. ( 2 колеса, цвет, размер, детали, материаллы – всё это прописано в спецификации. Всё классно и всё на месте). Но к примеру мы не учли, что целевая аудитория – это не только взрослые, но и дети. И там нужно 3 колеса. И это дело мы должны тоже учитывать.

В идеале верификацию и валидацию надо учитывать еще на моменте описания требований.

Тоже может быть и **с подпиской**. Мы даем человеку подписку на несколько сервисов, но к примеру этому человеку подписки вообще не нужны. Он может пользоваться одно, а остальное это ерунда – это тоже валидация. Функция как бы есть, прописали, что есть подписки на три сервиса, всё работает. Человек получает подписку на три сервиса, а они ему не нужн. Ему нужет только один из трех. И возможно что эта скидке будет дороже, чем покупка одного сервиса.

И мы как тестировщики можем повлиять на то, насколько у нас грамотно прописана и **верификация и валидация**.

Пример из жизни тестировщика. У нас есть форма для ввода данных. Мы можем ввести имя, email, создать пароль, можем ввести дату рождения. Это всё по сути прописано в требованиях. Как это выглядит, какой размер кнопки, какой шрифт используем, как работает интеграция с другими сервисами – это всё верификация. Мы проверили продукт, как он работает согасно требованиям.

Что касается **валидации** – мы уже должны подучать, что пользователь может сюда ввести различные символы, Различные инъекции ( о них мы будем говорить далее в разделе безопасности ) .

# **ПРИНЦИПЫ ТЕСТИРОВАНИЯ**

1. **Тестирование демонстрирует наличие дефектов а не их отсутствие**.  
   (даже если мы дефекты не нашли, мы не можем сказать, что их вообще нет.)
2. **Исчерпывающее тестирование недостижимо**. ( мы конечно будем использовать техники тест дизайна, которыми мы сможем оптимизировать наше тестирование, но сказать что мы сможем протестировать всё – нельзя!!!)
3. **Ранее тестирование сохраняет время и деньги**. (Исправить ошибку на этапе разработки требований на много дешевле, чем ошибка поступит к разработчику, он напишет код да и это еще попадет к конечному пользователю и будут критические ошибки ( медицинское оборудование, авиация) на начальных этапах лучше подключиться тестировщику, проанализировать требования, дать заключение)
4. **Кластеризация дефектов**. (В 20% ПО храниться 80% дефектов. Знание этого будет нам необходимо в будущем).
5. **Тестирование зависит от контекста**. (ПО которое отвечает за безопасность, тетстируется иначе чем web приложение для электронной комерции).
6. **Заблуждение о отсутствии ошибок**. (Отсылочка к ВАЛИДАЦИИ).
7. **Парадокс пестицида**.( Если мы вводим всегда одни и те же данные, в одно и то же поле, которое хотим протестировать ( допустим слово СОБАКА), то есть вероятно того что наша система изменится, возникнет новый баг, но мы его не сможем найти, так как необходимо вводить слово КОТ. Необходимо изключить что бы выполнялись всегда одни и те же тест-кейсы, вводились одни и те же тестовые данные, даже тесировали одни и те же ЛЮДИ. Это незначит что людей надо увольнять, их надо просто менять)

МОДЕЛИ РАЗРАБОТКИ ПО

**ЖИЗНЕНЫЙ ЦИКЛ РАЗРАБОТКИ ПО  
SDLS**

(software development lifecycle)

**Жизненный цикл разработки ПО** – активности, выполняемые на каждом этапе разработки программного обеспечения и то, как они соотносятся друг с другом в логическом и хронологическом порядке.

По сути это некие фазы разработки, которые идут друг за другом

Представим ситуацию, что мы хотим разработать некий продукт. И всё начинается с этапа сбора **ИДЕЙ.** Сама идея начинает обрастать **ТРЕБОВАНИЯМИ**. Далее наступает фаза **ДИЗАЙНА** ( но это не тот дизайн, который мы привыкли понимать как рисование некоего шаблона и разработка того, что будет видеть конечный пользователь). Наш ДИЗАЙН – это больше архитектура приложения, из каких составных частей будет оно состоять – база данных, сервер, клиент, условно будет использовано такое-то ПО для работы. Т.е. это техническая составляющая и на проектах она называется АРХИТЕКРУТА, и под нашим ДИЗАЙНОМ пониматься именно это. А обычный **дизайн** – это раздел ТРЕБОВАНИЙ. После уже наступает **РАЗРАБОТКА** ( тут трудятся разработчик). После наступает **ТЕСТИРОВАНИЕ.** После идет процесс **РАЗВЕРТЫВАНИЯ** ( наше ПО должно где-то разместиться, что бы на нем могли работать другие пользователи, либо другие члены команды. У нас есть несколько сред работы, о них будет речь идти позже. В одной среде трудятся разработчики, в другой тестировщики. После идет **ПОДДЕРЖКА** (если у пользователя возникают сложности, либо дефекты со стороны клиента). И когда проект заканчивается, у проекта мало пользователей, то происходит **ЗАКРЫТИЕ**.

Но по сути тестировщик в данной схеме подключается раньше чем на этапе ТЕСТИРОВАНИЯ. Он может работать на всех этапах жизненного цикла. Он может подключаться на этапе сбора идей. Можем анализировать и тестировать требования. В ДИЗАЙНЕ – мы можем влиять на то, как будет выглядить наша финальная архитектура продукта. В РАЗРАБОТКЕ тестировщик может анализировать код и проводить ревью кода. Да и сам разработчик может консультироваться с тестировщиком. РАЗВЕРТЫВАЕНИЕ – мы можем управлять релизом, мы можем сообщать что релиз будет или нет, что есть какие-то критические баги, можем запрещать релиз или разрешить. В ПОДДЕРЖКЕ тоже могут быть тестировщики. Есть компании в которые есть отделы поддержки, где сидят тестировщики. В ЗАКРЫТИИ тестировщики тоже применяются. Они должны прописать документацию, отчеты.

**МОДЕЛИ РАЗРАБОТКИ ПО**

Модель разработки ПО (Software Development Model) – структура, систематизирующая различные виды проектной деятельности, их взаимодействие и последовательность в процессе разработки ПО.

Если SDLS это общепринятая модель, как это может быть. То тут мы уже можем, исходя из конкретной модели разработки, управлять SDLS в каком-то специфическом аспекте.

**ВОДОПАДНАЯ МОДЕЛЬ (Waterfall)**

Является, самой старой и самой строгой моделью разработки. Либо ее называют КАСКАДНОЙ моделью. Мы видим, что SDLS повторяется, после окончания одного этапа мы переходим на следующий этап, **но не можем возвращаться на предыдущий**. В таких системах надо понимать, что бы требования были написаны ИДЕАЛЬНО. Данная модель встречается в гос. структурах, старых структурах. Но мы в повседневной деятельности с подобной моделью будем сталкиваться редко. Сейчас в основном работают с гибкими методологиями.

**V ОБРАЗНАЯ МОДЕЛЬ(V-model)**

В данной модели у нас те же SDLS процессы, но тестирование уже появляется с самого начала и оно есть на каждом из этапов жизненного цикла разработки. Данная модель уже более продвинутая чем waterfall, но уже не является самой распространённой.

**ИТЕРАЦИОННАЯ ИНКРЕМЕНТАЛЬНАЯ МОДЕЛЬ**

В рамках современности встречается чаще всего.

Либо отдельно итерационные либо инкрементальные модели.  
С точки зрения жизненного цикла, модель является **ИТЕРАЦИОННОЙ**, так как подразумевает многократное повторение одних и тех же стадий.

С точки зрения развития продукта, у нас был продукт с один набором полезных функций, мы что-то разработали, добавили. Еще что-то разработали, добавили и у нас нарастает как снежный ком все функции и модель называется **ИНКРЕМЕНТАЛЬНОЙ.**

ИТЕРАЦИЯ – повторение в определенный промежуток времени. Итерация длится от 2 до 4 недель. И мы за 2 недели сделали этап разработки, тестирования и далее переходим на следующий этап.

Эти модели чаще всего идут вместе и итерационная и инкрементальная.

Как всегда проходит «общее планирование» и далее запускается SDLS процесс. Прошло несколько итераций, мы получаем инкремент и в конце проекта делаем отчетность.

Существует так же еще **СПИРАЛЬНАЯ** модель разработки. В спиральной модели помимо того, что рассмотрено в ранее представленных моделях, проводится анализ рисков (риски связанные с нехваткой времени, связанные с тем, что продукт устарел, используются устаревшие технологии и т.п.).

Но чаще всего на практике встречается итерационно инкрементальная модель.

ГИБКАЯ МОДЕЛЬ (agile)

Все **ИТЕРАЦИОННЫЕ ИНКРИМЕНТАЛЬНЫЕ МОДЕЛИ** называются **ГИБКИМИ и** связанно это с тем что в них есть **agile.**

Существует Agile Manifesto он был подписан в начале 2000-х годов. Люди собрались и обсудили, какие есть на данный момент сложности в рзаработке и приняли манифест.

1. Люди и взаимодействие важнее процессов и инструментов (нам не важно какие мы используем процессы, какие у нас есть инструменты, самое главное, что бы людям было хорошо)
2. Работающий продукт важнее исчерпывающей документации (не важно насколько мы напишем подробную документацию, если продукт не будет работать)
3. Сотрудничество с заказчиком важнее согласования условий контракта (контракт есть, в нем что-то описано, но самое важное это хорошая химия между заказчиком и команды разработки)
4. Готовность к изменению важнее следования первоначальному плану. (в agile могут меняться требования, и к этому надо относится нормально).

Приведенная схема ГИБКОЙ МОДЕЛИ относится больше к SCRUM.

У нас есть некий **«Бэклок» проекта**, набор задач и идей, которые мы хотим реализовать в рамках нашего проекта.

Но у нас есть и **СПРИНТЫ,** мы не можем по сути работать в течении одного года в одном направлении, а должны разбить это время на части. Эти части называются спринтами, они длятся от 2 до 4 недель. И тут унас используется **Бэклог спринта.** Т.е. берутся задачи из Бэклога проекта которые мы хотим реализоваться в течении наших 4 недель, и формируем Бэглог спринта и запускаем наш спринт.

Раз в сутки проводится небольшое собрание (пятиминутка), где обсуждается, что делали за прошлый день, за этот день, что будем делать в будущем. Обсуждаем, возможно есть ограничения, которые не дают нам что-то делать, не написан код, не написан тест.

В результате наших спринтов происходит разработка, получаем ИНКРИМЕНТ и мы в рамках Результата возвращаем его заказчику.

УРОВНИ ТЕСТИРОВАНИЯ

Так как материал по этой теме обширный, сейчас мы рассмотрим какие виды тестирования есть, а потом на примере конкретного ресурса, мы рассмотрим, какие виды тестирования применяются.

**Уровни тестирования** – это группы активностей тестирования, которые организуются и управляются как единое целое.

Т.е. у нас есть определенный уровень, самый первый, потом появляется второй, третий, четвертый и на каждом из этих уровней, мы что-то можем делать и эти уровни растут.

В начале у нас есть код, потом появляются взаимодействия внутри этого кода, потом эти взаимодейтсвия преобразуются в то, что мы видем на экране и появляется frontend и на каждом из уровней мы проводим свои активности.

На каждую функцию в коде пишут **КОМПОНЕНТНЫЙ ТЕСТ**, Так называемый **UNIT тест**.

Тестируются: Компоненты, модули, код и структура данных, классы, модули БД. (Исполнитель – чаще всего **разработчик**)

Следующий уровень – это **ИНТЕГРАЦИОННОЕ ТЕСИРОВАНИЕ**

Интеграционное тестирование – уровень тестирования, который фокусируется на взаимодействии между компонентами или системами. (Очень часто будет идти речь о интеграционном тестировании, при тестировании API интерфейса – это будет взаимодействие между системами)

Тестируются: Подсистемы, базы данных, интерфейсы, программные интерфейсы, приложения (API), микро сервисы. (Исполнитель – если компонентное тестирование – разработчик. Системное тестирование – тестировщик)

3 типа ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ:

1. Снизу вверх (Bottom Up) При тестировании низкоуровневых модулей, и если чего-то не хватает, еще не разработано, а надо протестировать интеграцию, используются **заглушки (некий псевдокод).**
2. Сверху вниз (Top Down) В начале тестируются все высокоуровневые модули, если чего-то не хватает используют **драйвера.**
3. Большой взрыв (Big Bang) Когда объединяется всё вместе и тестируется.

# **СИСТЕМНОЕ ТЕСТИРОВАНИЕ**

Системное тестирование – уровень тестирования, который фокусируется на проверке соответствия системы в целом определенным требованиям.

Тестируется: Приложения, аппаратные и программные системы, тестируемая система, операционные системы, конфигурация системы, конфигурация данных. (Исполнитель – независимая группа тестировщиков)

**ПРИЕМОЧНОЕ ТЕСТИРОВАНИЕ**

Приемочное тестирование – уровень тестирования, целю которого является определение того, принимать ли систему. (Исполнитель – Клиенты, бизнес-пользователи, владельцы продуктов и операторы системы)

ПРИЕМОЧНОЕ ТЕСТИРОВАНИЕ:

1. Пользовательское ПТ – к примеру мы разрабатываем код для бухгалтеров компании, собираются бухгалтера и тестируют ПО.
2. Эксплуатационное ПТ – подключаются системные администраторы, создают среду эксплуатации и в ней тестируют.
3. Контрактное ПТ – когда заключен контракт и согласно контракту, определены критерии приёмки. На основании которых проводится тестирование.
4. Нормативное ПТ – проводится в соответствии с нормативами. Тесть существует ГОСТ и мы согласно ему проводит тестирование.
5. Альфа-тестирование – проводится на мощностях разработчика, но не командой самого разработчика.
6. Бета-тестирование – проводится не на мощностях компании, а его проводится выделенная группа конечных пользователей.

ВИДЫ ИНТЕРФЕЙСОВ

Определим, для чего создан тот или иной интерфейс.

Интерфейс – это способ взаимодействия.

Существуют 3 основных типа интерфейса.

1. Пользовательский интерфейс (User interface – UI). Это непосредственно взаимодействие пользователя с программой. У нас есть приложение, наша программа, в которую мы заходим. Это может быть браузер, может быть какая либо игра, это может быть операционная система и то, что мы видим на экране называется **Graphical User Interface (GUI)**. Т.е. GUI – это разновидность пользовательского интерфейса в виде картинок, кнопок, каких либо графических компонентов которые мы видим на экране. Помимо GUI может так же быть и другие виды пользовательского интерфейса. **Тактильные, Визуальные и Звуковые.** Иногда вам приходится слышать помимо User Interface еще UX - Us**er EXPERIANS**. Некоторые думают что UX – это тоже тип интерфейса, но это не так. По сути это опыт, который получает пользователь при взаимодействии с сайтом, или приложением. Включает в себя такие параметры, **насколько этим удобно пользоваться**, **насколько мы быстро обучаемся**, насколько мы понимаем, что в определенном месте, нам надо нажать на какую либо кнопку. Насколько в целом приложение выглядит красиво.
2. Интерфейс командной строки (Command Line Interface – CLI) На заре возникновения компьютеров и операционных систем, не было GUI, была только командная строка. CLI есть всегда, даже в системах которые используют GUI. Если мы говорим про Windows мы чаще обращаемся к GUI, если мы говорим про Ubuntu – мы первоначально работаем с командной строкой, а GUI в большей степени является надстройкой. В командной строке как выходит из названия, используются команды, которые позволяют открыть определенные папки, просмотреть файлы, возможно редактировать какой либо текст. CLI нам может пригодиться при тестировании web приложений со стороны backend.
3. Программный интерфейс приложения (Application Programming Interface – API) Это интерфейс создан для программ. Разработчик прописывает логику взаимодействий программ между собой. К примеру мы заходим на какой либо ресурс. И через смартфон мы хотим поделиться ресурсом в каком либо меседжере, в таком случае мы нажимаем на кнопочку на GUI и сразу подключается API нашей внутренней системы, к примеру системы Android и появляется отдельное окошко, что мы хотим поделиться и куда мы хотим отправить картинку или профиль. Мы хотим поделиться в телеграмме, Есть API телеграмм, который постоянно слушает, что происходит в телефоне и ловит этот сигнал. Так программы взаимодействуют между собой. В операционной системе ситуация может быть следующая, мы играем в какую либо игру и в какой-то момент, игра понимает, что ей не хватает памяти и она обращается через API к системе и просит выделить ей оперативную память. Когда мы говорим о web Ресусрах у нас есть какой либо общий агрегатор новостей, он обращается к другим ресурсам ( к примеру aviasales) и есть локальные компании, которые продают свои билеты. И aviasales при помощи API стучится к другим ресурсами и с помощью API информация из них передается в aviasales.

# **ТИПЫ ТЕСТИРОВАНИЯ**

Разберем типы тестирования, где-то их называют видами тестрирования, где-то классами тестирования.

Типы тестирования – это совокупность активностей тестирования, направленных на тестирования заданных характеристик системы или ее части, основываясь на конкретных целях.

ЦЕЛИ.

1. Оценка функциональных характеристик качества системы, таких как полнота, корректность, целесообразность. ( ЧТО ПРИЛОЖЕНИЕ ДЕЛАЕТ) это к примеру протестировать возможность добавить фотографию и т.п.
2. Оценка нефункциональных характеристик качества, таких как надёжность, продуктивность работы, безопасность, совместимость, удобство использования. (КАК ПРИЛОЖЕНИЕ ЭТО ДЕЛАЕТ)
3. Оценка правильности, полноты структуры или архитектуры компонента или системы и их соответствие спецификацией.
4. Оценка влияния изменений например, подверждение того, что дефекты были исправлены(подтверждающее тестирование) и поиск непреднамеренных изменений в поведении, вызванных изменением в программном обеспечении или окружение (**РЕГРИСИСОННОЕ ТЕСТИРОВАЕНИЕ**) .

**ФУНКЦИОНАЛЬНОЕ И НЕФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ**

Функциональное тестировование – тесты по оценке функций. Этот вид тестирования проводится абсолютно на всех уровнях.

Нефункциональное тестирование – выполняется для оценки характеристик, удобства использования, производительность, безопасность. Данный вид тестирования так же проводится на всех уровнях.

МЕТОДЫ ТЕСТИРОВАНИЯ

1. Тестирование черного ящика – У тестироващика нет доступа к коду. И мы можем только наблюдать frontend. Это больше работа начинающих тестировщиков.
2. Тестирование серого ящика - Если мы можем читать код, видеть сторонние приложения. Тестирование API, видеть что уходит на сервер и приходит с сервреа. Backend тестирование.
3. Тестирование белого ящика - Мы видим программный код, знаем архитектуру. Но это уже ближе к автоматизации тестирования. Ручное тестирование обычно на черном и сером ящике.

ТЕСТИРОВАНИЯ СВЯЗАННЫЕ С ИЗМЕНЕНИЯМИ.

1. **Подтверждающее тестирование** (re-test) – после того как дефект исправлен, программное обеспечение может быть протестировано с использованием тех же тестовых сценариев, которые завершились с ошибкой при найденном дефекте.( Когда мы исправляем баг, нам необходимо протестировать то, что он был исправлен.)
2. **Регрессионное тестирование** (regression testing) – тип тестирования связанного с изменениями, что бы найти привнесенные или ранее обнаруженные дефекты в неменяющихся областях программного обеспечения ( К примеру у нас был баг, из за того, что слетела функция «Забыли пароль» и часть этой функции была прописана в коде. Разработчик поменял код в этой функции «забыли пароль», но есть вероятность из за того что это был один модуль, но там были и друге функции, как создать пароль, залогиниться в систему. Нужно проверить, что бы другие функции никак не сломались, после того как мы пофиксилсли баг, либо добавили функциональность).

Возникает закономерный вопрос. Чем отличается re-test от регрессионного тестирования ? re-test – это когда баг пофиксили и мы посмотрели, что всё работает хорошо в конкретно в той функции в которой был пофикшен баг. Когда мы говорим про регрисионное тестирование, нам мало проверить то, что этот быг был пофикшен. Нам надо проверить, что это изменение не повлияло на другие функции в этом приложении, которые связанны с ним.

Есть 3 вида регрессии:

1. Регрессия багов – попытка доказать, что исправленная ошибка на самом деле не исправлена;
2. Регрессия старых багов – попытка доказать, что недавнее изменение кода или данных сломало исправления старых ошибок, т.е. старые баги стали снова воспроизводиться.
3. Регрессия побочного эффекта – попытка доказать, что надевание изменения кода или данных сломало другие части разрабатываемого приложения.

КАК ФОРМИРОВАТЬ РЕГРЕССИОННЫЕ НАБОРЫ

Так как мы часто будем работать с регрессионным тестированием, надо понимать, что мы не можем тестировать абсолютно всё. **Помните! Исчерпывающее тестирование невозможно**! Нет смысла нам вторично, если мы проводили тест ранее, все тест кейсы, чек-листы запускать опять. Для этого у нас есть подходы, как правильно сформировать регрессионный набор.

1. Высокоприоритетные задачи (Если у нас пришло какое-то изменение и оно завязано с приоритетной задачей. К примеру login в систему и мы что-то поменяли, нам обязательно надо проверить, что бы не сломалось всё что связанно с логином в систему.
2. Тестирование функций и модулей, которые чаще всего используются пользователями.
3. Функции и модули, в которых часто и регулярно обнаруживаются дефекты
4. Тестирование функций и модулей, которые связаны с изменением.

**Так же часто регрессионное тестирование автоматизируется**. Выделяются специально определенные дни на регрессию, так как это большое количество тест-кейсов которые необходимо пройти.

КЛАССИФИКАЦИЯ ПО ЗАПУСКУ КОДА НА ИСПОЛНЕНИЕ

1. Статическое тестирование – тестирование, которое не предполагает выполнение тестируемого компонента или системы. Примеры: тестирование документации, прототипов , кода (в рамкам code review), тестовых данных.
2. Динамическое тестирование – тестирование, проводимое во время выполнения тестируемого элемента. Пример: проверка реального поведения приложения при запуске кода на разных уровнях тестирования.

Многие говорят, что автоматизация заменит ручное тестирование, мол уже принимает участие искусственный интеллект, но такого не будет, так как есть часть функций, к примеру удобство использования, когда проверяется, на сколько удобно пользоваться человеку приложением, графический интерфейс никогда не проверить автоматизацией, AAA игры не смогут обойтись без ручного тестирования. С другой стороны вы должны понимать, что сейчас большой тренд на специалистов широкого профиля, которые умеют и писать код – заниматься автоматизацией и проводить ручное тестирование. Некоторые атомизаторы не пишут даже тест-кейсы а автоматизируют чужие.

ЧТО ЧАЩЕ ВСЕГО АВТОМАТИЗИРУЮТ

Все рутинные, важные и время затратные операции

1. Дымовое тестирование – это самый первый тест, который мы проводим в рамках нашего тестирования, мы должны проверить, что базовые, отправные функции нашего приложения работают (более подробно о дымовом тестировании далее). И так как этот тест самый первый и он проводится на каждом билде, на любой промежуточной версии продукта, следовательно этих тестов много и его целесообразно автоматизировать.
2. Регрессионное тестирование – регирисионных тестов так же очень много и их целесообразно автоматизировать.
3. Сквозное тестирование (end-to-end testing, E2E) – это вид тестирования, используемый для проверки программного обеспечения с начала до конца. Тут мы проверяем весь поток, от того как пользователь зарегистрировался в системе, до того как он купил какой либо продукт. Это один большой тест, зарегистрироваться, залогиниться, войти в профиль, найти товар, добавить в корзину, оплатить, оформить доставку – это и будет E2E тест.

ТЕСТИРОВАНИЕ ПО СТЕПЕНИ ВАЖНОСТИ

**Дымовое тестирование (smoke test) –** Набор тестов, который покрывает основные функциональные возможности компонента или системы, что бы определить, работает ли компонент или система должным образом до начала запланированного тестирования.( Условно пришел новый билд и мы проводим это тестирование. Мы проверяем, что можем зарегистрироваться, залогиниться, можем что либо купить. Если всё проходит, мы сообщаем что берем билд в тестирование. Если хотя бы одна функция не работает, мы сообщаем о необходимости их восстановить и только тогда сможем проводить дальнейшие тесты.

**Тестирование критического пути (critical path test)** – Направлено на исследование функциональности, используемой типичными пользователями в типичной повседневной деятельности. (Условно, это всё, что мы будем делать в рамках нашего тестирования. Есть требования, в которых описаны функции которые должно выполнять приложение, мы их и тестируем согласно этим требованиям.

**Расширенное тестирование** – Направлено на исследование всей заявленной в требованиях функциональности – даже той, которая низко проранжирована по степени важности.(Есть некие приоритеты с точки зрения функциональности, которые не так важны и иногда тестировщик может ими пожертвовать, если понимает, что не хватает время на тестирование. Но лучше тестировать конечно всё.

- Нетипичные, маловероятные, экзотические случаи и сценарии использования функций (Есть к примеру форма регистрации. И мы тужа можем ввести логин и пароль. Мы ввели правильные данные ( пароль должен содержать 6 символов, спец символы, больше буквы, числа и даже мы проверили полное несоответствие ( негативный тест - будет рассмотрен далее). Мы ввели туда SQL инъекцию – вредоносный участок кода.) Для этого тоже есть отдельный вид тестирования, который позволяет бороться с этим (он тоже будет рассмотрен далее по курсу)

**Т.Е. В ПЕРВУЮ ОЧЕРЕДЬ ПРОВОДЯТСЯ ДЫМОВОЕ И ТЕСТИРОВАНИЕ КРИТИЧЕСКОГО ПУТИ** а РАСШИРЕННОЕ если у тетсировщика есть на это время.

# **КЛАССИФИКАЦМЯ ПО ПРИНЦИПАМ РАБОТЫ С ПРИЛОЖЕНИЯМИ**

**Позитивное тестирование** – исследование приложения в ситуации, когда все действия выполняются по инструкции без каких бы то ни было ошибок, отклонений, ввода неверных данных. (Введи валидный правильный логин, валидный правильный пароль, зарегистрировались)

**Негативное тестирование** – направлено на исследование работы приложения в ситуациях, когда с ним выполняются некоторые (некорректные) операции и/или используются данные, потенциально приводящие к ошибкам.(Начинаем проводить проверки приводящие к ошибкам, как результат должно появиться валидационное сообщение)

**Деструктивное тестирование** – одна из форм негативного тестирования с целью нарушить работоспособность приложения и обнаружить точку отказа.

**ВСЕГДА НАЧИНАЙТЕ С ПОЗИТИВНОГО ТЕСТИРОВАНИЯ!!!**

Почему это важно? Самое главное для бизнеса, что бы ваше приложение работало. Нужно убедиться в том, что если мы ведем всё корректно приложение сработает.

**НЕЛЬЗЯ ОБЪЕДИНЯТЬ МЕЖДУ СОБОЙ НЕГАТИВНЫЕ ТЕСТЫ!!!**

КЛАССИФИКАЦИЯ ПО СТЕПЕНИ ФОРМАЛИЗАЦИИ

Условно у нас есть разная тестовая документация и тестирование может быть так же привязана к ней. Некая степень бюрократизации. Чаще всего мы будем работать в направлении тестирования на основе тест-кейсов и чек-листов.

1. **Тестирование на основе тест-кейсов** – формализованный подход в котором тестирование производится на основе заранее подготовленных тест-кейсов, наборов тест-кейсов и иной документации. ( Мы будем разбирать и составлять тест-кейсы и чек-листы. И тестирование будет проходить по уже заготовленному сценирию)
2. **Исследовательское тестирование** – частично формализованный подход, в рамках которого тестировщик выполняет работу с приложением по выбранному сценарию, который, в свою очередь дорабатывается в процессе выполнения с целью более полного исследования приложения. (Берем приложение и на основании своего опыта начинаем тестирование. Возможно в голове есть какой либо чек-лист, либо набросали чек-лист на бумаге. Но надо понимать что это не финальный тест-лист а просто набор идей) Когда мы приходим на проект, мы используем этот метод, пока нет официальной документации. Открываем приложение и начинаем смотреть как оно работать.
3. **Свободное (интуитивное) тестирование** – полностью неформализованный подход, в котором не предполагается использование ни такс-кейсов, ни чек-листов, ни сценариев. Тестировщик полностью опирается на свой профессионализм и интуицию для спонтанного выполнения с приложением действий, которые, как он считает, могут обнаружить ошибку.( Мы просто открываем приложение, у нас нет ни плана ни документации и начинаем что либо тестировать)

КЛАССИФИКАЦИЯ ПО ЦЕЛЯМ И ЗАДАЧАМ

**Инсталляционное тестирование** – тестирование направленное на выявление дефектов, влияющих на протекание стадии инсталяции(установки) приложения:

* Установка ПО.
* Удаление ПО.
* Обновление ПО.
* Откат на предыдущую версию.
* Повторный запуск установки после возникновения ошибки и исправления уже известных проблем.
* Автоматическая установка.
* Установка отдельного компонента из общего пакета программ

**Тестирование удобства пользователя** – тестирование направленное на исследование того, насколько конечному пользователю понятно, как работать с продуктом, а так же насколько ему нравится использовать продукт.

- Общая доступность.

- Скорость, производительность.

- Удобство навигации и интерфейс.

- Плавность.

**Тестирование доступности (accessibility testing, A11Y)** – тестирование, направленное на исследование пригодности продукта к использованию людьми с ограниченными возможностями.

- Использование вспомогательных технологий в ПО (распознание речи, экранная клавиатура и лупа, скринридеры).

- Возможность использовать приложение одной рукой.

- Настройки специальной цветопередачи.

- Наличие понятных инструкций и руководство пользователя.

**Тестирование безопасности** – тестирование, направленное на проверку способности приложения противостоять злонамеренным попыткам получения доступа к данным и функциям, права на доступ к которым у злоумышленника нет.

- SQL инъекции. (различные SQL Запросы которые позволяют обратиться к базе данных)

- XSS инъекции.(написанный на языка JS специальный скрипт, который позволяет получать данные, когда мы вводим в поле. Появляется окошко, в которое планируется, что мы введем свои личные данные)

- Перехват трафика.(есть специальные программные средства, и они так же будут нами рассмотрены далее, но мы будем их применять в нормальных целях, которые перехватывают, что мы передаем на сервер.)

- Брутфорсинг (полный перебор данных для доступа)

**Тестирование интернационализации** (internationalization testing, i18n test)– тестирование, направленное на проверку готовности продукта к работе с использованием различных языков и с учетом различных национальных и культурных особенностей.(Это проверка того, насколько наша программа в будущем будет готова выходить на различные локали, различные страны. Мы как разработчики понимаем, что в будущем наша программа будет работать допустим на Китайском рынке или азиатском. Условно сейчас у нас нет необходимости использовать эту функцию в приложении, но было бы хорошо, если в код была бы внесена возможность в будущем в поля ввода водить иероглифы. **МЫ НА БУДУЩЕЕ ПРЕДСУМАТРИВАЕМ ВОЗМОЖНОСТЬ ВЫХОДА НА ДРУГИЕ ЛОКАЛИ**)

**Тестирование локализации** (localization testing, l10n)– тестироваение, направленное на проверку корректности и качества адаптации продукта к использованию на том или ином языке с учетом национальных и культурных особенностей.( В данном случае, мы уже вышли на определенный рынок, мы уже перевели какие либо данные на странице. Мы уже учли особенности языка, форматы валют. И мы тестируем конкретно для этой локли наши приложение)

Для подобного тестирования, знать иностранные языки необязательно, есть специальные аналитики, технические писатели, которые напишут словарики в которых расписано, что мы конкретно будем тестировать.

**Тестирование совместимости** – тестирование, направленное на проверку способности приложения работать в указанном окружении.

- Конфигурационное тестирование (Совместимость с аппаратной платформой, опеационной системой и сетевой инфраструктурой)(проверим как отрабатывает процессор с приложением, как отрабатывает винда)

- Кросс-браузерное тестировение – Совместимость с браузерами и их версиями(Это не значит, что вам абсолютно всегда надо будет открывать три разных браузера. В команде тестирования несколько человек и можно разделить. Это так же позволит избежать парадокса Пестицида.)

- Кросс-платформерное тестирование – проверим как работает приложение на Андроид на IoC.

**Тестирование надежности** – тестирование способности приложения выполнять свои функции в заданных условиях на протяжении заданного времени или заданного количества операций. ( Мы задаем какое-то время, допустим один час , ставим условие работы к примеру 10 пользователей и смотрим хорошо ли работает система).

**Тестирование восстанавливаемости** – тестирование способности приложения восстанавливать свои функции и заданный уровень производительности, а так же восстанавливать данные в случае возникновения критической ситуации, приводящей к временной(частичной) утрате работоспособности приложения. (Допустим отключили свет, и нам надо посмотреть, насколько приложения себя быстро восстановит) к примеру отрубили свет и у нас word открыт, и при входе появляется сообщение, что была восстановлена промежуточная время докумнета)

**Тестирование отказоустойчивости** – тестирование заключается в эмуляции или реальном создании критической ситуации с целью проверки способности приложения задействовать соответствующее механизмы, предотвращающие нарушение работоспособности, производительности и повреждения данных.

**Тестирование производительности** – исследование показателей скорости реакции приложения на внешнее воздействие при различной по характеру интенсивности нагрузки.

- Нагрузочное тестирование.

- Тестирование масштабируемости.

- Объемное тестирование.

- Стрессовое тестирование.

- Конкурентное тестирование.

***Нагрузочное тестирование*** – исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором повышении этих пределов. ( Условно мы знаем, что системой могут без проблем пользоваться 10 человек, мы их добавляем и смотрим как приложение работает, после добавляем 11 12 13 14 и смотрим как приложение будет реагировать).

***Тестирование масштабируемости*** – исследование способности приложения увеличивать показатели производительности в соответствии с увеличением количества доступных приложению ресурсов (Мы работаем на одном сервер и понимаем что если мы добавим несколько серверов, то будем получать больше пользователей. И мы сморим как показатели производительности меняются, когда мы увеличиваем количество ресурсов)

***Объемное тестирование*** – исследование производительности приложения при обработке различных (как правило больших) объемов данных.(Загружаем большой файл в систему, условно на 100Гб и посмотреть, что произойдет)

***Стрессовое тестирование*** – исследование поведения приложения при нештатных изменениях нагрузки, значительно превышающих расчетный уровень, или в ситуациях недоступности значительной части необходимых приложению ресурсов. (Добавить вместо 10 , 20 пользователей и посмотреть как будет работать приложение).

***Конкурентное тестирование*** – исследование поведения приложения в ситуации, когда ему приходится обрабатывать большое количество одновременно поступающих запросов, что вызывает конкуренцию между запросами за ресурсы. (Обращаясь одновременно к базе данных, памяти, дискам, мы смотрим как приложение раставит приоритеты и как будет работать).

ПРИМЕР

Давайте определи какие виды тестирования можно использовать при тестировании саййта aliexpress

1. Это web приложение. Следовательно оно работает в разных браузерах, но надо опираться на то, что есть в требованиях. Если наши маркетологи, аналитики, заказчики хотят что бы приложение работало в конкретных браузерах, то нет особого смысла тестировать их абсолютно во всех. **Кросс-браузерное тестирование.**
2. **~~Кросс-платформерное тестировани~~**е тут работает поскольку постольку. Если это было бы десктопное приложение, то тогда мы бы тестировали его на разных ОС.
3. Стоит отметить, что это приложение мы можем установить на разные мобильные устройства Будет проводиться **мобильное тестирование и кросс-патформерное тестирование.**
4. Само собой, когда мы будем работать с приложением у нас будут **smoke test, Critical path text,** и нам надо определиться, что внеси в эти виды тестирования.   
   - Так как есть кнопка Войти – мы будем тестировать вход в приложение и регистрацию, добавление товаров к корзину, покупка т.е. это всё можно покрыть в рамках **smoke test.** Т.е. нам надо определить основные функции прложения, переговорить с командой и определить, что вносить в smoke test.  
   - Если будут появляться новые функции в приложении – это уже будет **critical path,** на это будут писаться отдельные тест кейсы и отдельные проверки.
5. В виду того, что будут какие либо изменения, нам надо проводить **Регрессионное тестирование.** Когда мы определимся с приоритетами, логин, добавление товара, навигация на сайте, покупка товара это всё будет вноситься в регрессионные наборы.
6. Мы можем перейти на страницу других стан, к примеру Турция. Тут можно выбрать, страну, язык, валюту. **Тестирование локализации**. Тестируются переводы, как переведены сами картинки, описание товаров, правильно отображается цена(валюта). Перевод идет автоматически, и если мы видим описание товара на непонятном языке, то это не баг ☺(это встроенный переводчик, это особенность платформы). Баг может быть в том, что есть ПОЛИТИКА БЕЗОПАСНОСТИ – вот она должна быть на том языке, который нас интересует.
7. Так как это ресурс огромный, на нем может быть много людей, необходимо проводить **тест производительности**. Учитывая то, сколько людей может его использовать, **стрессовое тестирование, нагрузочное тестирование.**
8. Так как много пользователей, много личной информации необходимо проводить **тестирование безопасности**.
9. **Тестирование интерфейса**, само собой, куда без него, так как у нас GUI. Мы можем изменить размер экрана, посмотреть как срабатывает адаптивная верстка. Смотрим как ведет себя приложение на разных устройствах, меняется ли его разрешение.
10. Определяем тип приложения
11. Определяем где его можно использоваться, барузеры, системы, мобильные платформы
12. Локализация
13. Интерфейс. Мы можем разделить приложение на модули (не модули в коде) а на шапку, на каталог перечни категорий, на центральные баннеры, на каталог товаров, на модуль оплаты, модуль регистрации, отдельный модуль логина.
14. Интеграционное тестирование API Так же проводиться

АНАЛИЗ ТРЕБОВАНИЙ

Любое тестирование начинается с анализа требований. Это первый этап, когда подключается тестировщик, если не считать тот, когда идет сбор идей, но очень редко тестировщик подключается на этом этапе.

**Требование** – условие которое включает обязательные для выполнения критерии.

Мы описываем в требованиях то, что мы хотим видеть на нашем экране, чир мы хотим видеть от приложения, какая бизнес логика заключается внутри нашего ПО. И в зависимости от требований, тестировщик пишет те проверки, которые будет выполнять в будущем.

У нас есть два вида документации. **Продуктная и проектаня**.

Продуктная документация – используется проектной командой во время разработки и поддержки продукта (мы будем с ней часще всего работать)

* План проекта, тест-план (Со стороны тестировщка, мы описываем, что мы будем тестировать)
* Требования к продукту (PRD) и функциональные спецификации(FSD, SRS) ( Тут описано как будет выглядеть наш продукт и какие функции он должен выполнять)
* Архитектура и дизайн (Тут описана структура нашего приложения, какая в нем база данных, какой сервер, какой клиент, какие дополнительные сервисы подключаются)
* Тест-кейсы и тестовые наборы (Это уже документация тестировщика. Тут мы пишем, как мы будем проводить тестирование в виде кейсов, и эти кейсы будут еще объединяться в наборы) Об это будет далее более подробно рассмотрено
* Технические спецификации (Это больше информация для технических специалистов)

Проектная документация – используется не только на стадии разработки, но и на более ранних и поздних стадиях (например стадии внедрения и эксплуатации)

* Продуктная документация
* Пользовательская и сопроводительная документация (Инструкции, мануалы)
* Маркетинговая документация (вещи которые помогают продвигать продукт)

ИСТОЧНИКИ И ПУТИ ВЫЯВЛЕНИЯ ТРЕБОВАНИЙ

**Интервью** – мы конкретно общаемся с заказчиком / стейкхолдерами. Бизнес-аналитик собирает эту информацию и переводит в нормальные требования которые понятны всем участникам проекта.

**Работа с фокусными группами** – Собираются условно 20 представителей бизнеса или наших конечных пользователей, и они могут проанализировать наш продукт, выдать свои идеи которые нам так де помогут получить информацию о конечных требованиях.

**Анкетирование** – Создание анкет.

**Наблюдение** – кода мы приходим в офис компании, там сидят юристы, для которых мы разрабатываем ПО. Возможно у них уже что-то разработано, и мы следим за тем, что они выполняют в течении рабочего дня. Какие функции в приложении им необходимы.

**Семинары и мозговые штурмы** – Собирается некоторое количество людей, и накидывают идем, которые они хотят видеть в приложении.

**Прототипирование** – Условно мы создаем прототип, либо обращаем внимание на ПО наших коллег (другие приложения) и мы понимаем, что мы хотим видеть.

**Анализ документов** – мы можем анализировать какие-то дополнительные документы, возможно анализ законодательных норм, стандартов.

**Моделирование процессов и взаимодействий**. – мы можем что-то после наблюдения, интервью, понимая того, что мы хотим видеть в нашем продукте, мы можем нарисовать модель и нарисовать процессы, которые мы хотим открыть в рамках нашего продукта.

**Самостоятельное описание –** По аналогии понимая как должен работать продукт, мы можем написать своё виденье, как он должен работать.

Но всегда помните, что у нас есть заказчик, всегда нужно консультироваться с ним, что бы наши требования были в рамках того, что хочет видеть наш конечный клиент.

УРОВНИ И ТИПЫ ТРЕБОВАНИЙ

Выделяют 3 уровня:

1 – **Уровень бизнес-требований**. На этом уровне мы определяем, что приложение может сделать полезное для бизнеса. ( К примеру повысить количество конечных пользователей, либо улучшить ситуацию на рынке «Выпуск какого либо конкурента, который заткнет остальных». Существуют еще **Бизнес правила** помимо того, что видит наш заказчик, видит какую-то функцию нашего приложения, у нас есть еще и бизнес правила, которые ограничивают условно использование дланного приложения, с точки зрения бизнеса. Какие либо нормативы, какие то нормы, какие либо законодательные вещи, какие то контрактные условия. Это всё прописывается в «бизнес правилах»

2 – **Уровень пользовательских требований**. И тут мы конкретно должны определить, какие потребности закрывает наше приложение, конкретного пользователя. Для этого существуют «Пользовательские требования» и тут мы прописываем, что мы хотим получить от этого продукта. Есть еще **Атрибуты качества –** это некая надстройка над нефункциональными характеристиками. По сути тут описываются нефункциональные требования.

3 – Уровень продуктных требований. Тут описано, что конкретно будет реализовано в продукте, как мы должны его сделать, что бы покрыть все остальные желания из вышестоящих уровней.

Далее идет Спецификация требований, это всё что описано было в уровнях. Всё объединяется в одну большую спецификацию. Она бывает на сотни, тысячи страниц. Этот документ достаточно редко встречается и в нем уже могут быть описаны «ограничения», «требования к интерфейсам», «требования к данным»

**Это все требования ЯВНЫЕ, они должны быть задокументированы и оформлены**

Но бывает так, что мы приходим на проект, а требований нет. Что же делать в этом случае. Тогда нам необходимо обратиться к неявным требованиям.

1. Законы, регламенты, инструкции
2. Списки задач, существующие тесты и баг-репорты (Возможно это и наша тестовая документация. Наши тест-кейсы становятся отличной отправной точкой)
3. Руководство пользователя (В нем тоже есть информация, как работает наше приложение, как мы его включаем, какие функции оно выполняет)
4. Реклама продукта
5. Интервью с командой и заказчиков ( Мы должны общаться с нашими разработчиками, маркетологами, менеджерами, казачиком)
6. Чаты и email переписка
7. Прототип, дизайн-макет
8. Конкурентный анализ, личный опыт

СВОЙСТВА ТРЕБОВАНИЙ

Благодаря этим свойствам, мы можем оценить, нормальные эти требования или нет.

1. **Завершенность** – Требование является полным и законченным с точки зрения представления в нем всей необходимой информации, ничто не прощено по соображениями «Это и так всем понятно» Пример: 1. Пароли должны храниться в зашифрованном виде (В каком зашифрованном виде, что вы имеете под этим зашифрованным видом); 2. Экспорт осуществляется в форматы PDF, PNG и т.д. (Т.е. есть еще какие-то другие форматы, если они есть, их надо перечислить); 3. Приведенные ссылки неоднозначны (например: «см. выше» вместо «см. раздел 123.45.b» (Ссылки конечно противопоказаны в требованиях, от них желательно отходить, так как это нарушает прослеживаемость и прозрачность).  
   Как работать с такими несоответствиями, когда мы видим подобные несоответствия требованиям, они скорее всего занесены в какую-то систему и есть возможность оставлять комментарии, есть возможность упомянуть людей в комментариях. И мы должны написать «В таких-то пуантах есть несоответствия, нужно что-бы было так….. либо пожалуйста уточните, как это должно работать.» (В идеале, анализ требований должен проводиться до этапа разработки и написания тестовой документации). Либо писать конкретно тому аналитику, который создавал требования, что бы он это поправил. Главное сигнализировать и не молчать.
2. **Атомарность, единичность** – Требование является атомарным, если его нельзя разбить на отдельные требования без потери завершенности и оно описывает одну ситуацию. Пример: Кнопка «Restart» не должна отображаться при остановленном сервисе, окно «Log» должно вмещать не менее 20-ти записей о последних действиях пользователя.(Это необходимо разбить на два отдельных требования); 2. Если пользователь подтверждает заказ и редактирует заказ или откладывает заказ, должен выдаваться запрос на оплату.(Может возникнуть недопонимание, он редактирует и подтверждает, а может еще какие либо взаимодействия должны быть. Всё должно быть разбито на отдельные требования и на отдельные предложения); 3. Когда пользователь входит в систему, ему должно отображаться приветствие; когда пользователь вошел в систему, должно отображаться имя пользователя; когда пользователь выходит из системы, должно отображаться прощание.(Аналогично надо разбить на отдельные приложение, Так же необходимо описать, какое прощения и приветствие должны высвечиваться, контекст может быть разным).
3. **Непротиворечивость и последовательность** – Требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам. Пример: 1. После успешного входа в систему пользователя, не имеющего права входить в систему…(Возникает противоречие); 2. Кнопка 712.а «Close» всегда должны быть красной, и 36452.x Кнопка «Close» всегда должна быть синей – противоречия в разных требованиях. (Либо на дизайн макете один цвет, а в требованиях другой); 3. «В случае, если разрешение окна составляет менее 800x600…» разрешение у экрана, у окна есть размер.
4. **Недвусмысленность** – Требование должно быть описано без использования жаргона, неочевидных аббревиатур и расплывчатых формулировок, должно допускаться только однозначное объективное понимание и быть атомарным в плане невозможности различной трактовки сочетания отдельных фраз. Пример: «Доступ к ФС осуществляется посредством системы прозрачного шифрования» и «ФС предоставляет возможность фиксировать сообщения в их текущем состоянии с сравнением истории всех изменений.» Что такое ФС ?; 2. «Система конвертирует выходной файл из формата PDF в выходной файл формата PNG» - некоторые вези опущены, так как автор считает их очевидными.(Непонятно каким образом он его конвертирует, не написано куда он его загружает).
5. **Выполнимость** – Требование должно быть технологически выполнимым и реализуемым в рамках бюджета и сроков разработки проекта. Примеры: 1. Анализ договоров должен выполняться с применением искусственного интеллекта, который будет выносить однозначное корректное заключение о степени выгоды от заключения договора – технически невыполнимо на данный момент. ( ИИ такое не умеет делать); 2. Система поиска должна заранее предусматривать все возможные варианты поисковых запросов и кэшировать из результаты – нельзя реализовать (Невозможно учитывать все варианты);3. «озолочение» - требование которые крайне долго или дорого реализуются и при этом практически бесполезны для конечных пользователей.
6. **Обязательность, нужность и актуальность** – Если требование необязательное , оно должно быть просто исключено из набора требований. Если требование нужное, но «не очень важное», для указания этого факта используется указание приоритета. Также исключены (или переработаны) должны быть неактуальные требования. Примеры: 1. Требование было добавлено «на всякий случай», хотя реальной потребности в нем не было и нет; 2. Требование выставлены неверные значения приоритета по критериям важности и/или срочности; 3. Требование устарело, но не было переработано или удалено.
7. **Прослеживаемость** – Вертикальная позволяет соотносить между собой требования на различных уровнях требований, горизонтальная позволяет соотносить требования с тест-планом, тест-кейсами, архитектурными решениями и.т.д. Примеры: 1. Требования не пронумерованы, не структурированы, не имеют оглавления, на имеют работящих перекрёстных ссылок. 2. При разработке требований не были использованы инструменты и техники управления требованиями.(один большой вордовский документ, в котором сложно найти требования) 3. Набор требований неполный, носит обрывочный характер с явными «пробелами».
8. **Модифицируемость** – это свойство характеризует простоту внесения изменений в отдельные требования и в набор требований. Пример:1. Требования неатомарные и непрослеживаемые, а потому их изменение с высокой вероятностью порождает противоречивость. 2. Требования изначально противоречивы. В такой ситуации внесение изменений(не связанных с устранением противоречивости) только усугубляет ситуацию, увеличивая противоречивость и снижая прослеживаемость. 3. Требования предоставлены в неудобной для обработки форме(например, не использованы инструменты управления требованиями, и в итоге команде приходиться работать с десятками огромных текстовых документов).
9. **Проранжированность по важности, стабильности, срочности** – Важность характеризует зависимость успеха проекта от успеха реализации требований. Стабильность характеризует вероятность того, что в обозримом будущем в требование не будет внесено никаких изменений. Срочность определяет распределение во времени усилий проектной команды по реализации того или иного требования.
10. **Корректность и проверяемость** – эти свойства вытекают из соблюдения всех вышеперечисленных. В дополнение можно отметить, что проверяемость подразумевает возможность создания объективного тест-кейса, однозначно показывающего, что требование реализовано верно и поведение приложения в точности соответствует требованию.

ТЕХНИКИ ТЕСТИРОВАНИЯ ТРЕБОВАНИЙ

**Взаимный просмотр** – используется чаще всего ( мы открываем требования и их вычитываем, мы получаем понимание о продукте и мы смотрим, можем ли мы написать тест кейс или нет. Если можем – хорошо. Если нет, то с требованиями что-то не то.

**Вопросы** – можно задавать вопросы, если что-то непонятно

**Тест-кейсы, чек-листы** – если можем написать, значит с требованиями всё норм.

**Исследование поведения системы**

**Прототипирование**

**Рисунки**

ВАРИНАТЫ НАПИСАНИЯ ТРЕБОВАНИЙ

Use case

В данном варианте требований используется графическое изображение. Изображаются люди которые что-то делают, с какой функцией они работают и с кем они сталкиваются в конце. И далее в табличном варианте расписывается Акторы и какие шаги они проводят. Так же расширения – они по сути являются нашими негативными кейсами. Такие виды описания требований иногда встречаются.

ПОЛЬЗОВАТЕЛЬСКИЕ ИСТОРИИ

Данный вид описания требований встречается чаще всего. Тут прописывается, как пользователь я хочу то-то, что бы делать что-то

Пример:

Как Макс, я хочу пригласить друзей, чтобы мы вместе могли пользоваться этим замечательным сервисом.

Как Саша, я хочу организовать свою работу, чтобы лучше контролировать ситуацию.

Как менеджер, я хочу видеть, как продвигается работа у моих коллег, чтобы можно было составлять более точные отчеты о наших успехах и неудачах.

В таком видео оформлено большинство требований.

А далее уже расписывается

Так же в требованиях есть Mockup – это макет, в котором мы видим как будет выгладить приложение.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Summary** | **Priority** | **Do we have an automation?** | **Sprint 1: Bugs Quantity** | **Sprint 2: Bugs Qty** | **Sprint 3: Bugs Qty** | **Regression** |
| Log in under admin | High | No | 0 | 1 | 5 | + |
| Registration via Google | High | Yes | 10 | 2 | 0 | - |
| Update a personal profile | Medium | No | 1 | 5 | 10 | + |
| Update a login button (GUI) | Trivial | No | 0 | 0 | 0 |  |
| Update a header (GUI) | Low | No | 13 | 12 | 11 | + |
| Delete all outdated data | High | No | 2 | 4 | 2 | + |
| Update a naming of social media area (GUI) | Trivial | No | 0 | 1 | 2 | - |
| Create an a user story (the feature should be deleted after 3rd sprint) | Low | Yes | 15 | 20 | 20 | - |
| Adding new goods in the basket | Medium | Yes | 0 | 2 | 2 | - |
| Adding the high contrast mode | High | Yes | 12 | 10 | 8 | + |
| Update a footer (GUI) | Trivial | No | 1 | 1 | 1 |  |

Мое решение

1. Log in under admin – Нет автоматизации, высокий приоритет, рост количества багов
2. Registration via Google – приоритет высокий, есть автоматизация, баги сокращаются
3. Update a personal profile – Количество багов растет от спринта к спринту
4. Update a login button (GUI) – низкий приоритет, GUI задача, регрессия не нужна
5. Update a header (GUI) - Большое количество багов каждый спринт
6. Delete all outdated data – Высокий приоритет, баги продолжают находиться
7. Update a naming of social media area (GUI) - Низкий приоритет, GUI задача, регрессия не нужна
8. Create an a user story (the feature should be deleted after 3rd sprint) - приоритет низкий, есть авто, фича будет не актуальная после 3 спринта
9. Adding new goods in the basket – Средний приоритет, много багов
10. Adding the high contrast mode – Несмотря на наличие авто, лучше продолжить регрессию, баги всеравно находятся.
11. Update a footer (GUI) = низкий приоритет. GUI задача, регрессия не нужна

ТЕХНИКИ ТЕСТ-ДИЗАЙНА

Не всегда у нас есть возможность провести исчерпывающее тестирование. По сути такой возможности никогда и нет, если не идет речь про какие либо примитивные тесты типа 2+2. И для того, что бы провести тестирование тщательно и не потратить на это большое количество времени, существуют техники тест-дизайна.

Выделают Анализ тестирования и Проектирование теста.

**Анализ тестирования** – это деятельность которая направлена на описание тестовых условий путем анализа базиса тестирования (Базис тестирования – это требования, дизайн. И на основании этого базиса мы можем определять какие либо тестовые условия, в том числе оптимизировать количество кейсов, которые нам необходимы для финального тестирования.

**Проектирование теста** – активность по получению и определению тест-кейсов в соответствии с условиями тестирования ( Тут мы создаем уже тест-кейсы. Мы понимает что у нас есть приложение, допустим web. Мы можем использовать в нем различные формы, мы можем заходить, регистрироваться, покупать товар. Эти условия и будут влиять на то, какие у нас в итоге получатся тест-кейсы.

ЭКВИВАЛЕНТНОЕ РАЗБИЕНИЕ

Самая популярная техника тест-дизайна. Пример. У нас есть некоторое приложение, это онлайн магазин и мы можем рассчитывать там скидку. Скидка у нас рассчитывается согласно возрасту. И есть некое воле, куда мы можем ввести этот возраст. Допускается ввод только чисел. Расчет скидки проходит следующим образом. Возраст покупателя от 0 до 17 – Скидка 50%, от 18 до 54 – Скидка 25%, от 55 и выше – Скидка 75%.

Мы видим что у нас есть некоторые закономерности, один класс людей, одна группа имеет скидку 50%, есть группа людей где скидка 25% и еще одна группа со скидкой 75%. Это и есть наш класс эквивалентности.

Класс эквивалентности – набор данных обрабатываемых одинаковым образом и приводящим к одинаковому результату. Т.е. независимо от того, какое число мы напишем, у нас всеравно скидка будет 50%, в другои классе 25% и в третьем 75%

Это позитивные классы, но нельзя забывать о НЕГАТИВНЫХ классах. А если мы введем меньше( возраст -1) то появится ошибка

Так же у нас есть граничные условия. Это значения которые содержатся на границах класса эквивалентности

По сути мы имеем позитивный класс

Это наши определенные скидки

Негативный класс

Это значения меньше 0, которые должны приводить к ошибке

А так же **Ввод букв и спец. Символов, код, эмодзи.** То есть все значения, которые пользователь может внести. Дополнительно **проверяют пустое значение и 0** так как именно в них возможно высокое возникновение дефектов.

НЕГАТИВНЫЕ КЛАССЫ ЭКВИВАЛЕТНОСТИ тестируют РАЗДЕЛЬНО!!!

Итак последовательность.

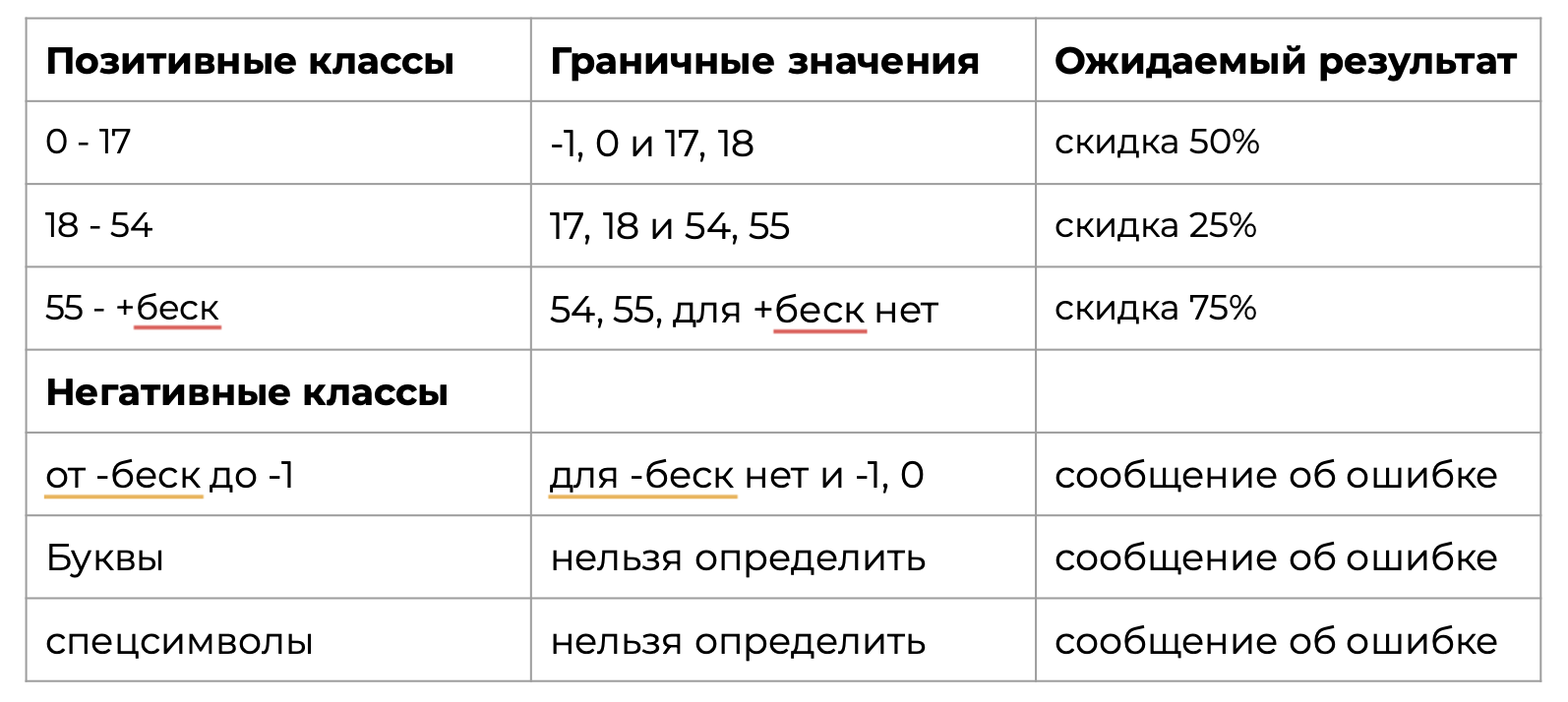
1. Мы определяем класс эквивалентности
2. Определяем негативные и позитивные классы
3. Переходим к тестированию и тестируем хотя бы одно значение из класа эквивалентности.



Но эквивалентное разбиение не закрывает все потребности и его недостаточно. Всегда когда мы говорим о эквивалентом разбиение, подмешивается **анализ граничных значений**.

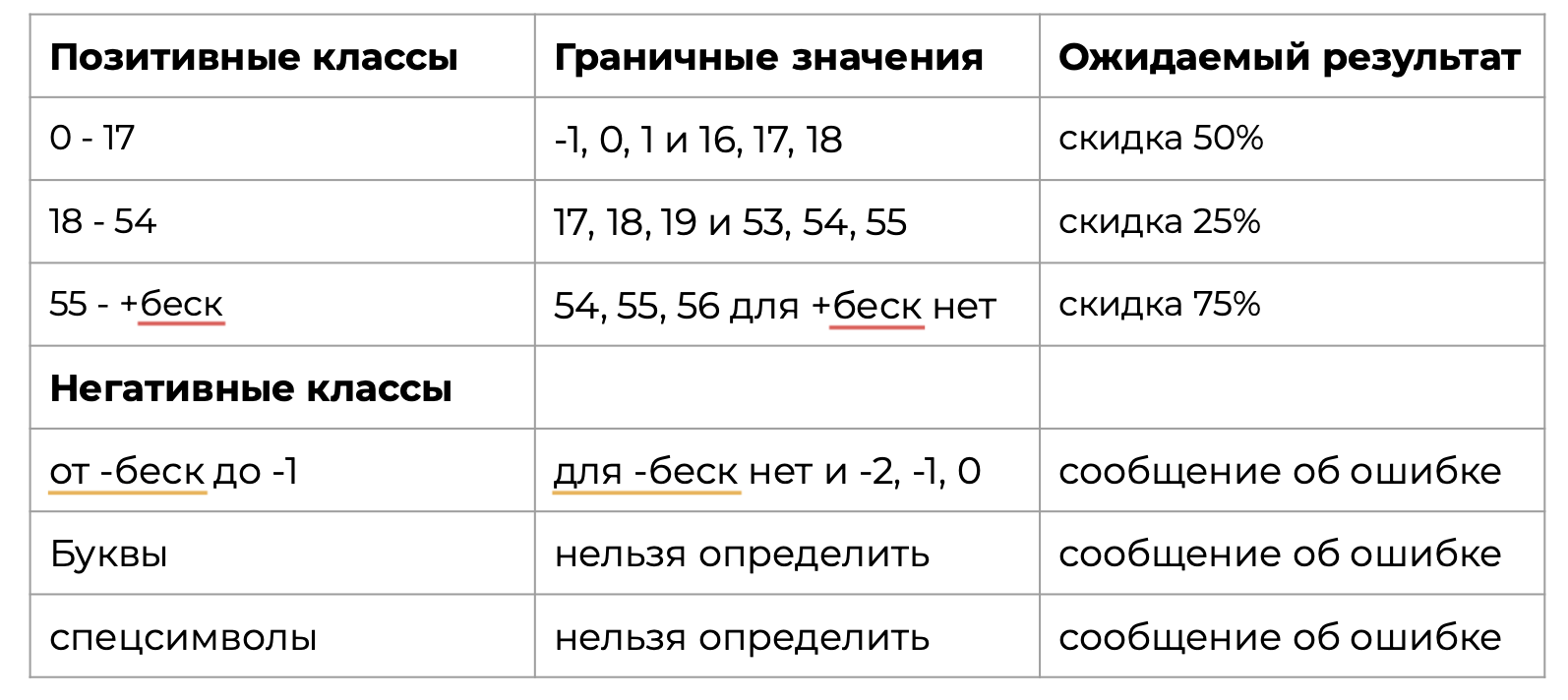
Метод Анализа граничных значение – является продолжением метода эквивалентного разбиения, но может быть применим, если классы состоят из упорядоченных числовых значений.

Недостаточно взять одно граничное значение, надо определить остальные граничные значения. Есть 2 метода: 1. Берётся наша нижняя граница и значение до нее и мы берем верхнюю границу и значение после ее. Мы должны протестировать 0, -1, 17, 18



Как мы видим, появляется бесконечность, Для бесконечности мы не можем вводить граничные значения

Второй метод – Тестируется значение нижней границы и значение как до нее так и после нее. И верхней границы, как до нее и после нее



Еще важный пункт, если у нас появляются десятичные значения после запятой, то нам надо искать граничные значения по дробной части

Например от 0.01 до 10.00 – граничные значения будут 0.00, 0.01, 0.02 и 9.99

ПОПАРНОЕ ТЕСТИРОВАНИЕ

(pairwise testing)

Попарное тестирование – это техника тестирования методом черного ящика, при котором тестовые сценарии разрабатываются таким образом, чтобы выполнить тестирование для всех комбинаций параметр-значение.

Представим, что у нас есть ресурс, который должен работать в 8 бразузерах, используются различные плагины, запускается на различных операционных системах и мы должны получать страницы от разных веб-серверов и работать с серверными операционными системами.

И того 8 бразузеров, 3 плагина, 6 клиентских операционных систем, 3 сервера, 3 серверных операционных системы – 1296 комбинаций. Для тестирования данного ресурса, нам надо будет написать 1296 тест-кейсов. Данной возможности мы себе позволить не можем, так как это очень долго а команда постоянно торопиться. Данное тестирование необходимо каким-то образом оптимизировать.

Есть 2 техники попарного тестирования. **Ортогональные массивы** и алгоритм **All Pairs**

Ортогональный массив работаеть следующим образом. В независимости от того какие мы два стобца выберем в системе, в каждой паре столбцов будут встречаться все комбинации значний этих стобцов. Но на практике то всетрчается редко. Чаще будут встречаться уже написанные алгоритмы которые будут работать по формуле вычисления ортогональных массивов. Мы просто будем вводить значения, которые нас интересуют, и система будет выводить ту таблицу, которая нам нужна.

Мы будем чаще всего использовать All Pairs, которые позволяет атоматически генерировать пары. И самым популярный инструмент для этого PICT. На практике мы не тратим время для вычисления и заполнения ортогональных массивов, мы вбиваем все данные в таблицу и получаем выходные данные, которые мы можем тестировать.  
<https://pairwise.teremokgames.com>  
Как работает система, мы заполняем таблицу интересующими нас параметрами. В первый столбец, мы ставим тот параметр, у которого больше всего значений. Далее столбы расписываются по убыванию.

После ввода данных мы нажимаем кнопку Generate Pairwise. У нас формируется таблица со всеми комбинациями, которые мы должны провести.

Когда технику попарного тестирования удобно применять ?  
Есть ресурсы в которых есть большое количество полей для ввода, фильтраций, каталоги. Поступила задача проверить фильтрацию, мы не будем сидеть и условно проверять все комбинации. Мы сформируем табличку. Открыв Pairwise пропишем конкретное название тех фильтров, которые нас интересуют (производитель, цена, магазины) – это будут столбцы. И у этих параметров есть значения – мы их заносим в строки.

Всегда когда мы видим большое количество полей для ввода, использующих информацию дающих нам искомый результат. Мы используем попарное тестирование. Во второй закладке pairwise позволяет генерировать файлы нужного расширения и объема, для тестирования.

ДИАГРАММА ПЕРЕХОДА И СОСТОЯНИЙ

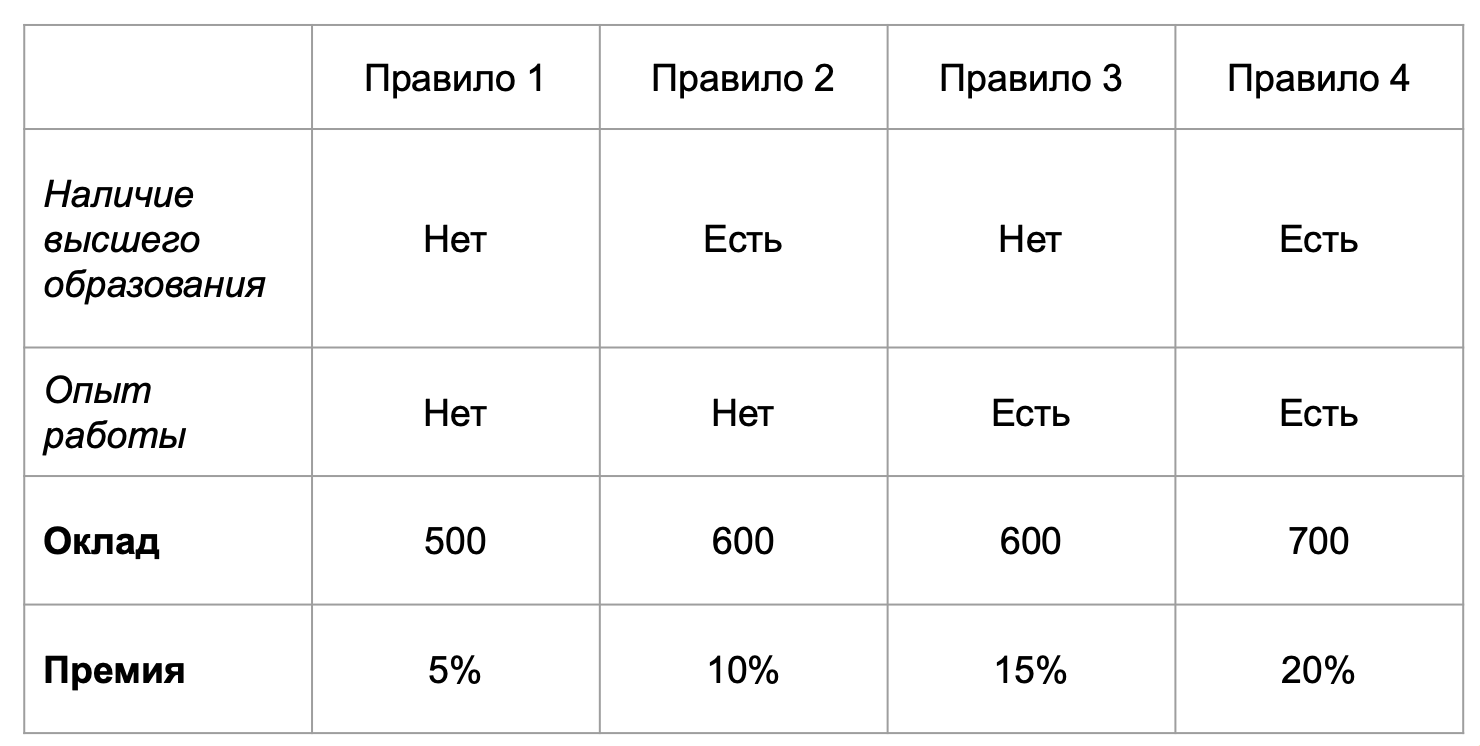
ТАБЛИЦА ПРИНЯТИЯ РЕШЕНИЙ

**Таблица принятия решений** – тестирование методом черного ящика, в котором тестовые сценарии проектируются для проверки комбинаций условий и действий, отраженных в таблице решений.

Из самого названия понятно, что речь пойдет про некую таблицу. ЕЕ так же называют таблицей альтернатив.

Таблица альтернатив – это хороший способ записи сложных бизнес-правил, которые должны быть реализованы в системе. В процессе создания таблицы тестировщик определяет условия ( входы) и результирующие действия (выходы). Пары условий и действий образуют строки таблицы, при этом условия указываются сверху а действия снизу.

К примеру у нас есть система, определяющая, какую заработную плату будет получать наш кандидат.



Есть несколько правил, наличие высшего образования и опыт работы, и какой будет оклад и какая премия. И в зависимости от комбинаций этих условий, мы определяем оклад и премию.

Ну либо на сайте, если человек принял условие соглашения, он может зайти на сайт. Если не принял, то может высветиться ошибка с просьбой принять соглашение.

Либо это может быть админитсратор. У нас будет набор ролей, и на выходе будет информация, будет ли у человека возможность редактировать что-то.

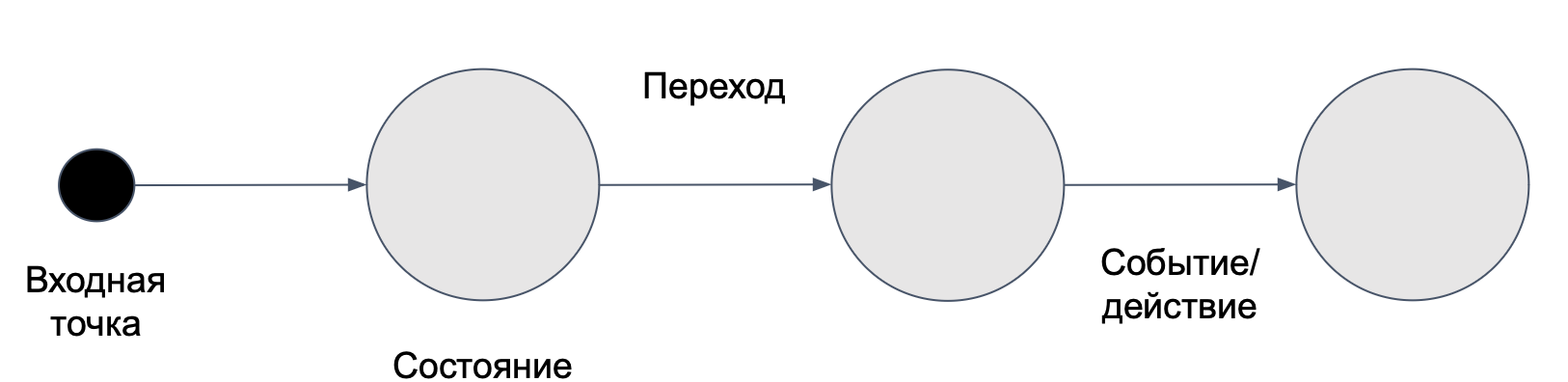
Тестирование таблиц принятия решений можно использовать, когда система должна реализовывать сложные бизнес-правила. Когда эти правила могут быть представлены в виде комбинаций условий, и когда эти условия имеют дискретные действия, связанные с ним.

Задачка!  
В дождливую погоду ты надеваешь дождевик, когда идет снег – ты надеваешь пальто, когда светит солнце – надеваешь солнечные очки, когда нет дождя и снега, ты носишь плащ. Считать, что может идти только один вид отсадков (дождь и снег не идут одновременно)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Правило 1 | Правило 2 | Правило 3 | Правило 4 | Правило 5 |
| Дождливая погода | Нет | Да | Да | Нет | Нет |
| Идет снег | Нет | Нет | Нет | Да | Да |
| Светит солнце | Нет | Нет | Да | Нет | Да |
| Надеть дождевик | Нет | Да | Да | Нет | Нет |
| Надеть пальто | Нет | Нет | Нет | Да | Да |
| Надеть солнечные очки | Нет | Нет | Да | Нет | Да |
| Надеть плащ | да | Нет | Нет | Нет | Нет |

Тестирование таблицы переходов – разработка тестов методом черного ящика, при котором сценарии тестирования строятся на основе модели переходов состояний.

У нас есть диаграмма состоящая из входной точки, состояния, перехода, события/действия



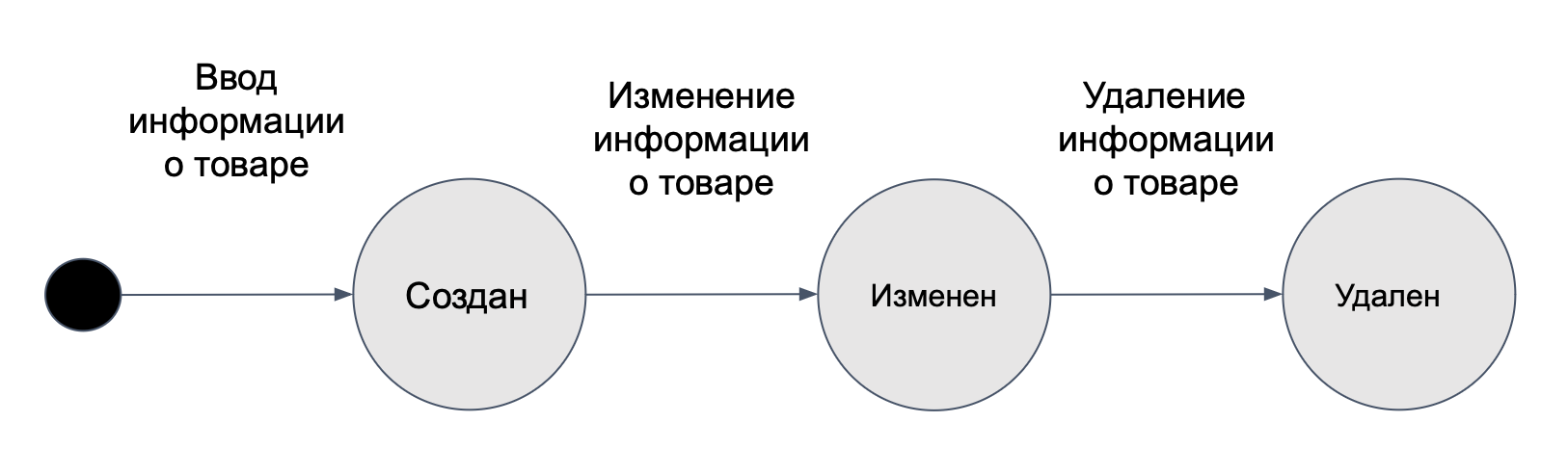
Состояние – это состояние в котором система ожидает возникновение одного или нескольких событий. ( Условно пользователь не зарегистриован)

Переход – это изменение состояния из одного в другое, произошедшее благодаря какому-то событию.

Событие – что-то, что вызывает изменение состояния системы.

Дейстиве – операция, которая вызвана изменением состояния.

Входная точка – это то с чего мы начинаем ( условно пользователя нет в системе)



У нас есть входная точка, мы вводим информацию о товаре. У нас создается карточка товара, далее мы изменяем информацию о товаре и у нас тоже изменяется состояние этого товара. Далее мы удаляем информацию о товаре и у нас карточка удаляется.

На практике на первых парах эти диаграммы состояний не встречаются, но в своей работе, если они будут вам необходимы, то можно будет их рисовать.

Тестирование с помощью таблицы переходов наиболее распространено в сфере встроенного ПО и при тестировании программных меню. Метод также подходит для моделирования бизнес-сценариев, имеющих конкретные состояния, или для тестирования переходов по экранным формам.

Дополнительные техники тест дизайна:

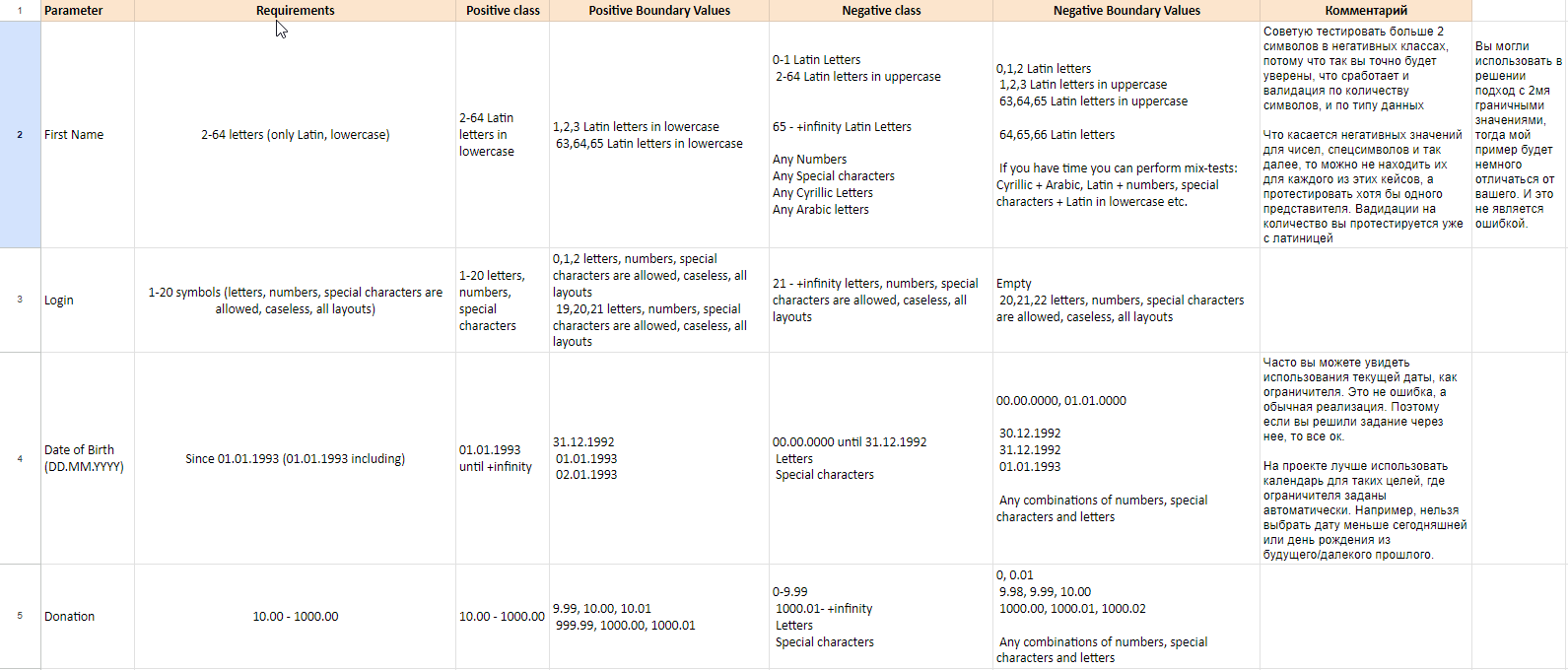
**Предположение о ошибках** – техника тестирования, в которой тесты получены на основе знаний тестировщика о ранее обнаруженных сбоях или общих знаниях о типах отказов. (Мы работаем и знаем, что ошибки возникают, когда мы вводим допустим не валидные данные) т.е. мы уже работали и можем предполагать, какие ошибки у нас получатся, при каких-то дейтсивях.

Отображение причино-следсвенных связей – метод тестирования черного ящика, при котором тестовые сценарии разрабатываются на основе диаграмм прчинно-следсвенных связей. ( Что-то произошло и появилось какое-то следствие)

Диаграмма причинно-следственных связей – графическое представления логических отношений между выходными данными (причинами) и связанными с ними выходными данными (следствиями)

Ваша задача: определить классы эквивалентности и граничные значения для каждого из них. Не забывайте, что классы делятся на позитивные и негативные.

Поля ввода могут принимать любые значения помимо описанных в таблице, поэтому обязательно удостоверьтесь, что вы проверяете доступность только тех данных, которые есть в требованиях. Например, если поле принимает только буквы, попробуйте ввести числа и спецсимволы. Для выполнения задания скопируйте шаблон, заполните недостающие поля и приложите ссылку в форму ниже.



ТЕСТОВАЯ ДОКУМЕНТАЦИЯ

Это очень важный раздел – тестовая документация. Именно с ней мы будем проводить большее количество времени. Сюда входят и тест-кейсы, чек-листы, отчет о дефектах, тест-планы, тестовые стратегии и требования которые мы с вами рассматривали ранее. И нам необходимо их научиться правильно писать, что бы не у кого не было вопросов к качеству тех артефактов которые мы представляем.

Итак, начнем с плана тестирования. Любые SDLC и STLC процессы начинаются с планирования. И нам надо научиться создавать эту документацию. Описывать как мы будем проводить тестирование, какие виды тестирования применять, какие у нас есть риски, какие есть устройства, какие есть тестовые среды. Вся эта информация будет содержаться внутри нашего плана.

**План тестирования** – документация, описывающая цели тестирования, которые должны быть достигнуты, средства и график их достижения, организованная для координации тестовой деятельности.

Существует 2 вида тест плана:



Главный пан пишется один раз, в начале проекта, само собой он может видоизменяться и в нем содержится вся информация которая нам необходима. Любая команда, которая у нас работает, может к нему обратиться на верхнем уровне (верхний уровень – значит без какой бы то ни было излишней детализации) и понять с чем мы работаем.

Уровневый план тестирования – это план тестирования, который обычно относится к одному уровню тестирования. В нашем случае можно сказать, что он не просто к уровню тестирования относится, а к тестированию какой либо отдельной команды. Он может создаваться как для каждой команды, так и для каждой итерации. Это условно то, на основании чего проводится тестирование на данный момент у нас.

Т.е. берется один общий продукт, делятся на команды, одна предположим Frontend а вторая Backend. Одна занимается общей информацией на сайте, - логин, регистрация. Вторая команда углубляется в функции к примеру покупка товара, оформление доставки и.т.д.

Если большой проект и большой продукт, то всегда есть разделение на разные команды, которые отвечают за определенные функции.

**СТРУКТУРА ТЕСТ ПЛАНА**

Структура тест плана +- везде одинакова, в интернете очень много шаблонов и все они в принципе правильные. Мы рассмотрим некоторые вещи из тест плана посмотрим за что они отвечают. (Далее по курсу у нас будет задание написать тест-план). Но волноваться не стоит. Тест-план обычно не пишут начинающие специалисты, мы его рассмотрим для общего понимания, как это происходит.

1. **Цель –** само собой есть цель тестирования. Цель формируется согласно количественным и качественным характеристикам. Есть принцип **SMART** – цель должна быть осуществима, цель должна быть определена, у цели должны быть количественные показатели, цель должная быть измерима. Таким образом формируется цель, чего мы добиваемся.
2. **Области подвергаемые тестированию –** У нас есть большое приложение, и большое количество функций. Само собой в конкретный момент времени, мы не можем протестировать всё, либо нет необходимости это делать. К примеру был ранее разработан модуль регистрации и нам сейчас нет смысла его тестировать. Либо его мы разрабатываем, но у него небольшой приоритет.
3. **Области, не подвергаемые тестированию –** тут указываются модули с небольшим приоритетом .
4. **Тестовая стратегия и подходы –** это может быть как отдельный вид документации, так она может быть и внутри нашего тест-плана. Тестовая стратегия, содержит информация, каким образом мы будем проводить тестирование.
5. **Критерии.** Чаще всего это критерии начала и завершения тестирования. Всегда эти критерии надо определять, иначе мы не сможем определить когда завершать тестирование. Начало – все требования описаны, вся тестовая документация написана, тест-кейсы, чек-листы. Команда разработки создала то, что мы будем тестирования. Окончание – 80% багов пофикшено, критических багов нет.
6. **Ресурсы –** программные, аппаратные, человеческие, финансовые ( размер команды, сколько денег понадобиться, какая понадобиться техника, наличие необходимого ПО. Возможно потребуется дополнительная закупка мобильных устройств, ПО…)
7. **Расписание –** расписание нашего тестирования. Даты, когда мы должны протестировать ту или иную функциональноть.
8. **Роли и ответственность –** Прописываем к примеру. В команде 15 тестировщиков, из них одни Лид - Иванов И.И. столько то мидлов.
9. **Оценка рисков –** Мы всегда должны понимать, что у нас есть риски. Кто-то болеет, кто-то в декрете, кто-то недавно пришел на проект, кто-то на испытательном, проблемы финансовые – не можем купить какую либо аппаратуру, программу. Всё это оценивается на данном этапе.
10. **Докумнетация –** Описывается какую документацию мы будем использоваться. Чек-листы, тест-кейсы, отчеты о дефекте, тест-план, тестовая стратегия (эти документы мы будем рассматривать)
11. **Метрики –** числовые характеристики показателей качества. (Оценка количества найденых багов, как в динамике они уменьшаются или увеличиваются. Тестовое покрытие – мы должны понимать какое количество требований покрыто тест-кейсами. Можно смотреть на то, сколько у нас написано автотестов на регрессионный тест.)

# **ОТЧЕТЫ ПО ТЕСТРИРОВАНИЮ**

Помимо того, что мы занимаемся тестированием, нам о этом тестировании необходимо еще рассказывать. Для этого служит отчет о результатах тестирование, который делается в конце итерации, перед релизом, для того что бы заказчик и вся команда понимала, готовы ли они к релизу, какое качество финального инкремента и в целом это задача тестировщика, по составлению данного отчета. Не всегда это делают начинающие специалисты, это могут делать и мидлы и менеджеры. Но бывает что и начинающих тестировщиков привлекают к этой деятельности, поэтому надо быть к этому готовыми. Тем более если вы один на проекте, вам надо будет доносить информацию о том, как работает отдел качества, то надо будет уметь составлять эти отчёты.

Это не сложно, сейчас рассмотрим основные атрибуты и так же будет задание, на котором мы потренируемся создавать отчеты.

Отчет о результатах тестирования – вид отчета о тестировании, составляемый через регулярные промежутки времени, содержащий данные о ходе тестирования в сравнении с исходным планом, о рисках и альтернативах, требующих принятия решения.

**Краткое описание** – отражает основные достижения, проблемы, выводы и рекомендации. Условно у нас прошла итерация и мы провели тестирование. Тут мы должны написать общее **качество продукта (высокое, низкое, среднее**), либо процентовкой (90%, 80%, 60%), у нас такое-то количество критических багов, команда тестирования советует не проводить релиз сейчас

**Команда тестировщиков** – прописывается информация о команде тестировщиков, которая проводила это тестирование.

**Описание процесса тестирования** – перечень работ за конкретный период. Мы пишем, сколько кейсов было запущено, сколько автотестов было запущено, сколько багов было найдено. Желательно сравнение в динамике, допусти в прошлую итерацию обнаружено 10 багов в эту итерацию 8. Условно можем сказать, что качество продукта повышается.

**Расписание** – детализированное расписание работы команды тестировщиков и/или личные расписания участников команды.

**Статистика по новым дефектам** в виде таблицы. Мы создаем графики, диаграммы либо таблицу, с информацией о том, сколько дефектов было найдено.

Список новых дефектов.

Статистика по всем дефектам – таблица, содержащая данные обо всех дефектах за время проекта.

Рекомендации – перечень работа за конкретный период. Тут мы можем предложить, то , что можно улучшить. Допустим у нас есть задержка по код-ревью. Т.е. разработчики тратят на код очень много времени и у нас тестирование задерживатеся. Либо мы обнаружили, что в каком-то модуле у нас концентрируется большое количество багов, и с каждой итерацией и спринтом, мы замечаем что только в этом модуле идет скопление дефектов.

Приложения – фактические данные (графики диагарммы)

ТЕСТ-КЕЙСЫ и ЧЕК-ЛИСТЫ

Это одна из важных тем. Это тестовая документация в виде чек-листов и тест-кейсов. А так же баг репорты и отчеты о дефекте. Так как с этой документацией тестировщику приходится больше всех работать.

**Тестирование на основе чек-листов** – это метод тестирования, основыанный на опыте, при котором тестироващик использует высокоуровневые списки. Список состоит из набора правил или критериев, согласно которым верифицируется программный продукт.

Чек лист это по сути список, в котором написаны наборы наших проверок.

Чек-лист – набор идей. Идеи по тестированию, идеи по разработке, идеи по планированию и управлению.

Сейчас мы рассмотрим больше теоретическую часть аспекта. В рамках самостоятельной работы мы так же напишем несколько чек-листов.

Обычно чек-листы оформляются в гугл таблицах, так как специализированного ПО для оформления чек-листов нет

**Шапка** – содержит информацию о названии приложения, его версии, окружении, на котором проводится тестирование (версия OC, браузера, эмулятора), ответственного за тестирование, дату тестировния.

**Тестируемые** модули, субмодули: например, регистрация, аутентификация, авторизация.

**Список** проверок – они должны отражать основную суть, без лишней детализации.

**Статус** – информация о статусе прохождения проверки – пройдено/не пройдено.

Иногда встречаются:

**Ожидаемый результат** – то, что мы ожидаем увидеть после запуска проверки согласно требованиям.

**Типы тестирования** – к какому типу относится проверка.

**Отчеты о дефекте** – ссылки на отчеты о дефектах для прослеживаемости.

**Заметки** если нужно добавить комментарий.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Project | | | | Alexpress |  |  |  |  |
| Date | | | | 06.08.2022 |  |  |  |  |
| Build | | | | 1.0 |  |  |  |  |
| Tester | | | | Artsiom Rusau |  |  |  |  |
| Environment | | | | Windows 11 Version 22H2 Google Chrome103.0.5060.134 | Safari | Firefox | Opera | Edge |
| Module | Submodule | Element/ function | Summary | Status | Status | Status | Status | Status |
| Registration | Choose a location | Dropdown | All countries are loaded | passed |  |  |  |  |
| User can choose a country | failed |  |  |  |  |
| The country search is working |  |  |  |  |  |
| Cancel button |  |  |  |  |  |
| Input numbers > error message |  |  |  |  |  |
| Input special characters > error message |  |  |  |  |  |
| Empty state > error message |  |  |  |  |  |
| Registration via phone | Phone | Existing phone |  |  |  |  |  |
| The verification code was received |  |  |  |  |  |
| Non-existing phone |  |  |  |  |  |
| Input letters |  |  |  |  |  |
| Input special characters |  |  |  |  |  |
| The button is disabled/active |  |  |  |  |  |
| Empty value |  |  |  |  |  |
| Registration via email | Email | With exisiting email and password (6-20 symbols) |  |  |  |  |  |
| Dropdown with emails |  |  |  |  |  |
| The user was created and try to register again |  |  |  |  |  |
| Cancel button |  |  |  |  |  |
| Modify the email after receiving the verification code |  |  |  |  |  |
| Close the verification window without code inputing |  |  |  |  |  |
| Resend code |  |  |  |  |  |
| Fill in wrong verification code |  |  |  |  |  |
| Empty value |  |  |  |  |  |
| Only letters |  |  |  |  |  |
| Only .com |  |  |  |  |  |
| Input special characters |  |  |  |  |  |
| Input numbers |  |  |  |  |  |
| Password | Cancel button |  |  |  |  |  |
| Hide/unhide the password |  |  |  |  |  |
| Empty value |  |  |  |  |  |
| Only letters |  |  |  |  |  |
| Input special characters |  |  |  |  |  |
| Input numbers |  |  |  |  |  |
| Input letters and special characters |  |  |  |  |  |
| Input letters and numbers |  |  |  |  |  |
| Input numbers and special characters |  |  |  |  |  |
| Input letters, numbers and special characters |  |  |  |  |  |
| Input 5 symbols |  |  |  |  |  |
| Input 7 symbols |  |  |  |  |  |
| Input 19 symbols |  |  |  |  |  |
| Input 21 symbols |  |  |  |  |  |
| Registration/log in via social media | Registration via AppleID | With exisiting Apple ID |  |  |  |  |  |
| With non-existing Apple ID |  |  |  |  |  |
| With decline of verification code |  |  |  |  |  |
| Registration via Google | With exisiting Google account |  |  |  |  |  |
| Without exisiting Google account |  |  |  |  |  |
| Registration via Facebook | With exisiting Facebook account |  |  |  |  |  |
| Without exisiting Facebook account |  |  |  |  |  |
| Registration via Twitter | With exisiting Twitter account |  |  |  |  |  |
| Without exisiting Twitter account |  |  |  |  |  |
| Registration via VK | With exisiting VK account |  |  |  |  |  |
| Without exisiting VK account |  |  |  |  |  |
| Registration via OK | With exisiting OK account |  |  |  |  |  |
| Without exisiting OK account |  |  |  |  |  |
| Show more | Show more/Hide feature |  |  |  |  |  |
| Policy | Links for Policy | AliExpress.com Free Membership Agreement |  |  |  |  |  |
| Privacy Policy |  |  |  |  |  |
| Triggers | Close the registration window | The window is closed by button |  |  |  |  |  |
| The window is closed by empty space clicking |  |  |  |  |  |

ЧИТ-ЛИСТЫ

Обычно в них объединяют часто повторяющиеся проверки:

1. Общие проверки для веб и мобильных приложений
2. Список проверок для тестирования веб-форм
3. Список проверок для определенного типа тестирования, например usability или локализация.

ТЕТСТОВЫЕ СЦЕНАРИИ / ТЕСТОВЫЕ КЕЙСЫ / ТЕСТОВЫЕ СЛУЧАИ

**Тест-кейс** – набор предусловий, входных данных, действий (где применимо), ожидаемых результатов и постусловий, разработанных на основе тестовых условий.

Судя по определению мы видим, что у нас есть входные данные, видим, что есть какие-то действия, какие-то шаги, которые мы должны сделать, что бы получить результат. И увидеть что результат фактический совпадает с ожидаемым.

**Тестовый сценарий** – последовательность действий над продуктом, которые связанны единым ограниченным бизнес-процессом использования, и сообразных им проверок корректности поведения продукта в ходе этих действий.

Тут мы видим уже есть **бизнес-процесс** использования. Тут в отличии от тест-кейса мы делаем уже высокоуровневые проверки которые относятся к пользовательским требованиями. \* Далее о этом еще поговорим\*

ВИДЫ ТЕСТ-КЕЙСОВ

**Высокоуровневый тест-кейс** – тест-кейс без конкретных входных данных и ожидаемых результатов. Данные тест кейсы редко встречаются.

**Низкоуровневый тест-кейс** – тест-кейс с конкретными входными данными и ожидаемыми результатами. Мы будем работать именно с этим типом.

Как и в чек-листе, в тест-кейсе есть обязательные поля – атрибуты.

**Идетификатор (ID)** уникальный томер, необходимый для прослеживаемости. В системах по управлению кейсами (TMS test manager system) – проставляется автоматически.

**Приоритет (Priority)** срочность и важность задачи

**Требования (Requirement)** ссылка на требование, для проверки которого служит кейс

**Модуль (Module)** название структурной части, в которой находится предмет тестирования

**Заголовок (Title)** Отражает суть проверки.

**Тестовые данные и предусловия** – мы само собой что-то будем вводить, будь это пароль, логин, размеры файла, с определенным разрешением. **Предусловие** – это состояния системы до начала тестирования. К примеру пользователь должен быть зарегистрирован до начала выполнения тест-кейса, либо что-то должно быть создано в базе данных. (Открыть сайт, или ресурс – это не предусловие.)

**Шаги** – последовательность действий для получения ожидаемого результата. Тут мы прописываем. Открыть такой-то ресурс, нажать такую-то кнопку, создаться пользователя. Полностью прописываем, что мы делаем, что бы получит наш ожидаемый результат, в том числе прописанный в заголовке. Рекомендации. Если пишем на английском языке, мы используем инфинитив, (Open Close Click). В Русском языке, люди любят писать ( Я открываю…. Он открывал… - это неправильно. Тут так же надо использовать инфинитивы. Открыть, нажать, перейти)

**Ожидаемые результаты (Expected results)** – ссылка на требования, для проверки которого служит кейс. Результат должен быть для каждого шага. Описываем ожидаемые результаты, не фактические. Фактические результаты в тест-кейсах не используют. Они могут использоваться в некоторых TMS системах. Шаг – открыть страницу – результат – запрашиваемая страница открыта.

**Постусловия (Postconditions)** – возвращение системы в исходное состояния. Создали 5 пользователей, удалили 5 пользователей. Загрузили файл на 500гб, удалили. Включили виртуальную машину – отключили.

Предусловия и постусловия – это не обязательные кейсы. Если кейс небольшой их можно не использовать. Объем кейса желательный примерно шагов 10, так что действия, которые не относятся к кейсу, обычно выносят в предусловия.

**Статус** – чаще всего используются в TMS системах. ( Пройдено, завалено, тест-кейс устарел)

Есть жизненный цикл тест-кейса. Кейсы создаются и находятся в системе, их необходимо прогонять. Нам необходимо провести проверку. И мы вставляем тест-кейсу статус запланирован. При выполнении – статус «выполняется». Если тест «провален» - то мы создаем баг репорты. «Заблокирован» если функциональность сломана, и тест кейс не будет выполнен.

**ОТЧЕТ О ДЕФЕКТЕ**

Мы дошли до момента, когда мы будем учиться писать дефекты. И узнаем о том, что такое отчеты о дефекте.

**Отчет о дефекте** – документирование возникновения, характера и состояния дефекта.

На самом деле мы будем находить баги не так часто, как это может показаться с первого раза. Нам будут платить деньги за создание документации, которая позволит нам протестировать ПО и сказать, что оно соответствует определенному уровню качества.

Как только возникает дефект, нам надо его зарепортить. И сообщить, что надо что-то делать, менять ситуацию.

**Отчет о дефекте** – документирование отклонения ожидаемого результата от фактического.

Очень часто есть недопонимание, что такое **ошибка**, **дефект** и **отказ**.

Это разные вещи и всё называть дефектами неправильно.

**Ошибка** (error) – Действие человека, которое приводит к неправильному результату. Аналитик, допускает ошибку в требовании, после разработки ПО возникает ошибка аналитика. Либо тестировщик, что-то не дотестировал, что привело к неправильному результату. Разработчик что-то не правильно написал и ошибка в коде привела к неправильному результату.

**Дефект** (defect) – Несовершенство или недостаток рабочего продукта, проявляющийся в несоответствии требованиям или спецификациям. К примеру Аналитик допустил ошибку, разработчик взял в работу и написал неправильный код. Реализация этой ошибки и есть дефект. Дефект – это реализация ошибки.

**Отказ** (failure) – Событие, при котором компонент или система не выполняют требуемую функцию в соответствии со спецификацией. Если допущенная ошибка превратилась в дефект и он еще привел к отказу, система перестала работать – это и есть отказ.

АТРИБУТЫ ОТЧЕТА О ДЕФЕКТЕ

**Идентификатор**: уникальный номер отчета. Присваивается автоматически в багтрекенговой системе (BTS, bug tracking system)

**Краткое описание(summary)**: отвечает на три вопроса: что, где, когда? Что произошло? Где это произошло? При каких условиях? Мы в начале прописываем в каком модуле произошла ошибка. К примеру «Регистрация». При вводе символов в поле Log in, у нас появляется сообщение о ошибке. Это вот баг, дефект, в требованиях не написано. В принципе система может принимать пять символов.

**Подробное описание(description)**: информация о дефекте в развернутом виде + фактический и ожидаемый результат + ссылка на требование. Необходимо иногда давать больше информации, связанной с дефектом. На практике встречается очень редко, достаточно просто summary. Сам фактический результат, очень часто записывают в заголовок summary

**Шаги по воспроизведению(steps to reproduce, STR)**

**Окружение(environment)** информация о среде, на которой был обнаружен баг (версия ОС, браузера, мобильного устройства)

**Важность(severity)** степень ущерба, который наноситься проекту существованием дефекта. Серьезность для разрабатываемого ПО.

***Критическая(critical)*** – существование дефекта приводит к масштабным последствиям катастрофического характера. Функция вообще ничего не может делать. \*сломалась функция оплаты\*

***Высокая(major)*** – существование дефекта приносит ощутимые неудобства многим пользователям в красках их типичной деятельности. Что-то не работает, в целом нам плохо от этого, но мы еще не сошли с ума ☺ \*Нет возможности добавить какой либо товар в корзину, допустим товар очень дорогой на ресурсе\*

***Средняя(medium)*** – существование дефекта слабо влияет на типичные сценарии работы пользователей, и/иди существует обходной путь достижения цели. Не работает функция, возможно частично не работает, но мы можем это сделать каким либо другим способом. Допустим добавления продукта в корзину, не работает со страницы общего каталога, но зайдя в продукт, мы сможем его добавить.

***Низкая(minor)*** – существование дефекта редко обнаруживается незначительными процентом пользователей и (почти) не влияет на их работу.

**Срочность(priority)** показывает, как быстро дефект должны быть устранен.

***Наивысшая(ASAP, as soon as possible)*** срочность указывает на необходимость устранить дефект настолько быстро, насколько это возможно.

***Высокая(high)*** – срочность означает, что дефект следует исправить вне очереди, т.к. его существование или уже объективно мешает работе, или начинает создавать такие помехи в самом ближайшем будущем.

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

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

Примеры **ВЫСОКОЙ** серьезности и **НИЗКОГО** приоритета:

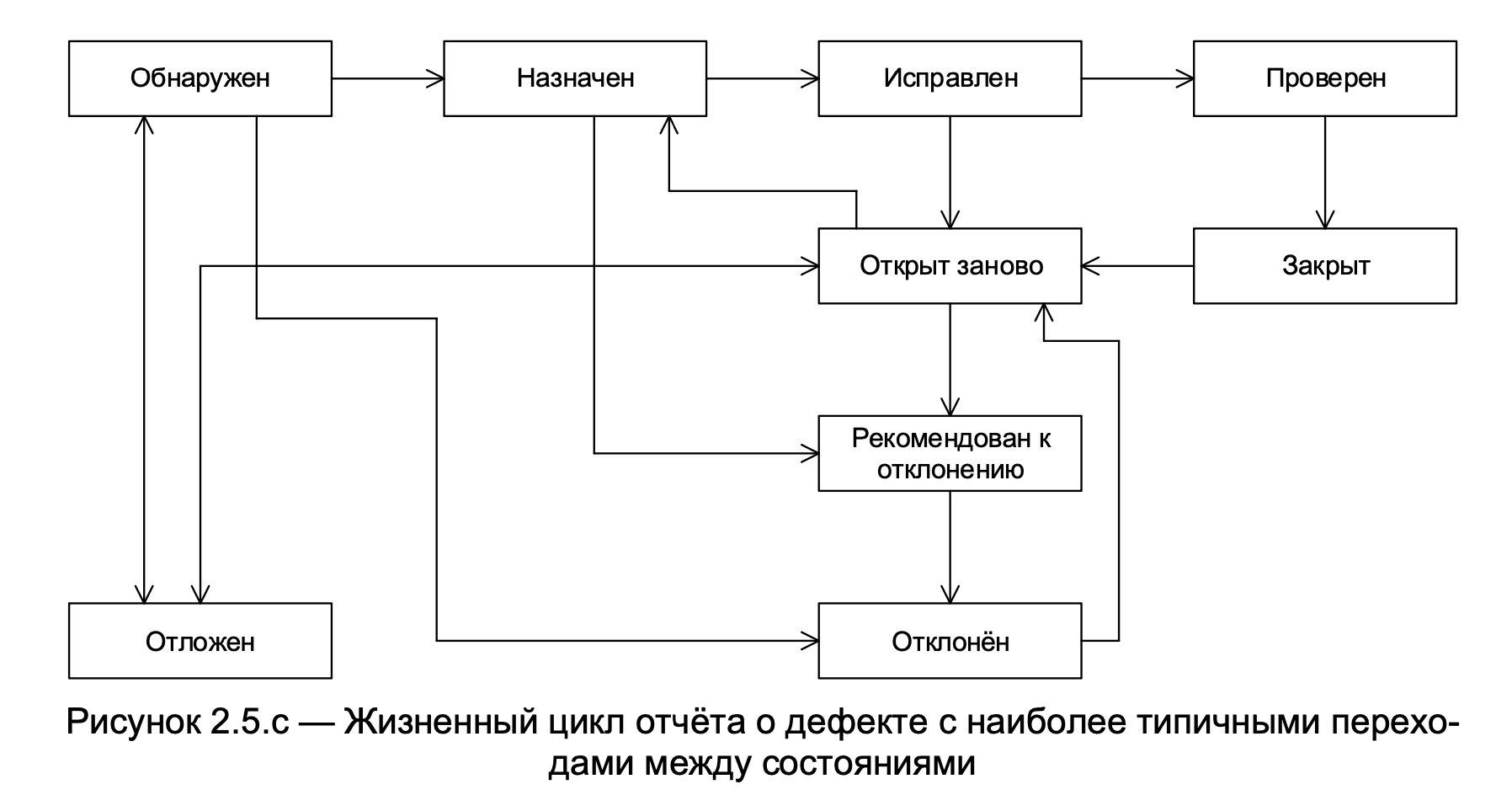
У нас есть функциональность, которой мы не пользуемся долго, ее откладывают на потом, что бы мы ее починили.

ВЫСОКИЙ приоритет, но НИЗКАЯ серьезность:

Заходим на сайт Гугл, у нас есть опечатка, она по сути не влияет на работу продукта, важность будет низкая, но с точки зрения бизнеса, если у нас опечатка на главной странице, это критическая, мы должны сразу это дело чинить.

**Комментарий(comments, additional info)** дополнительные данные о дефекте, возможность упомянуть исполнителя.

Вложения(attachments) подтверждение дефекта, фото, видео, логи, тестовые данные, архивы и т д.



В начале дефект обнаруживается, после мы его должны на кого-то назначить (на разработчика) и если всё хорошо, то разработчик берет его в работу. Его исправляют, после мы проверяем (re-test) и закрываем.

Бывает так, что дефект обнаружен, его назначают, но он плохо описан, либо это не дефект а какая либо функция. Бывает так, что тестировщик может не знать как это должно работать \* только пришел на проект\* а это было сделано 10 итераций до него, в требованиях он не нашел этого. Менеджер смотрит и говорит, что есть такое требование, дает ссылку и мы можем отклонить.

Либо бывают мы завели дефект и его откладывают, так как низкий приоритет, либо сейчас эта функция будет перерабатываться и в этом случае нет необходимости его фиксить.

**ТИПЫ УЛУЧШЕНИЙ**

Помимо багов, в системе есть улучшения. Их надо каким-то образом хранить и документировать. Отчеты о дефекте, похожи на улучшения.

**Improvement** – запрос на улучшение существующей функциональности. К примеру, у нас есть функциональность по регистрации в систему, но нет возможности регистрироваться через социальные сети.

**Feature** – Запрос на расширение ПО новой функцией или изменение существующей функциональности.

**Change Request** – запрос для настройки или изменения системы. Обычно создается бизнес-аналитиками когда они уже написали какую-то функциональность, она была реализована и если мы вносим изменения в существующие требования , то делаем это через change request, и она оформляется как user story.

**Enhancement** – запрос на воплощение новых идей, нового поведения или новой функциональности. Этого не было, и мы можем предложить это.

Возможно менеджер заметит предложенное нами улучшение и захочет внедрить в продукт.

ПОКАЗ ВОЗМОЖНОСТИ ФУНКЦИЙ JIRA и AZURE

(описать более подробно при необходимости)

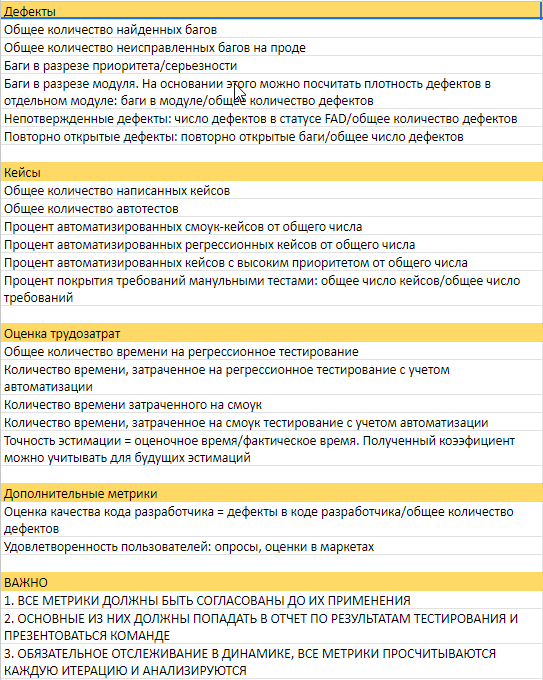
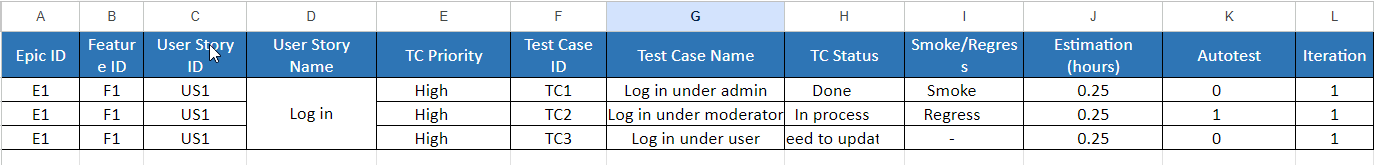
МАТРИЦА ТРАССИРОВКИ

Матрица трассировки позволяет нам показать на какие требования были написаны кейсы.

**Матрица трассировки** – это двумерная таблица, описывающая связь двух сущностей (например требований и тестовых сценариев). Таблица позволяет производить прямую и обратную трассировку от одной сущности к другой. Обеспечивая таким образом возможность определение покрытия и оценки влияния предполагаемых изменений. Помимо того, что мы сможем оценить тестовое покрытие, мы можем увидеть какое именно требования повлияет на ту или иную функциональность.



В первом приближении матрица трассировки выглядит так. В первом столбе описываются все требования и потом делаются столбы на каждый кейс в системе. На пересечении ставим плюсик, и дальше анализируем на какие требования написаны кейсы.



(подробности рассказывает в курсе, взять оттуда, если есть необходимость)

ОЦЕНКА ТРУДОЗАТРАТ В ТЕСТИРОВАНИИ

Оценка затрат на тестирование – это аппроксимация объема трудозатрат, связанная с различными аспектами тестирования.

Популярные техники эстимации.

1. **Пальцем в небо (методом проб и ошибок)**. Не рекомендуется применять, однако встречается очень часто. Когда вы сидите на совещаниях и разработчик говорит. \* Ну на эту задачу мне потребуется примерно часов 15\*. И все соглашаются, так как разработчик так решил.
2. **Процентное отношение к разработке**. Например кто-то когда-то сказал, что на одного тестировщика приходится 2-3 разработчика. Это по некоторым меркам норма. Разработчик потратил 20 часов, значит тестировщик потратит в 2 – 3 раза меньше времени. Это но точная оценка, но можно применять, если сложно что-то оценивать.
3. **Процентное распределение**. На все фазы SDLS выделяется определенный процент времени. Время выделенное на тестирование, разделяется на отдельные фазы STLS.
4. **Эстимация, основанная на предыдущем опыте**. В различных TMS системах, есть счетчики времени тестирования. В одних они запускаются в ручную, в других автоматически. После нескольких прогонов тест- кейсов уже можно определить, сколько в среднем мы тратим время на тестирование.
5. **Структурная декомпозиция.** При которой крупные задачи разделяются на более мелкие, которые легко оценить. И условно у нас записано, сколько мы потратили времени на тот или иной кейс раньше и можно из этих данных формировать информацию, сколько времени мы потратим на общее тестирование.

**ПРИМЕР ЭСТИМАЦИИ.**

Команда состоит из 3 человек, все тестировщики и мы проводим регрессионное тестирование. В регрессионном наборе 30 кейсов и на прохождение одного кейса тратиться 10 минут. Плюс ко всему мы должны понимать, что при тестировнии находятся баги и мы должны закладывать время на проверку исправления этих багов 20%. Для подсчета мы не будем учитывать время написания документации и уточнение требований.

***Время на регрессионное тестирование (30 \* 10)/3 = 100минут***

***Время на re-test 100\*0.2 = 20минут***

***Примерное время тестирования 120 минут.***

Обязательно надо учесть риски: Недоступность членов команды, проблемы с ПО, внештатное увеличение количества багов. Заложим на это еще 10% времени.

***Общее время тестирования 132 минуты***

1. Эстимация по трем точкам:

По сути мы используем те же самые данные которые у нас есть, из прошлого примера. Но в том числе мы вводим некую неопределенность.

У нас есть формула

(О + (4 x М) + Е) / 6

О - оптимистичная оценка, когда всё идет по плану.

М - наиболее вероятная оценка, с учетом возникающих проблем

Е – пессимистичная оценка, когда всё идет не по плану

Оптимистичная оценка – это время которое мы потратим на регрессионное тестирование 100 минут.

Пессимистичная оценка – это оценка с нашими рисками и будет составлять 132 минуты

И предположим, что нет рисков но есть re-test

В итоге получаем **( 100 + (4 \* 120) + 132 ) / 6 = 118.7 минут**

Так же необходимо определить стандартное отклонение, оно используется во всех расчетах **( Е – О ) / 6**

Финальный вариант **118.7 +/- 5.3 минуты**

От 1200 px и меньше отсутствует адаптивная верстка.

# **ТЕСТИРОВАНИЕ WEB-ПРИЛОЖЕНИЙ**

Интернет – основа сети, техническая инфраструктура, благодаря которой и существует всемирная паутина.

Давайте разберемся, что такое сеть. Первоначально это 2 ПК связанных вместе. Связаны они могут быть через провод – Ethernet либо через wifi.

Если у нас будет много ПК, то связать из одними проводами будет сложно, для этого и придумали маршрутизатор.

Маршрутизатор, тоже своего рода компьютер, к которому могут быть подключены другие устройства.

Но проблема в том, что через маршрутизатор, мы можем связать ПК лишь локально.

Но мы можем связать между собой несколько маршрутизаторов. И появляется Сеть сетей. Таким образом мы сможем протянуть провода в одном районе ☺

Для дальнейшего пути соединения придумали МОДЕМ. К модему были присоединены телефонные кабеля, теперь уже оптоволокно. И эти телефонные кабеля и оптоволокно, позволяет связывать весь мир с нами.

Помимо модемов есть еще ПРОВАЙДЕРЫ. Провайдеры – это компании, которые предоставляют услуги в конкретной стране. Наши модемы, связываются с провайдерами, у них тоже встроены модемы «на максималках» и мы можем платя им деньги, выходить с помощью них в мировую сеть.

Итак, наше устройство находится в сети. Но что бы определять устройства в сети есть IP АДРЕСА, которые работают по интернет протоколу.

**IP адрес** – уникальный адрес устройства в сети.

Существуют две версии протокола IPv4 и IPv6

4 версия более старая и в ней ip записывается в виде 4 чисел, 32 битный адрес

192.168.2.10

Ввиду ограниченного количества IPv4 в 1999 году была представлена 128 битная версия IPv6. Например 3ffe:1900:fe21:4545:0000:0000:0000:0000

Так же разделяют **статические** и **динамические** адреса.

Статические адреса – назначаются одному устройству и не меняются.

Динамические адреса будут меняться при каждом новом подключении к сети.

IP адреса имеют псевдонимиы – доменные имена. Именно их мы видим в адресной строке.

Если говорить сильно упрощенно, когда мы обращаемся к какому либо сайту, мы должны вводить его ip адрес, так как у сайтов тоже есть ip в интернете. Но так как людям не очень хочется видеть набор цифр в адресной строке, ввели доменные имена.

Есть еще MAC-адрес.

Это серийный номер который присевается каждой единице активного оборудования или некоторым их интерфейсам в компьютерных сетях Ethernet

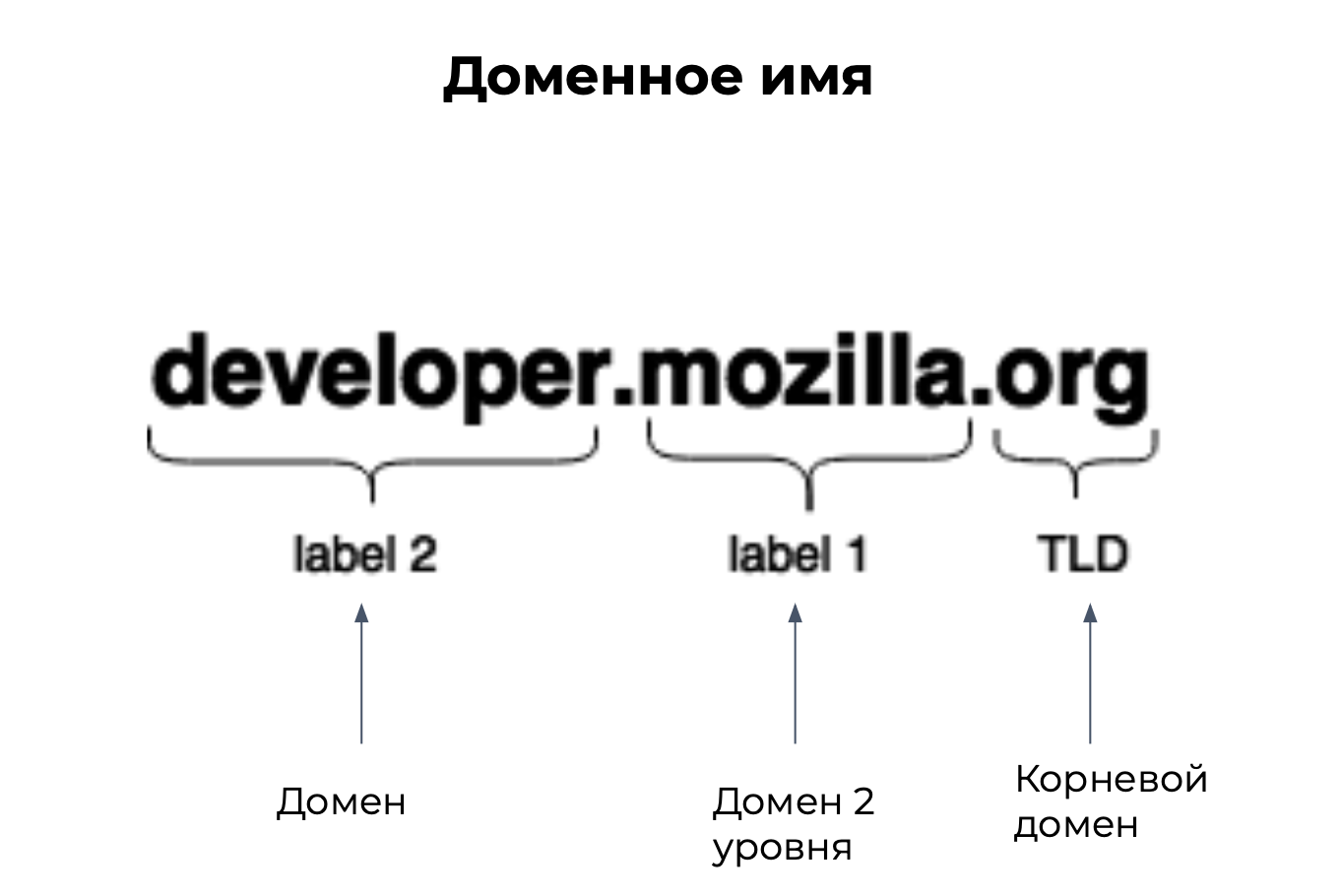
Любое устройство имеющее сетевую карту, будь то принтер, телефон, ноутбук, у них у всех есть МАС адреса.

МАС адрес состоит из шести групп по два символа, разделенных двоеточиями, например 00:1B:44:11:3A:B7

Как вы думаете, сколько МАС адресов у вашего компьютера ? У компьютера только MAC адресов, соклько устройств с сетевыми картами в него встроено.

**ДОМЕННОЕ ИМЯ**

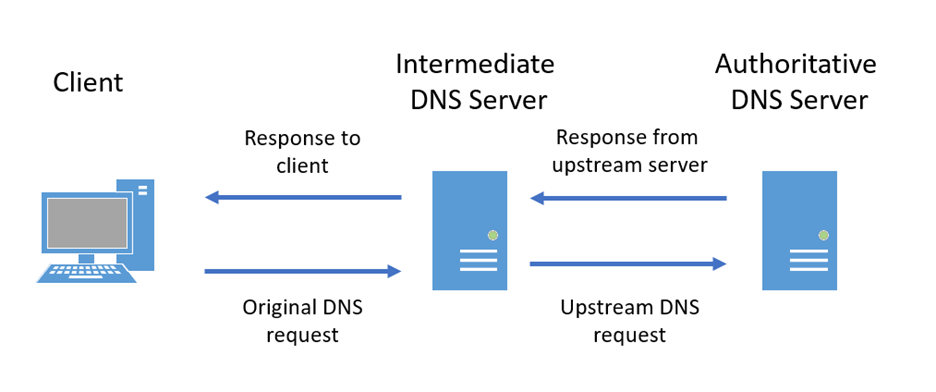
Итак, у нас есть IP адрес, и что бы он был удобо-читаемым для пользователя, то и существует доменное имя. Оно состоит из нескольких частей. Это корневой домен



Корневой домен общий, и покупается. Есть специальные компании регистраторы, у которых покупается этот корневой домен и регистрируется свое доменное имя и платим за это деньги на определенное количество времени.

Что касается домена и домена второго уровня, они уже могут настраиваться нами самостоятельно.

DNS – сервер. (Domain Name System)



У нас есть IP адрес наш и доменное имя которое мы вводим в URL. Мы должны как то перевести доменное имя в IP адрес, что бы понимать к какому серверу подключаться. У нас есть клиент, отправляется запрос с нашим доменным именем, он попадает на промежуточный DNS сервер и пытается найти на промежуточном DNS сервере информацию про наше доменное имя. ( да промежуточном DNS сервере есть список имен, которые размещены на промежуточном сервере ) Если в этом списке есть доменное имя, оно переводиться в IP, и мы по этому IP уже обращаемся к серверу, где размещен искомый нами ресурс. Если же информации нет на промежуточном, то идет переход на другие сервера.

Если чaсто мы обращаемся к одному ресурсу, его данные заносятся в DNS кэш.

ЧТО ТАКОЕ URL

URL (Uniform Resource Locator) – адрес, который выдан уникальному ресурсу в интернете

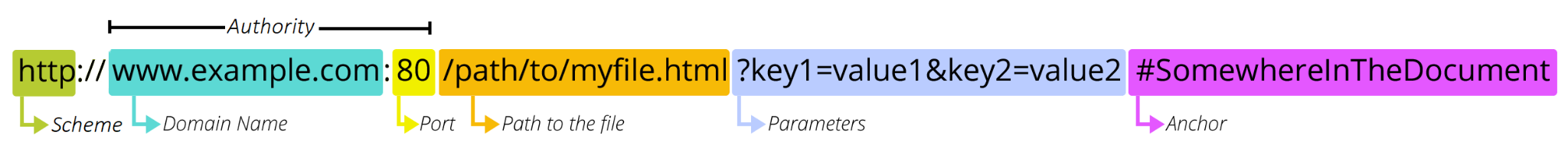


Схема (Scheme) – протокол по которому мы работаем. Есть еще протокол https – защищенный. (об этом будем говорить чуть дальше)

Доменное имя

Порт(Port) – это выделенное место для нашего ресурса

Путь до нашего файла (можно представить это как вложенные папки)

После знака ? идут параеметры. Это в принципе фильтрация, каждый фильтр является одним из парамертов.

Якорь(Anchor) если ресурс состоит из нескольких блоков, то у каждого блока есть название. И через этот якорь # мы можем перейти сразу на нужный блок.

Использовать параметры мы будет при тестировании API, пути так же, домен у нас само собой будет фигурировать. По типу протокола, мы будем понимать, защищен ли у нас ресурс или нет.

Если вы в жизни видите протокол http, без s на конце, то на такой сайт не стоит размещать личные данные.

Так же есть еще две модели OSI и TCP IP

Модель OSI – это набор протоколов, каурые служат для обмена информацией между компьютерами в локальной сети и интернет. Данная модель описывает работу сетевых устройств.

Всего в моделе 7 уровней, Мы чаше всего работаем с прикладным (HTTP FTP POP3 SMTP WebSocket) и Транспортным (TCP, UDP, SCTP, Порты)

Первые два уровня – это на уровне железа (Физический , канальный)

На сетевом уровне, появляются маршрутизаторы, сетевые шлюзы.

Далее идут 4 уровня, которые уже более интересны с точки зрения тестирования:

На транспортном уровне мы должны связать наш конечный пункт (ресурс который весть в интернете) и нашего клиента. И для этого есть несколько протоколов (далее мы их рассомтрим).

На сеансовом уровне идет управление сеансами связи ( с этими протоколами тестировка не сталкивается)

Уровень представления – представление шифрования данных

Прикладной уровнеь, самый интересный для нас, мы работает там с данными. Работаем на уровне, когда клиент видит какой-то ресурс. Больше всего будет работать с протоколом http и https

МОДЕЛЬ TCP/IP

В ней меньше уровней, но встречаются протоколы, которые мы видели и раньше

ТРАСПОРТНЫЕ ПРОТОКОЛЫ ( подробнее)



TCP – это надежность. При передаче через TCP мы будем всегда уверены, что у нас всё дойдет в том виде, в котором мы отправили.

UDP не проверяется, дошла ли информация. На нем устроены все потоковые сервисы ( к примеру стриминговые серверсы)

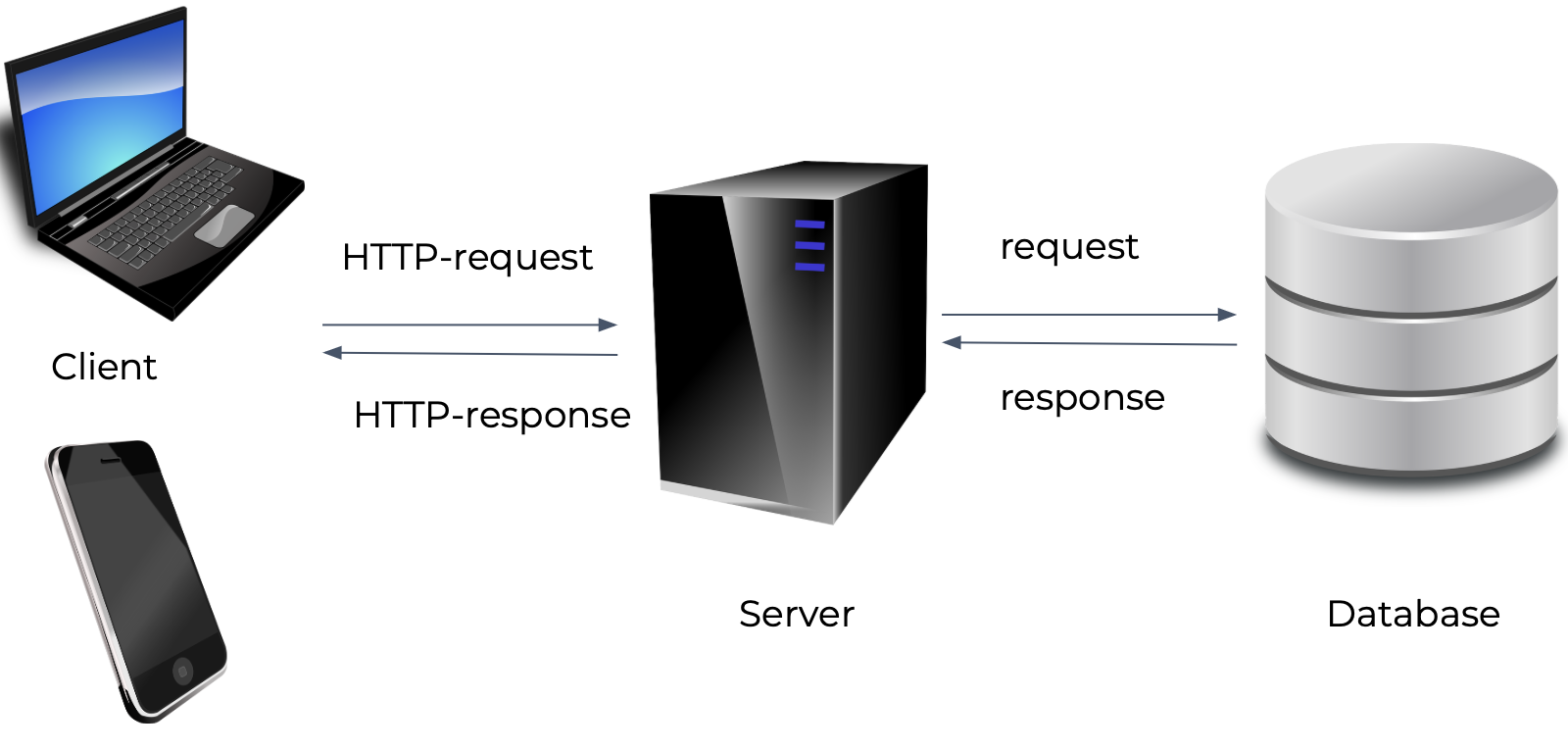
QUIC – надстройка над UDP.

UDP он быстрый, но не надежный. Так надстройка позволяет давать ему надежность.

КЛИЕНТ-СЕРВЕРНАЯ АРХИТЕКТУРА

Это база. Мы разобрались как устроен интернет, теперь мы должны разобраться в процессах, которые происходят в сети.

Клиент-серверная архитектура – вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемые серверами и заказчиками услуг, называемые клиентами. Фактические клиент и сервер – это программное обеспечение.



У нас всегда есть клиент, это то, с чего мы выходим в интернет и с помощью чего пользователь может взаимодействовать с интернет. Это может быть браузер на ПК, это может быть любое приложение, которое установлено на телефон. Инсаграмм, мобильные игры, это всё клиенты.

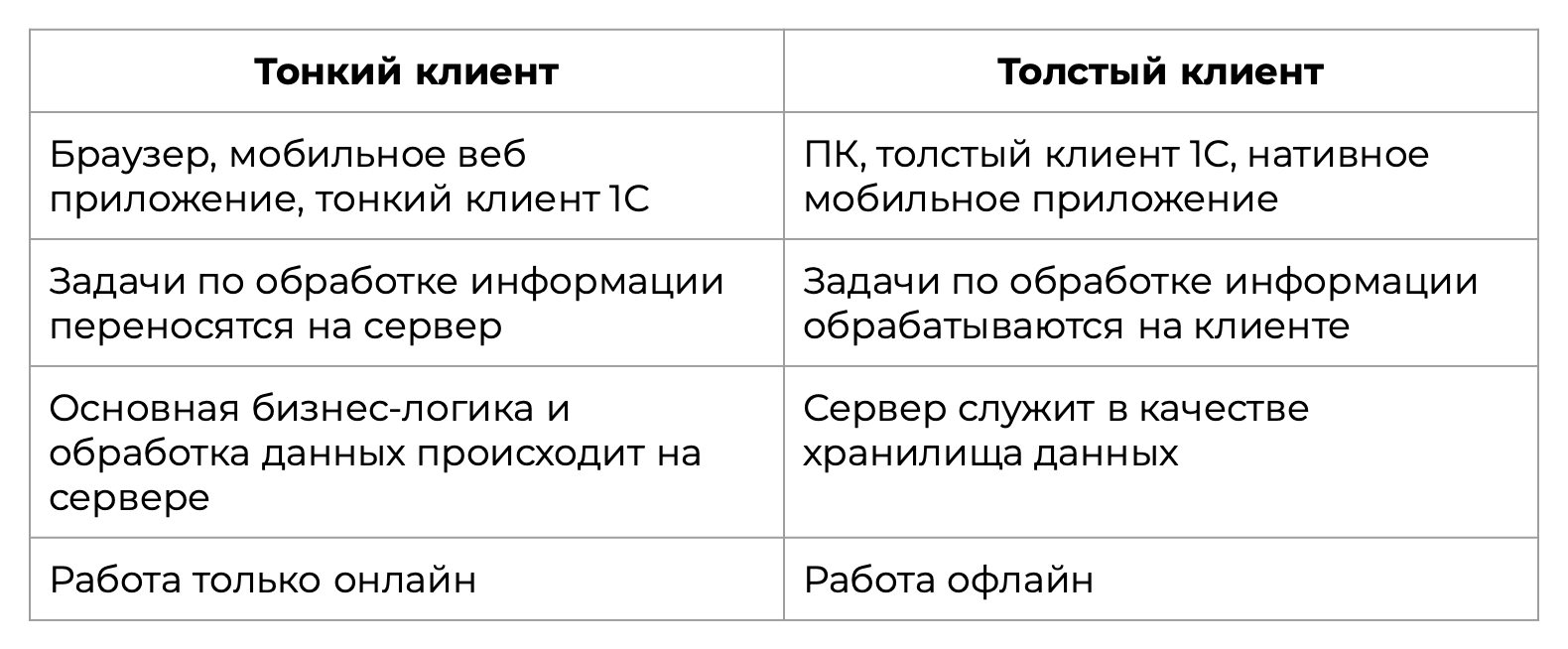
Есть еще сервер. Это компьютер на котором находится вся информация о ресурсе, к которому мы хотим обраться. Компьютер наш по сути тоже сервер, в котором находятся данные, которые лежат в папках, но это сервер только без выхода в интернет ☺

Но мы говорим про сервера, которые находятся в разных частях мира и мы к ним можем обращаться.

И есть база данных, всегда мы должны где-то хранить данные. Заходя на любой ресурс, допустим в интернет магазин, все товары хранятся в базе данных. Так же там храниться информация о пользователях, пароли, логины по которым мы входим в систему.

Итак у нас есть 3 звена. Трехуровневая клиент-серверная архитектура. Давайте разберемся как она работает.

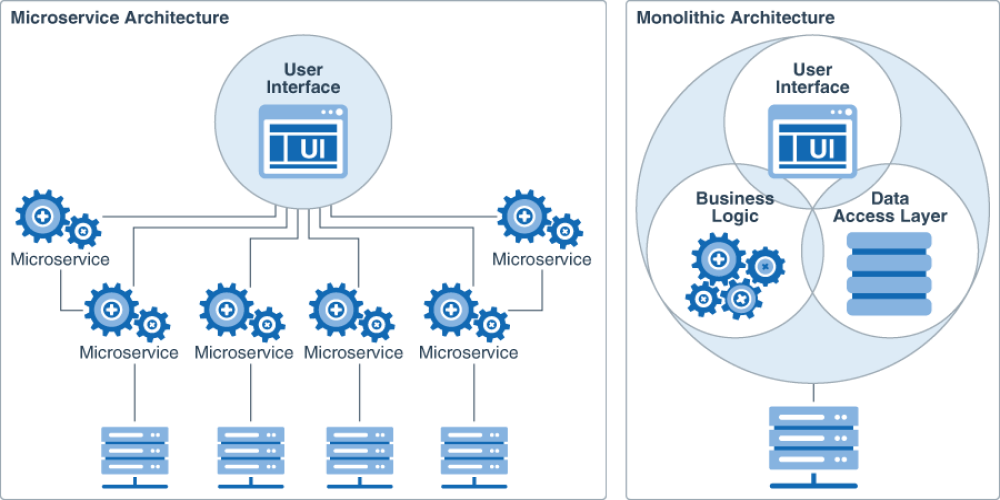
В нашем браузере мы вводим что-то в адресную строку. У нас отправляется **Http request (http запрос)**. Сервер получает это http запрос, обрабатывает, возможно если ему необходимо получить информацию из базы данных он обращается к базе данных. База данных получив запрос передает серверу информацию. Отправляет **response.** Сервер его анализирует и отправляет http response на наш клиент



В тонком клиенте все задачи выполняются на сервере, клиент предназначен для того, что бы отправить задачу и сервер всё сделает за него.

В толстом клиенте наоборот, все происходить на ПК а сервер уже выступает как хранилище данных.

МОНОЛИТЫ И МИКРОСЕРВЕРСЫ



Что касается монолита – у нас всё расположено в одном нашем приложении. Все интерфейсы, вся бизнес-логика, все данные

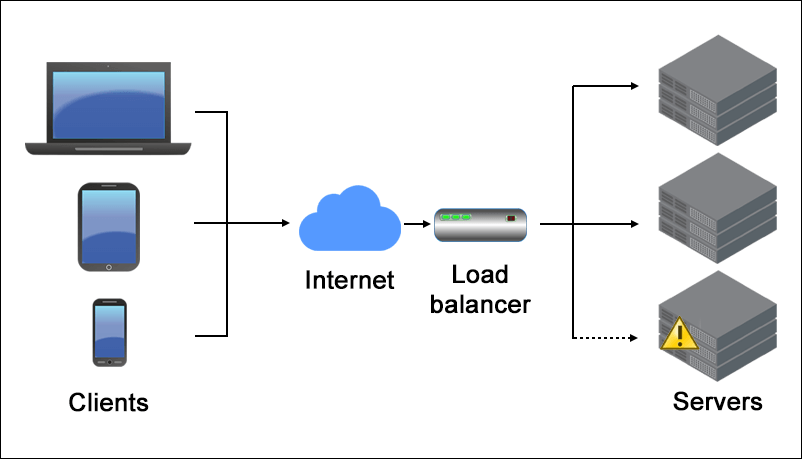
В микро сервисах наоборот у нас разделяются сервисы на определенные функции. К примеру в интернет магазине есть разные модули, регистрации, авторизации, модуль продажи товара, каталог, корзина. Для каждой отдельной задачи есть микро сервис со своими данными и со своей бизнес логикой. И эти микросервисы связаны между собой API интерфейсом.

В чем преимущество микросервиса. Если мы работаем под монолитом, что бы изменить любую функцию приложения, нам надо полностью задействовать весь монолит.

Если же мы находимся в микросервисе, если мы работаем к примеру над функцией регистрации, мы будем работать конкретно в одном микро сервисе. А все остальные задачи можно разрабатывать так же отдельно. Кто-то будет заниматься регистрацией, кто-то продажей товаров, кто-то корзиной. Все они самостоятельны, у них есть своя база данных. Сейчас отдают предпочтения микросрвисам.

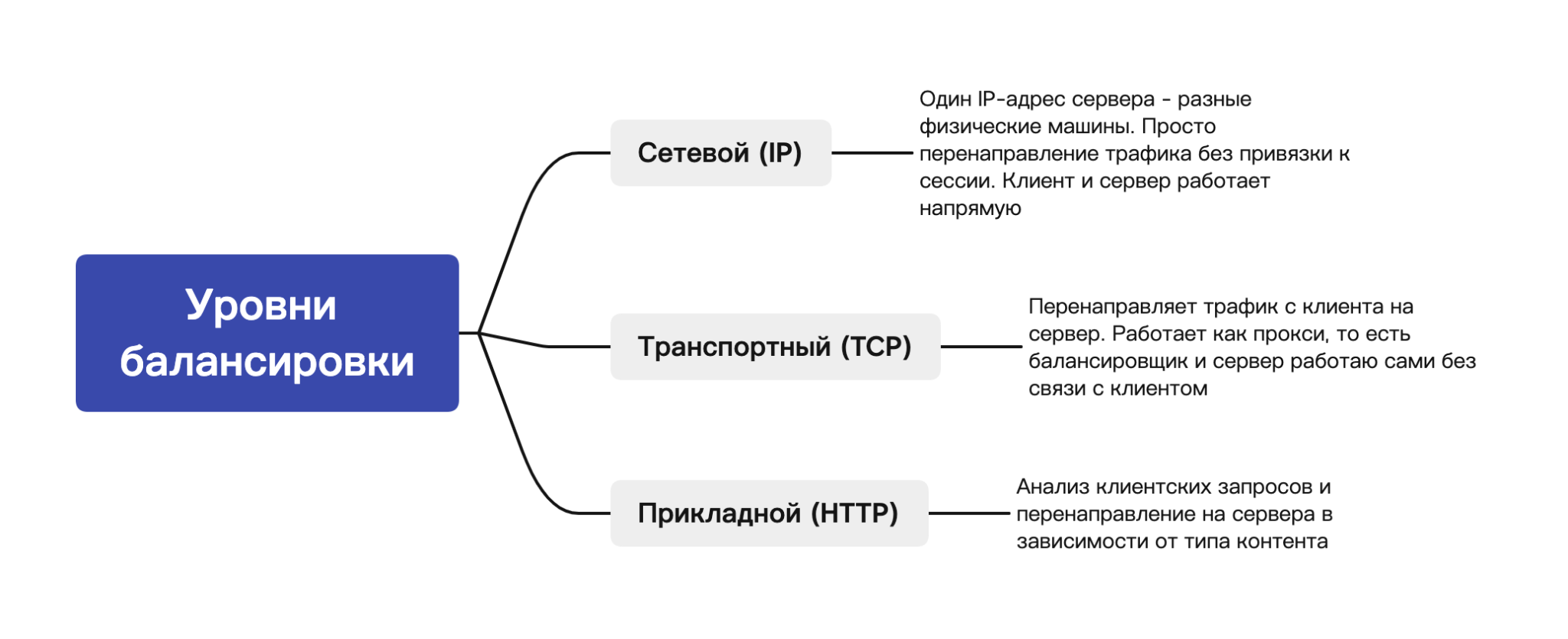
Даже тестирование в микросервисной системе удобнее. Те баги, которые мы будем находить в микросервисе, будут влиять на микросервис, а не на всю систему.

БАЛАНСИРОВЩИКИ ПРОИЗВОДИТЕЛЬНОСТИ



Это устройства, которые перераспределяют нагрузку. Не дают нагружать какой то один сервер.

Существуют три уровня балансирования.



Сетевой – Один IP адрес на все физические машины которые у нас есть. Представьте, что у нас несколько серверов, один отвечает за обработку данных, один за бизне-логику. И у нас для разных серверов есть один ip адрес, это очень удобно. И у нас идет просто балланисровщиком перенаправление трафика.

Транспортный – у нас идет перенаправления трафика с клиента на сервер. Есть такое понятие как PROCY – промежуточный сервер. Помимо сущетсующих клиента и сервера, между ними появляется proxy сервер. С помощью такого балансировщика, который работает как прокси, У нас клиент отправляет запрос, который попадает на балансировщик, и всё, клиент больше не взаимодействует с этим запросом. Прокси является основным источником коммуникации.

Прикладной уровень – будет проходить анализ, какие запросы ушли от клиента, и в зависимости от того какой тип контента был отправлен, то тот сервер и будет выбран.

HTTP – протокол. HTTP – методы. Запросы и ответы. Коды ответов.

**HTTP (HyperText Transfer Protocol)**— протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных

Всё, что передается в сети интернет, передается через HTTP протокол.

Есть много разных форматов, картинки, видео, реклама это все передается через GET запрос в сеть интернет.

HTTP – поток

Опишем каким образом происходит инициализация нашего соединения.

1. Открытие TCP – соединения. Для того что бы нам передавать информацию, мы должны его открыть.
2. То что мы отправляем называется HTTP сообщение.
3. Далее происходит чтение ответа сервера.
4. И когда мы закончили работу с соединением, мы закрываем TCP соединение.

HTTP методы

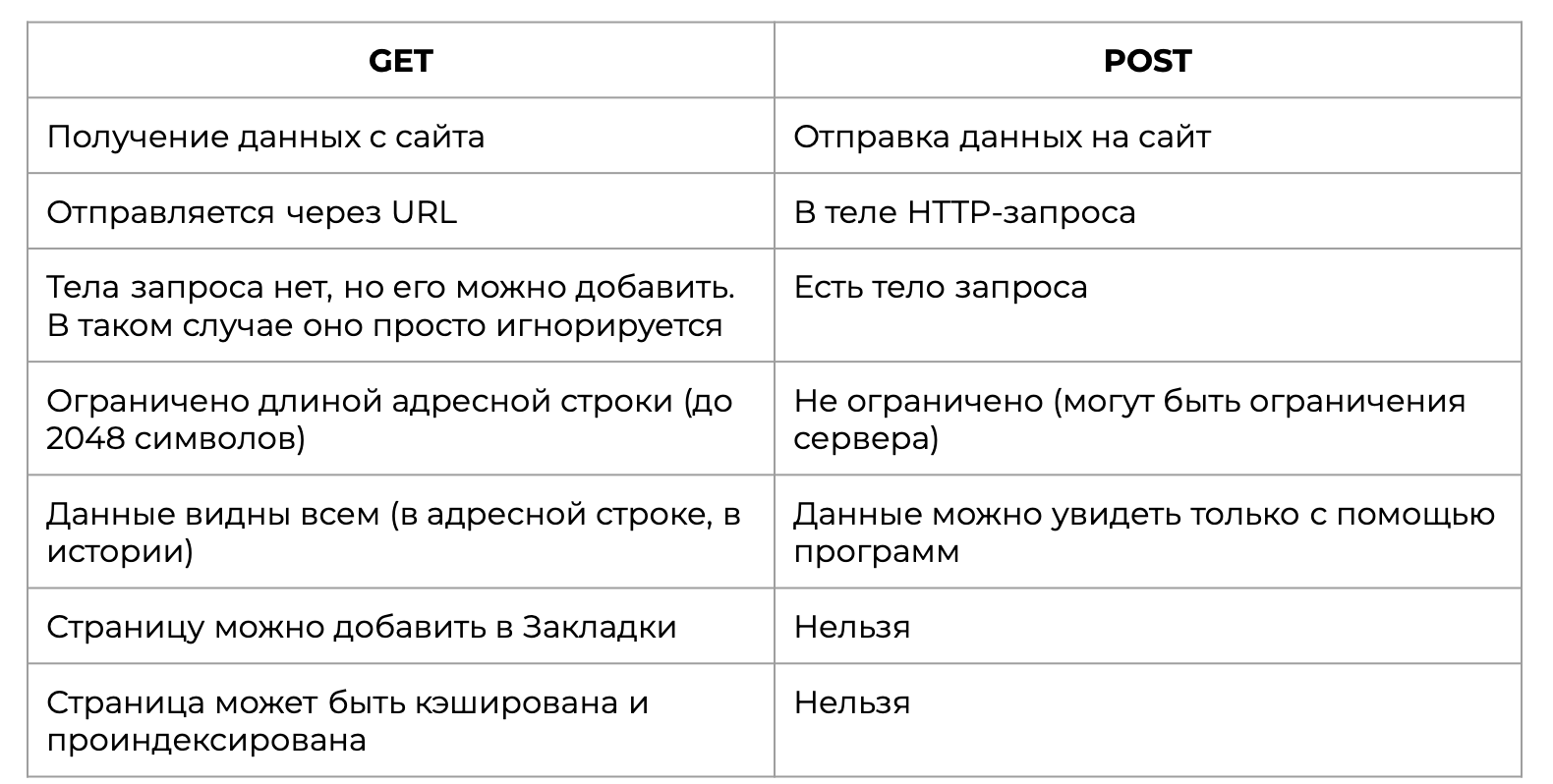
CRUD – создание, чтение, модификация, удаление.

C – создание : Post – создает какую либо сущность на сервере. В теле запроса POST передается полезная нагрузка и она сохраняется на сервере.

R – чтение GET – ничего не создает а просто читает инфомарцию с сервера.

U – модфикация PUT PATCH – изменение. Если нам надо заменить какую либо информацию на сервере. Мы используем эти методы

D – удаление Delete



GET – получает данные с сайте, а POST отправляет данные на сайт. Post – загрузка картинок, загрузка информации на сайт о себе к примеру.

GET – отправляется через URL. Post в теле запроса.

В Get – нет полезной нагрузки



Коды

200 **Успешно**. Всё прошло хорошо. Мы открываем ресурс и получаем 200 код.

201 При создании сущности через POST приходит 201 код.

301 **Перемещен на постоянной** основе. Был ранее ресурс aliexpress.com и решили из него сделать ali.com. И если пользователи вводили старый aliexpress.com их принудительно переводило на ali.com

304 **не модифицировано**. Кэш позволяет нам сохранить слепок страницы. И когда мы будем обращаться к ней, не заходить сразу на сервер, а искать на компьютере слепок и его загружать. И если ресурс подгружается из КЭШ приходит код 304.

400 **Плохой запрос**. Если мы ошиблись в синтаксисте при отправлении запроса, будет Bed Request. P.s. Если эта страница когда-то существовала будет 404.

401 **Не авторизован** При вводе логина и пароля проходит авторизация в системе. Если этот процесс не пройдет то получается 401

403 **Запрещено** Мы прошли на ресурс, прошли аутентификацию (ввод логина и пароля), но не прошла авторизация (выдача конкретных прав этому пользователю) К примеру пользователю были выданы права модератора, а он что-то хочет удалить из системы, как администратор. И приходит данный код.

404 Запрашиваемый ресурс не найден

500 Внутренняя ошибка сервера.

501 Не реализовано – Метод запроса не поддерживается на сервере.

502 Плохой шлюз. Сервер обрабатывал запрос и что-то пошло не так.

503 Сервис не доступен. Сервер не может обработать запрс.

504 В системе есть условно правило, что на выдачу ответа серверу дается 15 секунд, если это время привесилось, то нам приходит данный код 504

(Дописать из курсе, если будут лекции)

# **Кэш и куки. Разница в определениях веб-сайта, приложения, сервиса и сервера.**

Мы очень часто слышим такие понятия как веб-страница, веб-приложение, веб-сервис, веб-сайт. Но что такое web? Давайте разбираться.

Веб-страница – это документ который может быть отображён веб-браузером.

Эта страница написан на html, к ней применены различные стили, + подключается Js который дает динамику.

Когда мы связываем страницы под одним доменным именем – это Веб-сайт.

Бывают веб-сайты одностраничные. Но если к веб сайту подключают какую нибудь бизнес-логику. Внедряют в него калькуляторы, корзины. Связи между другими веб- веб сайтами. То это уже называется **веб-приложение**.

# **Веб-сайты выполняют информационную функцию, веб-приложение – это уже программное обеспечение.**

Веб-сервер – это компьютер, который имеет программное обеспечение для работы веб-сайтов

Веб-сервис – реализация абсолютно четких интерфейсов обмена данными между различными приложениями.

КЭШ – некаторе хранилище данных к которому есть быстрый доступ. В нем содержится информация, которая может быть запрошена с большей вероятностью. К примеру список товаров, который нас интересует.

**Серверный КЭШ**. Клиент отправляет запрос, сервер его обрабатывает, что-то загружет из БД. Условно мы хотим загрузить список товаров. Клиент отправляет запрос, он идет на сервер и на сервере есть КЭШ, мы стучимся к КЭШ и проверяем, есть ли там информация. По сути это выделенный объем памяти в котором храниться информация, к которой чаще всего идет обращение. Если этой информации нет, сервер уже обращается к базе данных, возможно сохранит эту информацию на сервере.

Существуют два виде подобно кэша.

1. Service Cache – Компания предоставляет определенный объем быстрой выделенной памяти
2. In-memory Cache – на самом сервере выделена часть памяти

**Клиентский КЭШ** – он хранится на компьютере. Мы обращаемся к ресурсу и попадаем на галвную страницу. Эту информацию браузер запомнит и сохранит в КЭШ который ест на ПК

Цепочка такая, в начале идет обращение к клиентскому КЭШ, если в нем нет информации то подключается к Серверному КЭШ, если там нет, то данные скачиваются с базы данных.

Управление и валидация кэширования.

Есть специальные хэдеры.

Cache-control – указывает необходимость сохранения кэша ресурса.

Expires header – прописывается информация, сколько должен жить кэш

Пример:

Cache-control: no-store (не сохраняют в том случае, если данные важны и надо их постоянно обновлять)

Expires header: Wed, 21 Oct 2024 07:28:00 GMT

При тестировании всегда очищаем КЭШ

Cookies

Это небольшие фрагменты данных, которые сервер отправляет браузеру пользователя, который храниться на компьютере пользователя.

Они используются для управления сеансом. В нем сохраняются наши логины, токены авторизации, корзины на сайте.

Персонализации. Пользовательские предпочтения. Когда мы соглашаемся и принимаем все КУКИ. Это стандарт GDPR. О нас сохраняется информация, куда мы заходим, что ищем, какой возраст. Это всё сохраняется в браузере и на том ресурсе, на котором мы находимся. И эту информацию используют в маркетинговых историях, реклама.

Даже если мы не согласимся с созданием кукис файлов, но будем продолжать пользоваться сайтом, то ресурс всё равно их будет собирать.

Рассмотрим как работают КУКИ на примере логина в систему.

1 запрос – логинимся в систему. – отправляем информацию на сервер. – сервер проверят есть ли в базе данных информация о нас и если есть, то подтверждает, что такой пользователь существует.

Нам выдается ТОКЕН – это набор символов, который подтверждает наши права на ресурсе.

ТОКЕН отправляется на КУКИ, но не просто так, а через Set-Cookie header. Нам сервер говорит, установи такие-то куки в твоем браузере. Информация с нашим ТОКЕНОМ сохранится в браузере, в специальном КУКИ файле.

И с каждым последующим запросом, это КУКИ будет с нашим запросом

У ТОКЕНА авторизации так же есть срок жизни, как только он устаревает, либо меняется, тогда при отправке запроса мы вновь будем совершать аутентификацию на ресурс.

**Виды Куки**

1. Сессионные куки – они удаляются при закрытии клиента.
2. Постоянные куки – удаляются при наступлени определенной даты ( атрибут Expires) или после определенного интервала времени (атрибут Max-Age)

Атрибуты куки

1. Secure – отсылаются на сервер только тогда, когда запрос отправлятся по протоколу SSL или HTTPS
2. HttpOnly – если нашему ресурсу не нужно что бы к нам обращались с помощью JS, мы пишем HttpOnly и никакой злоумышленник не сможет добавить скрипт, который будет информацию с нашего ресурса, передавать на другой.
3. Domain=github.com – указывает, на каком домене будут работать куки
4. SameSite – не дает возможности куки передавать на другие ресурсы. Только на один ресурс.

**Ctrs + Shift + N – Режим инкогнито.**

**Основы HTML и CSS**

Рассмотрев вкладку Chrome DevTools и обратив внимание на вкладку Elements мы видели HTML и CSS. Определимся с основами.

HTML – HyperText Markup Language – стандартизированный язык гипертекстовой разметки документов для просмотра веб-страниц в браузере.

1. Актуальная версия HTML 5 (5.3 на март 2023)
2. Структура HTML – документа
3. Основные теги и атрибуты

HTML Разместил блоки. И для того что бы выровнять их и сделать красивыми существует CSS

CSS - формальный язык описания внешнего вида документа, написнаого с использованием языка разметки.

В нем прописываются цвета размеры, различные формы и анимации.

1. Актуальная версия – CSS3
2. Способы подключения (index.html и style.css)
3. Основные атрибуты

Для того что бы разбираться в html и css скачать и установить visualstudio

Рассказать по информации из курса

Заголовки. Их всего 6 <h1></h1> - он отвечает за SEO оптимизацию. Он всего один на странице.

Расказать про абзацы <p>

Ссылки <a href="http://www.google.com/">Link</a>

Картинки

<img src=" " alt=" " width="500px">

Применение атрибута style

Рассказать про таблицы и списки

Рассказать про наполнение файла style.css. Как по тегам применяются стили.

Идентификаторы и классы

ТЕСТИРОВАНИЕ ГРАФИЧЕСКГО ИНТЕРФЕЙСА (GUI)

Тестирование графического интерфейса, является основной работой начинающего тестировщика. Хотя рынок меняется и вполне возможно с самых первых дней работы тестировщик уже будет тестироваться API.

Интерфейс – это способ взаимодействия. Либо пользователь взаимодействует с ПО, Либо ПО с другим ПО, либо ПО взаимодействует с железом.

В нашем случае мы говорим о взаимодействии пользователя и всего, что происходит в интернете. **User Interface**



Один из видов интерфейса – это интерфейс командной строки.

Графическией пользовательский интерфейс – кнопочки, поля ввода и т п. \*frontend\*

Жестовый, голосовой, тактильный, нейронный. - это и прослушивание чего либо, либо голосовое управление.

UX – User Experience – это то, что должно делать наше приложение, это его функции, это пользовательский опыт.

UI – это как приложение должно выглядеть, UX – это как приложение долго работать.

Перед тем как мы будем рассматривать GUI Надо обратить внимание на **ВИДЫ ВЕРСТОК**

На экране располагаются элементы, и в зависимости от видов верстки, эти элементы будут меняться в зависимости от разрешения экрана.

Раньше были два вида верстки:

1. Фиксированная – На экране находится элемент 100px и вне зависимости от размера экрана он всегда оставался 100px. Современное время это не актульано, но что бы ускорить разработку, на начальных этапах проекта можно использовать фиксированную верстку.
2. Резиновая – отличается тем, что у нас задано определенное число, размер изображения. И в зависимости от размера экрана, элемент не будет меняться, но возможно будет меняться его расположение.
3. Табличная – так же одна из старых видов верстки. Верстается страница таблицами, но ее не очень любят индексаторы Google, поэтому она уже не используется.
4. Блочная – используется. В ней мы разделяем код html На определенные блоки, и она часто используется с адаптивной версткой.
5. Адаптивная – в зависимости от размера экрана, у нас будет изменяться и его содержимое, размеры и могут появляться какие либо элементы, изменяться и исчезать. Те элементы, которые заточены под нужное устройство с такими размерами.
6. Flex-верстка, - это адаптивная на максималках верстка.

Работа в Figma

Расширение perfectpixel

**ЭЛЕМЕНТЫ GUI И КАК ИХ ТЕСТИРОВАТЬ**

w3schools.com

Слайдер – набор картинок, который собран в слайд-шок и переключается при помощью кнопок и стрелок.

* Тестируем переключение в лево в право; последний должен переводить на первый
* Проверяем разрешение картинки, которое мы можем вставить. ( можем тестировать классом эквивалентности. Размер изображения)

Форма для логина: Зависит от нашего сайта, есть юзернейм, пассворд и логин. Возможно ссылки на различные социальные сайты. Кнопка «забыли пароль» - которая переводит на специальную страницу

* Тестируются все валидации. ( используя классы эквивалентности)
* При позитивном тестировании, мы должны проверить, что в базе данных создался пользователь
* При негативных, мы должны проверить, что не можем создать пользователя с таким же username , с таким же password.
* Проерить что произойдет при вводе неверного логина и врного пароля, верного пароля и неверного логина, оставления полей пустыми
* Возможно кнопка Login должна быть disable если мы что-то не ввели.

Hover Dropdowns – реакция чего бы то нибыло на наведение.

* Тестирование цвета при наведении
* Появляющийся список
* Переход по ссылкам из спискам.

Modal Box , Modal Image

* Модальные окна нельзя закрыть просто нажатием на любое место на экране, только на крестик.

Serch bar – поиск

* Мы должны проверить, что, что-то есть на странице.
* Очень часто есть еще Автокомплит, когда мы вводим текст, появляется подсказка.
* Бывает в требованиях описан поиск по одной букве, по трем буквам.

Filter list

* Вмещался либо весь текст, либо появлялись три точки в конце

Sort List

* Проводится тестирования сортировки, по возрастанию, по убыванию и т п.

Pagination

* Тестирование переключение страниц, изменения их в зависимости от количества выводимого товара

Notification Button – счетчик в корзине

* Протестирования появление счетчкиа на иконке корзины, при добавлении товара.

ЛОГИРОВАНИЕ

Логи это некий файл, в котором прописываются все наши действия.

Существует три вида логов.

* Логи приложения - Относятся к работе внутри какого-то приложения.
* Логи сервера – Относятся к веб приложениям, развернутым на каком-то сервере. Разработчик прописывается логику того, как будет логироваться информация о тех действих, которые мы совершаем с сервером. Часто логи установлены на серверах по дефолту, но время от времени разработчики сами их пропи
* Системные логи

ТЕСТИРОВАНИЕ WEB СЕРВИСОВ

API

Рассмотрим теоретическую часть, а после будем непосредсвенно работать в Postman – это самое популярное решение которое есть сейчас на рынке.

Web-сервис – это реализация абсолютно четких интерфейсов обмена данными между различными приложениями, которые написаны не только на разных языках, но и распределены на разных узлах сети.

Что бы реализовать взаимодействие – необходимо использовать определенный интерфейс. У нас уже были интерфейсы CLI – интерфейс командной строки. GUI – пользовательский интерфейс. Но сейчас мы затронем интерфейс, который помогает общению разных web серверов, web приложений. Всё это реализовано через web сервисы.

API -- \*Aplication Programming Intergace\* - это механизмы, которые позволяют двум программным компонентам взаимодействовать друг с другом. Используя набор определений и протоколов.

По сути это то, что позволяет нам связать одну программу с другой программой

Или клиент с сервером.

С точки зрения API архитектуры – клиент – это тот, кто отправляет запрос. А сервер – приложение, которое отправляет нам ответ.

К примеру мы разрабатываем программу по прогнозу погоды. Мы являемся клиентом и отправляем запросы на сервера (это базы данных какой либо авиакомпании, либо метеослужбы ), которые могут дать нам метеосводки.

Или к примеру мы хотим подключить к нашей информационной системе платежную систему. Что бы не разрабатывать свою платежную систему, мы можем взять готовый вариант, на сервере.

И с помощью нашего клиента, обращаться именно конкретно к этой базе данных. И вот в этих случаях будет работать API

Есть несколько способов вызова API интерфейса.

Мы подробно затронем SOAP API и REST API

Websocket API, RPC APi мы не будем рассматривать ( но по ним есть очень много видео в интернете, и если вы захотите разобраться, можно это там спокойно найти).

Но при поиске работы, чаще всего требуется SOAP и REST,

**ЭТО ПРОТОКОЛ** - SOAP API (использует специальный протокол Simple Object Access Protocol) – Клиент и сервер обмениваются сообщениями по средством XML (XML это специальный формат передачи данных. О нем мы будем дальше говорить)

**ЭТО АРХИТЕКТУРНЫЙ СТИЛЬ** - REST API

Если SOAP – это набор каких-то правил, то REST – не имеет правил. Тут отправляются запросы на сервер в виде данных **JSON** – данный формат немножко похож на JS, так что тем кто изучает JS сейчас, немножко упроститься задача.

Сервер использует этот клиентский ввод для запуска внутренних функций и возвращает выходные данные обратно клиенту.

По REST API работает большенство приложений, которе мы знаем.

Наши HTTP запрос и ответ, методы GET POST DELETE – это всё работает в REST.

SOAP

Это протокол. Он работаем с небольшими файлами формата XML – они чем-то похожи на файлы html. Они предназначены для передачи и хранения данных.

XSD ( SML Schema) – это язык описания XML Документа. Это некая расшифровка, какие нам нужны атрибуты, сколько их должно быть, какие типы данных мы можем вводить. Всё это содержится в XML документе. Где-то на сервере лежит эта схема, которая проверяет отправляемые SML файлы.

WSDL – это документ написаный на XML – и является набором правил.

(слайд в презинтации)

XML Документ состоит из версии

Далее идет корневой эелемент <note>

И далее идет разделение на другие элементы. У элементов есть теги как и в html и есть информация внутри них.

Мы их должны тестировать.

Well Formed XML Documents

1. Должен быть корневой элемент
2. У элементов должны быть закрывающие теги
3. XML теги – регистрозависимые.
4. Правильное вложение элементов
5. Значения атрибутов должны быть в ковычках “ “
6. Некоторые значения должны экранироваться (>, <, &, ‘, ‘’)

Всё что отправляется в SOAP делается через метод POST/

REST API

Representational State Transfer – архитектурный стиль, который определяет условия работы API/

Если простым языком, это набор правил, который позволяет общаться нашему клиенту и серверу.

Шесть ограничений для RESTful API:

1. Единый интерфейс - мы всегда должны обращаться к одному url
2. Отсутствие состояний – сервер не сохраняет информацию о стстояниях и переходах клиента. Клиент запросил информацию, данные загрузили, больше данные никакие не сохраняются на сервере.
3. Кеширование ответа – всегда должна быть возможность определить, сохраняем мы кэш или. Эта информация должна быть в Хэдерах.
4. Клиент-сервер. В клиенте мы запрашиваем что-то от сервера. А на сервере хранятся все данные. Есть четкое разделение, что делает клиент, а что делает сервер.
5. Многоуровневая система. Сервер может находиться на разных уровнях. И сервер будет взаимодействовать с ближайшими уровнями и не будет сосредоточен на более высокие иди низкие уровни.
6. «Код по требованию» Если клиент хочет загрузить видео, то мы высылаем ему скрипт на это видео.

Это правила не обязательны, у нас API может быть и не RESTful, и это не будет ошибкой. Этим всем занимаются Backend разработчики, и если что-то не понятно в API необходимо обратиться к ним.

Для тестирования необходима документация, в которой написаны, какие запросы отсылаются, какие ответы получаются. И если тестировшику не выдают документацию, он не сможет провести тестирование API.

(обчыно по RESTful не спрашивают )

Архитектура REST похожа на архитектуру Http протокола.

Есть у нас клиент, мы отправляем GET POST DELETE, включается REST API и мы отправляемся к серверу. И возвращается JSON data.

JSON формат ( JavaScript Object Notation)

{“name”: “John”}

Ключ Значение

Информация которая отправляется в JSON хранится в ключах и значениях. В JSON храняться различные типы данных и необходимо так же выделять кавычками ключ. А значение в зависимости от типа данных.

1. String {“name”: “John” }
2. Number {“age”: 30 }
3. Object(JSON object){

“employee” : {“name”: “John” , “age”: 30, “city”: “New York”}

}

1. Arry {

“employee” : [“John” ,“Anna”, “Peter”]

}

1. Boolean {“sale”: true}
2. Null {“midlename”: null}

Как тестировать API

Если мы пришли на проект и получили такую задачу.

1. Запросить документацию a) В SOAP всё просто, есть документация по умолчанию в виде WSDL (читается через парсеры SoapUI, Wizdler). Нам дают ссылку, мы ее загружаем в специальные парсеры. Видем все методы и на основании того, какие данные мы можем отправлять, изучаем. Придумываем позитивные и негативные проверки и тестируем. b) для REST описывается в Swagger или других системах ( далее мы это подробнее рассмотрим)
2. Изучаем доступные методы.
3. Создаем коллекции, если работаем в Postman
4. Запускаем процесс тестирования. (Soap UI, Postman)

Что проверяют

1. Обязательные/необязательные значнеия
2. Отпарвка различных типов данных
3. Стстус-коды
4. Ответ от сервера ( заголовки, тело, ключи, значнеия, типы данных)
5. Авторизация и разрешения
6. Время ответа.

Методы используемые в POSTMAN

GET – позволяет найти какую либо информацию внутри сервера. Этот метод ограничен длинной адресной строки. Мы используем параметры, это не тело запроса. И эти **параметры всегда открытые.**

POST – предназначен для отправки чего-то на сервер. Создание какой-то сущности. Он не особо подходит для изменения данных, так как он **не идемпотентен.** Т.е. создавая новую сущность, он будет постоянно создавать что-то новое. Он не ограничен длинной адресной строки и в нем мы используем тело. (raw Json)

PUT – используется для изменения данных, так же он может и создавать новые данные. В него всегда нужно добавлять обязательный атрибут. Мы должны написать и обязательный атрибут и то, что хотим изменить. PATCH – позволяет изменить нужные данные без обязательных атрибутов.

DELETE – удаление данных

ТИПЫ АВТОРИЗАЦИИ В POSTMAN

**Идентификация**. Вы заходите на ресурс, и вас просят ввести логин. Это и есть идентификация. Есть такой человек в системе.

После ввода логина вводится пароль – это уже процесс **Аутентификации**.

Система понимает что мы есть назвавшийся пользователь, наш пароль совпадает с тем, что находится в базе данных системы. И это человек к примеру – администратор. И системе необходимо выдать ему права – **Авторизация.**

Авторизацию в Postman задают на уровне коллекции, и далее что бы не создавать авторизации по отдельным запросам, необходимо использоваться первый пункт «Inherit auth from parent» - наследование от авторизации коллекции.

Еще некоторые виды.

API Key – У нашей системы будет зашит для нашего профиля один API ключ, и будет информация о названии ключа и значении.

Basic Auth – базовая авторизация используется редко, когда мы используем логин и пароль. Несмотря на то, что вся информация будет хэшироваться, есть специальные инструменты, которые позволяют ее декодировать. Это не самый надежный способ.

OAuth1 и OAuth 2 – авторизация с использованием стороннего сервера. Когда мы хотим войти на аккаунт с использованием какой либо соц. Сети. Мы даем разрешение, что можно считать какие-то данные. И система действительно выдает какую-то часть наших данных. Выдается ресурсом специальный токен, который после используется в авторизации.

Bearer Token – мы так же будем вводить свой логин и пароль. Сервис выдаст нам токен, и вот его можно использоваться для автризации.

(Информация по перехвату трафика, настройки проски и Postman interceptor)

**АРХИТЕКТУРА ANDROID OS**

Принцип функционирования устройств основан на работе ~~не протоколов~~ а интерфейсов. Это уже нам знакомые ранее API интерфейсы, которые. Они есть на каждом уровне и связывают уровни между собой, начиная с нижнего первого и поднимаясь вверх.

Эта цепочка заканчивается СИСТЕМЫНМИ ПРИЛОЖЕНИЯМИ, это по сути то, с чем будут взаимодействовать тестировщики.

Итак, самый нижний уровень – это ЛИНУКС ЯДРО. Которое в первую очередь включает в себя драйверы, необходимые для взаимодействия с оборудованием. **В основе Android – лежит Linux.** Как ЛИНУКС так и все приложения на андроид – это открытый исходный код.

Что лежит в основе IOS до селе полностью неизвестно. Возможно это упрощенная версия MacOS.

Помимо драйверов в ядре есть различные системы. К примеру Power Management – система управления питанием, которая отвечает за реакцию на режимы питания. Поддержание компонентов активными. Распределение питания. К примеру некоторые приложения могут работать в фоновом режиме, вы выключаете экран, а музыка продолжает играть – в этом заслуга power management.

Управление памятью – позволяет распределять память равноценно, что бы каждое приложение могло брать память, которое ему нужно

Управление процессами – каждое приложение вызывает свой процесс, и система контролирует, что бы процессы работали одновременно и не конфликтовали.

**Следующий уровень это HAL ART Native c/c++**

**HAL –** обеспечивает связь между драйверами и библиотеками. Драйвера – связанны с работой аппаратных средств. Библиотеки – это некий код, написанный до нас разработчиками, которые упрощают взаимодействие с программами. И мы должны эти библиотеки связывать с драйверами с помощью специального API. Он состоит из отдельных модулей, которые организуют компонентный интерфейс.

ART – виртуальная машина для JAVA. JAVA – язык программирования. Многие приложения на андроид нативно написаны на нем. Но Javа – не может работать сама по себе, для этого нужна виртуальная машина.

Nativ c/c++ - набор библиотек. Они облегчают работу различных компонентов нашей системы.

**JAVA API Framework System Apps**

**JAVA API Framework –** пердоставляет разработчикам доступ ко всем функциям OS android. По факту это тот же самый режим разработчика, который позволяет при активации достучаться до тех элементом, которые не доступны обычному пользователю. Это API используются разработчиками при разработке приложений.

**System Apps –** установленные системные приложения. Когда мы в первый раз включаем андроид, у нас уже установлен google, playmarket, камера, календарь и т д. А так же всё, что мы устанавливаем из магазина, подтягивается на этот уровень системных приложений.

КОМПОНЕНТЫ АНДРОИД ПРИЛОЖЕНИЙ

Андроид приложения состоят из компонентов

* Activities (обсудим ниже)
* Services – позволяет выполнять фоновые задачи
* Broadcast receivers - отработка внешних событий, например – получение смс.
* Content providers – работа с данными. Базы данных, выкачка данных

Работу Андроид приложения можно разбить на Activities и fragments

Активити – это экран нашего приложения. Вот к примеру при открытии приложения появляется экран логина в систему – это Activity. В этом activity есть фрагменты – это отдельные ее элементы.

Если представить окно браузера – оно и есть Activity, а вкладки на этом окне – это фрагменты.

У Activities есть жизненный цикл. Мы должны их как то открывать, сворачивать в фоновый режим, останавливать и для каждого такого статуса есть **МЕТОДЫ ОБРАТНОГО ВЫЗОВА**

Эти методы позволяют перейти из одного **статуса** в другой.

**Created** – Активити создается, пользователь еще не может с ней взаимодействовать. ( мы нажимаем на иконку, и у нас открывается приложение)

**Started** – Активити запускается, получатель еще не может с ней взаимодействовать, но она уже видна ( к примеру появляется какой-то лоадер)

**Resumed** – Активити видна на экране и пользователь может с ней взаимодействовать. ( к примеру появляется экран логина в систему и мы уже можем с ним взаимодействовать)

**Paused** – Активити частично видна на экране и пользователь не может с ней взаимодействовать (к примеру мы хотим зарегистрироваться с помощью google, нажимаем на иконку и открывается страница, Остальное всё находится в Pause, мы его видим (если есть второй экран) но не можем с ним взаимодействовать.

**Stopped** – Активити остановлена и не видна на экране, но еще хранится в памяти. (работа в фоновом режиме)

**Destroyed** – Активити уничтожена и больше не храниться в памяти. (Зарыть приложение не сворачивая его)

МЕТОДЫ

onCreate() – переводит нашу Активити в статус Created

onStart() – запускает статус Started

onResume() - в работе статус Resumed

onPause()

onStop()

onDestroy()

На каждом из этих методов, есть свои особенности тетсирования, но это уже отдельная тема. Чаще всего тестировщики особенно на начальных этапах своей карьеры не переходят к ним.

АРХИТЕКТУРА IOS

В ней так же существуют уровни, которые связанны через API

Core OS – это ядро операционной системы – системные функции: аутентификация, безопасность, bluetoth и т д.

Core Services – Сервисы системы: геолокация, акселерометр, базы данных, сеть

Media Layer - графика, видео, аудио

Cocoa Touch – системные библиотеки. Основной уровень для разработки.

Тестирование IOS так же проходит на этом уровне Cocoa Tuch

АРХИТЕКТУРА ПРИЛОЖЕНИЙ РАБОТАЕТ НА ПРИМЕРЕ MVC

Model Controller View

Model – это по сути наши данные, которые тянутся из самого устройства

View – это интерфейс приложение

Controller – это API которая связывает наши данные

Работает на основе метода ОБРАТНОЙ СВЯЗИ.

Если пользователь что-то изменил через View на устройстве у нас идет сигнал на Controller и эти данные меняются в моделе.

Более подробно о элементах

Controller включает в себя:

* UIApplication Служит для управления событиями
* Application Delegate - Служит для управления событиями и позволяет переходить к Model ( модели данных)
* View Controller – обеспечивает размещение UI элементов на экране View (Активити) и UI Object (фрагменты)



ТЕСТОВАЯ ФЕРМА

Не всегда есть возможность использовать реальные девайсы и для некоторых целей используют специальные эмуляторы, симуляторы, фермы.

Необходимо уметь создавать эти фермы. Само собой начинающего специалиста не будут просить купить какие либо устройства, но в будущем при продвижении в карьере, такие задачи могут появляться.

Тестовая ферма – набор специальных устройств, программного обеспечения, для тестирования мобильных приложений.

**Реальные устройства** – гарантируют, что приложение будет работать без проблем на клиентских телефонах. Можно использоваться реальную и облачную ферму устройств. (Основная проблема в том, что они стоят очень много денег, мы никогда не сможем закрыть все потребности бизнеса используя реальные устройства, они постоянно обновляются и поэтому нашу базу, тестовую ферму, надо постоянно обновлять). Да, реальные устройства используют, но так же включают в тестирование эмуляторы и симуляторы.

**Эмуляторы** – воспроизводят все основные компоненты устройства, в том числе процессор, память и устройства ввода/вывода + поведение системы и ее интерфейса (Android Studio). Они более предпочтительны в использовании, но к примеру нет эмулятора для IOS. Но есть эмуляторы для Andriod

Симуляторы – воспроизводят поведение системы и ее интерфейс (xCode). Мы не можем симулировать железо, только графический интерфейс.

***Для тестирования IOS приходят только облачные фермы устройств.***

Далее надо определить, что мы будем включать в ферму.

Для этого необходимо изучить:

1. **Требования и особенности приложения** (Требования, которые написали аналитики и само приложение, будет понятно, для каких целей оно используется, возможно какие функции приложения оно использует - камера, блютус. Так же мы будем понимать какие функции завязаны на операционную систему, ее версии. В какой-то версии они могут поддерживаться а в какой-то нет.)
2. **Статистика** (Зайти в интернет и посмотреть, сколько IOS и Android больше в какой-то стране. Какие версии приложений используются, возможно какие-то версии уже давно не используют. Какие фирмы и для каких версий востребованы, к примеру Samsung – для Android, Apple – для IOS)
3. **Изучение целевой аудитории** (возможно подключение каких-то девайсов, к примеру работы телефона с умными часами, разработанными для спортсменов.)
4. **Наличие устройств в тестовой ферме (**Возможно ранее мы уже купили какие либо устройства)
5. **Использование эмуляторов и симуляторов**.(Тут необходимо изучить, какие части функциональности мы можем отдать этим устройствам)

ТИПЫ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

Нативные приложения:

1. Написаны на родной для определенной платформы языке программирования Android – Java/Kotlin, IOS – objective-C или Swift
2. Приложения хранятся на самом устройстве. (при скачивании, они хранятся на устройстве, на карте памяти. Если IOS то там карта памяти не используется, и они хранятся в облаке
3. Не обязателен выход в интернет
4. Могут быть установленны из официального магазина \*App Store Google Play\*
5. Могут использовтаься возможности устройства, камера, уведомления, gps и и т.д.

**Примеры:** Pokemon GO, Spotify

ВЕБ ПРИЛОЖЕНИЯ

1. Веб-приложения, адаптивны и оптимизированные под мобильное устройство (как пример это наши веб-сайты)
2. Часто похожи на мобильные приложения
3. Есть возможность добавить закладку на экран «Домой»
4. Кроссплатформенные
5. Не могут использовать возможность устройства

**Progressive Web Apps (PWA)** - позволяют создавать ярлыки определенных веб-страниц на рабочем столе мобильного устройства. Они выглядят как приложения для конкретной платформы и иногда могут работать в офлайн режиме.

**Accelerated Mobile Pages (AMP)**- это технология с открытым исходным кодом, позволяющая создавать веб-страницы, которые быстро загружаются в мобильных браузерах.

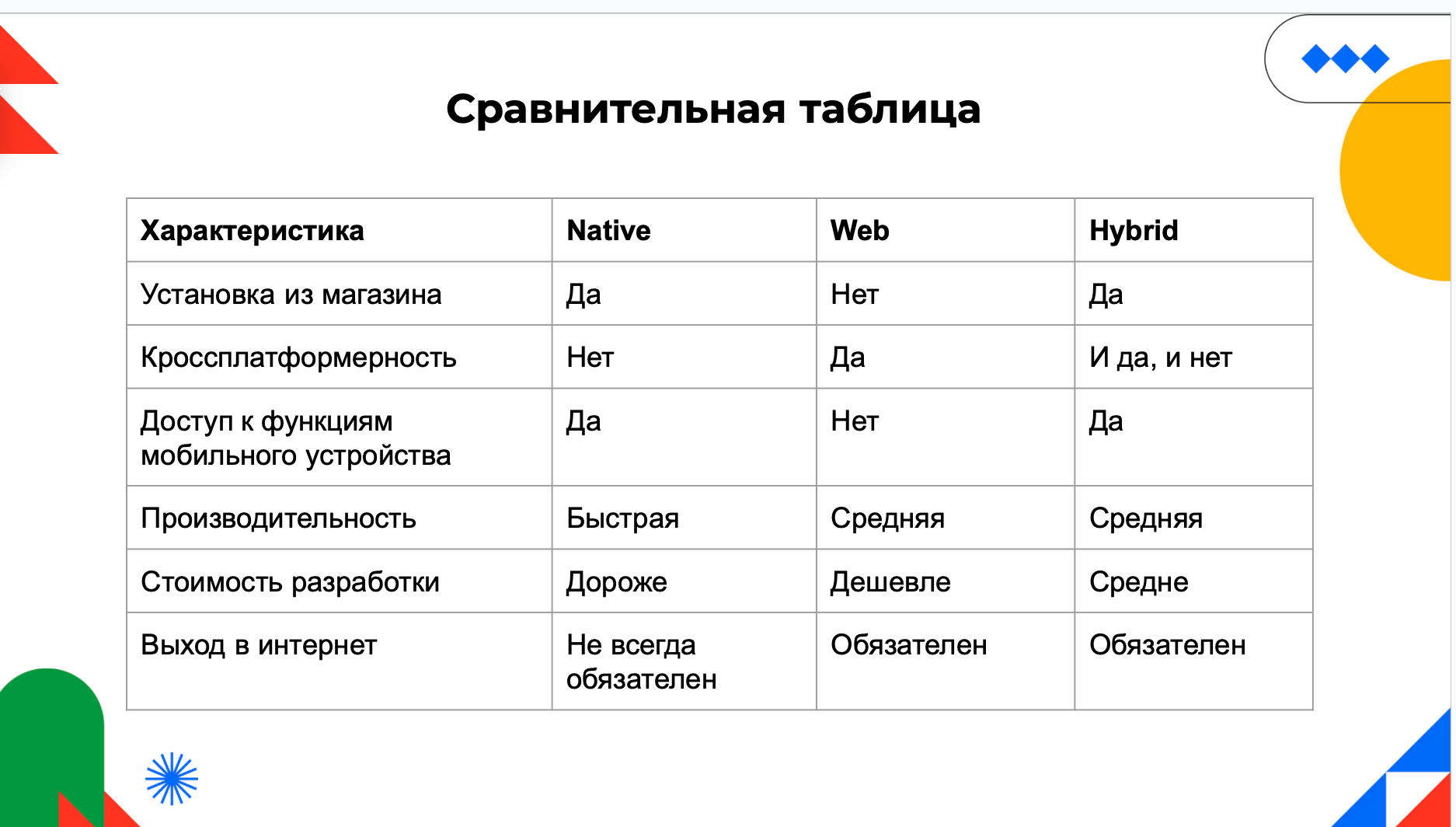
Гибридные и кроссплатформенные приложения.

**Гибридные приложения**

1. Веб-приложение в обертке нативного приложения, что служит контейнером для отображения веб-приложения через встроенный упрощенный браузер (Chrome webview в Android и WKWebView в iOS)
2. Приложения хранятся на самом устройстве
3. Кроссплатформерность
4. Могут быть установлены из официального магазина (App Store, Google Play)
5. Могут использовать возможности устройства - камера, уведомления, GPS и т.д.

**Примеры:** Uber, Evernote

**Кроссплатформенные приложения** разрабатываются с помощью кроссплатформенных фреймворков: React Native (JavaScript), Flutter (Dart), Ionic (JavaScript), Xamarin (.NET and C#) и т.п. и имеют общий код для iOS и Android.



ОСОБЕННОСТИ ТЕСТИРОВАНИЯ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

Как и упоминалось ранее, для тестирования мобильных приложений будут использоваться эмуляторы и симуляторы. Давайте определим, что конкретно можно будет тестировать с их использованием.

Совместимость с аппаратным обеспечением устройств.

Сюда включен процессор, внутренние функции.

1. **Разные способы выключения** (Специальная кнопка, нажатие нескольких кнопок, использовать программный бар – кнопку для выключения, Выключение устройства при полной разрядки батареи)
2. **Разные способы навигации** (Жесты, физические кнопки \*громкость, выключение\*, кнопки нажимаемые на экране)
3. **Использование аппаратной и программной клавиатуры** (можно использоваться как встроенную клавиатуру в устройство, либо внешней которая подключается через usb. Наши устройства не только мобильные телефоны и смарт-фоны. Это еще и планшеты, а к ним подключение клавиатуры – это норма.
4. **Различные аппаратные средства**

* Радио
* USB
* Bluetooth
* Камера
* Динамик
* Микрофон
* Гнездо для наушников

**Тестирование разных экранов**

1. Приложение масштабирует все элементы UI в соответствии с размером и разрешением экрана.
2. Элементы интерфейса не перекрывают друг друга
3. Не возникает проблем с практичностью или датчиками (датчики движения, освещения, датчики отслеживания нас в пространстве.
4. Не возникает проблем связанных с масштабированием изображений из-за высокого dpi (пиксели на дюйм) и ppi (точки на дюйм).

**Тестирование температуры устройства**

При зарядке устройства сильно нагреваются и необходимо обращать внимание как ведут себя приложения. В веб приложениях температура не сильно влияет на работу, а вот для мобильных – это критично. Если на iphon играть и одновренно заряжать, то при нагреве высоком, может приостановиться зарядка, для его охлаждения. Либо при съемке изображений в режиме 4K он может сильно нагреться, и система сообщит, что пока не может вести съемку в данном режиме и необходимо охладить устройство. При нагреве может снижаться частота процессора.

**Тестирование датчиков ввода данных**

GPS, гироскоп, магнитометр, аксилиромерт.

К примеру Акселирометр определяет наше положение в пространстве и различные режимы \*портретный и альбомный\* (горизонтально или вертикально держим телефон) меняется интерфейс.

1. Приложение работает должным образом по отношению к каждому из доступных датчико.
2. Функции, реагирующие на внешнее освещение, ведут себя должным образом при различных условиях освещения. (фото приложения, телефон меняет яркость от освещения помещений)
3. Датчики ввода и вывода звуковых данных реагируют правильно
4. Точное определение местоположения при следующих условиях:

* GPS включен или выключен
* Различный уровень сигнала GPS
* Использование других методов определения положения (если мы выходим в интенет по wifi тоже может определить положение)

|  |  |
| --- | --- |
| **Датчики** | **Функция** |
| GPS | Местоположение |
| Акселерометр | Изменение положения устройства в пространстве |
| Гироскоп | [Перемещения телефона в пространстве, скорость перемещения, определение стороны света](https://hs-store.ru/articles/interesnoe/chto-takoe-giroskop-v-telefone/) |
| Магнитометр | Компас |
| Датчики освещения, давления, температуры, влажности и т.д. | Понятно из названия |

**Совместимость с аппаратным обеспечением устройства**.

Тестирование различных способов ввода данных

* Программные клавиатуры. Работа и совместимость.(установка клавиатуры, переключение языков)
* Жесты и команды
* Камеры, используемые приложеним, могут захватывать изображения и видео, сканирование шитрх-кода.

**ЖЕСТЫ**

Некоторые жесты необходимо знать, они потребуются для написания тестовой документации.

**Tap** – Касание, нажание на сенсорный экран, что бы открыть любое приложение на смартфоне – мы тапаем его иконку.

**Double Tap –** два коротких нажатия.

**Multi Tap** – Три и более тапов подряд на одному элементу

**Long Tap** – нажатие с удержанием на несколько секунд. Позволит открыть дополнительные опции.

**Scroll** – вертикальное пролистывание содержимого скольжением пальца по экрану сверху в низ или с низу вверх

**Swipe** – Смахивание вниз, вверх, влево или в право.

**Pull to refresh** –довловный перевод (потяни до обновления)

**Drag&Drop** – изменение положения элементов интерфейса с помощью перетягивания

**Pinch** – жест, используемый для изменения масштаба картинки, пальцы разводят в стороны или сводят.

Тестирование изменения ориентации экрана.

1. Практичность и правильное функционировании приложения при переключения в портретный или альбомный режим
2. Приложение остается в том же состоянии
3. Поля ввода данных сохраняют собранные данные
4. Поля вывода данных отображают те же данные, сохраняя текущий сеанс.

Тестирование типичных прерываний.

Звонки, сообщения, подключение зарядного устройства, нехватка памяти. Переключение между приложениями.

1. Приложение правильно обрабатывает все прерывания без негативного влияния на поведение приложения
2. Приложение продолжает работать правильно сохраняя свой состояние, данные и сеант независимо от того , какое прерываени происходит
3. Если у устройства есть режим «не беспокоить» который блокирует уведомления, приложение должно гарантировать правильную работу.
4. Прерывания должны происходить во время использования приложения

**Тестирование прав доступа к функциям устройства**.

Это запрос разрешения на использования приложением функций устройства. Камры и т п. Необходимо смотреть, как будет влиять отклонения. Мы выключили камеру, значит какая-то функция приложения не должна работать. Либо после повторного запуска камеры через приложения проверить, как это повлияет на приложение.

**Тестирование энергопотребления и состояния**. Необходимо сомтреть, как работает устройства при разных уровнях заряда батарей. Сохранение данных, режим энергосбережения.

Тестирование взаимодействий приложения с ПО устройства

Тестирование уведомлений

1. Приложение должны иметь доступ к уведомлениям. Если приложение не работает, уведомления так же должны приходить. Либо если приложение работает, уведомления должны приходить в нутри приложения.
2. Переход в приложение по средствам нажатия на уведомления, уведомление должно содержать глубокую ссылку на страницу.

Тестирование ссылок быстрого доступа. ……..

Тетсирование пользовательских настроеке………….

ТИПЫ ТЕСТИРОВАНИЯ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

Тестирование установки.

Установка, обновление и деинсталяция приложения через следующие источники:

1. Магазины приложений
2. Загрузка неопубликованного приложения \*копирование и установка\* apk файлы.
3. Десктоптные приложения (Apple iTunes или Android App Installer)
4. Установка по беспроводной сети «Wifi или мобильные данные»
5. С помощью кабеля передачи данных (связь девайса через провод к компьютеру)

Более детально:

1. Установка, удаление и обновление приложения на встроенных и внешних запоминающих устройствах.
2. Переустановка приложения в случае, когда при деинсталляции была выбрана опция “Сохранить данные приложения”.
3. Переустановка приложения в случае, когда при деинсталляции не была выбрана опция “Сохранить данные приложения”.
4. Отмена или прерывание установки/деинсталляции приложения, например, при выключении мобильного устройства в процессе установки/деинсталляции или при потере соединения с Интернет.
5. Возобновление установки, деинсталляции и обновления после отмены или прерывания.
6. Тестирование прав доступа в процессе установки.
7. Обновление приложения с последующей проверкой, что данные не были потеряны.

Стрессовое тестирование, тестирование производительности

1. Высокая загрузка центрального процессора
2. Нехватка памяти
3. Нехватка места на диске
4. Загрузка батареи
5. Отказы
6. Низкая пропускная способность сети
7. Большое количество взаимодействий пользователя с приложением

Тестирование защищенности

1. Проверка входных данных для внедрения кола или переполнения буфера
2. Шифрование передаваемых данных
3. Шифрование локально хранимых данных
4. Удаление временных данных после использования или после аварийного завершения
5. Очистка текста в полях для ввода пароля.

Тестирование удобства использования

1. Быть интуитивно понятным
2. Позволять пользователю допускать ошибки
3. Быть последовательным в формулировках и логике работы
4. Не противоречить руководством по проектированию платформ
5. Обеспечивать видимость и доступность необходимой информации для каждого размера и типа экрана.

Тестирование ГЛОБАЛИЗАЦИИ , ЛОКАЛИЗАЦИИ и ДОСТУПНОСТИ

**Тестирование интернеционализации/голобализации** – приложение включает тестирование прложения для различных местоположений, форматов дат, чисел и валют.

**Тестирование локализации** – включает тестирование приложения с локализованными строками, изображениями и рабочими процессами для определенного региона

**Тестирование доступности** – проводиться для того, что бы выявить насколько легко пользователи с ограниченными возможностями могут использовать компонент или систему. Для мобильных приложений это можно сделать, используя настройки доступности устройства и тестируя приложениям для каждого параметра.

**Тестирование баз данных**

1. Проверка правильности хранения данных:

Синхронизация

Конфликты загрузки

Безопасность данных

Ограничения на данные

Функциональности CRUD

Поиск

1. Тестирование интеграции данных для данных, представляемых устройства( например, контактов) или сторонними приложения\*например, изображения/, видео и сообщений)
2. Производительность хранения данных на устройстве.