Тестирование – проверка соответствия между реальным и ожидаемым поведением программы.

Цели тестирования:

***– Предоставление информации о качестве ПО конечному заказчику;***

***– Повышение качества ПО;***

***– Предотвращение появления дефектов.***

Главная задача тестирования - ***поиск дефектов***.

**Ка́чество програ́ммного обеспечения** — способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям.

**ПРИНЦИПЫ ТЕСТИРОВАНИЯ**

***1. Тестирование показывает наличие дефектов***

Тестирование может показать наличие дефектов в программе, но не доказать их отсутствие.

***2. Исчерпывающее тестирование невозможно***

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

***3. Раннее тестирование***

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

***4. Скопление дефектов***

Разные модули системы могут содержать разное количество дефектов – то есть, плотность скопления дефектов в разных элементах программы может отличаться. Как правило, 80% проблем содержатся в 20% модулей.

***5. Парадокс пестицида***

Прогоняя одни и те же тесты вновь и вновь, Вы столкнетесь с тем, что они находят все меньше новых ошибок.

Чтобы преодолеть этот парадокс, необходимо периодически вносить изменения в используемые наборы тестов.

***6. Тестирование зависит от контекста***

Выбор методологии, техники и типа тестирования будет напрямую зависеть от природы самой программы. Например, программное обеспечение для медицинских нужд требует гораздо более строгой и тщательной проверки, чем, скажем, компьютерная игра.

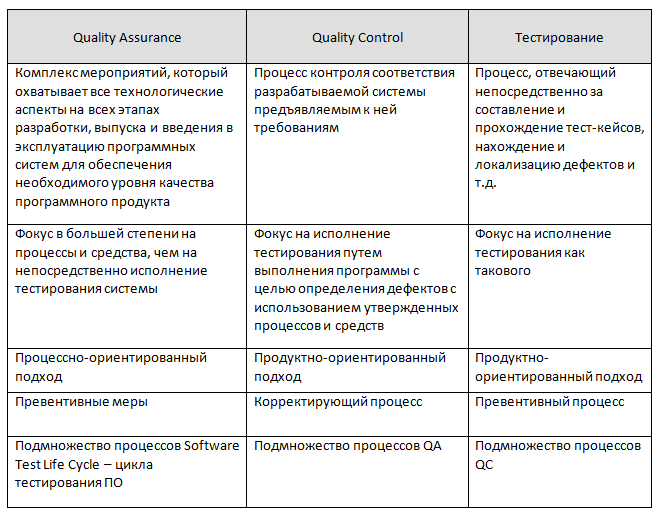
***7. Заблуждение об отсутствии ошибок***

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

ПОЛНАЯ СТАТЬЯ: <https://qalight.ua/ru/baza-znaniy/printsipyi-testirovaniya/>

**QA, QC, ТЕСТИРОВАНИЕ - РАЗЛИЧИЯ,**

**ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ**





ПОЛНАЯ СТАТЬЯ: <https://qalight.ua/ru/baza-znaniy/qa-qc-i-testirovanie/>



**Верификация** – подтверждение того, что определенные требования были выполнены.

**Валидация** – проверка того, что продукт отвечает ожиданиям и потребностям пользователей.

МОДЕЛИ РАЗРАБОТКИ ПО

**«Waterfall Model» (каскадная модель или «водопад»)**

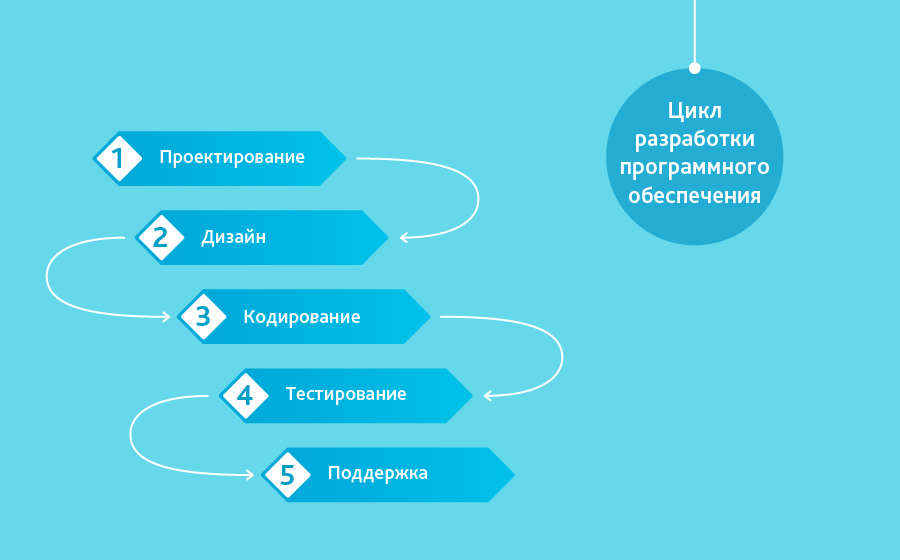
Одна из самых старых, подразумевает последовательное прохождение стадий, каждая из которых должна завершиться полностью до начала следующей.

Используется тогда, когда:

- требования известны, понятны и зафиксированы. Противоречивых требований не имеется.

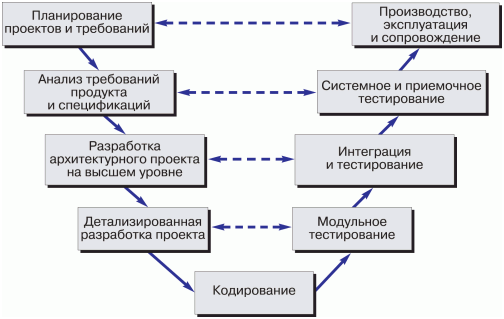
- нет проблем с доступностью программистов нужной квалификации.

- в относительно небольших проектах.



**V-образная модель**

V-модель – это улучшенная версия классической каскадной модели. Здесь на каждом этапе происходит контроль текущего процесса, для того чтобы убедиться в возможности перехода на следующий уровень.



**Инкрементная модель**

Инкрементная модель — это процесс разработки программного обеспечения, в котором требования разбиваются на несколько отдельных модулей цикла разработки программного обеспечения.

Процедура разработки по инкрементной модели предполагает выпуск на первом большом этапе продукта в базовой функциональности, а затем уже последовательное добавление новых функций, так называемых «инкрементов». Процесс продолжается до тех пор, пока не будет создана полная система.



**Agile model (гибкая модель)**

**Agile** – это методология, можно сказать даже некая философия, используемая при разработке. Она имеет 4 главных ценности, которые отражают суть Эджайл:

***1. люди и взаимодействие важнее процессов и инструментов;***

***2. работающий продукт важнее исчерпывающей документации;***

***3. сотрудничество с заказчиком важнее согласования условий контракта;***

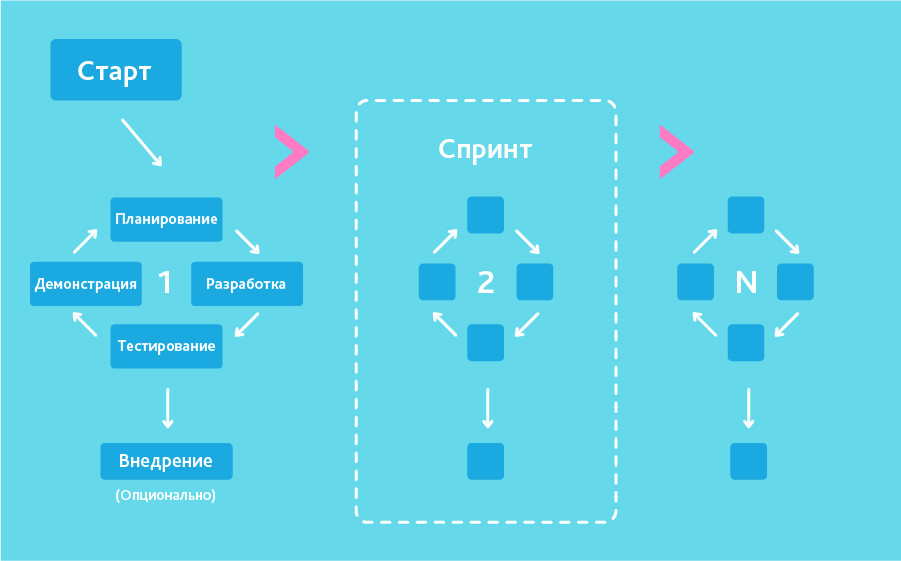
***4. готовность к изменениям важнее следования плану***

Существуют два основных фреймворка, которые основываются на принципах этой методологии. Это ***Scrum*** и ***Kanban.***

Суть ***Scrum*** в том, что вся работа делится на спринты — промежутки времени от 1 до 3 недель. Как только спринт выполняется, происходит его оценка и анализ проделанной работы, чтобы понять, что можно улучшить.

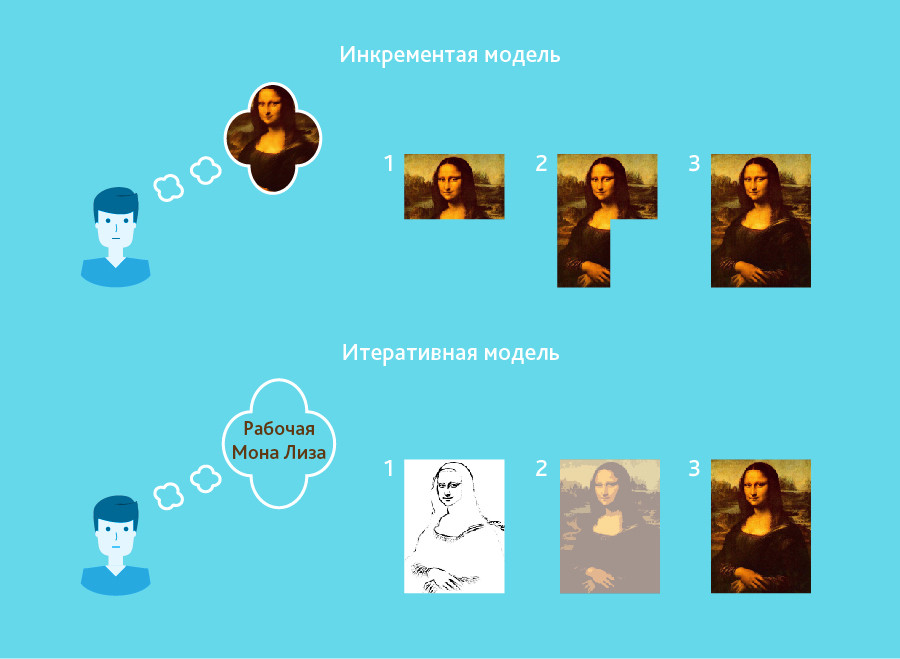
***Scrum*** подойдет для объемных проектов со строгим дедлайнами, где всю работу можно распланировать по четким спринтам, зная, сколько времени тратится на выполнение той или иной задачи.

Работа по ***Kanban*** строится вокруг ***kanban-доски***, которая используется для визуализации и оптимизации рабочего процесса. На стандартной Kanban-доске процесс состоит из трех шагов: «Запланировано», «В работе» и «Сделано». Однако доску можно настроить в соответствии с процессом, принятым в той или иной команде.



**Итеративная (итерационная) модель**

Итерационная модель жизненного цикла не требует для начала полной спецификации требований. Вместо этого, создание начинается с реализации части функционала, становящейся базой для определения дальнейших требований. Этот процесс повторяется. Версия может быть неидеальна, главное, чтобы она работала.



**Когда использовать?**

- требования к конечной системе заранее четко определены и понятны.

- проект большой или очень большой.

- основная задача должна быть определена, но детали реализации могут эволюционировать с течением времени.

**Спиральная модель**

Спиральная модель представляет собой повторяющуюся последовательность циклов разработки с непрерывным контролем рисков.

Спиральная модель состоит из четырех главных повторяющихся стадий. Это:

*1. планирование;*

*2. анализ рисков;*

*3. конструирование;*

*4. оценка результата (и если результат удовлетворительное качество, то происходит переход на следующий виток спирали)*

В ходе процесса разработки проект по нескольку раз проходит через все эти стадии. Каждый такой цикл прохождения через эти стадии называется ***спиралью***.



Подходит для сложных и дорогих проектов, например, таких, как разработка системы документооборота для банка, когда каждый следующий шаг требует большего анализа для оценки последствий, чем программирование.

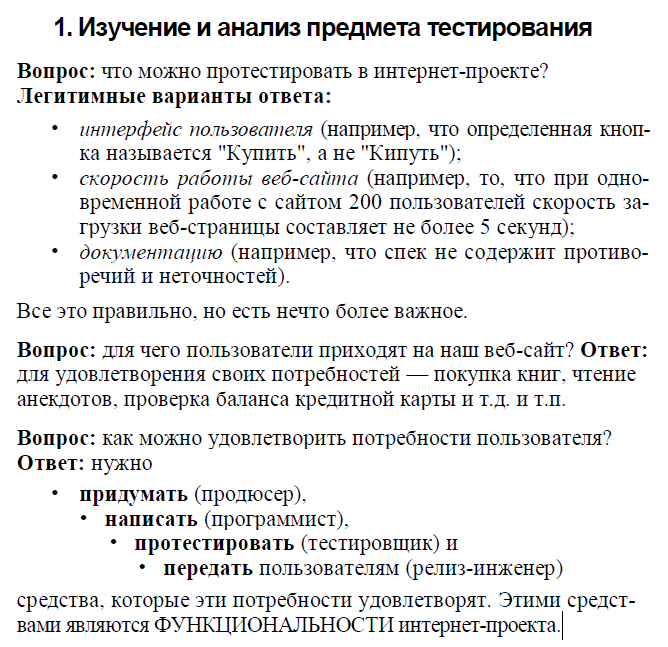
ЦИКЛ ТЕСТИРОВАНИЯ ПО

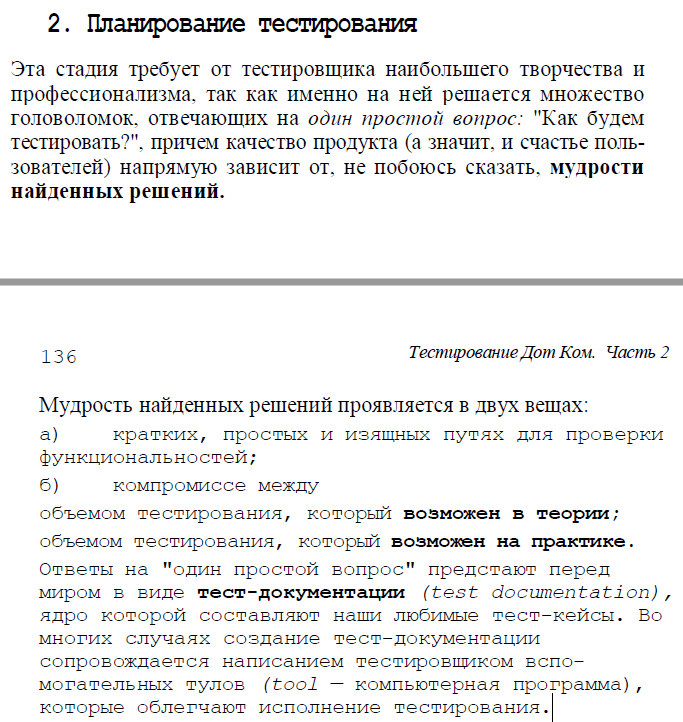
Цикл тестирования ПО состоит из трёх этапов:

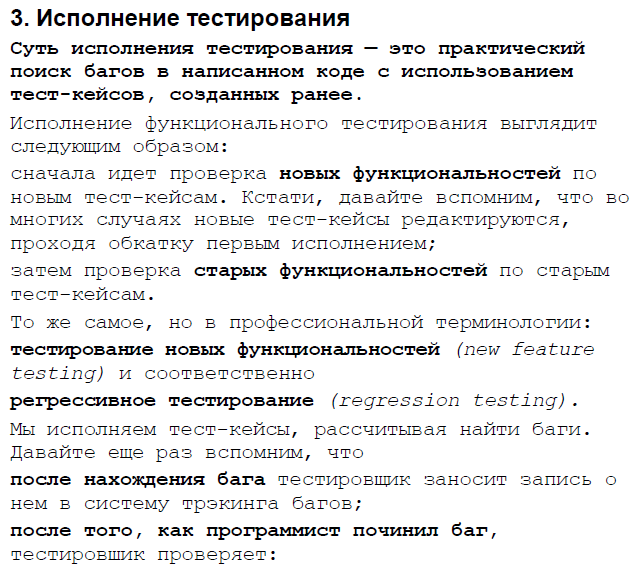
*1. Изучение и анализ предмета тестирования.*

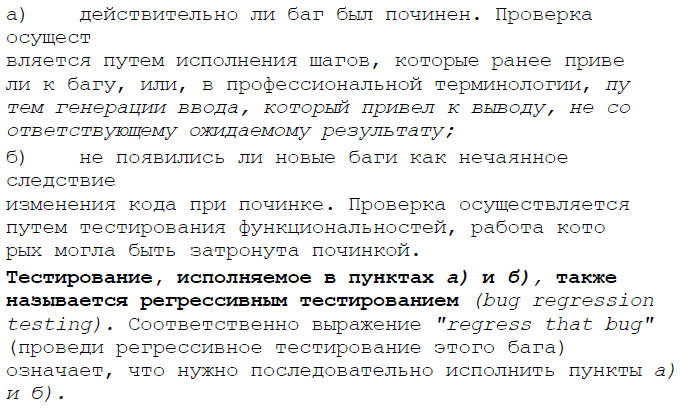
*2. Планирование тестирования.*

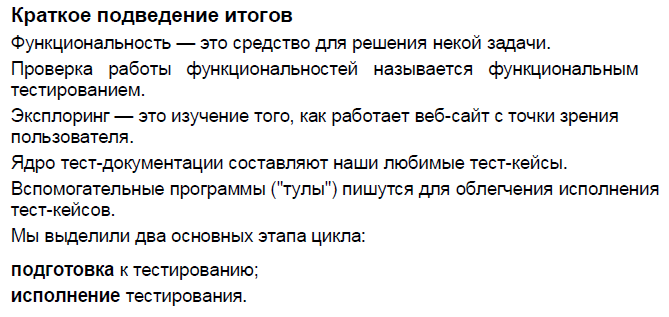
*3. Исполнение тестирования.*

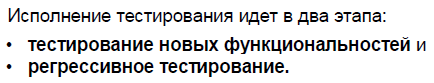
**

******

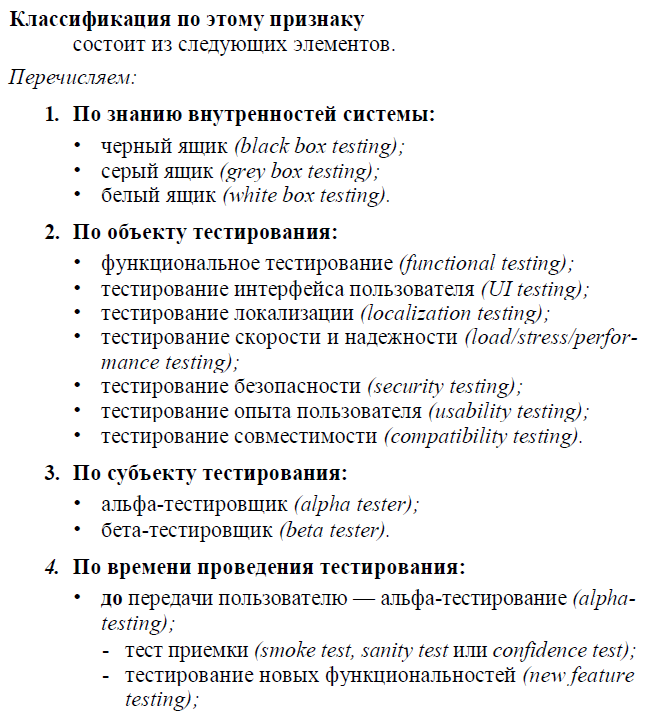


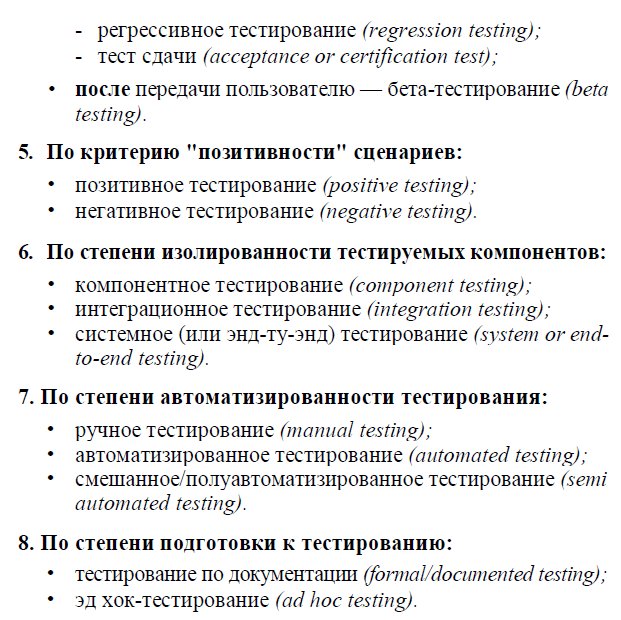






ОСНОВНЫЕ ВИДЫ ТЕСТИРОВАНИЯ





SEVERITY/PRIORITY

***Градация Серьезности дефекта (Severity) -*** это атрибут, характеризующий влияние дефекта на работоспособность приложения. Проставляется тестировщиком или техническим специалистом, который может оценить степень влияния дефекта на работу системы.

**S1 Блокирующая (Blocker)**

Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.

**S2 Критическая (Critical)**

Критическая ошибка, неправильно работающая ключевая бизнес логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, без возможности решения проблемы, используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системой.

**S3 Значительная (Major)**

Значительная ошибка, часть основной бизнес логики работает некорректно. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки.

**S4 Незначительная (Minor)**

Незначительная ошибка, не нарушающая бизнес логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.

**S5 Тривиальная (Trivial)**

Тривиальная ошибка, не касающаяся бизнес логики приложения, плохо воспроизводимая проблема, малозаметная посредствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

***Градация Приоритета дефекта (Priority) -*** это атрибут, указывающий на очередность выполнения задачи или устранения дефекта. Проставляется руководителем или менеджером проекта.

**P1 Высокий (High)**

Ошибка должна быть исправлена как можно быстрее, т.к. ее наличие является критической для проекта.

**P2 Средний (Medium)**

Ошибка должна быть исправлена, ее наличие не является критичной, но требует обязательного решения.

**P3 Низкий (Low)**

Ошибка должна быть исправлена, ее наличие не является критичной, и не требует срочного решения.

Порядок исправления ошибок по их приоритетам:

***High -> Medium -> Low***

**УРОВНИ ТЕСТИРОВАНИЯ (ПО СТЕПЕНИ ИЗОЛИРОВАННОСТИ). «ПОЗИТИВНОСТЬ» ТЕСТОВ**



***1.*** ***Модульное (компонентное) тестирование (Unit testing)*** – тестирование каждой атомарной функциональности (модуля) приложения отдельно, в искусственно созданной среде.

Например, к таким модулям можно отнести ***страницу авторизации, поиск товара, перемещение его в корзину, оплата заказа*** в интернет магазинах.

ПОЛНАЯ СТАТЬЯ: <https://qalight.ua/ru/baza-znaniy/modulnoe-testirovanie/>

***2.*** ***Интеграционное тестирование*** – вид тестирования, при котором на соответствие требований проверяется интеграция модулей, их взаимодействие между собой, а также интеграция подсистем в одну общую систему.

Исходя из различий между модульным тестированием и системным тестированием, интеграционное тестирование является **переходным этапом** между представлением программы в виде отдельных модулей в вид полностью функциональной системы.

*ПРИМЕР: есть модуль авторизации и есть модуль отправки сообщения на почту. При успешной авторизации отправляется сообщение на почту с уведомлением об успешной авторизации. Тестим эти два модуля, т.е. отправится ли сообщение при авторизации?*

Уровни интеграционного тестирования:

I. Компонентный интеграционный уровень (Component Integration testing) - проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

II. Системный интеграционный уровень (System Integration Testing) - проверяется взаимодействие между разными системами после проведения системного тестирования (см. далее).

***ВИДЫ ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ -*** [***https://testmatick.com/ru/integratsionnoe-testirovanie-osnovnye-ponyatiya-harakternye-osobennosti-i-primery/***](https://testmatick.com/ru/integratsionnoe-testirovanie-osnovnye-ponyatiya-harakternye-osobennosti-i-primery/)

***3****.* ***Системное тестирование*** – это тестирование программного обеспечения выполняемое на полной, интегрированной системе, с целью проверки соответствия системы исходным требованиям, как функциональным, так и не функциональным.

*Например, в интернет магазине мы будем проверять, можно ли зарегистрироваться, найти товар, положить товар в корзину, оплатить заказ и в целом проверить, как взаимодействуют модули между собой.* ***То есть, мы тестируем всё приложение целиком.***

ПОЛНАЯ СТАТЬЯ: <https://qalight.ua/ru/baza-znaniy/sistemnoe-testirovanie/>

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

Результатом приемочного тестирования может стать:

• Отправка проекта на доработку.

• Принятие его заказчиком, в качестве выполненной задачи.

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

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

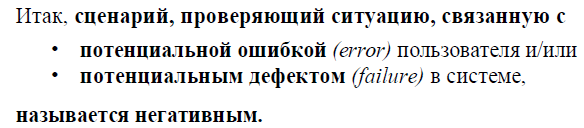
***User Acceptance Testing (UAT)*** — это приемочное тестирование, которое проводится конечными пользователями системы для принятия решения по внедрению. При любой разработке или доработке программного обеспечения есть заключительная стадия такого тестирования, причем оно проводится не профессиональными тестировщиками, а обычными пользователями. Так как обычный пользователь не может полноценно проверить совершенные доработки, важно проводить UAT-тестирование по заранее подготовленным сценариям. Они повышают качество проверки и существенно облегчают задачу конечных пользователей.

***Подробно с примерами – с.160 Савин – «Тестирование дот ком»***

Классификация тестирования «по позитивности»

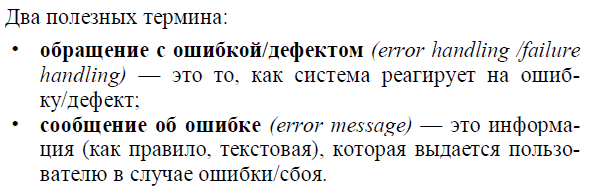
***Позитивное тестирование*** – это тестирование с применением сценариев, которые соответствуют нормальному (штатному, ожидаемому) поведению системы. С его помощью мы можем определить, что система делает то, для чего и была создана. Например, умножение на калькуляторе цифр 3 и 5.

***Негативным*** называют ***тестирование***, в рамках которого применяются сценарии, которые соответствуют внештатному поведению тестируемой системы. Это могут быть, например, исключительные ситуации или неверные данные. На примере калькулятора, это умножения числа 3 на грушу. Значение “груша” не является валидным для калькулятора.



Создание позитивных сценариев (тест-кейсов), как правило, предшествует созданию негативных тест-кейсов. Сначала мы проверяем работу системы, когда наш условный пользователь работает с системой “правильно”, а потом приступаем к проверке отклика системы на пользователя, который допускает различные ошибки (ввод неверных данных, например). И наша система должна быть готова ответить на неверный запрос. Это и есть цель негативного тестирования.





**ТИПЫ ТЕСТОВЫХ ИСПЫТАНИЙ ПО ГЛУБИНЕ ТЕСТИРОВАНИЯ**

***1.*** ***Дымное тестирование/тест приёмки*** (*smoke testing* ***a.k.a.*** *confidence testing, sanity testing, build verification test (BVT)*) – это тестирование системы с использованием минимальных наборов тестов для обнаружения самых явных багов на первых этапах работы ПО.

Дымное тестирование можно определить, как некоторый короткий цикл тестов, который осуществляется после выхода очередного ***билда*** (***промежуточной версии продукта***). Выполняется это для проверки работы основного функционала разрабатываемой программной системы. После успешного прохождения дымного тестирования, разрабатываемая система отправляется на следующие циклы более серьезных видов тестов.

*Если ПО не прошло смоук-тест, то весь процесс тестирования приостанавливается и баг отправляется разработчику.*

***2.******Тест критического пути (critical path test)*** — основной тип тестовых испытаний, во время которого значимые элементы и функции приложения проверяются на предмет правильности работы при стандартном их использовании. Чаще всего на практике, на данном уровне тестирования проверяется основная масса требований к продукту. Пример: выбор шрифта, возможность набора текста, вставки картинок и т.д.

**ТКП (CPT)** может быть как позитивным, так и негативным.

**Позитивный тест критического пути** — это проверка работоспособности функций программного продукта, с которыми пользователь сталкивается ежедневно.

**Негативный тест критического пути** — это проверка всевозможных вариантов нестандартного использования функциональности, используемой пользователем каждый день.

***3.*** ***Расширенный тест (Extended test)*** – вид углубленного тестирования, при котором проверяется нестандартное использование программного продукта, границы переполнения массивов данных, ввод специальных символов и т.п.

Например, в поле, предназначенное для ввода чисел, можно попробовать ввести буквы, знаки и прочие символы.

**ТИПЫ ТЕСТОВЫХ ИСПЫТАНИЙ ПО ЦЕЛИ ТЕСТИРОВАНИЯ**

***1.*** ***Тестирование новой функциональности (new feature testing)***

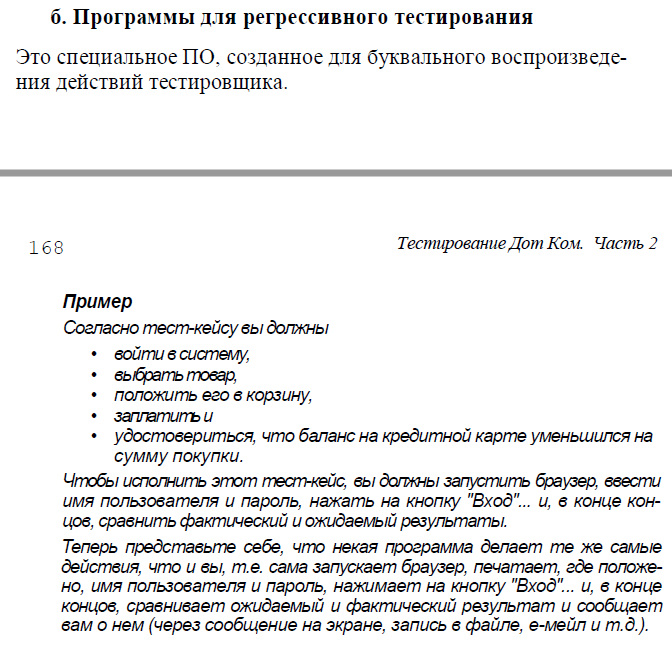
В данном виде тестирования акцент делается на тестировании новой функциональности, появившейся в конкретном выпуске (билде) программного продукта.

***2.*** ***Регрессионное тестирование (regression testing)***

Повторное выполнение тестов для проверки того, что изменения, внесённые в программу в результате разработки новой или изменения существующей функциональности, устранения ошибок, не повлияли на функциональность, которая не изменялась (т.е. текущая версия ведёт себя идентично предыдущей, за исключением измененных областей).

Иными словами, это тестирование с целью удостовериться, что найденные баги в новой фиче были пофикшены разработчиком и не появились ли новые баги в ПО после фикса.

Причём, проверять рекомендуется только те участки, которые взаимосвязаны с новой фичей, а не всё приложение целиком.



Как правило, юзают Selenium или Silk Test.

***3.*** ***Ретест (Re-test)*** - перепроверяет и подтверждает факт того, что ранее заваленные тест-кейсы проходят после того, как дефекты исправлены



**ДРУГИЕ ТИПЫ ТЕСТИРОВАНИЯ**

*По степени автоматизации* тестирование делится на **ручное**, **автоматизированное и полуавтоматизированное**

При ***ручном тестировании (manualtesting)*** тестировщики вручную выполняют тесты, не используя никаких средств автоматизации. Ручное тестирование – самый низкоуровневый и простой тип тестирования, не требующих большого количества дополнительных знаний.

***Автоматизированное тестирование*** предполагает использование специального программного обеспечения (помимо тестируемого) для контроля выполнения тестов и сравнения ожидаемого фактического результата работы программы. Этот тип тестирования помогает автоматизировать часто повторяющиеся, но необходимые для максимизации тестового покрытия задачи.

Некоторые задачи тестирования, такие как низкоуровневое регрессионное тестирование, могут быть трудозатратными и требующими много времени если выполнять их вручную. Кроме того, мануальное тестирование может недостаточно эффективно находить некоторые классы ошибок. В таких случаях автоматизация может помочь сэкономить время и усилия проектной команды.

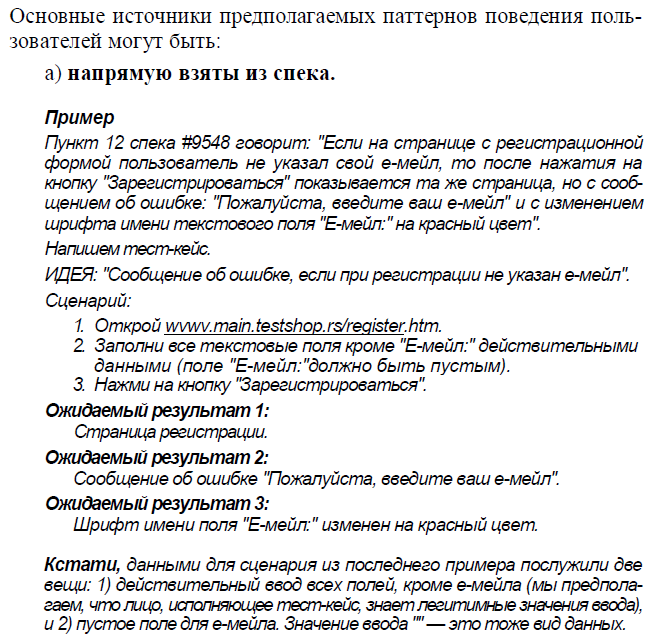
В общем, автотесты пишутся для случаев, в которых нужна частоповторяющаяся ручная работа.

*По степени знания внутреннего устройства объекта* тестирование делится на тестирование **чёрного ящика**, **белого ящика** и **серого ящика**.

***Тестирование чёрного ящика*** – это тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы, т.к. мы не знаем, как устроена тестируемая система.

Пример: тестировщик проводит тестирование веб-сайта, не зная особенностей его реализации, используя только предусмотренные разработчиком поля ввода и кнопки. Источник ожидаемого результата – спецификация. Мы тестируем GUI.

Тест-кейсы в первую очередь могут составляться по предполагаемым паттернам (шаблонам) поведения пользователя.



б) **найдены путем эксплоринга**

Иногда "брожение" по сайту является лучшим источником для понимания того, как реальный пользователь будет с ним обращаться.

в) **получены путем применения методики черноящичного тестирования (black box testing methodology)**

г) **подарены интуицией**

Интуиция не менее важна для настоящего профессионала-тестировщика, чем прикладные знания и опыт работы.

д) **присоветованы программистом или продюсером.**

Общение, общение и еще раз общение. Самое дорогое — это информация, и общение — один из главных ее источников.

е) др.

Например, мы прочитали статью в Интернете, давшую классную идею для сценария.

***Тестирование методом белого ящика (также: прозрачного, открытого, стеклянного ящика; основанное на коде или структурное тестирование)*** – метод тестирования программного обеспечения, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику, т.е. при белоящичном тестировании сценарии создаются с мыслью о том, чтобы протестировать определенную часть бэк-энда, а не определенный паттерн поведения пользователя. Мы выбираем входные значения, основываясь на знании кода, который будет их обрабатывать. Точно так же мы знаем, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование белого ящика – углубление во внутренне устройство системы, за пределы ее внешних интерфейсов.

Тестировщик, который, как правило, является программистом, изучает реализацию кода поля ввода на веб-странице, определяет все предусмотренные (как правильные, так и неправильные) и не предусмотренные пользовательские вводы, и сравнивает фактический результат выполнения программы с ожидаемым. При этом ожидаемый результат определяется именно тем, как должен работать код программы.



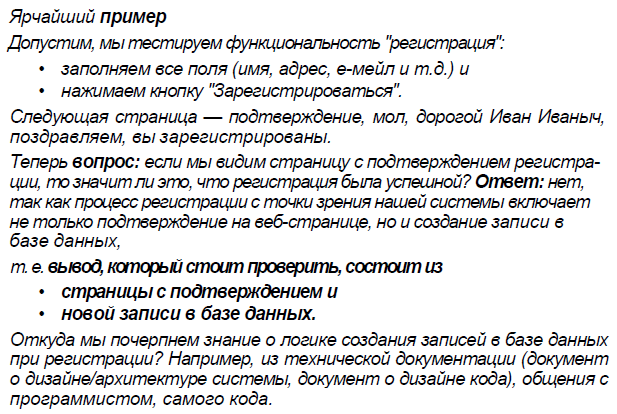
***Тестирование методом серого ящика*** – метод тестирования программного обеспечения, который предполагает, комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично. Предполагается, например, доступ к внутренней структуре и алгоритмам работы ПО для написания максимально эффективных тест-кейсов, но само тестирование проводится с помощью техники черного ящика, то есть, с позиции пользователя.

Эту технику тестирования также называют методом полупрозрачного ящика: что-то мы видим, а что-то – нет.

Тестировщик изучает код программы с тем, чтобы лучше понимать принципы ее работы и изучить возможные пути ее выполнения. Такое знание поможет написать тест-кейс, который наверняка будет проверять определенную функциональность.

Техника серого ящика применима на разных уровнях тестирования – от модульного до системного, но главным образом применяется на интеграционном уровне для проверки взаимодействия разных модулей программы.

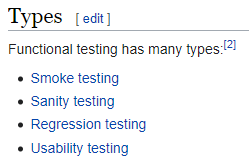
*Деятельность большинства профессиональных тестировщиков интернет-проектов протекает именно в разрезе сероящичного тестирования, т.к. оно работает «по двум фронтам» - с точки зрения пользователя и с точки зрения программиста (тестится бек-энд часть).*

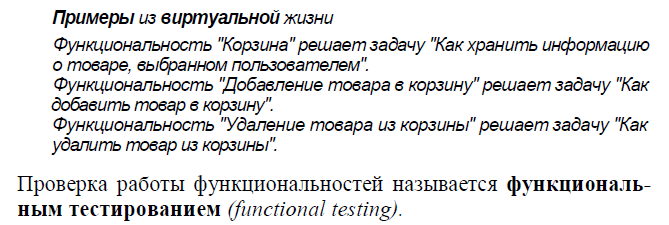
**

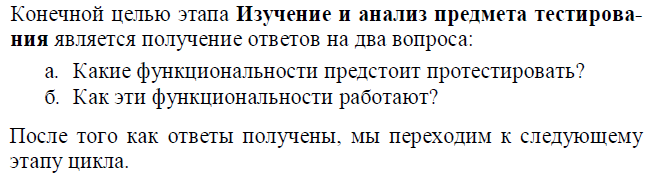
***Функциональное*** *и* ***нефункциональное*** *тестирование*

**Функциональное тестирование (Functional testing)** – направлено на проверку того, какие функции ПО реализованы, и того, насколько верно они реализованы и насколько они соответствуют требованиям.

Частью функционального тестирования являются:







Соответственно, **нефункциональное тестирование** – проверка корректности работы нефункциональных требований. Оценивается, КАК программный продукт работает. Данные свойства определяются нефункциональными требованиями, которые характеризуют продукт с таких сторон, как:

*Надежность* (реакция системы на непредвиденные ситуации).

*Производительность* (Работоспособность системы под разными нагрузками).

*Удобство* (Исследование удобности работы с приложением с точки зрения пользователя).

*Масштабируемость* (Требования к горизонтальному или вертикальному масштабированию приложения).

*Безопасность* (Защищенность пользовательских данных).

*Портируемость* (Переносимость приложения на различные платформы).

И много других качеств.

Данные свойства системы можно исследовать, используя следующие виды тестирования:

Тестирование установки (Installation testing) – проверка успешности установки приложения, его настройки и удаления. Снижает риски потери пользовательских данных, потери работоспособности приложения и пр.

Тестирование удобства использования (Usability testing) – характеризует систему с точки зрения удобства использования конечного пользователя.

Конфигурационное тестирование (или тестирование портируемости) – исследование работоспособности программной системы в условиях различных программных конфигураций.

Тестирование на отказ и восстановление (Failover and Recovery Testing) – исследование программной системы на предмет восстановления после ошибок, сбоев, каких-то потерь соединения. Оценивание реакции защитных свойств приложения.

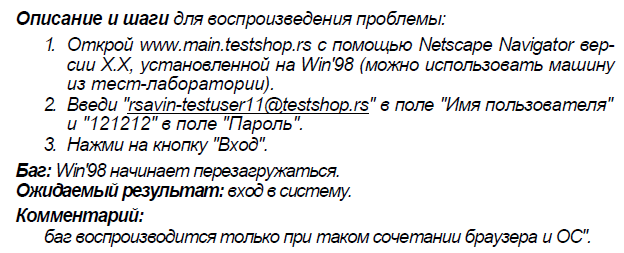
Тестирование производительности – это комплекс типов тестирования, целью которого является определение работоспособности, стабильности, потребления ресурсов и других атрибутов качества приложения в условиях различных сценариев использования и нагрузок.

Тестирование совместимости (compability testing) - Это проверка того, как наш веб-сайт взаимодействует с

• "железом" (например, модемами) и

• ПО (браузерами/операционными системами) наших пользователей.

Пример:



Основные моменты для тестирования кроссбраузерности: вёрстка (цвет, шрифты, расположение графических картинок и динамических элементов) и JavaScript.

Следует отметить, что кроссбраузерное тестирование необходимо выполнять когда система стабильна и весь функционал отлажен, иначе будут возникать ошибки, которые не являются кроссбраузерными. Это чревато лишними финансовыми затратами.

Тестирование скорости и надежности (load/stress/performance testing) – это отдельная техническая дисциплина, за хорошее знание которой получают очень большие деньги в иностранной валюте.

Как правило, целью такого тестирования является обнаружение слабого места (bottleneck) в системе. Под системой подразумеваются все компоненты, включая код, базу данных, "железо" и т.д.

Тестирование безопасности (security testing) — это множество вещей, суть которых заключается в том, чтобы усложнить условия для кражи — кражи данных, денег и информации.

Тестирование локализации

***Локализация (Localization testing, L10n)*** – процесс адаптации программного продукта к языку и культуре клиента. Данный процесс адаптации включает в себя:

• Перевод пользовательского интерфейса.

• Перевод документации.

• Контроль формата даты и времени.

• Внимание к денежным единицам.

• Внимание к правовым особенностям.

• Раскладка клавиатуры пользователя.

• Контроль символики и цветов.

• Толкование текста, символов, знаков.

• И прочие подобные аспекты.

Тестирование интернационализации

***Интернационализация (I18n)*** – более обобщенное понятие, подразумевающее проектирование и реализацию программного продукта или документации таким образом, который максимально упростит локализацию приложения.

Интернационализация включает в себя:

• Создание продукта с учетом возможности кодировки Unicode (стандарт кодирования, поддерживающий практически все языки мира).

• Создание в приложении возможности поддержки элементов, которые невозможно локализовать обычным образом (вертикальный текст азиатских стран, чтение с права на лево арабских стран и т.д.).

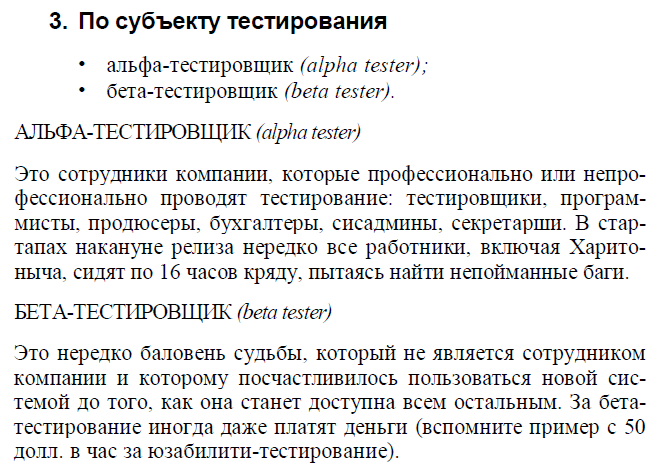
• Возможность загрузки локализированных элементов в будущем при желании пользователя.

Термин интернационализации не обязывает переводить текст программ или документацию на другой язык, он подразумевает разработку приложений таким образом, который сделает локализацию максимально простой и удобной, а также позволит избежать проблем при интеграции продукта для стран с отличающейся культурой.

***Важно разбавлять функциональное тестирование нефункциональным, иначе будет плёхо!***

ТЕСТИРОВАНИЕ ПО СУБЪЕКТУ ТЕСТИРОВАНИЯ

Альфа-тестирование – тестирование кода **ДО** передачи его рядовым пользователям для использования.





**Тестирование *по исполнению сценария***

**Интуитивное тестирование (ad hoc testing)** – вид тестирования, который выполняется без подготовки к тестам, без определения ожидаемых результатов, проектирования тестовых сценариев. Это неформальное, импровизационное тестирование. Он не требует никакой документации, планирования, процессов которых следует придерживаться в выполнении. Также на данный вид тестирования не пишутся тест-кейсы, что в свою очередь может вызвать определенные затруднения в попытках воспроизвести дефект в системе. *Такой вид зачастую может дать сходу больше результата чем тестирование по заранее определенным сценариям. Это обусловлено тем, что тестировщик на первых шагах приступает к тестированию основного функционала и выполняет нестандартные проверки, точнее некоторые из его проверок будут нестандартными.*

По ISTQB и по Джеймсу Баху **исследовательское тестирование (exploratory testing)** – это неформальный метод проектирования тестов, при котором тестировщик активно контролирует проектирование тестов в то время, как эти тесты выполняются, и использует полученную во время тестирования информацию для проектирования новых и улучшенных тестов.

Если каждый следующий тест, который мы выполняем, выбирается по результатам предыдущего теста, это означает, что мы используем исследовательское тестирование.

***По ISTQB и по Джеймсу Баху исследовательское тестирование*** *– это неформальный метод проектирования тестов, при котором тестировщик активно контролирует проектирование тестов в то время, как эти тесты выполняются, и использует полученную во время тестирования информацию для для проектирования новых и улучшенных тестов.*

**Тестирование *по критерию запуска программы***

**Статическое тестирование** – тип тестирования, который предполагает, что программный код во время тестирования не будет выполняться. При этом само тестирование может быть как ручным, так и автоматизированным.

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации. Для этого типа тестирования в некоторых случаях даже не нужен компьютер – например, при проверке требований.

Виды статического тестирования:

– вычитка исходного кода программы;

– проверка требований.

**Динамическое тестирование** – тип тестирования, который предполагает запуск программного кода. Таким образом, анализируется поведение программы во время ее работы.

Для выполнения динамического тестирования необходимо чтобы тестируемый программный код был написан, скомпилирован и запущен. При этом, может выполняться проверка внешних параметров работы программы: загрузка процессора, использование памяти, время отклика и т.д. – то есть, ее производительность.

Динамическое тестирование является частью процесса валидации программного обеспечения.

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

• Доступа к коду (тестирование черным, белым и серым ящиками).

• Уровня тестирования (модульное, интеграционное, системное и приемочное тестирование).

• Сферы использования приложения (функциональное, нагрузочное, тестирование безопасности и пр.).

***Тестирование GUI***

При **тестировании GUI** проверяется в целом общий вид приложения и в отдельности формы, расположенные на странице.

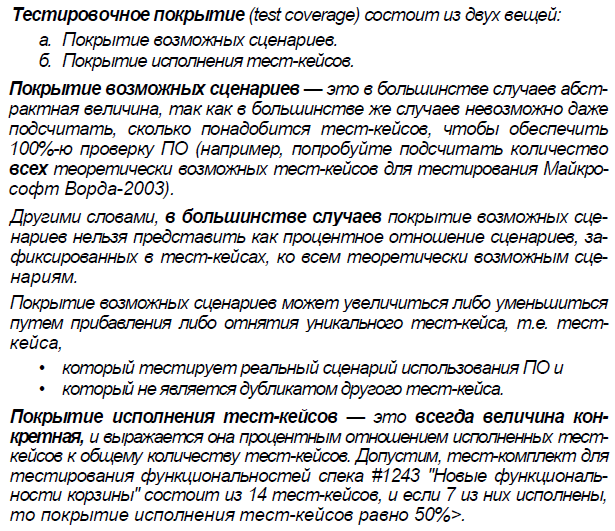
Все важные подробности в статье: <https://qaevolution.ru/testirovanie-po/vidy-testirovaniya-po/testirovanie-polzovatelskogo-interfejsa/>

UI / UX – разница

UI — это про то, как продукт выглядит, каким его воспринимает пользователь.

UX — это про взаимодействие пользователя с продуктом. Это про то, как сделать так, чтобы каждый открыл приложение и сходу разобрался, как оно работает.

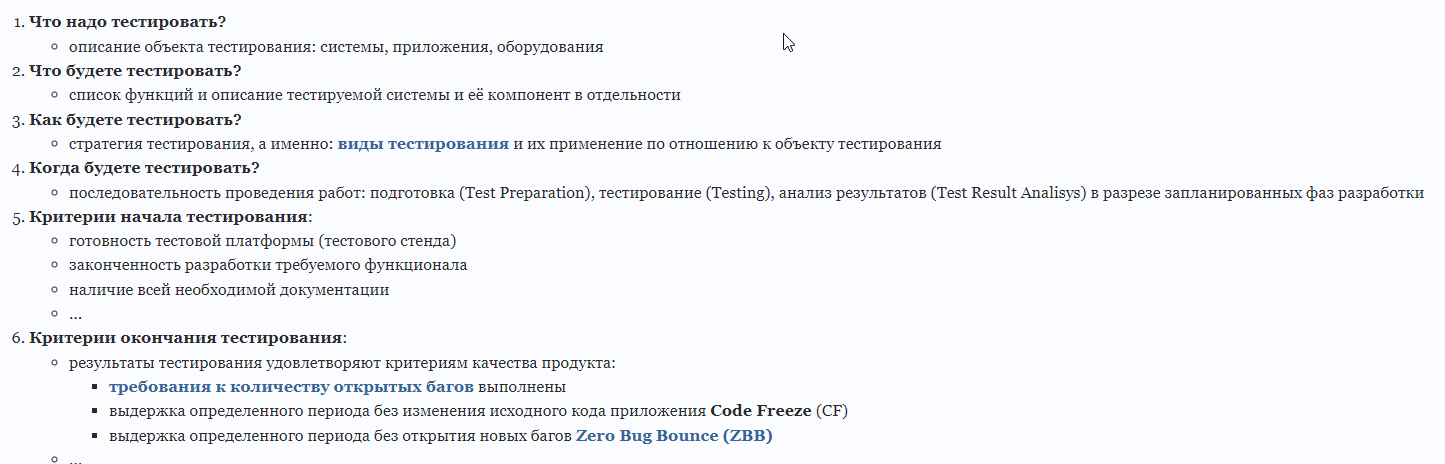
ТЕСТИРОВОЧНОЕ ПОКРЫТИЕ



ТЕСТОВЫЕ АРТЕФАКТЫ

**Тест план (Test Plan)** - это документ, описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Хороший тест-план как правило содержит следующую информацию:



# Виды тест планов

Чаще всего на практике приходится сталкиваться со следующими видами тест планов:

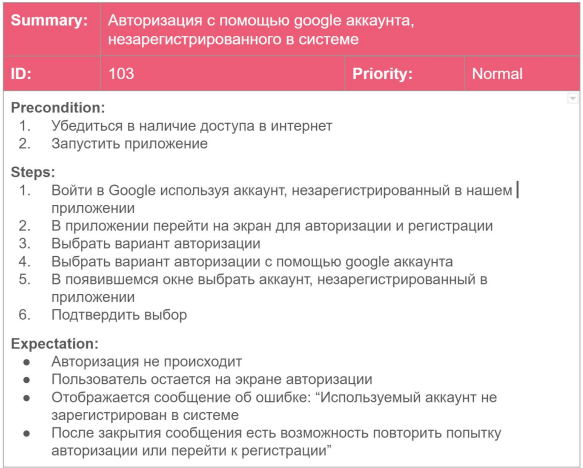
1. **Мастер Тест План (Master Plan or Master Test Plan)**
2. **Тест План (Test Plan)**, назовем его **детальный тест план)**
3. **План Приемочных Испытаний (Product Acceptance Plan)** - документ, описывающий набор действий, связанных с [**приемочным тестированием**](http://www.protesting.ru/testing/levels/acceptance.html) (стратегия, дата проведения, ответственные работники и т.д.) ([**Шаблон плана приемо-сдаточных испытаний от RUP**](http://www.protesting.ru/documentation/product_acceptance_plan_rup.zip))

Явное отличие Мастер Тест Плана от просто Тест Плана в том, что **мастер тест план является более статичным** в силу того, что содержит в себе высокоуровневую (**High Level**) информацию, которая не подвержена частому изменению в процессе тестирования и пересмотра требований. Сам же **детальный тест план**, который содержит более конкретную информацию по стратегии, видам тестировании, расписанию выполнения работ, **является "живым" документом**, который постоянно претерпевает изменения, отражающие реальное положение дел на проекте.

**Тест-кейс** — это профессиональная документация тестировщика, последовательность действий направленная на проверку какого-либо функционала, описывающая как прийти к фактическому результату.

Любой тест-кейс обязательно включает в себя:

* **Уникальный идентификатор тест-кейса** — необходим для удобной организации хранения и навигации по нашим тест-наборам.
* **Название**— основная тема, или идея тест-кейса. Кратное описание его сути.
* **Предусловия** — описание условий, которые не имеют прямого отношения к проверяемому функционалу, но должны быть выполнены.  
  Например, оставить комментарий на вашем портале может только зарегистрированный пользователь. Значит для тест-кейса «Создание комментария» будет необходимо выполнение предусловия «пользователь зарегистрирован», и «пользователь авторизован»
* **Шаги** — описание последовательности действий, которая должна привести нас к ожидаемому результату
* **Ожидаемый результат** — результат: что мы ожидаем увидеть после выполнения шагов.



**Юзкейс (Use Case)** — это перечень действий, сценарий по которому пользователь взаимодействует с приложением, программой для выполнения какого-либо действия для достижения конкретной цели. Тестирование по юзкейсам проводится для того чтобы обнаружить дополнительные логические дыры и баги в приложении, которые сложно найти в тестировании индивидуальных модулей, частей приложения отдельно друг от друга.

**Чек-лист** – набор проверок, которые необходимо произвести, описанные кратко (без шагов).

**Тест-комплект** – документ с набором определённых тест-кейсов.

ТРЕБОВАНИЯ

Виды требований

- Бизнес-требования — содержат высокоуровневые цели организации или заказчиков системы. В бизнес-требованиях объясняется, почему организации нужна такая система, то есть описаны цели, которые организация намерена достичь с ее помощью.

- Пользовательские требования — определяют набор пользовательских задач, которые должна решать программа, а также способы (сценарии) их решения в системе. Пользовательские требования могут выражаться в виде фраз утверждений, в виде сценариев использования (англ. use case), пользовательских историй (англ. user stories), сценариев взаимодействия (scenario).

- Функциональные требования описывают функции, которые должно выполнять разрабатываемое ПО. Функциональные требования вытекают из пользовательских требований.

Источники требований, способы выявления требований



Характеристики требований

- Корректность;

- Недвусмысленность;

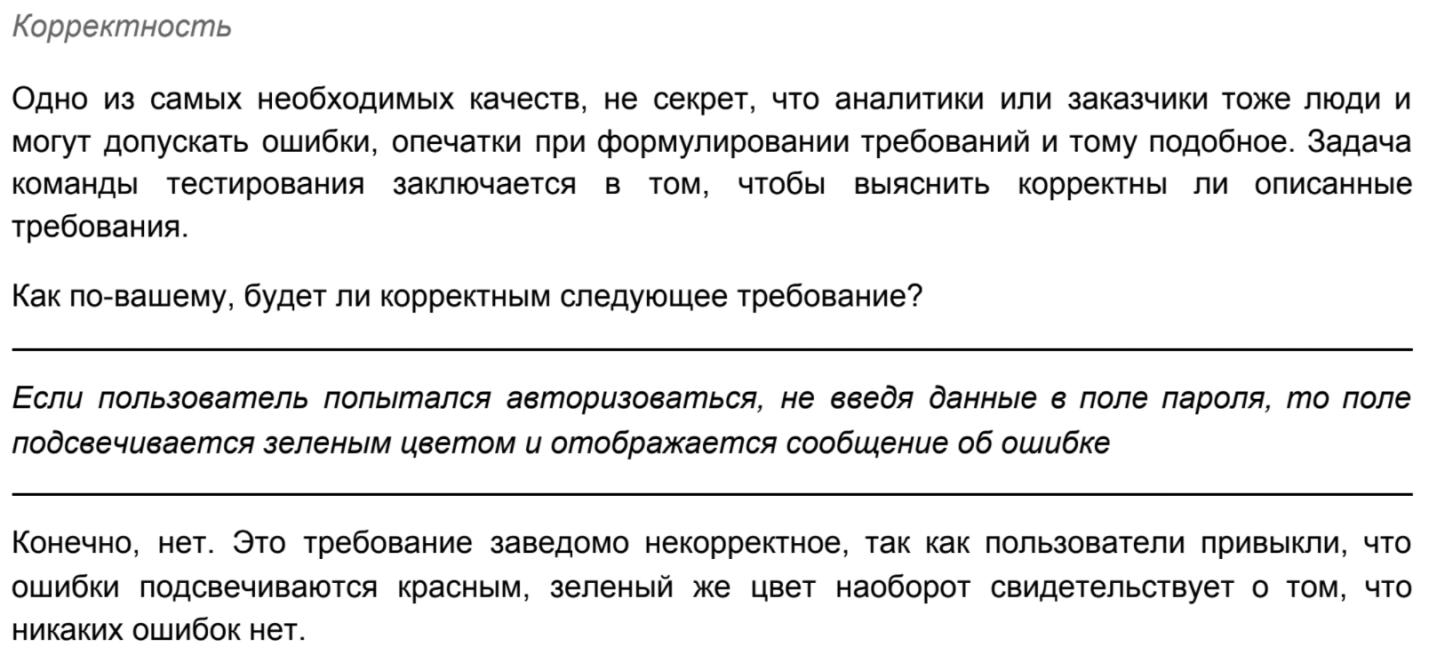
- Полнота набора требований;

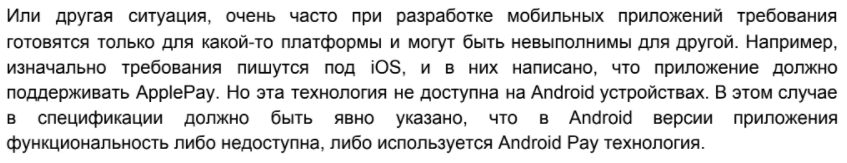
- Непротиворечивость;

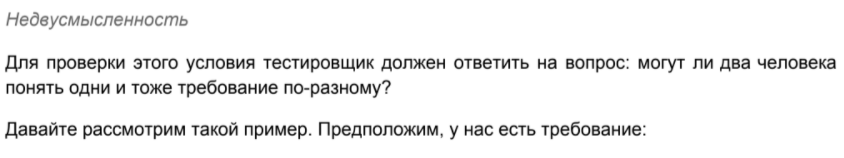
- Проверяемость;

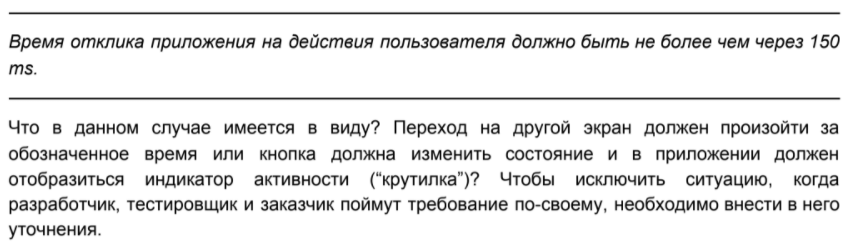
- Трассируемость;

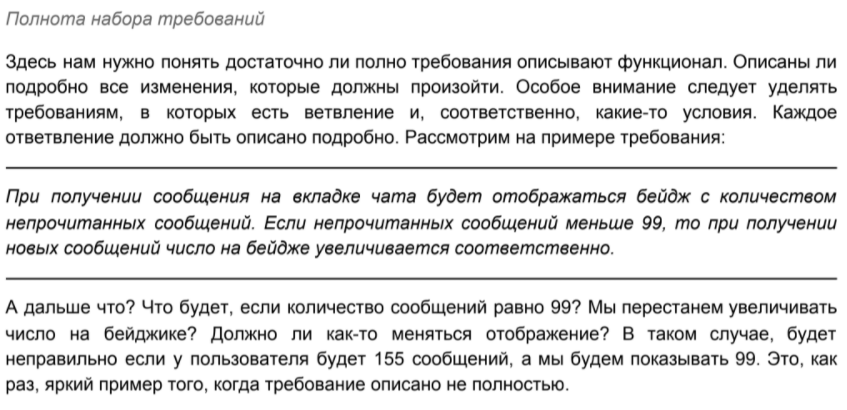
- Понятность.

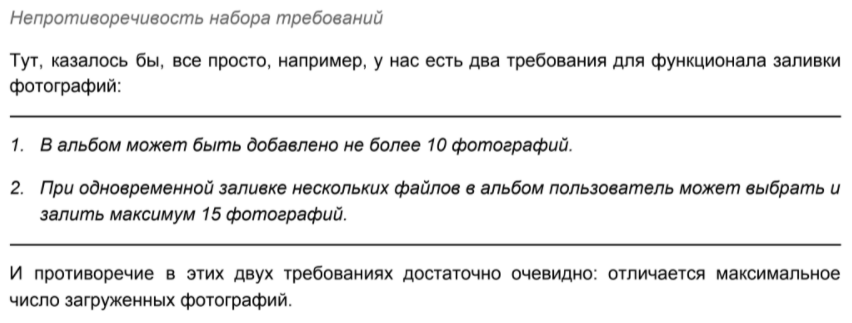


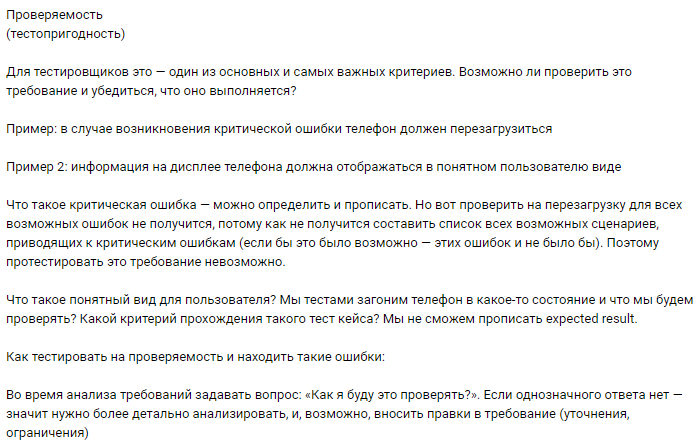




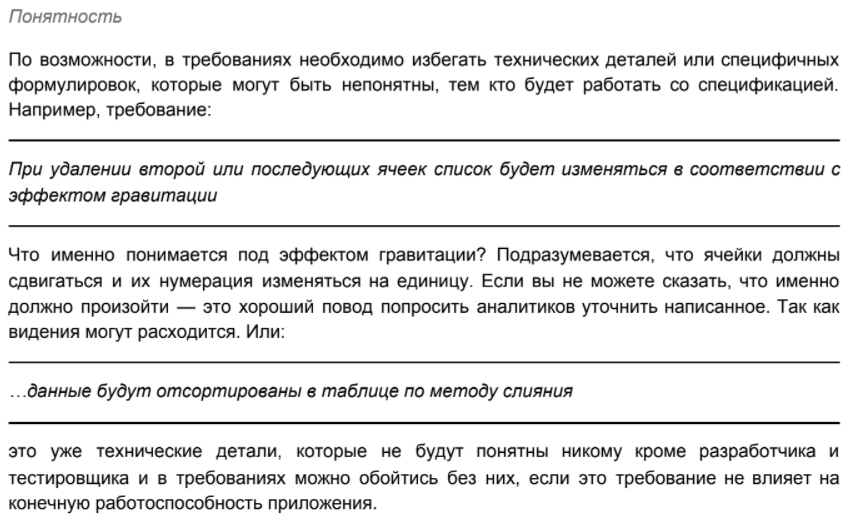












ТЕХНИКИ ТЕСТ-ДИЗАЙНА

**Тест-дизайн** – это процесс проектирования и создания тест-кейсов.

**Выделение классов эквивалентности**

***Класс эквивалентности*** – это специальный набор входных параметров ПО, которые проходят процесс обработки программой по 1 алгоритму или приводят к одному и тому же результату.

Тестовые данные разбиваются на определенные классы допустимых значений. В рамках каждого класса выполнение теста с любым значением тестовых данных приводит к эквивалентному результату

**ПРИМЕР:**

Если товар стоит от 2 до 10 долларов, то предоставляется скидка в размере 10%;

Если товар стоит от 11 до 20 долларов, то скидка 20%

Тут два класса с допустимыми значениями и ещё как минимум несколько с недопустимыми (от 0 до 1, от 20 до +бесконечности или ввести вообще другие символы)

**Анализ граничных значений**

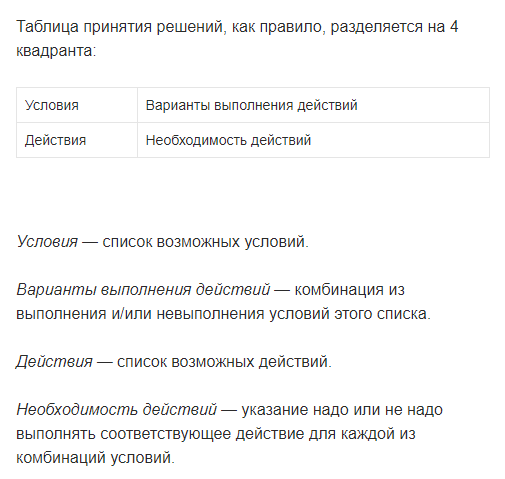
Для начала выбираются диапазоны значений – как правило, это классы эквивалентности. Затем определяются границы диапазонов. На каждую из границ создается 3 тест-кейса: первый проверяет значение границы, второй – значение ниже границы, третий – значение выше границы.

В первом классе берём 1 2 3, во втором – 10 11 12.

**Таблица принятия решений**

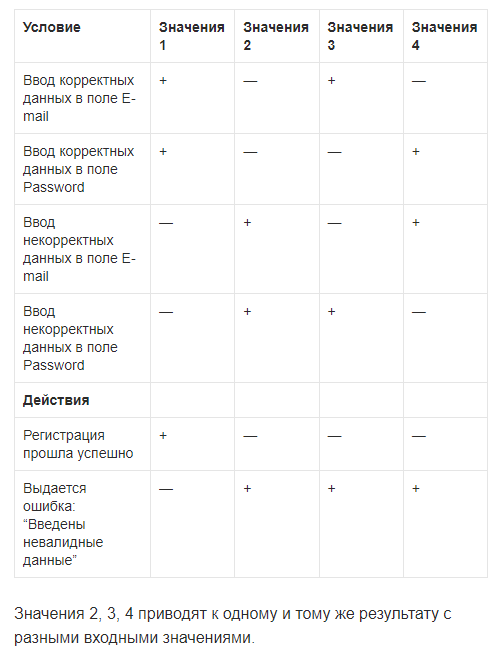
Это взаимосвязь между множеством условий и действий.

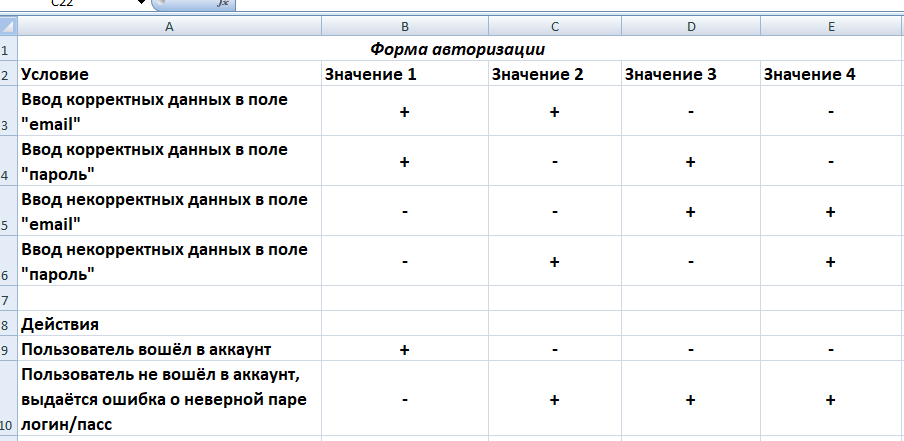
В таких таблицах представлен набор условий, одновременное выполнение которых должно привести к определённому действию



***Как составлять?*** Я так понял, кол-во колонок «Значение №» равно кол-ву строк с условиями. Значения «+» и «-» проставляются в рандомных комбинациях сначала сверху вниз в первой колонке «Значение 1», потом сверху вниз в колонке «Значение 2» и т.д. Кол-во плюсов должно быть равно кол-ву минусов в каждой из колонок

Примеры:





**Анализ переходов из состояния в состояние (State-transition testing)**

Система переходит в то или иное состояние в зависимости от того, какие операции над нею выполняются.

Эта техника подразумевает создание некоторых блок-схем или таблиц с различными вариантами поведения ПО в зависимости от тех или иных принятых пользователем решений.

**Метод попарного тестирования**

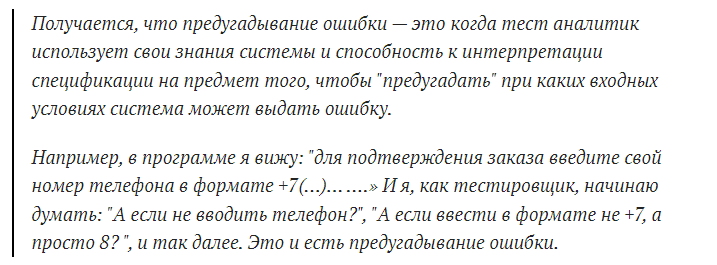
<https://vk.com/@qaqc_ru-pairwise-testing-chto-takoe-poparnoe-testirovanie>

Суть метода в парах, и в уникальной комбинации пар. История (и опыт) гласят нам, что большинство багов встречаются в тестах, которые проверяют 1 или 2 параметра. Вернее, чтобы значение одного параметра хотя бы 1 раз пересекалось с каждым другим значением другого параметра. Если баги выявляются в >2 параметров, то они скорей всего менее критичны.

Обычно для составления уникальных пар используют *инструмент* ***PICT***, который позволяет сделать это автоматически.

**Предугадывание ошибок**



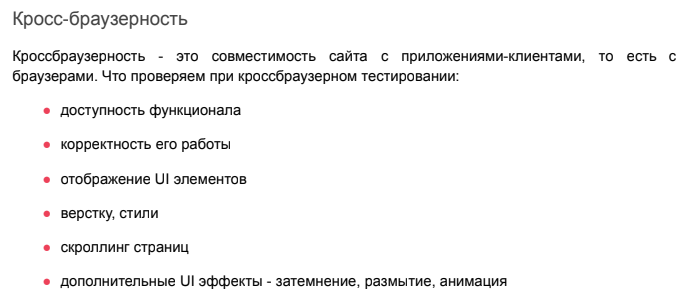
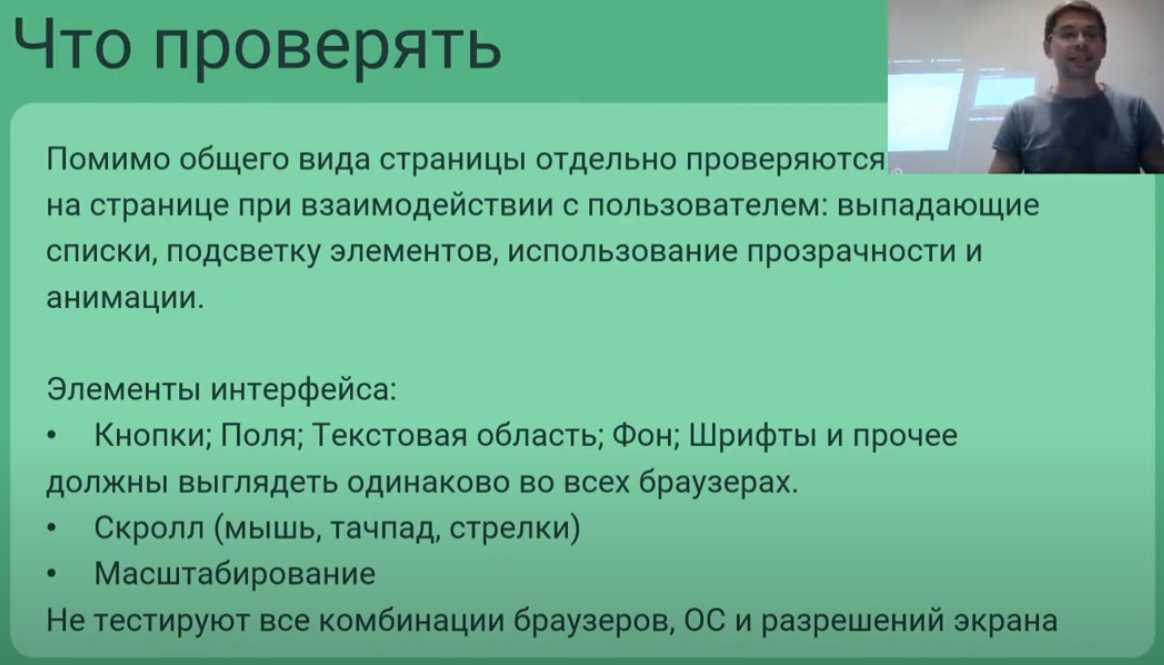


Доп. инфа: <https://vk.com/@zapiskisedogotestera-predugadyvanie-oshibok>

ФИШКИ ПРИ ТЕСТИРОВАНИИ ВЕБА

Важно очищать кэш перед выходом каждого билда, т.к. баг может так же воспроизвестись, хотя он пофикшен. Это из-за того, что старые данные с прошлого билда приложения остались в кеше.

ЧЕКЛИСТ КРОССБРАУЗЕРНОСТИ



КЛИЕНТ-СЕРВЕРНАЯ АРХИТЕКТУРА

ПРОТОКОЛЫ ПЕРЕДАЧИ ДАННЫХ СЕТИ ИНТЕРНЕТ

IP-протокол (Internet Protocol)

Этот протокол является наиболее простым. А еще он ненадежен, так как не подтверждает доставку пакетов получателю и не контролирует целостность данных. По IP-протоколу передача данных происходит без установки соединения.

TCP/IP-протокол (Transmission Control Protocol/Internet Protocol)

Это уже стек протоколов TCP и IP. TCP обеспечивает и контролирует передачу данных и следит за надежностью и целостностью, а IP отвечает за маршрутизацию.

UDP (User Datagram Protocol)

Обеспечивает передачу данных, без создания предварительного соединения. UDP считают недостаточно надежным, т. к. пакеты могут не только не дойти, продублироваться или прийти не по порядку.

Из плюсов – быстрая скорость доставки данных. Его нередко используют в приложениях, особо чувствительных к сетевым задержкам

FTP-протокол (File Transfer Protocol)

Служит для передачи файлов. Надежен, гарантирует передачу данных, функционирует по принципу клиент-серверной архитектуры.

SMTP-протокол (Simple Mail Transfer Protocol)

SMPT - сетевой протокол, предназначенный для передачи электронной почты в сетях TCP/IP.

DNS (Domain Name System/Система доменных имён)

Протокол дает возможность клиентским компьютерам запрашивать у DNS-сервера IP-адрес какого-нибудь сайта, плюс он помогает осуществлять обмен БД между серверами DNS. В работе системы также используются протоколы TCP и UDP.

Протокол NTP (Network Time Protocol)

Протокол NTP служит для синхронизации локальных часов устройства со временем в глобальной/локальной сети.

SSH-протокол (Secure SHell)

Протокол для удалённого управления операционной системой с использованием TCP. В SSH шифруется весь трафик.

SSH часто применяется при работе с хостингами, когда клиент может удалённо подключиться к серверу и работать уже оттуда.

HTTP-ПРОТОКОЛ

*HTTP-протокол* – это протокол, который позволяет обмениваться данными между клиентом (обычно в роли клиента выступает браузер) и сервером. Он также определяет то, КАК ИМЕННО будет происходить обмен данными.

У этого протокола есть широко используемое расширение – *HTTPS (HyperText Transfer Protocol Secure)*. Данные, передаваемые через этот протокол, зашифровываются в криптографический вид (в протоколы SSL или TLS).

**СТРУКТУРА HTTP ЗАПРОСА**

HTTP запрос состоит из трех основных частей, которые идут в нем именно в том порядке, который указан ниже. Между заголовками и телом сообщения находится пустая строка (в качестве разделителя), она представляет собой символ перевода строки.



1. строка запроса (Request Line)

2. заголовки (Message Headers)

Пустая строка (разделитель)

3. тело сообщения (Entity Body) – необязательный параметр. Также зовётся как Payload или полезная нагрузка.

*Строка запроса* – указывает метод передачи, URL-адрес, к которому нужно обратиться и версию протокола HTTP.

***URI (Uniform Resource Identifier)*** – это идентификатор ресурса на который отправляется запрос.

*Заголовки* – описывают тело сообщений, передают различные параметры и др. сведения и информацию.

*тело сообщения (Payload)* - это сами данные, которые передаются в запросе. Тело сообщения – это необязательный параметр и может отсутствовать.

Когда мы получаем ответный запрос от сервера, тело сообщения, чаще всего представляет собой содержимое веб-страницы. Но, при запросах к серверу, оно тоже может иногда присутствовать, например, когда мы передаем данные, которые заполнили в форме обратной связи на сервер.

**МЕТОДЫ**

HTTP-методы GET и POST — самые распространённые способы отправить или получить данные с сервера. Также есть ещё два известных метода – PUT и DELETE. Кроме них есть и другие.

**GET** — метод, который передаёт параметры через ссылку, т.е. URL. Его можно использовать, например, для доступа к указанной странице. Он говорит серверу, что клиент хочет прочитать указанный документ. На практике этот метод используется чаще всего, например, в интернет-магазинах на странице каталога. Фильтры, которые выбирает пользователь, передаются через метод **GET**.

**GET-запрос ограничен по символам.**

**GET-запросы кешируются на стороне клиента.**

**GET-запросы можно добавлять в закладки.**

**POST** — метод, который передаёт данные уже через тело запроса, и таким образом данные скрываются от посторонних глаз, что делает его более безопасным. Чаще всего с помощью метода **POST** передаются данные из каких-либо инпутов.



<https://guruweba.com/html/metody-get-i-post-ispolzovanie-i-otlichiya/>

**PUT** очень похож на POST, но он используется для замены или обновления каких-то существующих объектов.

**DELETE** удаляет указанный ресурс.

Также есть:

**HEAD** - запрашивает ресурс так же, как и метод GET, но без тела ответа.

**CONNECT** устанавливает "туннель" к серверу, определённому по ресурсу.

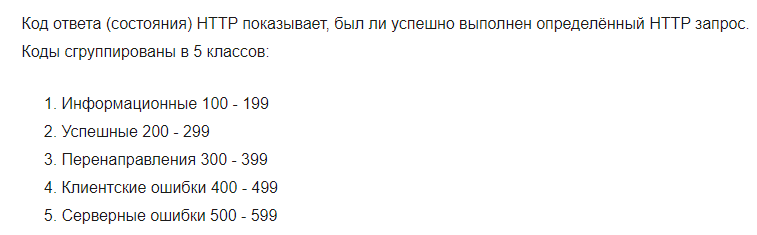
**OPTIONS** используется для описания параметров соединения с ресурсом.

**TRACE** выполняет вызов возвращаемого тестового сообщения с ресурса.

**PATCH** используется для частичного изменения ресурса.

**Всего 9.**

**Коды ответа HTTP**

****

**Важные заголовки запросов/ответов**

**ЗАГОЛОВКИ ОТВЕТОВ *Content-type* определяет MIME-тип (медиа-тип) ресурса, т.е. сообщает клиенту о том, какой будет тип передаваемого файла. Клиент должен знать, с чем он вообще имеет дело.  
  
*Content-encoding* используется для сжатия тела ответа и оптимизации таким образом процесса передачи данных, а значение этого параметра показывает какие кодировки были применены к телу ответа и в каком порядке. Это позволяет клиенту узнать, как правильно декодировать тело ответа и получить доступ к оригинальному телу. *Last-Modified* сообщает клиенту время последнего изменения страницы (объекта). Если клиент (браузер, поисковый робот) получил заголовок Last-Modified, то при следующем обращении к адресу, при условии, что страница (объект) есть в локальном кеше, он добавит вопрос If-Modified-Since (не изменилась ли страница после даты, полученной в Last-Modified).  
Если страница не изменилась, то сервер прекратит передачу данных после отправки заголовков с кодом 304 Not Modified, а тело страницы, изображения и другие объекты передаваться не будут, а загрузятся из кэша, и таким образом будет происходить экономия трафика. Также при правильной реализации проверки времени последнего изменения нагрузка на сервер может значительно снизиться, что очень важно для сайтов с высокой посещаемостью. *If-Modified-Since* делает запрос условным: сервер отправит обратно запрошенный ресурс с статусом 200, только если он был изменён после указанной даты. Если запрос не был изменён после указанной даты, ответ будет 304 без какого-либо тела, т.к. запрашиваемый ресурс был прокэширован ранее.  
  
*Access-Control-Allow-Origin* показывает, может ли ответ сервера быть доступен коду, отправляющему запрос с определённого источника.  
  
ЗАГОЛОВКИ ЗАПРОСОВ  
 *Accept* указывает, какие типы ресурсов, выраженные как MIME-типы (медиа-типы), клиент может понять. Затем сервер выбирает наиболее подходящий для клиента тип и информирует его при помощи значения заголовка ответа Content-type. Таким образом происходит "согласование контента", и для клиента выбирается наиболее желаемый тип запрашиваемого ресурса, а не все сразу.  
  
*Accept-Encoding* показывает серверу какой алгоритм сжатия и шифрования может понять клиент, чтобы тот мог расшифровать данные.  
  
*Authorization* указывает данные пользователя для авторизации.  
  
*Cookie* используется для возврата серверу ранее переданных клиенту cookies, чтобы сервер «распознал» клиент и выдал ему информацию, заключённую в них, а не передавал данные снова.  
  
*User-Agent* указывает серверу информацию о браузере пользователя, в зависимости от которой сервер предоставляет то или иное содержимое страницы, которое будет корректно отображаться и в целом работать.**

ВИДЫ ИНТЕРФЕЙСОВ

1. Пользовательский интерфейс (UI/GUI) – видимая часть, которая видна пользователю для удобства взаимодействия с программой

2. Программный интерфейс приложения (API – Application Programming Interface) – см. далее

3. Интерфейс командной строки (CLI – Command Line Interface) – разновидность текстового интерфейса(TUI) между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд), в UNIX-системах возможно применение мыши. Также известен под названием консоль.

ТОЛСТЫЙ И ТОНКИЙ КЛИЕНТЫ

**Толстый клиент** — это приложение, обеспечивающее полную функциональность и независимость от центрального сервера, т.е. *вся работа по обработке и представлению данных переносится на машину клиента, а сервер часто в этом случае является лишь хранилищем данных.*

**Тонкий клиент** — компьютер или программа-клиент, в которой *большая часть задач по обработке информации перенесена на сервер и права доступа клиента строго ограничены*. Примером тонкого клиента может служить компьютер с браузером, использующийся для работы с веб-приложениями.

API, SWAGGER, cURL

**API («Application Programming Interface» или «Программный интерфейс приложения») –** это такой набор способов (а вернее – функций), с помощью которых одна программа может обращаться к другой программе минуя пользовательский интерфейс.

Например, если нам нужно добавить на сайт актуальную информацию о количестве подписчиков в группе ВК, то это можно сделать, обратясь по специальному запросу к API Вконтакте.

Годная статья по основам REST API: <https://wp-kama.ru/handbook/rest/basic>

Для API часто разработчиками пишется документация в формате сваггера (Swagger)

**Swagger** — это набор инструментов, который позволяет автоматически описывать API на основе его кода.

**cURL** — инструмент для передачи данных с сервера или на него, при этом используется один из поддерживаемых протоколов: DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET и TFTP. Команда предназначена для работы без взаимодействия с пользователем.

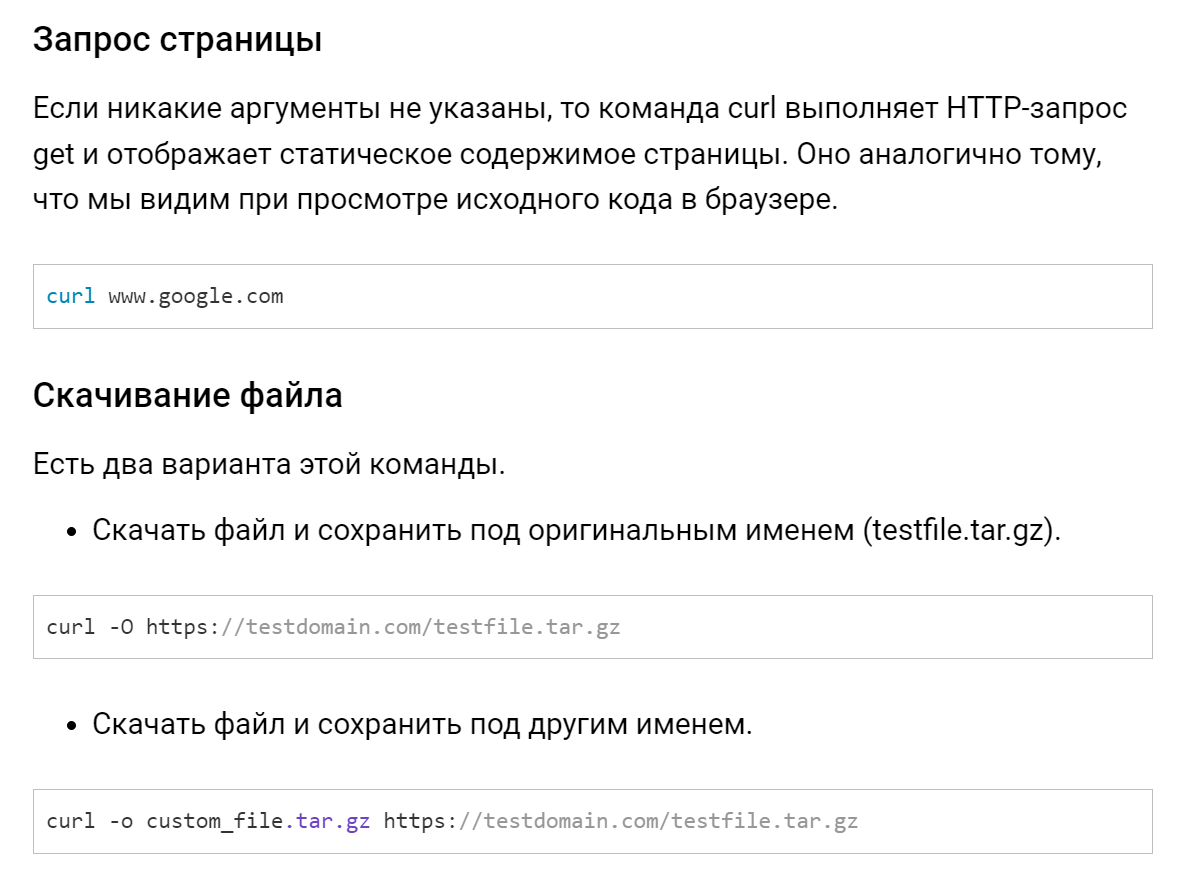
Команда curl запускается из командной строки и предустановлена в большинстве дистрибутивов Linux.

Варианты применения:

- доступ без браузера;

- внутри shell-скриптов;

- для тестирования API.



ЧТО ТАКОЕ КУКИЗ (COOKIES)

Кукиз (cookies) или «печеньки» - это текстовые файлы небольшого объема со служебной информацией для сайтов, которые посещал пользователь и которые сохраняются на жёстком диске пользователя. Часто в таких файлах хранится статистика посещений, логины и пароли от сайтов или сервисов, индивидуальные настройки пользователя (регион, дизайн оформления и прочее).

Они нужны для того, чтобы обеспечить комфортную работу пользователя в интернете. Например, с помощью них можно автоматически авторизовываться (как один из ярких примеров)

Посмотреть куки можно в девтулзе во вкладке Applicatiion > Storage > Cookies

Столбцы таблицы cookie: <http://joxi.ru/bmoQXBktO91k7A>

МОДЕЛЬ OSI

**Модель Open Systems Interconnection (OSI)** – это скелет, фундамент и база всех сетевых сущностей. Модель определяет сетевые протоколы, распределяя их на **7** логических уровней.

***1. ФИЗИЧЕСКИЙ (PHYSICAL) УРОВЕНЬ***

На первом уровне модели OSI происходит передача физических сигналов (токов, света, радио) от источника к получателю. На этом уровне мы оперируем кабелями, контактами в разъемах, кодированием единиц и нулей, модуляцией и так далее. Среди основных стандартов передачи – это стандарт **Ethernet**.

***2. КАНАЛЬНЫЙ (DATA LINK) УРОВЕНЬ***

После того, как мы получили физический сигнал с первого уровня, на втором уровне проверяются и исправляются ошибки передачи этих физических сигналов. На этом уровне появляются такие термины, как «фрейм» (кадр) и добавляются служебные данные, которые несут информацию о том, от кого и кому передаются данные.

***3. СЕТЕВОЙ (NETWORK) УРОВЕНЬ***

Идем вверх! Сетевой уровень вводит термин «маршрутизация» и, соответственно, IP – адрес. Кстати, для преобразования IP – адресов в MAC – адреса и обратно используется протокол ARP.

Именно на этом уровне происходит маршрутизация трафика, как таковая. Если мы хотим попасть на сайт wiki.merionet.ru, то мы отправляем DNS – запрос, получаем ответ в виде IP – адреса и подставляем его в пакет. Да – да, если на втором уровне мы используем термин фрейм/кадр, как мы говорили ранее, то здесь мы используем пакет.

***4. ТРАНСПОРТНЫЙ (TRANSPORT) УРОВЕНЬ***

Транспортный уровень, как можно понять из названия, обеспечивает передачу данных по сети. Здесь две основных рок – звезды – TCP и UDP. Разница в том, что различный транспорт применяется для разной категории трафика. Принцип такой:

- Трафик чувствителен к потерям - нет проблем, TCP (Transmission Control Protocol)! Он обеспечивает контроль за передачей данных;

- Немного потеряем – не страшно - по факту, сейчас, когда вы читаете эту статью, пару пакетов могло и потеряться. Но это не чувствуется для вас, как для пользователя. UDP (User Datagram Protocol) вам подойдет. А если бы это была телефония? Потеря пакетов там критична, так как голос в реальном времени начнет попросту «квакать».

***5. СЕАНСОВЫЙ (SESSION) УРОВЕНЬ***

Сеансовый уровень занимается тем, что управляет соединениями, или попросту говоря, сессиями. Он их разрывает. Помните мем про «НЕ БЫЛО НИ ЕДИНОГО РАЗРЫВА»? Мы помним. Так вот, это пятый уровень постарался :)

***6. УРОВЕНЬ ПРЕДСТАВЛЕНИЯ (PRESENTATION)***

На шестом уровне творится преобразование форматов сообщений, такое как кодирование или сжатие. Тут живут JPEG и GIF, например. Так же уровень ответственен за передачу потока на четвертый (транспортный уровень).

***7. УРОВЕНЬ ПРИЛОЖЕНИЯ (APPLICATION)***

На седьмом этаже, на самой верхушке айсберга, обитает уровень приложений! Тут находятся сетевые службы, которые позволяют нам, как конечным пользователям, серфить просторы интернета. Гляньте, по какому протоколу у вас открыта наша база знаний? Правильно, HTTPS. Этот парень с седьмого этажа. Еще тут живут простой HTTP, FTP и SMTP.

КРИТЕРИИ НАЧАЛА И ОКОНЧАНИЯ ТЕСТИРОВАНИЯ

**Критерии начала тестирования такие:** *чем раньше в жизненном цикле программы начнется тестирование, тем в большей степени мы можем быть уверены в ее качестве.*

**Критерии окончания тестирования такие:**

*1. граничные сроки, установленные заранее;*

*2. выполнение всех предусмотренных тест-кейсов;*

*3. достижение определенного уровня тестового покрытия;*

*4. когда после определенного момента, мы практически не находим новых багов или критических дефектов;*

*5. решение менеджмента.*

REST И SOAP – ЧТО ЭТО?

Многие сравнивают REST и SOAP, хотя это не совсем корректно. REST и SOAP на самом деле не сопоставимы. REST — это архитектурный стиль, а SOAP — это формат обмена сообщениями.

**SOAP (от англ. Simple Object Access Protocol — простой протокол доступа к объектам)** – протокол обмена данными, представленными в формате XML. А XML — это язык для описания данных, причём он очень похож на HTML, но предназначен для хранения и транспортировки данных, в то время как HTML предназначен, в первую очередь, для правильности отображения данных.

**REST (от англ. Representational State Transfer — «передача состояния представления») —** это набор правил, по которым программист организовывает написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

*- В* ***REST*** *уже может использоваться обмен данными в форматах и* ***XML****, и* ***JSON****, и какие-нибудь другие. Наиболее популярным является JSON.*

*-* ***REST*** *работает только по протоколам HTTP/HTTPS*

**RESTful веб-сервис/система** – такой сервис/система, которые полностью отвечают требованиям REST

Кратко об отличиях:

- SOAP это протокол, REST – это архитектурный стиль, SOAP имеет веб-сервис WSDL с прописанными методами, которые можно удаленно вызывать.

- REST использует Json и XML, SOAP только XML.

- REST работает только по HTTP/HTTPS, SOAP с любым протоколом прикладного уровня: SMPT, FTP, HTTP, HTTPS, POP3

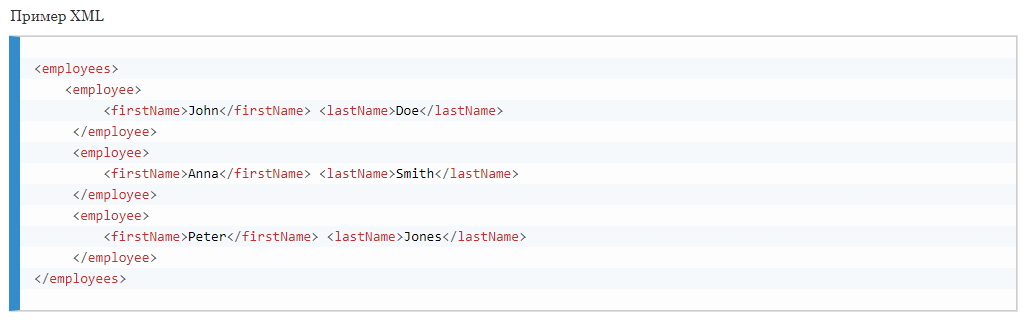
- REST более простой, гибкий и быстрый, SOAP типизированный, но в некоторых случаях лучше визуализируется за счет применения им синтаксиса похожего на HTML разметку.

XML И JSON ФОРМАТЫ

И JSON, и XML могут использоваться для получения данных с веб-сервера.

В следующих примерах JSON и XML определяется один и тот же объект employees с массивом из 3 элементов:





### JSON похож на XML тем, что

* И JSON, и XML имеют "самоописательную" структуру (понятную для человека)
* И JSON, и XML имеют иерархическую структуру (значения внутри значений)
* И JSON, и XML могут парситься и использоваться любым языком программирования
* И JSON, и XML могут извлекаться при помощи XMLHttpRequest

### JSON не похож на XML тем, что

* JSON не использует конечный тег
* JSON короче
* JSON быстрее читать и писать
* JSON может использовать массивы

Самое значительное отличие JSON от XML заключается в том, что XML приходится парсить при помощи специального XML парсера, а JSON может парситься стандартной функцией JavaScript.

### Почему JSON лучше XML

Парсить XML гораздо сложнее, чем JSON. При этом JSON парсится в уже готовый для использования объект JavaScript.

Для AJAX приложений JSON быстрее и проще XML

Основные принципы программирования

KISS

KISS («Keep it simple, stupid» - «Делай проще, тупица») - провозглашает, что простота кода – превыше всего, потому что простой код – наиболее понятный.

DRY

DRY («Don’t repeat yourself» - «Не повторяйся») - это принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода. Принцип DRY формулируется как: «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы».

SOLID

SOLID — это аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании.

S - Single responsibility — принцип единственной ответственности

O - Open-closed — принцип открытости / закрытости

L - Liskov substitution — принцип подстановки Барбары Лисков

I - Interface segregation — принцип разделения интерфейса

D - Dependency inversion — принцип инверсии зависимостей

[S]Принцип единственной обязанности / ответственности (single responsibility principle / SRP) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.

[O]Принцип открытости / закрытости (open-closed principle / OCP) декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

[L] Следование принципу подстановки Барбары Лисков заключается в том, что при построении иерархий наследования создаваемые наследники должны корректно реализовывать поведение базового типа. То есть если базовый тип реализует определённое поведение, то это поведение должно быть корректно реализовано и для всех его наследников.

[I] Принцип разделения интерфейсов заключается в том, что большие, объемные интерфейсы надо разбивать на мелкие таким образом, чтобы клиенты  
маленьких интерфейсов знали только о тех методах которые необходимы им в работе.

[D] Принцип инверсии зависимостей – такой принцип объектно-ориентированного программирования, суть которого состоит в том, что классы должны зависеть от абстракций, а не от конкретных деталей.

YAGNI

YAGNI («You aren’t gonna need it» - «Тебе это не понадобится») - принцип проектирования ПО, при котором в качестве основной цели и/или ценности декларируется отказ от избыточной функциональности.

Отличия mocks от stubs

Главное различие между stubs и mocks заключается в том, что в одном случае мы управляем состоянием, а в другом - поведением.

Когда мы используем mocks, мы заменяем весь модуль на mock (ложный, тестовый объект, имитирующий настоящий). А stub - это функция, которая всегда выводит один и тот же результат, вне зависимости от того, что было подано на вход. Mocks используют для того, чтобы проверить, была ли функция вызвана с правильными аргументами, а stubs, чтобы протестировать, как функция работает с полученным ответом. Стабы нужны для проверки состояния метода, а моки используются для регулировки поведения.

Ещё статья: <https://testengineer.ru/mocks-and-stubs/>

DOM (Document Object Model)

DOM (Document Object Model) — это специальная структура, представленная в виде древа HTML-тегов, которая позволяет управлять HTML-разметкой из JavaScript-кода

Браузер создаёт DOM при загрузке страницы, складывает его в переменную document и сообщает, что DOM создан, с помощью [события](https://doka.guide/js/events/) [DOMContentLoaded](https://doka.guide/js/event-domcontentloaded/). С переменной document начинается любая работа с HTML-разметкой в JavaScript.

Парадигмы разработки

Существует несколько парадигм разработки. Одни из популярных:

*TDD (Test Driven Development)* – парадигма разработки через тестирование. **Изначально пишется тест, покрывающий желаемое изменение, затем пишется программный код, который реализует желаемое поведение системы и позволит пройти написанный тест**

*BDD (Behaviour Driven Development) -* это разработка, основанная на описании поведения. **Определенный человек (или люди) пишет описания вида "я как пользователь хочу когда нажали кнопку пуск тогда показывалось меню как на картинке". В последствии такие описания конвертируются в код и тестируются**

*DDD (Data Driven Development)* **-** парадигма разработки, при которой **логика программы и написанный код определяется входными данными**