----- Оглавление -----

----- Введение (F.A.Q.) -----

Хочу войти в айти (в разработку) через тестирование, хороший план?
Это имело смысл (и то спорно) несколько лет назад, но после эпидемии и агрессивной рекламы работы в IT со стороны обучающих компаний появились тысячи выпускников курсов по тестированию, так что этого “легкого пути вкатиться в айтишечку” уже нет и не будет. Конкурс на одно место на удаленку может доходить до нескольких сотен человек. Конечно, в разработке тоже конкуренции поприбавилось, но если вы нацелены стать программистом, то никакие вхождения через другие специальности не имеют ни малейшего смысла - тестирование и разработка - разные профессии и опыт работы джуниор тестировщиком вам практически ничем не пригодится, вы лишь потратите время на подготовку к собеседованиям (а требования теперь весьма высокие), а потом еще на обучение непосредственно той специальности, куда хотели изначально, т.к. опыт по сути не релевантен и спрашивать будут с нуля.

Доп. материал:
Мифы о тестировании #2 / О чем не говорят на курсах по тестированию / Правда о работе в IT
Хочу зарабатывать много денег, мне сюда?
Первые зарплаты будут мизерными (особенно учитывая конкурс на места), а подняться выше по карьерной лестнице без искреннего интереса и запала не получится, тестирование - слишком обширная область знаний. Без внутренней мотивации к ежедневному самообучению не получится зарабатывать больше чем в любой другой профессии на старте. Так что сферу деятельности стоит менять только если вы всю жизнь чувствовали, что занимаетесь не тем, а тут прям вот екнуло и хочется взахлеб осваивать именно тестирование. Но даже в этих случаях нужно понимать, что далеко не всем компаниям требуются профессиональные тестировщики, разработчикам в этом смысле найти применение своим навыкам куда проще. Именно по этой причине существует отток уже проработавших какое-то время в тестировании специалистов в другие направления: менеджмент, чистая автоматизация, разработка. Фактически они просто не смогли найти дальнейшие пути развития своих навыков. Соответственно и зарплаты здесь в среднем ниже, чем на многих специальностях в IT.
Статистика зарплат: Россия, Украина, Беларусь
Хочу работать удаленно джуном, это возможно?
К сожалению, шансы устроиться на удаленку специалисту без опыта довольно низкие - высокая конкуренция, к тому же компании охотнее берут начинающих в офис (так эффективнее обучать). Поэтому попробовать, конечно, стоит, но рассчитывать на это особо не нужно. Да и начинающему действительно продуктивнее находиться в офисе вместе со всей командой, в гуще событий.
Что реально должен знать junior? А что спросят на собеседовании?
Вот пара ссылок на карты знаний для тестировщиков, но можете на просторах найти и другие.
https://github.com/anas-qa/Quality-Assurance-Road-Map
https://www.mindmeister.com/ru/1324282825/junior-qa?fullscreen=1
https://www.mindmeister.com/ru/1558647509?t=973hdS2AKb

Некоторые компании подробно расписывают на своих порталах ожидания от каждой стадии развития сотрудника, тоже не проблема найти, по этой же теме много видео на Youtube (раз, два, три, четыре, …). Еще один ориентир – просто открыть и почитать вакансии в своем городе, что в среднем у вас требуется от новичка.
Если же попытаться выделить самые частые вопросы на собеседованиях, то получится примерно следующее:

- Что такое тестирование?
- Зачем оно нужно?
- QA/QC/Тестирование - разница?
- Что такое качество ПО?
- Верификация и валидация?
- Severity priority?
- Виды, типы, уровни тестирования?
- Виды документации? Тест-план, тест-кейс, чек-лист?
- Что такое баг? Его жизненный цикл?
- Техники тест-дизайна?
- SDLC, STLC; Методологии разработки ПО?
- Мобильное тестирование: его особенности (и в частности жизненный цикл Android и iOS приложения)?
- Клиент-серверная архитектура?
- API?
- Базовое знание сетей: HTTP(S), его методы, коды ответов, Query-параметры, REST/SOAP, JSON/XML
- Базы данных: что такое SQL, СУБД, основные команды (особенно любят джойны).
- Инструменты: Chrome DevTools, Postman, Charles/Fiddler, GIT, TMS
- Практика: тестирование форм или какого либо сайта, приложения (в частности составление тест-кейсов и баг-репортов), придумать хороший summary для репорта, определение severity/priority; SQL запросы; что-нибудь на “подумать”.

В случае gamedev могут еще спросить про последнее во что играл, что понравилось/не понравилось и т.п.

Помимо вышеперечисленного нужно помнить об английском языке и его важности в работе. Конечно, есть небольшое количество компаний, которые не станут уделять этому внимания, но если в вакансии указан необходимый уровень владения языком, то будьте готовы к тому, что как минимум попросят ответить на какой-нибудь простенький житейский или из списка HR-вопросов на английском. В отдельных случаях, где явно указана необходимость разговорного уровня, все собеседование вполне может пройти на английском.

Доп. материал:
Образ современного тестировщика. Что нужно знать и уметь
Что должен знать тестировщик бэкенда
Тестировщик ПО / что делает QA Engineer / интервью с Artsiom Rusau QA
Исследование рынка труда в QA
Гид по профессии тестировщик: чем занимается специалист в сфере QA, сколько зарабатывает, что надо знать и где учиться

Качественное тестирование ПО
С чего начать обучение?
Начать нужно с ознакомления с тестированием и я считаю, что лучший выбор для этого - книга Романа Савина “Тестирование дот ком”. После ознакомления я бы посоветовал выбрать по отзывам в коммьюнити хороший базовый онлайн-курс и пройти его, либо по возможности пойти на офлайн-курсы местной компании с возможностью последующего трудоустройства - это вообще лучший вариант. Если нет желания или возможности, то после Савина стоит начать с книги Святослава Куликова “Тестирование программного обеспечения. Базовый курс” и далее уже имея общее представление и понимая азы равномерно восполнять пробелы, подготавливаясь к собеседованиям.
Тестирование – самая широкая область в IT. Т.к. теория хоть и не сильно сложная, но ее настолько много, что невозможно изучить все, нужно пытаться как можно быстрее найти применение своим навыкам. Начать стоит с классики типа тестирования форм, тренировочных сайтов с дефектами специально для тестировщиков и т.п. Не стоит забывать и о софт скилах и базовой грамотности. Бич многих современных тестировщиков даже высокого уровня – безграмотность, поэтому смолоду тренируйтесь в составлении тестовых артефактов.
По мере роста компетенций как можно раньше стоит начать проходить собеседования и пытаться устроиться на любую стажировку, вообще любой вариант, где вы сможете применять знания и указать этот опыт в резюме, т.к. без опыта сейчас найти работу очень трудно. Если нет никаких оффлайн вариантов, как было у меня, можете регистрироваться на краудтестинговых платформах (но зачастую это гиблое дело + многие работодатели игнорируют такой опыт), искать в тг-каналах возможности протестировать какие-то проекты за бесплатно (иногда там ищут волонтеров за опыт) либо придумать такой тестовый проект себе самому - взяться тестировать любимый(или какой-либо популярный) сервис, но делать это близко к тому, будто это ваша реальная работа. То есть чтобы было что потом рассказать и показать результаты (тест-кейсы, баг-репорты и т.п.). Именно поэтому эффективнее обучаться когда есть опытный наставник, который дает приближенные к реальным задания и дает обратную связь по выполненной работе.
Когда вы устроитесь на свою первую работу, спустя некоторые время сможете начать готовиться к дальнейшему развитию и выбору направления, ведь никто не заставляет всю жизнь быть ручным тестировщиком. Вы можете сосредоточиться на mobile/web/desktop платформе, профессионально развиваться в менеджеры или автоматизацию, готовиться к узкой специализации — безопасности или performance и т. д., а также сфокусироваться на подготовке по перспективным направлениям:

- ML&AI в QA
- QAOps
- Тестирование IoT
- Тестирование больших данных

Помимо прочего, специалисту, планирующему развиваться профессионально, желательно как можно раньше начать сначала посещать релевантные митапы и конференции, а когда-нибудь и начать выступать в роли докладчика. Также не лишними будут различные сертификации (хотя бы тот же ISTQB разных уровней).

Дополнительные ссылки:
Тестирование в эпоху ИИ

12 Important Software Testing Trends for 2021 You Need to Know
Как учиться, чтобы научиться

Какие есть полезные ресурсы кроме этого?

Telegram:
Must have! Каналы это: знакомство с коммьюнити, живое общение, уникальный опыт тысяч коллег; богатая история сообщений, где поиском по истории сообщений можно найти все что угодно; в шапке каждого канала закреплено сообщение со своим набором полезностей. Кроме того, некоторые каналы специализируются на мониторинге нового полезного материала с основных порталов, так что можно даже не погружаться с головой в хабр, доу, медиум и т.п., вам отберут все самое полезное. В конце концов, в этих каналах публикуются анонсы грядущих онлайн-мероприятий, чтобы ничего не упустить.

- Всем новичкам сюда! Тренировочные собеседования, интересные обсуждения https://t.me/qa_interviews
- Огромный чат, ориентированный на джуниоров https://t.me/qajuniors
- Огромный чат, ориентированный на уже работающих в сфере тестирования https://t.me/qa_ru
- Чат по геймдеву https://t.me/qa_gamedev
- Обсуждение курсов, отзывы о них https://t.me/qa_courses
- Тут можно размещать свое резюме https://t.me/qa_resumes
- Тут отзывы о компаниях https://t.me/qa_bad_company
- Обсуждение финансов https://t.me/qa_fin
- Публикация книг https://t.me/booksqa
- Авторский бложик https://t.me/shooandendlessagony
- Репосты новых статей и полезных ссылок с разных сайтов:
- https://t.me/qa_wiki
- https://t.me/serious_tester
- https://t.me/qa_pro
- https://t.me/qa_chillout
- https://t.me/yetanotherqa

Youtube-каналы:

- Вадим Ксендзов (тренировочные собеседования!) • https://www.youtube.com/channel/UC• 6• hNNlCXv• 1• ZgdGpziNf• 83• RA
- Mikhail Portnov: • https://www.youtube.com/channel/UCDbzkNMBNZJBKP• 4• C-• 9• qGw• 4• Q
- Podlodka Podcast • https://www.youtube.com/channel/UCOei• 1• E• 1• Vqq• 10• S• 913• OEqTWGw
- QA START UP: • https://www.youtube.com/channel/UCAlCZby• 7• zJHzyhOmS• 8• issDQ
- Компания DINS: • https://www.youtube.com/channel/UCmJYB• 3• hvbF• 3• AlVh• 9• HFeXX• 3• A
- Artsiom Rusau QA Life https://www.youtube.com/c/ArtsiomRusauQALife
- Леша Маршал • https://www.youtube.com/channel/UCTVciJQp• 8• eYwKLLQII-iSJw

- Тестирование: • https://www.youtube.com/channel/UC• 9• VeXtf• 7• fcCJUfmZ_cyweXA
- Fest Group: • https://www.youtube.com/channel/UCITnsvgTiW• 2• YcfP• 1• tcI• 2• oKA
- QAGuild: • https://www.youtube.com/channel/UCHtyBZ• 2• XbtsRmNiAxh• 48• RGg

Web:
Большой список тестовых площадок для тренировок
- https://cantunsee.space/
- https://software-testing.ru/ (+форум)
- http://www.mobileappstesting.com/

https://www.guru• 99• .com
- https://www.softwaretestingmaterial.com/
- https://www.ministryoftesting.com/
- http://www.testingeducation.org/BBST/foundations/
- https://www.learnqa.ru/

Книги:
Очень хвалят вот эту подборку
http://okiseleva.blogspot.com/2014/02/blog-post_6.html?m=1
Подборка книг по ИБ https://habr.com/ru/company/mailru/blog/282700/
Перевод книги Ли Копланда “A Practitioner's Guide to Software Test Design”

Еще встречал вот такой список:
- Роман Савин «Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах» (можно сказать худ.лит, «для чайников»)
- Святослав Куликов «Тестирование программного обеспечения. Базовый курс.»
- Арбон Джейсон, Каролло Джефф, Уиттакер Джеймс «Как тестируют в Google»
- Борис Бейзер «Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем»
- Рекс Блэк «Ключевые процессы тестирования»
- Гленфорд Майерс, Том Баджетт, Кори Сандлер «Искусство тестирования программ.»

Microsoft Corporation - Performance testing Guidance for Web Applications

Мобильные приложения:
StrimQa • —• • ваш карьерный навигатор!
Другие сборники материала и ответов на вопросы:
https://drive.google.com/file/d/• 1• x• 9• oeWuPiVqytNYTKunJgLLarq• 4• IBdeTY/view?usp=sharing
https://www.istqb.org/downloads/category/• 2• -foundation-level-documents.html
- https://github.com/ultragreatester/how-to-qa

https://habr.com/ru/post/• 257529• /
https://docs.google.com/file/d/• 18• yJqziwhxz• 5• khP• 1• mtrme•1• mNJW• 95• mwrgc/edit?filetype=msword

Словари терминов (в т.ч. элементов интерфейса):

Словарик айтишника или Что? Где? Куда? Часть • 1
IT-• словарик для не-айтишников
https://docs.google.com/spreadsheets/d/• 1• LgytNrl• 7• ep• 9• wlr• 3• A_• 3• u• 0• NitQsrZzKhEQwC-OTQfbLAM/edit?usp=sharing
https://docs.google.com/spreadsheets/d/• 1• r• 5• Ek• 83• V• 4• lHkOsW• 52• DyVT• 8• iJepR• 20• oZu_Jy• 5• vAkq• 7• Srl/edit?usp=sharing
Элементы интерфейса сайта
UI-• элементы и жесты в мобильных приложениях
Словарь тестировщика

Чек-листы и идеи для тестов:
Где брать идеи для тестов (подборка полезных ссылок)
37 • источников тест-идей
Checklists Base :)
Чек-лист тестирования мобильных приложений
Дополняем чек-лист тестирования при обновлении иконки и сплеша в мобильных приложениях
Чеклист для тестирования мобильных приложений
Чек-лист для тестирования числового поля
Чек-лист тестирования WEB приложений
Testing checklist for mobile applications
iOS App Testing Template
Getting started with mobile testing
Mobile testing in a nutshell
Am I Really Done Testing?
Mobile Testing Checklist
Testing Criteria for Android Applications
A mnemonic for mobile app testing
Test Mobile Applications with I SLICED UP FUN!
Mobile App Test Coverage Model : LONG FUN CUP
Тестирование новой фичи
Основные инструменты тестировщика?

- Мультитул: DevTools;
- Снифферы: Charles Proxy, Fiddler;
- Тестирование API: Postman, SoapUI;
- Тестирование производительности: JMeter;
- Тестирование безопасности: Kali linux, Santoku Linux + drozer, OWASP ZAP, …;
- Тестирование UI/UX: Figma, Zeplin, любой mindmap-like продукт;
- Фермы устройств для тестирования мобильных приложений: BrowserStack, Xamarin, AWS;

Инструменты тестирования Android приложений

- Системы контроля версий: GIT;
- Взаимодействие с базами данных: язык SQL, системы СУБД;
- Системы CI/CD: Jenkins/TeamCity;
- Прочее: мессенджеры, баг-трекинговые системы и TMS, генераторы тестовых данных и т.п.

DevTools:
В каждый современный браузер встроены инструменты разработчика — они позволяют быстро отловить и исправить ошибки в разметке или в коде. С их помощью можно узнать, как построилось DOM-дерево, какие теги и атрибуты есть на странице, почему не подгрузились шрифты и многое другое:
Проверка ответа сервера
Проверка мобильной адаптивности
Проверка мобильной выдачи
Региональная поисковая выдача
Изменение дизайна
Анализ протокола безопасности
Анализ скорости загрузки страницы
Средства консоли Chrome, которыми вы, возможно, никогда не пользовались

Postman:
Постман представляет собой мультитул для тестирования API. В нем можно создавать коллекции запросов, проектировать дизайн API и создавать для него моки (заглушки-имитации ответов реального сервера), настраивать мониторинг (периодическая отправка запросов с журналированием), для запросов возможно написание тестов на JS, есть собственный Runner и т.д. Однако постман сложно назвать подходящим инструментом для серьезной автоматизации ввиду сложности поддержки тестов, но при этом он хорошо подойдет в простых случаях или как инструмент поддержки а анализа: проверка работоспособности endpoint, дебаг тестов, простая передача информации о дефектах (можно сохранить запрос в curl, ответ в json и т.п.). Postman также может работать без графического интерфейса (newman).
• Аналог: https://hoppscotch.io/
Курс Тестирование ПО. Занятие • 30• . POSTMAN. Ручное тестирование API | QA START UP
API testing using Postman

Proxy (снифферы трафика):
Charles — инструмент для мониторинга HTTP/HTTPS трафика. Программа работает как прокси-сервер между приложением и сервером этого приложения. Charles записывает и сохраняет все запросы, которые проходят через него и позволяет их редактировать.
Charles: • незаменимый тул в арсенале QA-инженера
Breakpoints charles proxy • Подмена данных
Как приручить Charles Proxy?
Using Web Debugging Proxies for Application Testing
Перехват SSL трафика с Android-приложения
Certificate and Public Key Pinning

Тестирование безопасности:
Чем искать уязвимости веб-приложений: сравниваем восемь популярных сканеров
20 • мощных инструментов тестирования на проникновение в • 2019 • году
10 • лучших инструментов сканирования уязвимостей для тестирования на проникновение • –• • 2020
Пентест веб сайта с помощью Owasp Zap

Проверяем безопасность приложений с помощью Drozer

GIT:
Git - это консольная утилита, для отслеживания и ведения истории изменения файлов, в вашем проекте. Чаще всего его используют для кода, но можно и для других файлов. Например, для картинок - полезно для дизайнеров.
С помощью Git-а вы можете откатить свой проект до более старой версии, сравнивать, анализировать или сливать свои изменения в репозиторий.
Репозиторием называют хранилище вашего кода и историю его изменений. Git работает локально и все ваши репозитории хранятся в определенных папках на жестком диске.
Так же ваши репозитории можно хранить и в интернете. Обычно для этого используют три сервиса: GitHub, Bitbucket, GitLab
Знать команды нужно для:
“1. Если надо что-то сделать с гитом по SSH.
2. Если надо сравнить диффы огромных файлов. Vimdiff вытянет, многие тулы сдохнут или будут сильно тормозить.
3. Тулы часто тупят и работают неправильно.
4. Хорошие тулы платные)” (c) @anton_smolianin
В любом случае, даже нажимая кнопки, требуется понимать, как это работает под капотом хотя бы на элементарном уровне.

Все что нужно для работы с GIT
Git • для новичков (часть • 1•)
Git • изнутри и на практике
Git, • я хочу все отменить! Команды исправления допущенных ошибок

SQL:
Все что нужно для работы с SQL:
• Официальные сайты
SQLite
MySQL
PostgreSQL
• GUI клиенты
MySQL Workbench
HeidiSQL
Navicat for MySQL
dbForge Studio for MySQL
• Основы SQL
Алан Бьюли • HYPERLINK "https://www.amazon.com/Learning-SQL-Generate-Manipulate-Retrieve/dp/1492057614/ref=sr_1_1?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-1"• "• HYPERLINK "https://www.amazon.com/Learning-SQL-Generate-Manipulate-Retrieve/dp/1492057614/ref=sr_1_1?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-1"• Изучаем SQL• HYPERLINK "https://www.amazon.com/Learning-SQL-Generate-Manipulate-Retrieve/dp/1492057614/ref=

sr_1_1?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-1"• "
Линн Бейли • HYPERLINK "https://www.amazon.com/Head-First-SQL-Brain-Learners/dp/0596526849/ref=sr_1_10?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-10"• "• HYPERLINK "https://www.amazon.com/Head-First-SQL-Brain-Learners/dp/0596526849/ref=sr_1_10?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-10"• Изучаем SQL• HYPERLINK "https://www.amazon.com/Head-First-SQL-Brain-Learners/dp/0596526849/ref=sr_1_10?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-10"• "
W• 3• C Introduction to SQL
guru• 99 • | SQL Tutorial for Beginners: Learn SQL in • 7 • Days
• Продвинутый уровень
Энтони Молинаро • HYPERLINK "https://www.amazon.com/SQL-Cookbook-Query-Solutions-Techniques/dp/1492077445/ref=sr_1_2?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-2"• "• HYPERLINK "https://www.amazon.com/SQL-Cookbook-Query-Solutions-Techniques/dp/1492077445/ref=sr_1_2?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-2"• SQL. • Сборник рецептов• HYPERLINK "https://www.amazon.com/SQL-Cookbook-Query-Solutions-Techniques/dp/1492077445/ref=sr_1_2?dchild=1• &• keywords=sQL• &• qid=1613292997• &• s=books• &• sr=1-2"• "
Алекс Кригель • HYPERLINK "https://www.amazon.com/SQL-Bible-Alex-Kriegel/dp/0470229063/ref=sr_1_1?dchild=1• &• keywords=sQL+bible•&• qid=1613293063• &• s=books• &• sr=1-1"• "• HYPERLINK "https://www.amazon.com/SQL-Bible-Alex-Kriegel/dp/0470229063/ref=sr_1_1?dchild=1• &• keywords=sQL+bible•&• qid=1613293063• &• s=books• &• sr=1-1"• SQL. • Библия пользователя• HYPERLINK "https://www.amazon.com/SQL-Bible-Alex-Kriegel/dp/0470229063/ref=sr_1_1?dchild=1• &• keywords=sQL+bible•&• qid=1613293063• &• s=books• &• sr=1-1"• "
Джеймс Грофф, Пол Вайнберг, Эндрю Оппель • HYPERLINK "https://www.amazon.com/SQL-Complete-Reference-James-Groff-dp-0071592555/dp/0071592555/ref=mt_other?_encoding=UTF8"• "• HYPERLINK "https://www.amazon.com/SQL-Complete-Reference-James-Groff-dp-0071592555/dp/0071592555/ref=mt_other?_encoding=UTF8"• SQL • Полное руководство. Третье издание.• HYPERLINK "https://www.amazon.com/SQL-Complete-Reference-James-Groff-dp-0071592555/dp/0071592555/ref=mt_other?_encoding=UTF8"• "• HYPERLINK "https://www.amazon.com/SQL-Complete-Reference-James-Groff-dp-0071592555/dp/0071592555/ref=mt_other?_encoding=UTF8"•
• Практика
SQLAcademy | • Онлайн тренажер с упражнениями по SQL
SQLBolt | Introduction to SQL

W• 3• C | The Try-SQL Editor
HackerRack SQL
Упражнения по SQL
Тест на знание SQL
- https://www.db-fiddle.com/
- Shit happens

SQL Cheat Sheet
Основные команды SQL, которые должен знать каждый программист
27 • распространенных вопросов по SQL с собеседований и ответы на них

Mind maps:
12 • программ и сервисов для создания майндкарт
Как нарисовать карту приложения (mind map)
Mind map • вместо тест-кейса, или Как визуализация позволяет тестировать приложение быстрее
Mind Map • в помощь тестировщику
Mind Map • в тестировании • —• • или легкий способ тестировать сложные приложения

TMS:
Топ-12 лучших систем управления тестированием 2020

Как вообще происходит процесс найма?
Все зависит от компании и в меньшей степени от уровня позиции. В среднем это выглядит так:
- Отклик на вакансию;
- *Опционально: выполнение тестового задания, п.2 и п.3 могут меняться местами;
- Скрининг по телефону (небольшая беседа с HR);
- Полноценное собеседование с HR;
- Техническое собеседование, п.4 и п.5 иногда делают за раз;
- *Опционально: собеседование с боссом;

Качества, которыми нужно обладать тестировщику?
- Развитые софт-скиллы (например, уметь в коммуникацию, не перебивать собеседника и т.п.);
- Общая грамотность;
- Умение обучаться;
- Пытливый ум и желание выяснить первопричину проблемы;
- Устойчивость к рутине;

Доп. материал:
Миф об образе мышления в тестировании
Как составить резюме?
Кратко о базовых рекомендациях.
Резюме стажера или джуниора – ровно 1 страница (имеется в виду вариант в файле, а не на площадках). Помимо PDF желательно иметь вордовскую копию на google-диске. Язык резюме – русский, если нацелены на компании из РФ с клиентами из РФ. В остальных случаях – английский. Лучший вариант оформления шапки:

Просто нормальное фото, ФИО, на какую позицию претендуете, опционально локация и дата рождения, актуальные контакты.
После чего идет раздел с опытом работы (любые практические навыки), где максимально кратко и емко описывается чем конкретно вы занимались. Это самая важная часть резюме. Общее правило - использовать глаголы совершенного вида (сделал то, там-то; а делал, участвовал - ничего о вас не говорит), а еще лучше в формате «зона ответственности + достижения»
Следом – образование и затем ключевые навыки. Не забудьте упомянуть знание иностранных языков. Сориентироваться поможет, например, бесплатный тест EFSET с сертификатом. Никогда не используйте банальные ключевые навыки «ответственный, целеустремленный, …». Только конкретика. Помните, что HR часто ищут по ключевым словам, а вы не должны раздувать ваше резюме всяким мусором. Технологии, инструменты – хороший выбор. Но будьте готовы, что вас по ним детально будут спрашивать в первую очередь.
Раздел «О себе» можно включить, если есть что важного и интересного написать, опять же, коротко и если есть чем выделиться.
При отклике на вакансию встает вопрос о сопроводительном письме и мне понравилось это мнение:
[Переслано от Vincent Jozeph Mousekewitz] (@V_J_Mousekewitz)
“Рассматривайте свое резюме как коммерческое предложение, а сопроводительное письмо -- как быстрый и понятный ответ на вопрос "почему я должен рассмотреть детальнее именно Ваше предложение".
Сопроводительные письма почти всегда читают. Другой вопрос, что они почти всегда написаны плохо, сухо и "не цепляют", что автоматически идет в минус. Если не хотите/не можете/не готовы в эпистолярный жанр -- не пишите. Переформатируйте резюме, чтобы за 30 секунд чтения было понятно, какую боль Вы готовы снять заказчику(работодателю) и какими квалификациями для этого обладаете.
Сопроводительные -- тоже, в каком-то смысле "инструмент". Не надо их писать "чтобы было", такое отношение видно сразу и не играет на руку кандидату.
Однако, если есть "что сказать" -- не держите в себе. Увидели вакансию, считаете себя идеальным кандидатом и готовы аргументировать -- вперед.”

Вообще на тему составления резюме в IT есть миллион статей (пример) и видео на youtube, да и не стоит исключать фактор личных пристрастий нанимателя, так что следует просто ознакомиться с базовыми рекомендациями, при желании скинуть итоговый вариант на оценку в коммьюнити и заняться более насущными вопросами.
Насчет самих откликов, тут на мой взгляд уместна аналогия с холодными звонками, т.е. не нужно на начальном этапе выбирать место работы так, будто собираетесь в ней состариться. Вообще слать резюме стоит не только откликаясь на вакансии. Есть мнение, что когда компания уже выкинула вакансию на работные сайты, это уже тупик (т.к. не нашли кандидата по своим каналам). Шлите на почты компаний, HR-ов, расширяйте сеть контактов в linkedin и т.п.. Слышал, что активные джуны рассылали по несколько сотен писем в неделю. Стоит ли говорить, что они быстро нашли свою первую работу?

Доп. материал:
Что писать в резюме, если нет опыта работы

Инхаус, фриланс, аутсорс компания: куда приземлиться тестировщику, чтобы не разлюбить профессию и расти как на дрожжах

Как проходить собеседование?

Прежде всего, конечно, на него не нужно опаздывать. Стоит прийти немного заранее, оглядеться, привыкнуть к атмосфере и настроиться на нужный лад.

Главное, что нужно сказать о собеседованиях – на них надо ходить. Регулярно. Это отдельный навык, который утрачивается если не практиковать. Не стоит бояться и относиться к ним как к экзамену, вы и сами это поймете на практике. В знаниях это скорее это как калибровка, нахождение ориентира где вы сейчас находитесь и в какую сторону корректировать курс. Прошли – проанализировали – подкачались – повторяете, пока не будет достигнут приемлемый результат. Некоторые советуют даже после устройства на работу периодически ходить на собеседования, чтобы держать этот навык в тонусе и ориентироваться в своей ценности на рынке.

Если вы ищете первую работу, начать ходить на собеседования нужно как можно раньше еще и потому, что никто не задумывается, как долго в реальности может занять процесс поиска работы. Сначала нужно будет выбить себе возможность пройти интервью. Цифры примерно такие: 10 откликов на подходящие вакансии или 100 рассылкой = 1 собес. 10 не заваленных собесов = 1 оффер. Принятие решения компанией может занимать пару недель. В некоторых компаниях этапов может быть штук до 6. Средне-оптимистичный сценарий предполагает от 2-3 месяцев с нуля до начала работы, но это в случае наличия вокруг выбора и хорошей подготовки у кандидата (а на этот счет тоже многие заблуждаются). В не идеальном случае процесс займет от полугода.

Основное, что хочется сказать про само собеседование:

Узнай о компании, в которую идешь проходить собеседование.

Не скромничай и не занижай свои навыки. Ты, возможно, 50-й кандидат на этой неделе и еще 100 будет после тебя. Если будешь мяться – про тебя забудут, как только ты закроешь за собой дверь. Здоровая гипербола и немного красок еще никому не мешали. Но никогда не переходи в ложь.

Потренируйтесь в самопрезентации, записывая себя на видео и пересматривая. Обычно первое, что вас спросят – «расскажите о себе». Основная задача HR – проверить адекватность кандидата, в некоторых случаях еще соответствие определенному заказу из целевой команды (от возраста до хобби, все что угодно). Помните, что собеседуют в компанию в первую очередь человека, а уже потом специалиста. Кто-то сравнивает собеседование со свиданием, где за час вы должны понять, подходите ли вы друг другу. Вопросы на этом этапе в основном стандартные, ответы на них лучше расписать заранее, но не заучивать. Заученный ответ обычно выглядит нелепо, а вот сформулировать свои мысли заранее и понять себя бывает полезно.

Хороший базовый рассказ о себе определит дальнейший ход собеседования, HR будет копать вглубь от заинтересовавших фактов. Кстати, не на все вопросы вы будете знать ответ и это нормально. Одна из задач рекрутера проверить, как вы себя ведете и как отвечаете, когда не знаете ответ. Или вас попробуют заставить сомневаться в заведомо правильном ответе (в этот момент вы вспомните передачу “кто хочет стать миллионером”). Умение грамотно отстаивать свою точку зрения очень важно для тестировщика.

Еще хороший совет - всё, что вы скажете, может и будет использовано против вас. В том смысле, что не говорите лишнего или того, в чем плаваете, иначе очень быстро закопаетесь в новых вопросах.
В конце (хотя бывает и в начале) спрашивает собеседуемый! Помните, что это обоюдный процесс? Они выбирают себе кандидата, но и вы выбираете себе компанию. Немного информации для борьбы со стеснением: для компании найм сотрудника очень затратная затея. В крупных компаниях даже есть бонусы сотрудникам за удачную рекомендацию знакомого в размере тысяч 100 и это в контексте затрат на обычный процесс найма просто копейки. Обе стороны заинтересованы закрыть торги прямо здесь и сейчас, так что не бойтесь задавать вопросы и будьте полноценной второй стороной в этих переговорах.
По поводу спросить – вот безжалостный копипаст (есть мнение, что если все это начнет спрашивать начинающий специалист, то на это покрутят у виска. Просто умело и ненавязчиво вплетайте самое важное для вас в беседу и все будет ок):

- Официальное ли оформление, тип. Белая ли заработная плата? Предусмотрены ли премии?
- Есть ли KPI, что в них входит?
- Какой график работы, как происходят отработки?
- Как часто бывают переработки и оплачиваются ли они?
- Время начала рабочего дня и отношение к опозданиям?
- Схема карьерного роста, матрица компетенций, период и порядок пересмотра з.п.?
- Приветствуется ли инициатива и если нет, насколько она наказуема?
- Есть ли командировки, какие направления и как часто?
- Что входит в социальный пакет и когда он предоставляется?
- Если работа удаленная или частично удаленная — какая техника предоставляется?
- Как организовано рабочее место, что в него входит? Опенспейс или кабинет?
- Есть ли наставничество или менторство в первое время работы в компании?
- Практикуется ли компенсация обучения, заинтересован ли работодатель в сертификации, участии сотрудника в митапах, конференциях и т.п.?
- Как осуществляется контроль за сотрудниками: есть ли тайм-трекеры, камеры и т. д.?
- Наличие спортзала, столовой, душа?
- Наличие библиотеки с актуальной литературой?

Если на собеседовании несколько иерархически подчиненных человек (HR и директор, начальник отдела и директор или топ-менеджер), обратите внимание на модель их общения, на то, слаженно ли они работают, есть ли контакт или же только благоговейное молчание. Команду видно издалека.

- Как отнеслись к вашему резюме: оно одно из многих и вы здесь «на потоке» или оно лежит одно, и вы в фокусе.
- Как распределено время на собеседование и часто ли отвлекаются его участники, вовремя ли начато общение или вам пришлось ждать больше 15 минут.
- Как вам представили руководителя — с регалиями или нет, по имени-отчеству или имени, формально или неформально.
- Как отнеслись к ходу решения вами заданий — как к экзамену или как к деловому обсуждению задачи. Это говорит о вашем уровне в глазах собеседующего.

Собеседование на английском языке практическое такое же*, просто из-за языкового барьера могут возникать трудности. Практикуйтесь! И помните, что вашему собеседнику может быть так же трудно, как и вам.
*- при собеседовании в другую страну следует учитывать культурные особенности (да и законодательство), потому что некоторые ценности и взгляды могут совершенно не очевидным образом быть разными и при этом иметь решающее значение. Здесь нужно целенаправленно читать статьи о найме или релокации в интересующую страну, там упоминаются эти нюансы и особенности.
Доп. материал:
Вопросы для собеседования • —• • от кандидата к работодателю
Как собеседовать работодателя?
О чем поговорить на собеседовании с выпускником онлайн-курсов по тестированию
Как QA найти • «ту • самую» компанию и стать тимлидом
Собеседование для QA: резюме, вопросы на интервью, переговоры о зарплате + полезные ссылки
Сценарий идеального технического собеседования
Собеседование для собеседующих
Обратное собеседование
Чек-лист для подготовки к собеседованию на английском (linkedin, в РФ нужен VPN/proxy)
Ошибки в работе у начинающих тестировщиков?

- Во всем видят дефекты. Как избежать:
- Внимательно анализировать требования
- Владеть информацией о том, как должен работать продукт
- Если сомневаешься, что это дефект – спроси БА или ответственного
- Несколько раз перепроверь прежде чем регистрировать дефект
- Пытаются сразу все сломать. Как избежать:
- Начинать тестирование только с положительных тестов
- Акцентировать внимание на том, что в приоритете для заказчика
- Не проверять редкие сценарии в первую очередь
- Боятся задавать вопросы. Как избежать:
- Начать понимать, что коммуникация это важная и неотъемлемая часть работы
- Не интересуются, кто и за что отвечает, как устроены процессы. Как избежать:
- Узнать у ПМ об областях ответственности каждого члена команды
- Узнать у ПМ о всех процессах на проекте
- Паникуют при малейшей трудности. Как избежать:
- Без паники, п. 3 и 4 помогут разобраться
- Пытаются применить сразу все, что изучили. Как избежать:
- Помнить, что у каждого вида тестирования своя цель
- Бюджет всегда ограничен, расставлять приоритеты
- Задают один и тот же вопрос несколько раз
- Дёргают разработчиков по каждой мелочи (прерывают состояние потока, контекст. Разработка требует держать огромное количество абстракций в голове во время работы над задачей. Это очень легко сбить элементарным вопросом, который находится в первой строчке поисковой выдачи)

Доп. материал:

Как выжить на новой работе или онбординг снизу
Шесть тест-персон, с которыми не стоит иметь дела
Лучше не знать и спросить, чем притворяться, что знаешь, или Шесть подсказок новичку в тестировании
Мои • 3 • ошибки, которые я совершала как junior QA engineer
Я – единственный тестировщик на проекте. Что делать?

Начинать знакомство с проектом лучше с интервью. Станьте журналистом. Что из себя представляет структура организации, а именно кто над кем стоит и кто за что отвечает? Где больше всего багов, каким видят тестирование сейчас и какие ожидания в будущем? Оцените зрелость процессов по CMM и TMM, зрелость проекта (новый/старый-зрелый/старые-зрелые где будут глобальные изменения) и команды, определите методологию разработки. Проведите вводную лекцию команде: что такое QA, как оно может помочь, с какими проблемами обращаться и зачем вообще оно надо. Далее в зависимости от всего этого ищем и смотрим соответствующий вебинар/статью и т.п., в крайнем случае можно поинтересоваться у коллег в тематических сообществах, например, в tg. Вообще создавать отдел тестирования обычно нанимают QA Lead, а если вы джун, то либо работодатель не понимает что делает, ожидая от джуниора построения процессов, о которых он в лучшем случае где-то читал, либо от вас хотят чего-то вполне конкретного и проводить целое расследование не придется - наверняка все объяснят еще на собеседовании. В нормальной ситуации вы проведете исследовательское тестирование, составите набор кейсов, задокументируете все текущие баги и в дальнейшем будете заниматься тестированием новых сборок, проверкой исправления найденных дефектов и проведением регрессии.

Доп. материал:
Как организовать работу QA. Один практически примененный способ
Как QA организовать автоматизацию тестирования на проекте. Один практически примененный способ
Как QA выстроить эффективное взаимодействие с разработчиками. Один возможный путь
Никогда такого не было и вот опять: Построение отдела тестирования - Андрей Мясников. QA Fest • 2018
Процесс: как наладить, а не нагадить - Андрей Мясников. QA Fest • 2015
Как проходит организация тестирования и составление тест планов (в зависимости от проекта)
Концепция построения процесса тестирования в Agile-проектах: • 3• +• 1
Построение процессов тестирования на новом проекте
Мифы о тестировании #• 2 • / О чем не говорят на курсах по тестированию / Правда о работе в IT
• • Что нужно знать о Value Driven Testing. Анализируем ценность и экономическую целесообразность тестирования

----- HR-часть -----
Вопросы с реальных собеседований с этапа HR
Часть из них зададут в любом случае. Список, разумеется, не полный:

- Расскажи о себе (все что хочешь, что нам нужно знать о тебе)
- Есть ли релевантный опыт?
- Какие курсы проходил и вообще, что изучал?
- Что не устраивало на прошлом месте работы (если было), особенно если решил сменить сферу?
- Почему выбрал именно тестирование?
- Чем заинтересовала именно наша компания?
- Как часто бываешь на собеседованиях?
- Уровень английского? (вопрос могут задать на английском, многие теряются в этот момент)
- (Если требуется и уровень хороший) расскажите на английском: как доехали до собеседования/о себе (только не как в обществе анонимных алкоголиков) /почему считаешь, что можешь стать тестировщиком/ как прошел вчерашний день/о своих хобби/ и т.п.
- Как в целом смотришь на мир, как решаешь возникающие проблемы?
- 3 твоих сильных и 3 слабых стороны?
- Как отдыхаешь? Как проводишь свободное время?
- Какие хобби?
- Что последнее прочитал техническое? Не техническое?
- Если бы мог вернуться в начало осознанной жизни, выбрал бы иной карьерный путь?
- 3 примера, что тебе положительного дал предыдущий опыт работы (если есть)
- 3 плюса и 3 минуса в сфере тестирования лично для тебя
- Как видишь развитие в этой сфере, кем видишь себя через год, три?
- Какая-то одна вещь или ситуация, которой ты гордишься
- Представим, что остальных кандидатов много и они опытнее (обычно так и есть), может у тебя есть какие-то преимущества перед ними? Почему ты думаешь, что лучше других кандидатов?
- Зарплатные ожидания сейчас, после испытательного срока, через год?
- Есть ли какие-то факторы, с которыми ты согласишься на меньшие деньги?
- С чем точно не готов мириться в отношении компании или руководителя?
- Ожидания от работы?
- Отношение к переработкам?
- Представь, что ты работаешь уже полгода. Опиши свой рабочий день.
- Что если при выполнении задачи понимаешь, что не укладываешься в сроки?

----- Теоретическая часть -----

Общие понятия
Что означает тестирование ПО?
Тестирование (testing) программного обеспечения - с технической точки зрения заключается в выполнении приложения (Application Under Testing (AUT) или Implementation Under Testing (IUT)) на некотором множестве исходных данных и сверке получаемых результатов с заранее известными (эталонными) с целью установить соответствие различных свойств и характеристик приложения заказанным свойствам. Как одна из основных фаз процесса разработки программного продукта (Дизайн

приложения - Разработка кода - Тестирование), тестирование характеризуется достаточно большим вкладом в суммарную трудоемкость разработки продукта. Широко известна оценка распределения трудоемкости между фазами создания программного продукта: 40%-20%-40%, из чего следует, что наибольший эффект в снижении трудоемкости может быть получен прежде всего на фазах Design и Testing. В более широком смысле, тестирование ПО - это техника контроля качества программного продукта, включающая в себя проектирование тестов, выполнение тестирования и анализ полученных результатов.

Почему требуется тестирование ПО?

- Процесс тестирования гарантирует, что ПО будет работать в соответствии с ожиданиями клиентов и на имеющемся у них оборудовании.
- Это уменьшает циклы кодирования, выявляя проблемы на начальном этапе разработки. Обнаружение проблем на более ранних этапах SDLC обеспечивает правильное использование ресурсов и предотвращает повышение стоимости.
- Команда тестирования привносит взгляд клиента в процесс и находит варианты использования, о которых разработчик может не подумать.
- Любой сбой, дефект или ошибка, обнаруженные клиентом в готовом продукте, нарушают доверие к компании.

Что означает обеспечение качества (Quality Assurance - QA) при тестировании ПО?

Это совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации ПО и информационных систем, предпринимаемых на разных стадиях жизненного цикла ПО, для обеспечения требуемого уровня качества выпускаемого продукта.

Доп. материал: QA — специалист по пожарной безопасности вашего проекта

Что означает контроль качества (Quality Control - QC) при тестировании ПО?

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

Что означает качество ПО? (Software Quality)

Качеству очень сложно дать неформальное определение, оно заключается в соответствии требованиям (conformance to requirements) и пригодности к использованию (fitness for use). Качество программного продукта характеризуется набором свойств, определяющих, насколько продукт "хорош" с точки зрения заинтересованных сторон, таких как заказчик продукта, спонсор, конечный пользователь, разработчики и тестировщики продукта, инженеры поддержки, сотрудники отделов маркетинга, обучения и продаж. Каждый из участников может иметь различное представление о продукте и о том, насколько он хорош или плох, то есть о том, насколько высоко качество продукта. То есть, основная последовательность действий при выборе и оценке критериев качества программного продукта включает:

- Определение всех лиц, так или иначе заинтересованных в исполнении и результатах данного проекта.
- Определение критериев, формирующих представление о качестве для каждого из участников.
- Приоритезацию критериев, с учетом важности конкретного участника для компании, выполняющей проект, и важности каждого из критериев для данного участника.

• Определение набора критериев, которые будут отслежены и выполнены в рамках проекта, исходя из приоритетов и возможностей проектной команды. Постановка целей по каждому из критериев.
• Определение способов и механизмов достижения каждого критерия.
• Определение стратегии тестирования исходя из набора критериев, попадающих под ответственность группы тестирования, выбранных приоритетов и целей.
Доп. материал:
• https://analytics.infozone.pro/software-quality/ https://www.intuit.ru/studies/courses/• 48• /• 48• /lecture/• 1440• ?page=• 1
Кто несет ответственность за качество тестирования приложения? • 10 • причин попадания ошибки в продакшен
На ком лежит ответственность за качество программного обеспечения?
What Is Cost Of Quality (COQ): Cost Of Good And Poor Quality
Объясните отличия в QA, QC и тестировании

Разница между верификацией и валидацией? (Verification и Validation)
• Верификация — это процесс включающий в себя проверку Plans, Requirement Specifications, Design Specifications, Code, Test Cases, Check-Lists, etc.
• Верификация всегда проходит без запуска кода.
• Верификация использует методы — reviews, walkthroughs, inspections, etc.
• Верификация отвечает на вопрос “Делаем ли мы продукт правильно?”
• Верификация поможет определить, является ли программное обеспечение высокого качества, но оно не гарантирует, что система полезна. Проверка связана с тем, что система хорошо спроектирована и безошибочна.
• Верификация происходит до Validation.
Она содержит все активности которые позволяют достигнуть высокого качества программного обеспечения:
• Inspection in software engineering, refers to peer review of any work product by trained individuals who look for defects using a well defined process. (Fagan inspection)
• Walkthroughs In software engineering, a walkthrough or walk-through is a form of software peer review «in which a designer or programmer leads members of the development team and other interested parties go through a software product, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems».
• Reviews In software development, peer review is a type of software review in which a work product (document, code, or other) is examined by its author and one or more colleagues, in order to evaluate its technical content and quality.
Валидация (validation) – это процесс оценки конечного продукта, необходимо проверить, соответствует ли программное обеспечение ожиданиям и требованиям клиента. Это динамический механизм проверки и тестирования фактического продукта.
• Валидация всегда включает в себя запуск кода программы.
• Валидация использует методы, такие как тестирование Black Box, тестирование White Box и нефункциональное тестирование.
• Валидация отвечает на вопрос “Делаем ли мы правильный продукт?”
• Валидация проверяет, соответствует ли программное обеспечение требованиям и ожиданиям клиента.

- Валидация может найти ошибки, которые процесс Verification не может поймать.
- Валидация происходит после Verification.

На практике, отличия верификации и валидации имеют большое значение: заказчика интересует в большей степени валидация (удовлетворение собственных требований); исполнителя, в свою очередь, волнует не только соблюдение всех норм качества (верификация) при реализации продукта, а и соответствие всех особенностей продукта желаниям заказчика.

Верификация Валидация
По факту отвечает на вопрос, правильно ли создается и тестируется ПО и все ли требования учитываются при этом. Отвечает на вопрос, создается ли продукт правильно с точки зрения ожиданий клиента.
В процессе верификации убеждаемся в том, что весь созданный функционал приложения работает корректно и логически верно. При процессе валидации убеждаемся в том, что продукт полностью соответствует поведению, которое от него ожидается и то, что клиент знает о наличии подобного функционала.
В структуру верификации входят такие компоненты, как сверка завалидированным требованиям, технической документации и корректное выполнения программного кода на любом этапе создания и тестирования ПО. Валидация, по своей сути, в большей степени включает в себя общую оценку ПО и может основываться исключительно на субъективном мнении касательно правильности работы приложения или его компонентов.

Источник: Верификация и валидация
Доп. материал:
Большое обсуждение: Verification • HYPERLINK "https://software-testing.ru/forum/index.php?/topic/979-verification-validation-chto-eto-takoe/"
- &• HYPERLINK "https://software-testing.ru/forum/index.php?/topic/979-verification-validation-chto-eto-takoe/"
- Validation - • что это такое?

Чем отличается валидация от верификации
Принципы тестирования?
- Тестирование демонстрирует наличие дефектов
- Исчерпывающее тестирование недостижимо
- Раннее тестирование
- Скопление/кластеризация дефектов
- Парадокс пестицида
- Тестирование зависит от контекста
- Заблуждение об отсутствии ошибок
- Garbage in, garbage out (GIGO)

Принцип 1. Тестирование показывает наличие дефектов
Тестирование может показать, что дефекты присутствуют, но не может доказать, что дефектов нет.
Сколько бы успешных тестов вы не провели, вы не можете утверждать, что нет таких тестов, которые не нашли бы ошибку. Но если мы нашли хотя бы один дефект, мы уже можем утверждать, что в данном ПО присутствуют дефекты.

Принцип 2. Исчерпывающее тестирование невозможно
Вместо попыток «протестировать все» нам нужен некий подход к тестированию (стратегия), который обеспечит правильный объем тестирования для данного проекта, данных заказчиков (и других заинтересованных лиц) и данного продукта. При определении, какой объем тестирования достаточен, необходимо учитывать уровень риска, включая технические риски и риски, связанные с бизнесом, и такие ограничения проекта как время и бюджет. Оценка и управление рисками – одна из наиболее важных активностей в любом проекте.

Принцип 3. Раннее тестирование
Тестовые активности должны начинаться как можно раньше в цикле разработки и быть сфокусированы на определенных целях.
Этот принцип связан с понятием «цена дефекта» (cost of defect). Цена дефекта существенно растет на протяжении жизненного цикла разработки ПО. Чем раньше обнаружен дефект, тем быстрее, проще и дешевле его исправить. Дефект, найденный в требованиях, обходится дешевле всего.
Еще одно важное преимущество раннего тестирования – экономия времени. Тестовые активности могут начинаться еще до того, как написана первая строчка кода. По мере того, как готовятся требования и спецификации, тестировщики могут приступать к разработке и ревью тест-кейсов. И когда появится первая тестовая версия, можно будет сразу приступать к выполнению тестов.

Принцип 4. Скопление дефектов
Небольшое количество модулей содержит большинство дефектов, обнаруженных на этапе предрелизного тестирования, или же демонстрируют наибольшее количество отказов на этапе эксплуатации.
Многие тестировщики наблюдали такой эффект – дефекты «кучкуются». Это может происходить потому, что определенная область кода особенно сложна и запутана, или потому, что внесение изменений производит «эффект домино». Это знание часто используется для оценки рисков при планировании тестов – тестировщики фокусируются на известных «проблемных зонах». Также полезно проводить анализ первопричин (root cause analysis), чтобы предотвратить повторное появление дефектов, обнаружить причины возникновения скоплений дефектов и спрогнозировать потенциальные скопления дефектов в будущем.

Принцип 5. Парадокс пестицида
Если повторять те же тесты снова и снова, в какой-то момент этот набор тестов перестанет выявлять новые дефекты. Повторное применение тех же тестов и тех же методик приводит к тому, что в продукте остаются именно те дефекты, против которых эти тесты и эти методики неэффективны.
Чтобы преодолеть «парадокс пестицидов», необходимо регулярно пересматривать существующие тест-кейсы и создавать новые, разнообразные тесты, которые будут выполняться на различных частях системы.

Принцип 6. Тестирование зависит от контекста
Тестирование выполняется по-разному, в зависимости от контекста. Например, тестирование систем, критических с точки зрения безопасности, проводится иначе, чем тестирование сайта интернет-магазина.

Этот принцип тесно связан с понятием риска. Что такое риск? Риск – это потенциальная проблема. У риска есть вероятность (likelihood) – она всегда выше 0 и ниже 100% – и есть влияние (impact) – те негативные последствия, которых мы опасаемся. Анализируя риски, мы всегда взвешиваем эти два аспекта: вероятность и влияние.
То же можно сказать и о мире ПО: разные системы связаны с различными уровнями риска, влияние того или иного дефекта также сильно варьируется. Одни проблемы довольно тривиальны, другие могут дорого обойтись и привести к большим потерям денег, времени, деловой репутации, а в некоторых случаях даже привести к травмам и смерти.
Уровень риска влияет на выбор методологий, техник и типов тестирования.

Принцип 7. Заблуждение об отсутствии ошибок
Нахождение и исправление дефектов бесполезно, если построенная система неудобна для использования и не соответствует нуждам и ожиданиям пользователей. Заказчики ПО – люди и организации, которые покупают и используют его, чтобы выполнять свои повседневные задачи – на самом деле совершенно не интересуются дефектами и их количеством, кроме тех случаев, когда они непосредственно сталкиваются с нестабильностью продукта. Им также неинтересно, насколько ПО соответствует формальным требованиям, которые были задокументированы. Пользователи ПО более заинтересованы в том, чтобы оно помогало им эффективно выполнять задачи. ПО должно отвечать их потребностям, и именно с этой точки зрения они его оценивают.
Даже если вы выполнили все тесты и ошибок не обнаружили, это еще не гарантия того, что ПО будет соответствовать нуждам и ожиданиям пользователей.
Иначе говоря, верификация != валидация.

* Принцип 8. GIGO.
В компьютерной науке «garbage in – garbage out» (GIGO) — это концепция, в которой ошибочные или бессмысленные входные данные создают бессмысленный вывод или «мусор», т.е. при неверных входящих данных будут получены неверные результаты, даже если сам по себе алгоритм правилен. В тестировании такие случаи иногда создают намеренно, но я добавил этот принцип в общий список для того, чтобы подчеркнуть важность подготовки качественных тестовых данных, положительные они или отрицательные.

Доп. материал: The Cold Hard Truth About Zero-Defect Software
Критерии выбора тестов?
Требования к идеальному критерию тестирования:
- Критерий должен быть достаточным, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.
- Критерий должен быть полным, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.
- Критерий должен быть надежным, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы.
- Критерий должен быть легко проверяемым, например вычисляемым на тестах.

Для нетривиальных классов программ в общем случае не существует полного и надежного критерия, зависящего от программ или спецификаций. Поэтому мы стремимся к идеальному общему критерию через реальные частные. Классы критериев:

- Структурные критерии используют информацию о структуре программы (критерии так называемого "белого ящика").
- Функциональные критерии формулируются в описании требований к программному изделию (критерии так называемого "черного ящика").
- Критерии стохастического тестирования формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.
- Мутационные критерии ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

Структурные критерии используют модель программы в виде "белого ящика", что предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления. Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный класс критериев часто используется на этапах модульного и интеграционного тестирования (Unit testing, Integration testing). Структурные критерии базируются на основных элементах УГП (Управляющий граф программы), операторах, ветвях и путях.

- Условие критерия тестирования команд (критерий C0) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.
- Условие критерия тестирования ветвей (критерий C1) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах автоматизации тестирования.
- Условие критерия тестирования путей (критерий C2) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или числом классов выходных путей).

Структурные критерии не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию C2 мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

Функциональный критерий - важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, контроль степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При функциональном тестировании преимущественно используется модель "черного ящика". Проблема функционального тестирования - это, прежде всего, трудоемкость; дело в том, что документы, фиксирующие требования к программному изделию (Software requirement specification, Functional specification и т.п.), как правило, достаточно объемны, тем не менее, соответствующая проверка

должна быть всеобъемлющей. Ниже приведены частные виды функциональных критериев:

- Тестирование пунктов спецификации - набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза. Спецификация требований может содержать сотни и тысячи пунктов требований к программному продукту и каждое из этих требований при тестировании должно быть проверено в соответствии с критерием не менее чем одним тестом.
- Тестирование классов входных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза. При создании тестов классы входных данных сопоставляются с режимами использования тестируемого компонента или подсистемы приложения, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов. Следует заметить, что перебирая в соответствии с критерием величины входных переменных (например, различные файлы - источники входных данных), мы вынуждены применять мощные тестовые наборы. Действительно, наряду с ограничениями на величины входных данных, существуют ограничения на величины входных данных во всевозможных комбинациях, в том числе проверка реакций системы на появление ошибок в значениях или структурах входных данных. Учет этого многообразия - процесс трудоемкий, что создает сложности для применения критерия.
- Тестирование правил - набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики. Следует заметить, что грамматика должна быть достаточно простой, чтобы трудоемкость разработки соответствующего набора тестов была реальной (вписывалась в сроки и штат специалистов, выделенных для реализации фазы тестирования).
- Тестирование классов выходных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, причем отдельные классы результатов учитывают, в том числе, ограничения на ресурсы или на время (time out). При создании тестов классы выходных данных сопоставляются с режимами использования тестируемого компонента или подсистемы, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов.
- Тестирование функций - набор тестов в совокупности должен обеспечить проверку каждого действия, реализуемого тестируемым модулем, не менее одного раза. Очень популярный на практике критерий, который, однако, не обеспечивает покрытия части функциональности тестируемого компонента, связанной со структурными и поведенческими свойствами, описание которых не сосредоточено в отдельных функциях (т.е. описание рассредоточено по компоненту). Критерий тестирования функций объединяет отчасти особенности структурных и функциональных критериев. Он базируется на модели "полупрозрачного ящика", где явно указаны не только входы и выходы тестируемого компонента, но также состав и структура используемых методов (функций, процедур) и классов.
- Комбинированные критерии для программ и спецификаций - набор тестов в совокупности должен обеспечить проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза. При этом все комбинации непротиворечивых условий надо подтвердить, а условия противоречий следует обнаружить и ликвидировать.

Стохастическое тестирование применяется при тестировании сложных программных комплексов - когда набор детерминированных тестов (X,Y) имеет громадную мощность.
Мутационный критерий (класс IV). Постулируется, что профессиональные программисты пишут сразу почти правильные программы, отличающиеся от правильных мелкими ошибками или описками типа - перестановка местами максимальных значений индексов в описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы цикла на 1 и т.п. Предлагается подход, позволяющий на основе мелких ошибок оценить общее число ошибок, оставшихся в программе. Подход базируется на следующих понятиях: Мутации - мелкие ошибки в программе. Мутанты - программы, отличающиеся друг от друга мутациями . Метод мутационного тестирования - в разрабатываемую программу P вносят мутации, т.е. искусственно создают программы-мутанты P1, P2... Затем программа P и ее мутанты тестируются на одном и том же наборе тестов (X,Y). Если на наборе (X,Y) подтверждается правильность программы P и, кроме того, выявляются все внесенные в программы-мутанты ошибки, то набор тестов (X,Y) соответствует мутационному критерию, а тестируемая программа объявляется правильной. Если некоторые мутанты не выявили всех мутаций, то надо расширять набор тестов (X,Y) и продолжать тестирование.

Подробнее и с примерами в источнике:
https://www.intuit.ru/studies/courses/48/48/lecture/1428?page=1
Доп. материал: https://www.youtube.com/watch?v=qkUCzvP-5mg HYPERLINK "https://www.youtube.com/watch?v=qkUCzvP-5mg&t=20547s"& HYPERLINK "https://www.youtube.com/watch?v=qkUCzvP-5mg&t=20547s"t=20547s
Что такое импакт анализ (анализ влияния, Impact Analysis)?
Impact Analysis (импакт анализ) - это исследование, которое позволяет указать затронутые места (affected areas) в проекте при разработке новой или изменении старой функциональности, а также определить, насколько значительно они были затронуты.
Затронутые области требуют большего внимания во время проведения регрессионного тестирования.
Импакт анализ может быть полезным в следующих случаях:

- есть изменения в требованиях;
- получен запрос на внесение изменений в продукт;
- ожидается внедрение нового модуля или функциональности в существующий продукт;
- каждый раз, когда есть изменения в существующих модулях или функциональностях продукта.

Как мы знаем, в настоящее время продукты становятся все более большими и комплексными, а компоненты все чаще зависят друг от друга. Изменение строчки кода в таком проекте может "сломать" абсолютно все.
Информация о взаимосвязи и взаимном влиянии изменений могут помочь QA:

- сфокусироваться на тестировании функциональности, где изменения были представлены;
- принять во внимание части проекта, которые были затронуты изменениями и, возможно, пострадали;

- не тратить время на тестирование тех частей проекта, которые не были затронуты изменениями.

Источник: Impact Analysis: 6 шагов, которые облегчат тестирование изменений
Что подразумевается под тестовым покрытием? (Test Coverage)
Тестовое Покрытие - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода. Сложность современного ПО и инфраструктуры сделало невыполнимой задачу проведения тестирования со 100% тестовым покрытием. Поэтому для разработки набора тестов, обеспечивающего более-менее высокий уровень покрытия можно использовать специальные инструменты либо техники тест дизайна.
Существуют следующие подходы к оценке и измерению тестового покрытия:
Покрытие требований (Requirements Coverage) - оценка покрытия тестами функциональных и нефункциональных требований к продукту путем построения матриц трассировки (traceability matrix).
Покрытие кода (Code Coverage) - оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей ПО.
Тестовое покрытие на базе анализа потока управления - это одна из техник тестирования белого ящика, основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей.
Различия:
Метод покрытия требований сосредоточен на проверке соответствия набора проводимых тестов требованиям к продукту, в то время как анализ покрытия кода - на полноте проверки тестами разработанной части продукта (исходного кода), а анализ потока управления - на прохождении путей в графе или модели выполнения тестируемых функций (Control Flow Graph).
Ограничения:
- Метод оценки покрытия кода не выявит нереализованные требования, так как работает не с конечным продуктом, а с существующим исходным кодом.
- Метод покрытия требований может оставить непроверенными некоторые участки кода, потому что не учитывает конечную реализацию.

Альтернативное мнение:
Покрытие кода — совершенно бесполезная метрика. Не существует «правильного» показателя. Это вопрос-ловушка. У вас может быть проект со 100% покрытием кода, в котором по-прежнему остаются баги и проблемы. В реальности нужно следить за другими метриками — хорошо известными показателям CTM (Codepipes testing Metrics).

PDWT (процент разработчиков, пишущих тесты) — вероятно, самый важный показатель. Нет смысла говорить об антипаттернах тестирования ПО, если у вас вообще нет тестов. Все разработчики в команде должны писать тесты. Любую новую функцию можно объявлять сделанной только если она сопровождается одним или несколькими тестами.

PBCNT (процент багов, приводящих к созданию новых тестов). Каждый баг в продакшне — отличный повод для написания нового теста, проверяющего

соответствующее исправление. Любой баг должен появиться в продакшне не более одного раза. Если ваш проект страдает от появления повторных багов даже после их первоначального «исправления», то команда действительно выиграет от использования этой метрики.

PTVB (процент тестов, которые проверяют поведение, а не реализацию). Тесно связанные тесты пожирают массу времени при рефакторинге основного кода.

PTD (процент детерминированных тестов от общего числа). Тесты должны завершаться ошибкой только в том случае, если что-то не так с бизнес-кодом. Если тесты периодически ломаются без видимой причины — это огромная проблема.

Если после прочтения о метриках вы по-прежнему настаиваете на установке жесткого показателя для покрытия кода, я дам вам число 20%. Это число должно использоваться как эмпирическое правило, основанное на законе Парето. 20% вашего кода вызывает 80% ваших ошибок

Доп. материал:
Лекция 4: Оценка оттестированности проекта: метрики и методика интегральной оценки
Что такое модель зрелости тестирования (TMM - Test Maturity Model)?
Существует определение Maturity Models, то есть модели зрелости различных процессов в организации, состоящая из 5 уровней. Нас же в рамках этого материала интересует один конкретный подвид моделей MM - модель зрелости тестирования, которая тоже состоит из 5 уровней. TMM в свою очередь основан на модели зрелости возможностей (CMM — Capability Maturity Model). Модель зрелости ПО (CMM или SW-CMM) — это модель для оценки зрелости программных процессов в организации. В ней также перечислены некоторые стандартные практики, которые увеличивают зрелость этих процессов. TMM это подробная модель для улучшения процесса тестирования. Она может быть дополнена любой моделью улучшения процесса или может использоваться как одиночная модель.
Модель TMM имеет два основных компонента:

- Набор из 5 уровней, которые определяют возможности тестирования (testing capability)
- Модель оценки (An Assessment Model)

Пять уровней TMM помогают организации определить зрелость своего процесса и определить следующие шаги по улучшению, которые необходимы для достижения более высокого уровня зрелости тестирования. Приведенный далее текст является переводом, но есть и русскоязычные на эту тему. Однако материал везде несколько отличается, поэтому представляю несколько точек зрения:
https://devsday.ru/blog/details/5427
https://www.software-testing.ru/library/around-testing/processes/3092-test-maturity-model
https://habr.com/ru/company/otus/blog/479368/

- Уровень 1. Начальный. ПО должно успешно работать.
- На этом уровне области процессов не определены.
- Цель тестирования — убедиться, что ПО работает нормально.
- На этом уровне не хватает ресурсов, инструментов и обученного персонала.
- Нет проверки качества перед поставкой ПО.

- Уровень 2. Определенный. Разработка целей и политик тестирования и отладки.
- Этот уровень отличает тестирование от отладки, и они считаются различными действиями.
- Этап тестирования наступает после кодирования.
- Основная цель тестирования — показать, что ПО соответствует спецификации.
- Основные методы и методики тестирования.
- Уровень 3: Комплексный. Интеграция тестирования в жизненный цикл ПО.
- Тестирование интегрируется в весь жизненный цикл.
- На основании требований определяются цели испытаний.
- Структура тестирования существует.
- Тестирование на уровне профессиональной деятельности.
- Уровень 4: Управление и измерение. Создание программы тестовых измерений.
- Тестирование — это измеренный и количественный процесс.
- Проверка на всех этапах разработки признается как тестирование.
- Для повторного использования и регрессионного тестирования есть Test case и они записаны в базу тестов.
- Дефекты регистрируются и получают уровни серьезности.
- Уровень 5: Оптимизированный. Оптимизация процесса тестирования.
- Тестирование управляется и определено.
- Эффективность и стоимость тестирования можно отслеживать.
- Тестирование может постоянно настраиваться и улучшаться.
- Практика контроля качества и предотвращения дефектов.
- Практикуется процесс повторного использования (Reuse).
- Метрики, связанные с тестированием, также имеют средства поддержки.
- Инструменты обеспечивают поддержку разработки тестовых наборов и сбора дефектов.

Доп. материал: 7 подходов к тестированию
Что такое тестирование со сдвигом влево? (Shift left testing)
В попытке перенести тестирование на более ранний этап жизненного цикла разработки при одновременном улучшении показателей качества, задачи смещаются влево в схеме жизненного цикла разработки ПО. По возможности, тестирование должно проводиться с самого начала фазы проектирования, чтобы построить соответствующую стратегию тестирования. Проще говоря, это подход к тестированию программного обеспечения и тестированию системы, при котором тестирование выполняется на более раннем этапе жизненного цикла. Ключевые преимущества:
- Сокращение затрат
- Более высокое качество
- Повышение эффективности
- Конкурентные преимущества

Доп. материал:
Экономим ресурсы и успеваем в срок: зачем подключать QA-инженера в начале работы над фичей
Что такое независимое тестирование? (Independent testing)
Можете ли вы доверять вердикту судьи, который является частью внутреннего круга людей, которых он должен судить? Чтобы этот процесс был справедливым, лица, принимающие решения, должны быть беспристрастными. Теперь, когда вы активно

участвуете в разработке какого-либо продукта или программного обеспечения, тестировать его с нейтральным мышлением это легче сказать, чем сделать. Как разработчик, вы бы хотели отгружать продукт в кратчайшие сроки и считать его безупречным и в конечном итоге упустите из виду некоторые ошибки. Чтобы избежать такой ситуации, иногда следует нанять независимую организацию по тестированию, которая тщательно проверит ваш продукт на наличие сбоев, готовя его к развертыванию.

Тестирование по уровням независимости:

- Программист тестирует свой код
- Тестирование проводится другим программистом в организации
- Внутренняя команда тестирования
- Независимая организация тестирования

- Когда программист проверяет свой код: Вы бы никогда не попросили шеф-повара быть его собственным критиком. И даже если вы это сделаете, вам будет трудно поверить всему, что он говорит. Смысл - создатель никогда не может быть хорошим критиком своей собственной работы. Программист знает свой код от и до. Их цель - создать продукт и отправить его в кратчайшие сроки. Вместо того, чтобы искать ошибки со всех возможных точек зрения, они будут искушены найти способы обойти найденные ошибки. Писатель Гленфорд Майерс в своей книге «Искусство тестирования программного обеспечения» перечислил разницу в мышлении разработчика и тестировщика. Он сказал, что разработчик думает как строитель, сосредоточенный на строительстве, в то время как тестировщик ищет недостатки, которые приведут к разрушению здания, если не будут решены.
- Тестирование проводится другим программистом в организации: Компромисс - это найти кого-то в организации. Это может быть какой-то другой программист, который участвует в некоторых других проектах. Это дает определенный уровень независимости. Но проблема возникает из-за того же reporting manager. Менеджер может попросить программиста пропустить некоторые тесты, когда есть ограничения по времени. Это приведет к неполному тестированию продукта. Кроме того, если попросить других разработчиков провести тестирование, это приведет к развертыванию различных ресурсов в одном проекте. Это будет вредно для всей работы организации.
- Внутренняя команда тестирования: Наличие другой внутренней команды - это хорошее решение. Но поскольку они будут в организации, на них будут влиять ограничительные сроки. Кроме того, это будет дорого поддерживать внутреннюю команду. Это приведет к большим бюджетным и ресурсным ограничениям для команды. Команда может иметь доступ к ограниченным инструментам и программному обеспечению, таким образом, не отвечая требованиям всех проектов. Среда тестирования также будет варьироваться в зависимости от количества пользователей и числа выполненных интеграций. Затем тестирование будет проводиться в спешном порядке, что приведет к упущению некоторых ошибок, которые могут появиться после выпуска продукта. Решение, которое позаботится обо всех этих недостатках, - «Независимое тестирование».
- Почему независимое тестирование? Независимые тестирующие организации изучат все аспекты вашей продукции. Они работают с мышлением поиска недостатков и ошибок. Они не будут использовать ярлыки в процессе тестирования. И поскольку они не были частью процесса разработки, они будут проводить тесты на нейтральной

основе, чтобы прежние интересы не мешали процессу тестирования. Мысль о поиске максимальных «точек останова» пойдет на пользу вашему продукту. Почти все сторонние тестирующие организации предоставят вам подробные отчеты об ошибках и предложат корректирующие меры.

В чем разница между превентивным и реактивным подходами в тестировании? (Preventative and Reactive approaches)

- Превентивный (профилактический) подход: он также известен как Verification Process. Этот подход заключается в предотвращении дефектов. При таком подходе тесты разрабатываются на ранних этапах SDLC, то есть до того, как программное обеспечение было создано. Он подпадает под анализ качества (QA).
- Реактивный подход: он также известен как Validation Process. Этот подход заключается в выявлении дефектов. При таком подходе тесты предназначены для выполнения после того, как программное обеспечение было произведено. Здесь мы пытаемся найти недостатки. Подпадает под контроль качества (QC).

Перечислите типичные возможные обязанности инженера по обеспечению качества?

- Команда QA отвечает за мониторинг всего процесса разработки.
- Они несут ответственность за отслеживание результатов каждого этапа SDLC и корректировку их в соответствии с ожиданиями.
- Они несут ответственность за чтение и понимание необходимых документов.
- Анализируют требования к тестированию, а также разрабатывают и выполняют тесты.
- Разрабатывают Test case и расставляют приоритеты в тестировании.
- Записывают проблемы и инциденты в соответствии с задачами проекта и планами управления инцидентами.
- Работают с командой приложения и/или клиентом для решения любых проблем, возникающих в процессе тестирования.
- Проводят регрессионное тестирование каждый раз, когда в код вносятся изменения для исправления дефектов.
- Должны взаимодействовать с клиентами, чтобы лучше понять требования продукта.
- Принимают участие в прохождении процедур тестирования.

Что такое аудит качества?

Аудит качества - это процесс систематической и независимой проверки программного продукта или процесса для оценки соответствия спецификациям, стандартам, соглашениям и другим соответствующим критериям.

Почему тестирование делится на отдельные этапы?

- Каждый этап испытаний имеет свое назначение
- Проще управлять поэтапно
- Мы можем запустить разные тесты в разных средах
- Производительность и качество тестирования улучшаются с помощью поэтапного тестирования

Почему невозможно полностью протестировать ПО?

- Спецификации ПО могут быть субъективными и приводить к различным интерпретациям.
- ПО может потребоваться слишком много входов, слишком много выходов и слишком много комбинаций путей для тестирования.

Как вы тестируете продукт, если требования еще не зафиксированы?

Если спецификация требований недоступна для продукта, тогда план тестирования может быть создан на основе предположений, сделанных относительно продукта. Но мы должны хорошо документировать все предположения в плане тестирования.
Как вы узнаете, было ли создано достаточно тестов для тестирования продукта?
Прежде всего, мы проверим, охватывает ли каждое требование хотя бы один Test case. Если да, то можно сказать, что тестовых примеров достаточно для тестирования продукта.
Как вы понимаете инспекцию?
Это процесс проверки на уровне группы и улучшения качества документации по продукту. Он фокусируется на следующих двух аспектах:

- Улучшение документа продукта
- Улучшение процесса (как производства документов, так и проверки)

Какие есть роли/должности в команде?

Project manager
Это человек, который берет входящие от заказчика требования (несформулированные хотелки), уточняет все и нормальным языком передает разработчикам, следит за рисками, прогрессом и доводит до финала. На нем вся коммуникация с заказчиком, согласования и т. д.
Product Owner
Может выполнять немного разные роли в разных компаниях. Человек, который является хранителем информации о продукте. Его роль быть decision maker, он отвечает со стороны бизнеса за приложение, с него будет спрашивать заказчик. Противоположные роли с ПМ, как адвокат с обвинителем. ПМ со стороны команды, пытается извертеться на плюшки, а PO со стороны заказчика выбивает все по максимуму для себя.
Knowledge manager
Управление знаниями - это о том, как распоряжаться всеми имеющимися в компании знаниями, чтобы позволять всем сотрудникам максимально быстро находить ответы на вопросы, принимать решения, избегать ошибок, придумывать что-то новое, управлять проектами, подбирать и развивать сотрудников. А значит, нужно выстраивать коммуникации между подразделениями, учить их разговаривать друг с другом. Вопросы менеджера по знаниям ведущим специалистам имеют высший приоритет, поэтому тот всегда держит руку на пульсе. Полученные знания после доставки не теряются, а превращаются в обучающие документы, которые изучают все сотрудники.

Доп. материал:
Product Owner vs Product Manager • или Product Owner/Product Manager
Продакт-менеджмент как профессия: востребованность, зарплата и другие нюансы
Кто такой продакт-менеджер? Или не все PM• ’• ы • —• • проджект-менеджеры
Knowledge management: • как перестать изобретать велосипеды
Заметки knowledge manager'a. Как работает управление знаниями в Exness
Профессия CTO
Кто такой DevOps-инженер, что он делает, сколько зарабатывает и как им стать
Гайд по DevOps для начинающих

Опишите жизненный цикл продукта по этапам - какие участники на каждом этапе, какие у них роли? Какие артефакты на каждом этапе?
Хотелось бы уточнить, что описанные этапы не являются эталонными и от компании к компании процессы могут радикально отличаться. В данном случае представлены некие «сферические процессы в вакууме» для какого-то начального понимания. Параллельно описывается SDLC и STLC.

- Анализ. Участники: Product owner, BA(бизнес-аналитик), QA. Артефакты: спецификация требований к ПО (Software Requirement Specification, SRS).

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

- Проектирование. Участники: Product owner, разработчики, системные архитекторы, QA. Артефакты: дизайн-спецификация (Design Specification Document, DSD), Тест-план, тест-сценарии, тест-кейсы; вариативно: тестовая стратегия.

На стадии проектирования (называемой также стадией дизайна и архитектуры) программисты и системные архитекторы, руководствуясь требованиями, разрабатывают высокоуровневый дизайн системы. Разнообразные технические вопросы, возникающие в процессе проектирования, обсуждаются со всеми заинтересованными сторонами. Определяются технологии, которые будут использоваться в проекте, загрузка команды, ограничения, временные рамки и бюджет. В соответствии с уточненными требованиями выбираются наиболее подходящие проектные решения. Утвержденный дизайн системы определяет перечень разрабатываемых программных компонентов, взаимодействие с третьими сторонами, функциональные характеристики программы, используемые базы данных и многое другое. QA/test analyst проектируют процесс тестирования в тест плане, руководствуясь также (если есть) политикой и стратегией тестирования. Тестировщики начинают писать сценарии и по ним кейсы для тестирования.

- Разработка. Участники: разработчики. Артефакты: -.

Системные администраторы настраивают программное окружение, frontend программисты разрабатывают пользовательский интерфейс программы и логику ее взаимодействия с сервером.
Кроме того, программисты пишут Unit-тесты для проверки правильности работы кода каждого компонента системы, проводят ревью написанного кода, создают билды и разворачивают готовое ПО в программной среде. Этот цикл повторяется до тех пор, пока все требования не будут реализованы.

- Тестирование. Участники: QA. Артефакты: дефект-репорты, сводный отчет о тестировании.

Тестовый администратор (если есть) настраивает тестовые среды/стенды для проведения тестирования. Тестировщики занимаются поиском дефектов в программном обеспечении и сравнивают описанное в требованиях поведение системы с реальным. В фазе тестирования обнаруживаются пропущенные при разработке баги. При обнаружении дефекта, тестировщик составляет отчет об ошибке, который передается разработчикам. Последние его исправляют, после чего тестирование

повторяется – но на этот раз для того, чтобы убедиться, что проблема была исправлена, и само исправление не стало причиной появления новых дефектов в продукте. По итогам проведенного процесса тестирования составляется итоговый отчет.

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

Когда программа протестирована и в ней больше не осталось серьезных дефектов, приходит время релиза и передачи ее конечным пользователям. После выпуска новой версии программы в работу включается отдел технической поддержки. Его сотрудники обеспечивают обратную связь с пользователями, их консультирование и поддержку. В случае обнаружения пользователями тех или иных пост-релизных багов, информация о них передается в виде отчетов об ошибках команде разработки, которая, в зависимости от серьезности проблемы, либо немедленно выпускает исправление (т.н. hot-fix), либо откладывает его до следующей версии программы. Кроме того, команда технической поддержки помогает собирать и систематизировать различные метрики – показатели работы программы в реальных условиях и т.п.

Кто такой SDET?

SDET (Software Development Engineer in Test) инженер по разработке ПО в тестировании - это ИТ-специалист, который может одинаково эффективно работать в сфере разработки и тестирования, и он / она принимает участие в полном процессе разработки ПО. В отличие от Quality Analyst (QA), которым желательны базовые знания в программировании, но нет необходимости писать код, SDET это делает на постоянной основе, совмещая в себе и тестировщика и разработчика.

SDET Manual Tester

Знает всю систему от начала до конца Ограниченные знания о системе

SDET участвует в каждом этапе процесса разработки ПО, как Проектирование, разработка и тестирование. QA участвует только в жизненном цикле тестирования процесса разработки ПО.

Высококвалифицированный специалист со знаниями как в разработке, так и в тестировании Тестировщик ПО участвует только в подготовке и выполнении Test case

SDET может участвовать в разработке средств автоматизации тестирования Не ожидается разработка средств автоматизации тестирования.

SDET должны выполнять такие обязанности, как тестирование производительности, автоматическое создание тестовых данных и т. д. Только задачи по ручному тестированию

Знает требования и гайдлайны для продуктов От QA таких знаний не ожидается.

Что такое тестирование как сервис? (TaaS – testing as a Service)

ТЕСТИРОВАНИЕ КАК СЕРВИС (TaaS) - это модель аутсорсинга, при которой деятельность по тестированию передается третьей стороне. Здесь тестирование проводится сторонними подрядчиками, а не сотрудниками организации. TaaS используется, когда Компании не хватает навыков или ресурсов для внутреннего тестирования или чтобы получить свежий взгляд со стороны. Чаще всего на аутсорс отдают тестирование функциональности, производительности и безопасности.

Что подразумевается под тестовой средой? (Test Environment/Test Bed)

Вообще существует несколько сред:

- Среда разработки (Development Env) – в ней разработчики пишут код, проводят отладку, исправляют ошибки, выполняют Unit-тестирование. За эту среду отвечают также разработчики.
- Среда тестирования (Test Env) – в этой среде работают тестировщики. Тут тестируются новые билды. Здесь тестировщики проверяют функционал, проводят регрессионные проверки, воспроизводят ошибки. Эта среда появляется во время начала динамического тестирования;
- Интеграционная среда (Integration Env) – иногда реализована в рамках среды тестирования, а иногда в рамках превью среды. В этой среде собрана необходимая для end-to-end тестирования схема взаимодействующих друг с другом модулей, систем, продуктов. Собственно, необходима она для интеграционного тестирования. Поддержка среды – также, как и в случае со средой тестирования
- Превью среда (Preview, Preprod Env) – в идеале, это среда идентичная или максимально приближенная к продуктивной: те же данные, то же аппаратно-программное окружение, та же производительность. Она используется, чтобы сделать финальную проверку ПО в условиях максимально приближенным к «боевым». Здесь тестировщики проводят заключительное end-to-end тестирование функционала, бизнес и/или пользователи проводят UAT, а команды поддержки L3 и L2 выполняют DryRun (пробную установку релиза). Как правило за эту среду отвечает группа L3 поддержки.
- Продакшн среда (Production Env) – среда, в которой работают пользователи. С этой средой работает команда L2 поддержки устанавливая поставки ПО или патчи с исправлениями, выполняя настройки, отвечая за работоспособность всех систем. Инциденты и проблемы требующие исправления ПО передаются в работу команде на L3

В общем случае среда тестирования - это настройка программного и аппаратного обеспечения для тестирования.
Испытательный стенд (Test Bed) – более глобальная сущность и включает в себя operating system, configuration management for the products, hardware, network topology и т. д. Настраиваются в соответствии с требованиями тестируемого приложения. В некоторых случаях испытательный стенд может представлять собой комбинацию тестовой среды и тестовых данных, которые он использует.
Настройка правильной среды тестирования гарантирует успех тестирования ПО. Любые недостатки в этом процессе могут привести к дополнительным затратам и времени для клиента. Следующие люди участвуют в настройке тестовой среды: Системные администраторы, Разработчики, Тестировщики.

Доп. материал:
Тестовая среда
STLC •—• • настройка тестовой среды
Что подразумевается под тестовыми данными?
Тестовые данные - это набор входных значений, необходимых для выполнения Test case. тестировщики определяют данные в соответствии с требованиями. Они могут сделать это вручную или использовать инструменты генерации.
Основные фазы тестирования?
- Pre-Alpha: - ПО является прототипом. Пользовательский интерфейс завершен. Но не все функции завершены. На данном этапе ПО не публикуется.

- Alpha: является ранней версией программного продукта. Цель - вовлечь клиента в процесс разработки. Хороший Альфа-тест должен иметь четко определенный план тестирования с комплексными тестовыми примерами. Это дает лучшее представление о надежности программного обеспечения на ранних стадиях. В некоторых случаях тестирование может быть передано на аутсорс.
- Beta: ПО стабильно и выпускается для ограниченной пользовательской базы. Цель состоит в том, чтобы получить отзывы клиентов о продукте и внести соответствующие изменения в ПО.
- Release Candidate (RC): основываясь на отзывах Beta Test, вы вносите изменения в ПО и хотите проверить исправления ошибок. На этом этапе вы не хотите вносить радикальные изменения в функциональность, а просто проверяете наличие ошибок. RC также выпущен для общественности
- Release: Все работает, ПО выпущено для общественности.

Подробнее про бета-тестирование?

Бета-тестирование происходит на конечных пользователях. Это нужно для обеспечения обратной связи.

Существуют различные типы бета-тестов в тестировании ПО, и они заключаются в следующем:

- Традиционное бета-тестирование: продукт распространяется на целевой рынок, и соответствующие данные собираются по всем аспектам. Эти данные могут быть использованы для улучшения продукта.
- Публичное бета-тестирование: продукт публикуется во внешнем мире через онлайн-каналы, и данные могут быть получены от любого пользователя. На основе обратной связи могут быть сделаны улучшения продукта.
- Техническое бета-тестирование: продукт передается во внутреннюю группу организации и собирает отзывы / данные от сотрудников организации.
- Целевая бета-версия: продукт выпущен на рынок для сбора отзывов об особенностях программы.
- Бета-версия после выпуска. Продукт выпущен на рынок, и данные собираются для внесения улучшений в будущем выпуске продукта.

Что означает пилотное тестирование? (Pilot)

PILOT testing определяется как тип тестирования программного обеспечения, который проверяет компонент системы или всю систему в режиме реального времени. Целью пилотного теста является оценка осуществимости, времени, стоимости, риска и эффективности исследовательского проекта. Это тестирование проводится точно между UAT и Production. В пилотном тестировании выбранная группа конечных пользователей пробует тестируемую систему и предоставляет обратную связь до полного развертывания системы. Другими словами, это означает проведение генеральной репетиции для последующего теста на удобство использования. Пилотное тестирование помогает в раннем обнаружении ошибок в Системе. Пилотное тестирование связано с установкой системы на площадке заказчика (или в среде, моделируемой пользователем) для тестирования на предмет постоянного и регулярного использования. Выявленные недостатки затем отправляются команде разработчиков в виде отчетов об ошибках, и эти ошибки исправляются в следующей сборке системы. Во время этого процесса иногда приемочное тестирование также включается как часть тестирования на совместимость. Это происходит, когда система разрабатывается для замены старой.

В чем отличие build от release?

Билд это номер, даваемый ПО при передаче от разработчиков тестировщикам. Релиз — это номер, даваемый ПО при передаче конечному пользователю.
Что такое бизнес – логика (domain)?
Бизнес – логика (domain) это то, что конкретная программа по задумке должна сделать. Например, в складской программе проверка на возможность отправить товар (вдруг его нет в наличии). Это правила, которые должны соблюдаться в данной конкретной программе, определенные бизнес-клиентом. Слои приложения – слой пользовательского интерфейса, слой бизнес логики, слой сохранения данных.

----- Виды тестирования -----
Какие существуют основные виды тестирования ПО?

- Функциональные виды («Что?» - проверяет весь функционал продукта):
- Функциональное тестирование (Functional testing)
- Тестирование взаимодействия (Interoperability testing)
- Нефункциональное («Как?»):
- Производительности (Performance)
- Тестирование емкости/способностей (Capacity testing)
- Стрессовое (Stress testing)
- Нагрузочное (Load testing)
- Объемное тестирование (Volume testing)
- Выносливости (Soak/Endurance testing)
- Стабильности/надежности (Stability / Reliability testing)
- Шиповое (Spike)
- Отказоустойчивости (Stability testing)
- Масштабируемости (Scalability test)
- Отказ и восстановление (Failover and Recovery testing)
- Удобство пользования (Usability testing)
- Тестирование установки (Installation testing)
- Тестирование безопасности (Security and Access Control testing)
- Конфигурационное (Configuration testing)
- Связанное с изменениями:
- Регрессионное (Regression testing)
- Санитарное (Sanity testing)
- Дымовое (Smoke testing)
- Тестирование сборки (Build Verification testing)

В разных источниках предлагается разный взгляд:

Типы тестирования? (White/Black/Grey Box)
Самым высоким уровнем в иерархии подходов к тестированию будет понятие типа, которое может охватывать сразу несколько смежных техник тестирования. То есть, одному типу тестирования может соответствовать несколько его видов. Отличаются они знанием внутреннего устройства объекта тестирования.
Что означает тестирование черного ящика?
Summary: Мы не знаем, как устроена внутри тестируемая система.
Тестирование методом «черного ящика», также известное как тестирование, основанное на спецификации или тестирование поведения – техника тестирования, основанная на работе исключительно с внешними интерфейсами тестируемой системы.
– тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы.
– тест-дизайн, основанный на технике черного ящика – процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.
Почему именно «черный ящик»? Тестируемая программа для тестировщика – как черный непрозрачный ящик, содержания которого он не видит. Целью этой техники является поиск ошибок в таких категориях:
– неправильно реализованные или недостающие функции;
– ошибки интерфейса;
– ошибки в структурах данных или организации доступа к внешним базам данных;
– ошибки поведения или недостаточная производительности системы;
Таким образом, мы не имеем представления о структуре и внутреннем устройстве системы. Нужно концентрироваться на том, ЧТО программа делает, а не на том, КАК она это делает.
Пример:
Тестировщик проводит тестирование веб-сайта, не зная особенностей его реализации, используя только предусмотренные разработчиком поля ввода и кнопки. Источник ожидаемого результата – спецификация.
Поскольку это тип тестирования, по определению он может включать другие его виды. Тестирование черного ящика может быть как функциональным, так и нефункциональным. Функциональное тестирование предполагает проверку работы функций системы, а нефункциональное – соответственно, общие характеристики нашей программы.
Техника черного ящика применима на всех уровнях тестирования (от модульного до приемочного), для которых существует спецификация. Например, при осуществлении системного или интеграционного тестирования, требования или функциональная спецификация будут основой для написания тест-кейсов.
Техники тест-дизайна, основанные на использования черного ящика, включают:
– классы эквивалентности;
– анализ граничных значений;
– таблицы решений;
– диаграммы изменения состояния;

– тестирование всех пар.
Преимущества:
– тестирование производится с позиции конечного пользователя и может помочь обнаружить неточности и противоречия в спецификации;
– тестировщику нет необходимости знать языки программирования и углубляться в особенности реализации программы;
– тестирование может производиться специалистами, независимыми от отдела разработки, что помогает избежать предвзятого отношения;
– можно начинать писать тест-кейсы, как только готова спецификация.
Недостатки:
– тестируется только очень ограниченное количество путей выполнения программы;
– без четкой спецификации (а это скорее реальность на многих проектах) достаточно трудно составить эффективные тест-кейсы;
– некоторые тесты могут оказаться избыточными, если они уже были проведены разработчиком на уровне модульного тестирования;
Противоположностью техники черного ящика является тестирование методом белого ящика, речь о котором пойдет ниже.
Что означает тестирование белого ящика?
Тестирование методом белого ящика (также: прозрачного, открытого, стеклянного ящика; основанное на коде или структурное тестирование) – метод тестирования ПО, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику. Мы выбираем входные значения, основываясь на знании кода, который будет их обрабатывать. Точно так же мы знаем, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование белого ящика – углубление во внутреннее устройство системы, за пределы ее внешних интерфейсов.
Техника белого ящика применима на разных уровнях тестирования – от модульного до системного, но главным образом применяется именно для реализации модульного тестирования компонента его автором.
Преимущества:
– тестирование может производиться на ранних этапах: нет необходимости ждать создания пользовательского интерфейса;
– можно провести более тщательное тестирование, с покрытием большого количества путей выполнения программы.
Недостатки:
– для выполнения тестирования белого ящика необходимо большое количество специальных знаний
Основным методом тестирования Белого ящика является анализ покрытия кода. Анализ покрытия кода устраняет пробелы в наборе тестовых примеров. Он определяет области программы, которые не покрываются набором Test case. После выявления пробелов вы создаете контрольные примеры для проверки непроверенных частей кода, тем самым повышая качество программного продукта.

- Охват операторов: - Этот метод требует, чтобы каждое возможное утверждение в коде было проверено хотя бы один раз в процессе тестирования разработки ПО.
- Покрытие ветвления - этот метод проверяет все возможные пути (если-еще и другие условные циклы) программного приложения.

Помимо вышесказанного, существует множество типов покрытия, таких как покрытие условий, покрытие нескольких условий, покрытие пути, покрытие функций и т. д.

Каждый метод имеет свои достоинства и пытается протестировать (охватить) все части программного кода. Используя покрытие Statement и Branch, вы обычно достигаете 80-90% покрытия кода, что является достаточным.
Что означает тестирование серого ящика? (Grey box)
Тестирование методом серого ящика – метод тестирования ПО, 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. Либо нам не доступна внутренняя реализация функций, но мы знаем на уровень ниже, чем пользователи – интерфейсы/эндпоинты и т.п.
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.
Пример тестирования «серого ящика»: при тестировании веб-сайтов на битые ссылки, если тестировщик сталкивается с какой-либо проблемой с этими ссылками, он может сразу же внести изменения в HTML-код и проверить в реальном времени.
Методы:

- Матричное тестирование: этот метод тестирования включает в себя определение всех переменных, которые существуют в их программах.
- Регрессионное тестирование: чтобы проверить, повлияло ли изменение в предыдущей версии другие аспекты программы в новой версии.
- Тестирование ортогональных массивов или OAT: обеспечивает максимальное покрытие кода с минимальным количеством тестов.
- Pattern testing: это тестирование выполняется на данных истории предыдущих дефектов системы. В отличие от тестирования черного ящика, в тестировании серого ящика копаются в коде и определяют причину сбоя.

Основные отличия White/grey/black box?

Что такое пирамида / уровни тестирования? (Testing Levels)
«Пирамида тестов» — метафора, которая означает группировку тестов программного обеспечения по разным уровням детализации. Она также дает представление, какое количество тестов должно быть в каждой из этих групп.

… В тесте более высокого уровня вы не тестируете всю условную логику и пограничные случаи, которые уже покрыты юнит-тестами более низкого уровня. Убедитесь, что тест высокого уровня фокусируется только на том, что не покрыто тестами более низкого уровня.
Правило трех A(AAA) (arrange, act, assert) или триада «дано, когда, тогда» — хорошая мнемоника, чтобы поддерживать хорошую структуру тестов.

Доп. материал:
Пирамида тестов на практике
Антипаттерны тестирования ПО
Unit, API • и GUI тесты • —• • чем отличаются
Почему тестировать должны не только QA. Распределяем тест-кейсы между Dev, Analyst и QA

The Practical Test Pyramid
Test Pyramid
Что такое деструктивное/разрушающее/негативное тестирование? (DT - Destructive testing)
ОТРИЦАТЕЛЬНОЕ ТЕСТИРОВАНИЕ - тип тестирования ПО для поиска точек отказа в программном обеспечении, который проверяет систему на обработку исключительных ситуаций (срабатывание валидаторов на некорректные данные), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора. Неожиданные условия могут быть чем угодно, от неправильного типа данных до хакерской атаки. Целью отрицательного тестирования является предотвращение сбоя приложений из-за некорректных входных данных. Просто проводя положительное тестирование, мы можем только убедиться, что наша система работает в нормальных условиях. Мы должны убедиться, что наша система может справиться с непредвиденными условиями, чтобы обеспечить 100% безошибочную систему.
Типичные примеры: ввести неправильно составленный e-mail и номер телефона, загрузить файл не предусмотренного расширения или размера.
Для деструктивного тестирования существует множество способов его тестирования:

- Метод анализа точек отказа: это пошаговое прохождение системы, проводящее оценку того, что может пойти не так в разных точках. Для этой стратегии может быть использована помощь BA (Business Analyst).
- Экспертная проверка тестировщика: проанализируйте или дайте на ревью ваши Test вашему коллеге-тестировщику, который менее знаком с системой/функцией
- Бизнес-анализ тестовых случаев. Конечные пользователи или эксперты могут подумать о многих допустимых сценариях, которые иногда тестировщики могут не учитывать или упустить, так как все их внимание будет сосредоточено на тестировании требований.
- Проведите предварительное тестирование с использованием контрольных таблиц (run sheets). Исследовательское тестирование с использованием контрольных таблиц поможет определить, что было проверено, повторить тесты и позволит вам контролировать охват тестами.
- Используйте другой источник: вы можете попросить кого-нибудь сломать программный продукт и проанализировать различные сценарии.

Доп. материал:
Топ 10 негативных кейсов
Что такое недеструктивное/неразрушающее/позитивное тестирование? (NDT – Non Destructive testing)
НЕДЕСТРУКТИВНОЕ ТЕСТИРОВАНИЕ - это тип тестирования программного обеспечения, который включает в себя правильное взаимодействие с программным обеспечением. Другими словами, неразрушающее тестирование (NDT) также можно назвать позитивным тестированием или тестированием «счастливого пути». Это дает ожидаемые результаты и доказывает, что программное обеспечение ведет себя так, как ожидалось. Пример: - Ввод правильных данных в модуль входа в систему и проверка, принимает ли он учетные данные и переходит на следующую страницу
Что подразумевается под компонентным/модульным/юнит тестированием? (Component/Module/Unit testing)

С этими терминами происходит путаница и даже глоссарий ISTQB не проясняет ситуацию. Обычно эти термины используют как синонимы, а конкретика варьируется от компании к компании. Но если копнуть и попробовать разобраться, получается примерно следующее:

Модульное тестирование (юнит-тестирование). Модульные тесты используются для тестирования какого-либо одного логически выделенного и изолированного элемента системы (отдельные методы класса или простая функция, subprograms, subroutines, классы или процедуры) в коде. Очевидно, что это тестирование методом белого ящика и чаще всего оно проводится самими разработчиками. Целью тестирования модуля является не демонстрация правильного функционирования модуля, а демонстрация наличия ошибки в модуле, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне модульного тестирования и выявляются на более поздних стадиях тестирования. Изоляция тестируемого блока достигается с помощью заглушек (stubs), манекенов (dummies) и макетов (mockups). Являясь по способу исполнения структурным тестированием или тестированием "белого ящика", модульное тестирование характеризуется степенью, в которой тесты выполняют или покрывают логику программы (исходный текст). Тесты, связанные со структурным тестированием, строятся по следующим принципам:

- На основе анализа потока управления. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на основе структурных критериев тестирования C0, C1,C2. К ним относятся вершины, дуги, пути управляющего графа программы (УГП), условия, комбинации условий и т. п.
- На основе анализа потока данных, когда элементы, которые должны быть покрыты, определяются при помощи потока данных, т. е. информационного графа программы.

Тестирование на основе потока управления. Особенности использования структурных критериев тестирования C0,C1,C2 были рассмотрены ранее. К ним следует добавить критерий покрытия условий, заключающийся в покрытии всех логических (булевских) условий в программе. Критерии покрытия решений (ветвей - C1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

К популярным критериям относятся критерий покрытия функций программы, согласно которому каждая функция программы должна быть вызвана хотя бы один раз, и критерий покрытия вызовов, согласно которому каждый вызов каждой функции в программе должен быть осуществлен хотя бы один раз. Критерий покрытия вызовов известен также как критерий покрытия пар вызовов (call pair coverage).

Тестирование на основе потока данных. Этот вид тестирования направлен на выявление ссылок на неинициализированные переменные и избыточные присваивания (аномалий потока данных). Предложенная там стратегия требовала тестирования всех взаимосвязей, включающих в себя ссылку (использование) и определение переменной, на которую указывает ссылка (т. е. требуется покрытие дуг

информационного графа программы). Недостаток стратегии в том, что она не включает критерий C1, и не гарантирует покрытия решений.
Стратегия требуемых пар также тестирует упомянутые взаимосвязи. Использование переменной в предикате дублируется в соответствии с числом выходов решения, и каждая из таких требуемых взаимосвязей должна быть протестирована. К популярным критериям принадлежит критерий CP, заключающийся в покрытии всех таких пар дуг v и w, что из дуги v достижима дуга w, поскольку именно на дуге может произойти потеря значения переменной, которая в дальнейшем уже не должна использоваться. Для "покрытия" еще одного популярного критерия Cdu достаточно тестировать пары (вершина, дуга), поскольку определение переменной происходит в вершине УГП, а ее использование - на дугах, исходящих из решений, или в вычислительных вершинах.
Методы проектирования тестовых путей для достижения заданной степени тестированности в структурном тестировании. Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

- Конструирование УГП.
- Выбор тестовых путей.
- Генерация тестов, соответствующих тестовым путям.

Первая фаза соответствует статическому анализу программы, задача которого состоит в получении графа программы и зависящего от него и от критерия тестирования множества элементов, которые необходимо покрыть тестами. На третьей фазе по известным путям тестирования осуществляется поиск подходящих тестов, реализующих прохождение этих путей. Вторая фаза обеспечивает выбор тестовых путей. Выделяют три подхода к построению тестовых путей:

- Статические методы.
- Динамические методы.
- Методы реализуемых путей.

Статические методы. Самое простое и легко реализуемое решение - построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута выходная вершина управляющего графа программы. Эта идея может быть усилена в так называемых адаптивных методах, которые каждый раз добавляют только один тестовый путь (входной тест), используя предыдущие пути (тесты) как руководство для выбора последующих путей в соответствии с некоторой стратегией. Чаще всего адаптивные стратегии применяются по отношению к критерию C1. Основной недостаток статических методов заключается в том, что не учитывается возможная нереализуемость построенных путей тестирования.
Динамические методы. Такие методы предполагают построение полной системы тестов, удовлетворяющих заданному критерию, путем одновременного решения задачи построения покрывающего множества путей и тестовых данных. При этом можно автоматически учитывать реализуемость или нереализуемость ранее рассмотренных путей или их частей. Основной идеей динамических методов является подсоединение к начальным реализуемым отрезкам путей дальнейших их частей так, чтобы: 1) не терять при этом реализуемости вновь полученных путей; 2) покрыть требуемые элементы структуры программы.
Методы реализуемых путей. Данная методика заключается в выделении из множества путей подмножества всех реализуемых путей. После чего покрывающее множество путей строится из полученного подмножества реализуемых путей.
Достоинство статических методов состоит в сравнительно небольшом количестве необходимых ресурсов, как при использовании, так и при разработке. Однако их

реализация может содержать непредсказуемый процент брака (нереализуемых путей). Кроме того, в этих системах переход от покрывающего множества путей к полной системе тестов пользователь должен осуществить вручную, а эта работа достаточно трудоемкая. Динамические методы требуют значительно больших ресурсов как при разработке, так и при эксплуатации, однако увеличение затрат происходит, в основном, за счет разработки и эксплуатации аппарата определения реализуемости пути (символический интерпретатор, решатель неравенств). Достоинство этих методов заключается в том, что их продукция имеет некоторый качественный уровень - реализуемость путей. Методы реализуемых путей дают самый лучший результат.
Компонентное тестирование — тип тестирования ПО, при котором тестирование выполняется для каждого отдельного компонента отдельно, без интеграции с другими компонентами. Его также называют модульным тестированием (Module testing), если рассматривать его с точки зрения архитектуры. Как правило, любое программное обеспечение в целом состоит из нескольких компонентов. Тестирование на уровне компонентов (Component Level testing) имеет дело с тестированием этих компонентов индивидуально. Это один из самых частых типов тестирования черного ящика, который проводится командой QA. Для каждого из этих компонентов будет определен сценарий тестирования, который затем будет приведен к Test case высокого уровня -> детальным Test case низкого уровня с предварительными условиями.
Исходя из глубины уровней тестирования, компонентное тестирование можно классифицировать как:

- Тестирование компонентов в малом (CTIS — Component testing In Small). Тестирование компонентов может проводиться с или без изоляции остальных компонентов в тестируемом программном обеспечении или приложении. Если это выполняется с изоляцией другого компонента, то это называется CTIS. Пример: веб-сайт, на котором есть 5 разных веб-страниц, тестирование каждой веб-страницы отдельно и с изоляцией других компонентов.
- Тестирование компонентов в целом (CTIL — Component testing In Large). Тестирование компонентов, выполненное без изоляции других компонентов в тестируемом программном обеспечении или приложении, называется CTIL. Давайте рассмотрим пример, чтобы понять это лучше. Предположим, что есть приложение, состоящее из трех компонентов, таких как Компонент A, Компонент B и Компонент C. Разработчик разработал компонент B и хочет его протестировать. Но для того, чтобы полностью протестировать компонент B, некоторые его функции зависят от компонента A, а некоторые — от компонента C. Функциональный поток: A -> B -> C, что означает, что существует зависимость от B как от A, так и от C, заглушка - вызываемая функция, а драйвер - вызывающая функция. Но компонент A и компонент C еще не разработаны. В этом случае, чтобы полностью протестировать компонент B, мы можем заменить компонент A и компонент C заглушкой и драйверами по мере необходимости. Таким образом, в основном, компоненты A & C заменяются заглушками и драйверами, которые действуют как фиктивные объекты до тех пор, пока они фактически не будут разработаны.

Unit testing Component testing
Тестирование отдельных программ, модулей, функций для демонстрации того, что программа выполняется согласно спецификацииТестирование каждого объекта или частей программного обеспечения отдельно с или без изоляции других объектов

Проверка в(на) соответствии с design documents Проверка в(на) соответствии с test requirements, use case
Пишутся и выполняются(обычно) разработчиками Тестировщиками
Выполняется первым Выполняется после Unit

Другой источник:
Разница между компонентным и модульным тестированием: По-существу эти уровни тестирования представляют одно и тоже, разница лишь в том, что в компонентном тестировании в качестве параметров функций используют реальные объекты и драйверы, а в модульном тестировании - конкретные значения.

Доп. материал:
Я сомневался в юнит-тестах, но• …
Юнит-тесты переоценены
Что подразумевается под интеграционным тестированием? (Integration testing)
Интеграционное тестирование предназначено для проверки насколько хорошо два или более модулей ПО взаимодействуют друг с другом, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами). С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки (Stub) на месте отсутствующих модулей. Основная разница между модульным и интеграционным тестированием состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов, или анализ использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой. Больше:
https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=1
Уровни интеграционного тестирования:
- Компонентный интеграционный уровень (Component Integration testing): Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.
- Системный интеграционный уровень (System Integration testing): Проверяется взаимодействие между разными системами после проведения системного тестирования.

Подходы к интеграционному тестированию:
- Подход Большого взрыва:
- Инкрементальный подход:
- Нисходящий подход
- Подход снизу-вверх
- Сэндвич-подход

Некоторые утверждают, что всех участников (например, вызываемые классы) тестируемого субъекта следует заменить на имитации (mocks) или заглушки (stubs), чтобы создать идеальную изоляцию, избежать побочных эффектов и сложной настройки теста. Другие утверждают, что на имитации и заглушки следует заменять только участников, которые замедляют тест или проявляют сильные побочные

эффекты (например, классы с доступом к БД или сетевыми вызовами). Иногда эти два вида юнит-тестов называют одинокими (solitary) в случае тотального применения имитаций и заглушек или общительными (sociable) в случае реальных коммуникаций с другими участниками.
Информация должна приходить в течение нескольких секунд или нескольких минут с быстрых тестов на ранних этапах конвейера. И наоборот, более длительные тесты — обычно с более широкой областью — размещаются на более поздних этапах, чтобы не тормозить фидбек от быстрых тестов. Как видите, этапы конвейера развертывания определяются не типами тестов, а их скоростью и областью действия. Поэтому очень разумно может быть разместить некоторые из самых узких и быстрых интеграционных тестов на ту же стадию, что и юнит-тесты — просто потому что они дают более быструю обратную связь
Доп. материал:
Интеграционные тесты в микросервисах
Лекция • 6• : Интеграционное тестирование и его особенности для объектно-ориентированного программирования
Разница между Unit testing и Integration testing?
Именно здесь больше всего споров о названиях. «Область» интеграционных тестов также весьма противоречива, особенно по характеру доступа к приложению (тестирование в черном или белом ящике; разрешены mock-объекты или нет). На практике получается так: если тест…

- использует базу данных,
- использует сеть для вызова другого компонента/приложения,
- использует внешнюю систему (например, очередь или почтовый сервер),
- читает/записывает файлы или выполняет другие операции ввода-вывода,
- полагается не на исходный код, а на бинарник приложения,

… то это интеграционный, а не модульный тест

Подведем итог: хотя теоретически можно использовать только интеграционные тесты, на практике

- Юнит-тесты легче поддерживать.
- Юнит-тесты легко воспроизводят пограничные случаи и редкие ситуации.
- Юнит-тесты выполняются гораздо быстрее интеграционных тестов.
- Сбойные юнит-тесты легче исправить, чем интеграционные.

Если у вас есть только интеграционные тесты, то вы впустую тратите и время разработки, и деньги компании. Нужны как модульные, так и интеграционные тесты одновременно. Они не взаимоисключающие.
Что такое системное интеграционное тестирование? (SIT - System Integration testing)
Это тип тестирования программного обеспечения, проводимого в интегрированной аппаратной и программной среде для проверки поведения всей системы. Это тестирование, проведенное на полной интегрированной системе для оценки соответствия системы ее установленным требованиям. SIT выполняется для проверки взаимодействия между модулями программной системы. Оно занимается проверкой требований к программному обеспечению высокого и низкого уровня, указанных в Software Requirements Specification/Data and the Software Design Document. Он также проверяет сосуществование программной системы с другими и тестирует интерфейс между модулями программного приложения. В этом типе тестирования модули

сначала тестируются индивидуально, а затем объединяются в систему. Например, программные и / или аппаратные компоненты объединяются и тестируются постепенно, пока не будет интегрирована вся система.
Что подразумевается под инкрементальным подходом? (Incremental Approach)
При таком подходе тестирование выполняется путем объединения двух или более логически связанных модулей. Затем другие связанные модули поэтапно добавляются и тестируются для правильного функционирования. Процесс продолжается до тех пор, пока все модули не будут соединены и успешно протестированы. Осуществляется двумя разными методами: Снизу-вверх и сверху-вниз.
Что подразумевается под подходом снизу-вверх? (Bottom-Up Approach)
В восходящей стратегии каждый модуль на более низких уровнях последовательно тестируется с более высокоуровневыми модулями, пока не будут протестированы все модули. Требуется помощь драйверов для тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения.
Преимущества: Локализация ошибок проще. Не тратится время на ожидание разработки всех модулей, в отличие от подхода Большого взрыва
Недостатки: Критические модули (на верхнем уровне архитектуры ПО), которые контролируют поток приложения, тестируются последними и могут быть подвержены дефектам. Ранний прототип невозможен.

Что подразумевается под подходом сверху-вниз? (Top-Down Approach)
Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами.
Преимущества: Локализация неисправностей проще. Возможность получить ранний прототип. Основные недостатки дизайна могут быть найдены и исправлены в первую очередь. Недостатки: Нужно много заглушек. Модули на более низком уровне тестируются недостаточно.

Что подразумевается под гибридным/сэндвич-подходом? (Sandwich Approach)
Представляет собой комбинацию подходов сверху вниз и снизу-вверх. Здесь верхние модули тестируются с нижними модулями, а нижние модули интегрируются с верхними модулями и тестируются. Эта стратегия использует и заглушки и драйверы.

Что подразумевается под подходом Большого взрыва? (Big Bang Approach)
Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если Test case и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования
В чем разница между тест-драйвером и тест-заглушкой? (Test Driver and Test Stub)
Тестовый драйвер - это фрагмент кода, который вызывает тестируемый программный компонент. Это полезно при тестировании по принципу «снизу-вверх». Тестовая заглушка - это фиктивная программа, которая интегрируется с приложением для

полной функциональности. Они актуальны для тестирования, в котором используется нисходящий подход. Давайте возьмем пример.
1. Допустим, есть сценарий для проверки интерфейса между модулями A и B. Мы разработали только модуль-A. Затем мы можем проверить модуль-A, если у нас есть реальный модуль-B или фиктивный модуль для него. В этом случае мы называем модуль-B тестовой заглушкой.
2. Теперь модуль B не может отправлять или получать данные напрямую из модуля A. В таком сценарии мы перемещаем данные из одного модуля в другой, используя некоторые внешние функции, называемые Test Driver.
Заглушки и драйверы не реализуют всю логику программного модуля, а только моделируют обмен данными с вызывающим модулем. Заглушка: вызывается тестируемым модулем. Драйвер: вызывает модуль для тестирования.
Что подразумевается под системным тестированием?
Системное тестирование качественно отличается от интеграционного и модульного уровней. Системное тестирование рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов, в отличие от последних фаз интеграционного тестирования, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций. Основная задача системного тестирования - в выявлении дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и тому подобное.
Системное тестирование производится над проектом в целом с помощью метода "черного ящика". Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю.
Категории тестов системного тестирования:
- Полнота решения функциональных задач.
- Стрессовое тестирование - на предельных объемах нагрузки входного потока.
- Корректность использования ресурсов (утечка памяти, возврат ресурсов).
- Оценка производительности.
- Эффективность защиты от искажения данных и некорректных действий.
- Проверка инсталляции и конфигурации на разных платформах.
- Корректность документации

Для минимизации рисков, связанных с особенностями поведения системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи. Системное тестирование относят к черному ящику.
Можно выделить два подхода к системному тестированию:
- на базе требований (requirements based): Для каждого требования пишутся • Test case, проверяющие выполнение данного требования.
- на базе случаев использования (use case based): На основе представления о способах использования продукта создаются случаи использования системы (Use Cases). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся • Test case, которые должны быть протестированы.

Доп. материал:
Лекция 7: Разновидности тестирования: системное и регрессионное тестирование

Можем ли мы провести системное тестирование на любом этапе?
Нет. Системное тестирование следует начинать, только если все модули написаны и работают правильно. Тем не менее, это должно произойти до UAT (пользовательского приемочного тестирования).
Что такое функциональное тестирование?
Функциональное тестирование рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.
Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех уровнях тестирования (компонентном, интеграционном, системном, приемочном). Как правило, эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (use cases).
Тестирование в перспективе «требования» использует спецификацию функциональных требований к системе как основу для дизайна Test case. В этом случае необходимо сделать список того, что будет тестироваться, а что нет, приоритезировать требования на основе рисков (если это не сделано в документе с требованиями), а на основе этого приоритезировать тестовые сценарии. Это позволит сфокусироваться и не упустить при тестировании наиболее важный функционал.
Тестирование в перспективе «бизнес-процессы» использует знание этих самых бизнес-процессов, которые описывают сценарии ежедневного использования системы. В этой перспективе тестовые сценарии (test scripts), как правило, основываются на случаях использования системы (use cases).
Преимущества функционального тестирования:
- имитирует фактическое использование системы;

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

Что такое тестирование совместимости/взаимодействия? (Compatibility/Interoperability testing)
Тестирование взаимодействия - функциональное тестирование, проверяющее способность приложения/устройства взаимодействовать с одним и более компонентами/системами/устройствами и включающее в себя тестирование совместимости (compatibility testing) и интеграционное тестирование (integration testing).
ПО с хорошими характеристиками взаимодействия может быть легко интегрировано с другими системами, не требуя каких-либо серьезных модификаций. В этом случае, количество изменений и время, требуемое на их выполнение, могут быть использованы для измерения возможности взаимодействия. Например, тестирование совместимости проводится между смартфонами и планшетами для проверки передачи данных через Bluetooth.
Существуют разные уровни тестирования совместимости:
- Аппаратное обеспечение: проверяет совместимость программного обеспечения с различными аппаратными конфигурациями.

- Операционные системы: Он проверяет ваше программное обеспечение на совместимость с различными операционными системами, такими как Windows, Unix*, Mac OS и т. д.
- Программное обеспечение: проверяет ваше разработанное программное обеспечение на совместимость с другим программным обеспечением. Например, приложение MS Word должно быть совместимо с другими программами, такими как MS Outlook, MS Excel, VBA и т. д.
- Сеть: оценка производительности системы в сети с различными параметрами, такими как пропускная способность, скорость работы, емкость.
- Браузер: проверяет совместимость вашего сайта с различными браузерами, такими как Firefox, Google Chrome, Internet Explorer и т. д.
- Устройства: проверяет совместимость вашего программного обеспечения с различными устройствами, такими как устройства USB-порта, принтеры и сканеры, другие мультимедийные устройства и Bluetooth.
- Mobile: проверка совместимости вашего программного обеспечения с мобильными платформами, такими как Android, iOS и т. д.
- Версии программного обеспечения. Он проверяет совместимость вашего программного приложения с различными версиями программного обеспечения. Например, проверка вашего Microsoft Word на совместимость с Windows 7, Windows 7 SP1, Windows 7 SP2, Windows 7 SP3.

Существует два типа проверки версий:

- Тестирование обратной совместимости предназначено для проверки поведения разработанного аппаратного / программного обеспечения с использованием более старых версий аппаратного / программного обеспечения.
- Тестирование прямой совместимости заключается в проверке поведения разработанного аппаратного / программного обеспечения с использованием более новых версий аппаратного / программного обеспечения.

Пример тестирования взаимодействия:

- Подключите (connect) два или более устройств от разных производителей
- Проверьте связь между устройствами
- Проверьте, может ли устройство отправлять / получать пакеты или фреймы друг от друга
- Проверьте, правильно ли обрабатываются данные на уровне сети и объектов
- Проверьте, правильно ли работают реализованные алгоритмы
- Результат в порядке: проверьте следующий результат. Результат не в порядке: используйте инструменты мониторинга, чтобы обнаружить источник ошибки
- Отчет о результатах в тестовом отчете.

Что такое тестирование на соответствие? (Conformance/Compilance testing)

CONFORMANCE testing - это тип тестирования программного обеспечения, который удостоверяет, что система программного обеспечения соответствует стандартам и правилам, определенным IEEE, W3C или ETSI. Цель проверки соответствия состоит в том, чтобы определить, в какой степени отдельная реализация конкретного стандарта соответствует индивидуальным требованиям этого стандарта. Включает в себя:

- Производительность
- Функции
- Прочность (Robustness)
- Совместимость (Interoperability)
- Поведение системы

Тестирование соответствия может быть логическим или физическим, и оно включает в себя следующие типы тестирования:

- Тестирование на соответствие (Compliance testing)
- Нагрузочное тестирование (• Load testing)
- Стресс тестирование (• Stress testing)
- Объемное тестирование (Volume testing)

Conformance testing Compliance testing
Conformance является формальным и точным способом тестирования стандартов
Compliance является неформальным и менее точным способом тестирования стандартов
Сертификация Conformance применима только к операционной системе, имеющей официальный Certification Authority Операционная система, которая обеспечивает единый API (Portable Operating System Interface), считается совместимой
Conformance testing используется для тестирования системы, которая обеспечивает полную поддержку данных стандартов Compliance testing используется для тестирования системы, обеспечивающей поддержку некоторых из указанных стандартов
Тестирование соответствия также называется Type testing, который является формальным способом тестирования.
Что такое нефункциональное тестирование?
НЕФУНКЦИОНАЛЬНОЕ тестирование определяется как тип тестирования ПО для проверки нефункциональных аспектов ПО. Оно предназначено для проверки готовности системы по нефункциональным параметрам, которые никогда не учитываются при функциональном тестировании.

- Нефункциональное тестирование должно повысить удобство использования, эффективность, ремонтопригодность и portability продукта.
- Помогает снизить производственный риск и затраты, связанные с нефункциональными аспектами продукта.

Позволяет:

- оптимизировать способ установки, настройки, выполнения, управления и мониторинга продукта.
- Собирать и производить измерения и метрики для внутренних исследований и разработок.
- Улучшать и расширять знания о поведении продукта и используемых технологиях.

Основные нефункциональные типы тестирования:

- Производительности (Performance)
- Стрессовое (Stress testing)
- Тестирование емкости/способностей (Capacity testing)
- Нагрузочное (Load testing)
- Объемное тестирование (Volume testing)
- Выносливости/стабильности/надежности (Soak/Endurance/Stability/Reliability testing)
- Шиповое (Spike)
- Масштабируемости (Scalability Test)
- Тестирование времени отклика (Response Time testing)

- Тестирование на отказоустойчивость (Failover testing)
- Тестирование совместимости (Compatibility testing)
- Тестирование на удобство пользования (Usability testing)
- Тестирование на поддерживаемость/ремонтопригодность (Maintainability testing)
- Тестирование безопасности (Security testing)
- Тестирование аварийного восстановления (Disaster Recovery testing)
- Тестирование на соответствие (Compliance testing)
- Тестирование переносимости (Portability testing)
- Тестирование эффективности (Efficiency testing)
- Базовое тестирование (Baseline testing)
- Тестирование документации (Documentation testing)
- Тестирование восстановления (Recovery testing)
- Интернационализация (Globalization/Internationalization testing)
- Тестирование локализации (Localization testing)

Основные понятия в тестировании производительности?

- Время задержки (Latency) - временной интервал между запросом и ответом. Например, у вашего сервиса время задержки составляет 100ms, что означает, что такому сервису потребуется 100 миллисекунд на обработку запроса и генерирование ответа. Как правило, чем ниже время задержки, тем лучше клиентский опыт.
- Время ответа (Response time) - время, необходимое для ответа на запрос
- Пропускная способность (Throughput) - фактическое количество запросов (или пользователей), которое может обработать система за определенное время. В то время как время задержки говорит вам только о времени, метрика пропускной способности информирует об объеме данных, полученных и обработанных в момент времени. Важно не отделять показатели времени задержки от пропускной способности, т.к. высокий показатель времени задержки часто прямо связан с увеличением показателей метрики пропускной способности. Пропускная способность обычно измеряется в rps – (кол-во) запросов в секунду (requests per second).
- Ширина пропускания канала (Bandwidth) - максимальное число запросов (или пользователей), которое может обработать система. В отличие от пропускной способности ширина пропускания канала измеряет максимальный объем, который может обработать приложение.
- Транзакции в секунду. Пользовательские транзакции – это последовательность действий пользователя в интерфейсе. Сравнивая реальное время прохождения транзакции с ожидаемой (или количество транзакций в секунду), вы сможете сделать вывод о том, насколько успешной системой было пройдено нагрузочное тестирование.
- Процент ошибок рассчитывается как отношение невалидных ответов к валидным за промежуток времени.
- Про Average, медианы, перцентили и т.п. углубляться в рамках этой статьи не буду, есть в гугле.

Тестирование производительности клиентской части и серверной, в чем разница?

Оценка скорости работы клиентской и серверной частей веб-приложения осуществляется двумя разными видами тестирования: для Frontend применяется тестирование клиентской части, или Client-Side testing, а для Back-end – тестирование серверной части.

Основная цель тестирования клиентской части состоит в измерении времени, необходимого браузеру, для загрузки HTML-страницы. Наиболее важными

показателями здесь являются количество загружаемых данных, их объем, а также количество выполненных запросов.
Собрать данную статистику можно как с использованием встроенных инструментов браузера (DevTools), так и с помощью специализированных инструментов и онлайн-сервисов, которые позволяют замерить необходимые показатели с учетом интересующего региона.
Помимо общего веса страницы, инструменты предоставляют детализированную информацию по каждому из компонентов. Изучив параметры запросов, можно обнаружить ряд проблем, приводящих к ухудшению скорости отображения страницы. К примеру, подгружается слишком большая картинка и Javascript, или отправляется значительное количество запросов.
Другая необходимая проверка направлена на анализ заголовков кэширования, поскольку корректность его выполнения при повторном посещении страницы позволяет повысить скорость загрузки страницы до 80%.
Тестирование серверной части направлено на анализ выполнения запросов и получения соответствующего ответа от Back-end.
Цели данного вида тестирования:

- Измерить время отклика самых важных бизнес-транзакций;
- Определить предельный уровень допустимой нагрузки;
- Выявить «узкие» места в производительности системы;
- Составить рекомендации по улучшению производительности;
- Найти возможные дефекты, проявляющиеся только при одновременной работе большого количества пользователей.

В общем виде что такое тестирование производительности?
Это класс тестирования ПО, который фокусируется на производительности системы при определенной нагрузке. Он не ищет напрямую ошибки или дефекты. Он производит аналитику на основе эталонных тестов и предоставляет разработчику всю диагностическую информацию, необходимую для выявления проблем производительности и узких мест. При этом происходит:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций
- определение количества пользователей, одновременно работающих с приложением
- определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций)
- исследование производительности на высоких, предельных, стрессовых нагрузках

Важно понимать, что все подвиды тестирования производительности это, грубо говоря, одно и то же, просто в зависимости от конкретного подвида выбираются разные параметры (показатели нагрузки/пользователей, длительности и т.п.) и собираются соответствующие метрики. Точкой отсчета принято брать результаты Capacity testing.
В реальном мире проводят только часть из перечисленных подвидов в соответствии с бюджетом и приоритетами данного ПО, а параметры тестов и метрики могут корректироваться в разных ситуациях.

Небольшая заметка. Несмотря на необходимость понимания многих математических и статистических концепций, многие тестировщики и менеджеры либо не имеют достаточных знаний в области математики и статистики, либо не пользуются ими. Это приводит к значительным искажениям и неверной интерпретации результатов тестирования производительности. Поэтому хороший специалист должен обладать и смежными знаниями.

Доп. материал:
Анализ результатов нагрузочного тестирования
Нагрузочное тестирование выполнять сложно, а инструменты далеки от совершенства. Почему?
Нагрузочное тестирование: особенности профессии

Что такое тестирование емкости/способностей? (Capacity)
Capacity — базовый тест, который обычно выполняется первым. Все последующие тесты на среднее время ответа, регенерацию системы, утилизацию ресурсов нужно выполнять с оглядкой на результаты Capacity. Емкость системы измеряется в rps (requests per second). Используемый подход: ступенчато повышаем нагрузку до момента, когда время ответа начнет расти экспоненциально. Экспоненциальный рост времени ответа, как правило, связан с полной утилизацией одного из ресурсов, из-за которого запросы вместо моментальной обработки выстраиваются друг за другом и ждут своей очереди на обработку.

Capacity point – точка, где перестает расти Throughput, но увеличивается Response Time, то есть система перестает справляться с запросами.
Исходя из этого тестирования выбираются значения для stress, load и soak/endurance тестов. Для stress берется значение близкое к capacity point, но немного меньше. Для load количество пользователей из зоны деградации.
Важно, ваша цель, не получить кол-во rps, при котором все взрывается, а понять, какой именно ресурс станет узким местом, при повышении нагрузки. Поэтому, если обстреливаемый вами сервер не покрыт мониторингом — можете даже не начинать тест. Общий подход к сбору телеметрии — чем больше метрик собирается, тем лучше.
В некоторых случаях Capacity называют так же Scalability (масштабируемость), но в целом это не равнозначные понятия.
Что означает тестирование масштабируемости? (Scalability)
Профиль нагрузки тот же, что и при нагрузочном тестировании. Что получаем в результате? Ответы на следующие вопросы:

- Увеличится ли производительность приложения, если добавить дополнительные аппаратные ресурсы?
- Увеличится ли производительность пропорционально количеству добавленных аппаратных средств?

Разница между тестированием емкости/способностей и тестированием масштабируемости? (Capacity vs Scalability)
Тестирование масштабируемости - это тестирование программного приложения для измерения его способности увеличивать или уменьшать масштаб с точки зрения любых его нефункциональных возможностей. Тестирование производительности, масштабируемости и надежности обычно группируется аналитиками качества ПО.

Тестирование емкости измеряет, сколько пользователей может обработать приложение. Это подмножество тестирования масштабируемости, в котором при тестировании масштабируемости вы получите меру емкости приложения.
Тестирование масштабируемости измеряет, насколько хорошо приложение справляется с растущим числом пользователей. Если вы тестируете масштабируемость до тех пор, пока приложение не выйдет из строя, у вас будет мера того, сколько пользователей (емкость) может обработать приложение.
Расскажите о стрессовом тестировании? (Stress testing)
Стрессовое тестирование выполняется самым первым, если нет отдельного Capacity тестирования, хотя по факту это все равно будет Capacity, т.к. нагрузка берется «с потолка». Позволяет проверить насколько приложение и система в целом работоспособны в условиях высокой нагрузки. Нагрузка на систему будет возрастать непрерывно до тех пор, пока не будет достигнут один из критериев его остановки.
Пример: стресс-тест банковской системы был остановлен при превышении отметки в 1500 пользователей, когда высокая загруженность процессора (более 80%) привела к увеличению среднего времени отклика в пять раз и массовому появлению ошибок HTTP(S).

Расскажите о нагрузочном тестировании? (Load)
Нагрузочное тестирование - это тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе. Этот тип тестирования производительности выполняется для диагностики поведения системы при увеличении рабочей нагрузки.
Нагрузка на систему обычно подается на протяжении 1-2 часов (в некоторых источниках: 4-8 часов, хотя это уже больше endurance), количество пользователей для нагрузочного теста берется из зоны деградации (в некоторых источниках: определяется в количестве 80% от результата максимальной производительности). В это время собираются метрики производительности: количество запросов в секунду, транзакций в секунду, время отклика от сервера, процент ошибок в ответах, утилизация аппаратных ресурсов и т. д.

Что такое объемное тестирование? (Volume testing)
Объемное тестирование предназначено для прогнозирования того, может ли система / приложение обрабатывать большой объем данных. Это тестирование сосредоточено на наполнении базы данных продукта в реальных сценариях использования.
Пример 1: отправка через POST-запросы очень большого количества данных.
Пример 2: как изменится производительность приложения спустя X лет, если аудитория приложения вырастет в Y раз?
Тестирование выносливости/стабильности/надежности (Soak/Endurance/Stability/Reliability testing)
Задачей тестирования стабильности является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Время выполнения операций может играть в данном виде тестирования второстепенную роль. При этом на первое место выходит проверка на утечки памяти, время отклика, правильность подключения и закрытия подключения к модулям (например, БД) и т.п. в течение длительного времени, чтобы гарантировать, что после длительного периода время отклика системы останется таким же или лучше, чем на

начало теста. Этот тип тестирования выполняется в самом конце (а где-то по ночам). Так же он помогает управлять будущими нагрузками, ведь нам необходимо понять, сколько дополнительных ресурсов (таких как ЦП, емкость диска, использование памяти или пропускная способность сети) необходимы для поддержки использования в будущем.

Что такое спайк/шиповое тестирование? (Spike)

Этот вид тестирования предназначен для определения поведения системы при внезапном увеличении нагрузки (большого количества пользователей) на систему. Например, дни распродаж в интернет-магазине.

Что такое тестирование устойчивости? (Resilence)

Проверка устойчивости проводится для того, чтобы убедиться, что система способна вернуться в исходное состояние после кратковременного напряжения. Например, если в интернет-магазине действует скидка на определенные товары на короткое время, скажем, один час в день.

Доп.материал:

Chaos Monkey

Что такое тестирование времени отклика? (Response time testing)

Время ответа относится ко времени, которое требуется одному системному узлу для ответа на запрос другого. Среднее время ответа - это среднее время, затрачиваемое на каждый запрос в оба конца. Пиковое время отклика помогает нам определить, какие компоненты потенциально проблематичны. Коэффициент ошибок - это математический расчет, который отображает процент проблемных запросов. Три критических значения времени отклика: 0,1 секунды, 1,0 секунды и 10 секунд.

Что такое Ramp тестирование?

Это метод тестирования, который предлагает ступенчато поднимать нагрузку до тех пор, пока система не выйдет из строя.

Что такое тестирование хранилища? (Storage testing)

Тестирование хранилища определяется как тип тестирования программного обеспечения, который проверяет, сохраняет ли тестируемое приложение соответствующие данные в соответствующих каталогах и достаточно ли у него места для предотвращения неожиданного завершения из-за недостатка дискового пространства. Это также называется тестированием производительности хранилища (Storage Performance testing).

Зачем оно нужно?

- Медленное хранилище означает медленное время отклика, длительные запросы и более низкую availability приложений.
- Медленное хранилище - это накладные расходы на обслуживание серверной инфраструктуры.
- Также помогает найти практическое ограничение хранилища перед деплоем.
- Помогает понять, как система будет реагировать при замене или обновлении аппаратного обеспечения.

Типы:

- Application testing: Тестирование приложений с примерами запросов в production like environment
- Сравните время ответа OLTP
- Сравните время выполнения batch
- Сравните rates непрерывной потоковой передачи

- Application Simulation: Проведите тестирование с использованием стандартного программного обеспечения, аналогичного целевому приложению
- Протестировать на пиковые значения IOPS для баз данных
- Тест пика для data streaming environments
- Проверка задержек хранилища для обмена сообщениями или других однопоточных приложений
- Benchmarking: Провести тестирование с использованием стандартного программного обеспечения.
- Проверка на повреждение данных.

Что такое тестирование на отказ и восстановление? (Failover and Recovery testing)

Тестирование на отказ и восстановление (Failover and Recovery testing) проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками ПО, отказами оборудования или проблемами связи/сети. Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.

Тестирование на отказ и восстановление очень важно для систем, работающих по принципу “24x7”. Если Вы создаете продукт, который будет работать, например, в интернете, то без проведения данного вида тестирования Вам просто не обойтись. Т.к. каждая минута простоя или потеря данных в случае отказа оборудования, может стоить вам денег, потери клиентов и репутации на рынке.

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

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

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

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

Что вы знаете о Тестировании удобства пользования? (Usability testing)

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

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

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

Проверка удобства использования может проводиться как по отношению к готовому продукту, посредством тестирования черного ящика (black box testing), так и к интерфейсам приложения (API), используемым при разработке - тестирование белого ящика (white box testing). В этом случае проверяется удобство использования внутренних объектов, классов, методов и переменных, а также рассматривается удобство изменения, расширения системы и интеграции ее с другими модулями или системами. Использование удобных интерфейсов (API) может улучшить качество, увеличить скорость написания и поддержки разрабатываемого кода, и как следствие улучшить качество продукта в целом.

Отсюда становится очевидно, что тестирование удобства пользования может производиться на разных уровнях разработки ПО: модульном, интеграционном, системном и приемочном.

Доп. материал:

Вкусный и здоровый гайд по юзабилити-тестированиям

Отличия тестирование на удобство пользования и тестирования доступности? (Usability Vs. Accessibility testing)

USABILITY testing показывает, насколько проста в использовании и удобна система программного обеспечения. Здесь небольшой набор целевых конечных пользователей «использует» программную систему для выявления дефектов юзабилити. Основное внимание в этом тестировании уделяется простоте использования приложения пользователем, гибкости в управлении средствами управления и способности системы выполнять свои задачи. Это также называется тестированием пользовательского опыта (UX – “Ю-Экс”, user experience). Это тестирование рекомендуется на начальном этапе разработки SDLC, что позволяет лучше понять ожидания пользователей. Исследования (Virzi, 1992 и Neilsen Landauer, 1993) показывают, что 5 пользователей достаточно для выявления 80% проблем с юзабилити, хотя некоторые исследователи предлагают другие цифры.

Тестирование доступности (accessibility testing) - это подмножество юзабилити-тестирования. Его цель - убедиться в том, что наш продукт удобен в использовании для людей с различными видами инвалидности или особенностей восприятия. Это могут быть проблемы со зрением, слухом или ограничения в подвижности рук.

Ваш продукт должен правильно работать с соответствующим ПО. Примеры такого программного обеспечения:

- Speech Recognition Software - ПО преобразует произнесенное слово в текст, который служит вводом для компьютера.
- Программа для чтения с экрана - используется для озвучивания текста, отображаемого на экране
- Программное обеспечение для увеличения экрана - используется для увеличения масштаба элементов и облегчения чтения для пользователей с нарушениями зрения.
- Специальная клавиатура, облегчающая ввод для пользователей, у которых проблемы с двигательными функциями.

Еще один из примеров - люди с цветовой слепотой (дальтонизмом). Эта особенность довольно широко распространена. Различными видами цветовой слепоты страдают около 8 % мужчин и 0,4 % женщин - не так уж мало!

Цвет не должен быть единственным способом передачи информации. Если вы используете цвет для того, чтобы, допустим, отобразить статус, эту информацию стоит продублировать еще каким-то образом - геометрическими фигурами, иконками или текстовым комментарием.

Хорошая контрастность. Хорошая контрастность обеспечивает нормальную видимость элементов управления и текста даже для людей, не различающих те или иные оттенки. Есть отличный инструмент для тестирования веб-сайтов на предмет доступности для людей с различными формами цветовой слепоты: Color Blind Web Page Filter.

Если вы хотите сократить количество тестов, можно ограничиться только тремя фильтрами: дейтеранопия, протанопия и тританопия. Это наиболее выраженные формы цветовой слепоты (не считая крайне редкого черно-белого зрения). Остальные люди с особенностями цветовосприятия видят больше оттенков, и если ваш UI достаточно хорошо виден с этими тремя фильтрами, то и для остальных будет отображаться корректно.

Пример чек-листа:

- Предоставляет ли приложение клавиатурные эквиваленты для всех действий мышью и окон?
- Предоставляются ли инструкции как часть пользовательской документации или руководства? Легко ли понять и использовать приложение, используя документацию?
- Упорядочены ли вкладки логически для обеспечения плавной навигации?
- Предусмотрены ли сочетания клавиш для меню?
- Поддерживает ли приложение все операционные системы?
- Четко ли указано время отклика каждого экрана или страницы, чтобы конечные пользователи знали, как долго ждать?
- Все ли надписи правильно написаны?
- Являются ли цвета подходящим для всех пользователей?
- Правильно ли используются изображения или значки, чтобы их было легко понять конечным пользователям?
- Есть ли звуковые оповещения?
- Может ли пользователь настроить аудио или видео элементы управления?
- Может ли пользователь переопределить шрифты по умолчанию для печати и отображения текста?

- Может ли пользователь настроить или отключить мигание, вращение или перемещение элементов?
- Убедитесь, что цветовое кодирование никогда не используется в качестве единственного средства передачи информации или указания на действие
- Видна ли подсветка с инвертированными цветами?
- Тестирование цвета в приложении путем изменения контрастности
- Правильно ли слышат люди с ограниченными возможностями все имеющее отношение к аудио и видео?
- Протестируйте все мультимедийные страницы без мультимедиа-оборудования.
- Предоставляется ли обучение пользователям с ограниченными возможностями, что позволит им ознакомиться с программным обеспечением или приложением?

Доп. материал:
QA • и его роль в создании ресурсов для людей с ограниченными возможностями
Чеклист по UX из • 30 • пунктов для мобильных приложений
Web Content Accessibility Guidelines
Что такое тестирование интерфейса?
Это тип интеграционного теста, который проверяет, правильно ли установлена связь между двумя различными программными системами или их частями (модулями). Соединение, которое объединяет два компонента, называется интерфейсом. Этот интерфейс в компьютерном мире может быть чем угодно, как API, так и веб-сервисами и т. д. Тестирование этих подключаемых сервисов или интерфейса называется Тестированием интерфейса.
Тестирование интерфейса включает в себя тестирование двух основных сегментов:
- Интерфейс веб-сервера и сервера приложений
- Интерфейс сервера приложений и базы данных

Что такое тестирование рабочего процесса/воркфлоу? (Workflow testing)
Это тип тестирования программного обеспечения, который проверяет, что каждый software workflow точно отражает данный бизнес-процесс. Workflow - это серия задач для получения желаемого результата, которая обычно включает несколько этапов или шагов. Для любого бизнес-процесса тестирование этих последовательных шагов определяется как «WorkFlow testing».
Например, убедитесь, что система может быть установлена на платформе пользователя и работает правильно. Тестирование рабочего процесса проводится поэтапно. Вот как вы будете выполнять Workflow testing:
- Начальная фаза (Inception phase): эта фаза включает начальное планирование испытаний и тестирование прототипа
- Фаза разработки (Elaboration phase): Эта фаза включает базовую архитектуру тестирования
- Фаза строительства (Construction phase): эта фаза включает в себя значительные испытания в каждой сборке
- Фаза перехода (Transition phase): Эта фаза включает в себя регрессионные тесты и повторные тесты исправлений

Тестирование workflow выполняется:
- Test engineer: планирует цели теста и график. Определяет Test case и процедуры. Оценивает результаты теста.
- Component engineer: Разработка тестовых компонентов. Автоматизирует некоторые тестовые процедуры.

- Integration Tester: Выполнение интеграционных тестов и выявление дефектов
- System Testers: Выполнение системных тестов и отчеты о дефектах

Что вы знаете о пользовательском приемочном тестировании? (UAT – User Acceptance testing)

Пользовательское приемочное тестирование (UAT) - это тип тестирования, выполняемый конечным пользователем или клиентом для проверки / принятия ПО перед его перемещением в production. UAT выполняется на заключительном этапе тестирования после выполнения функциональных, интеграционных и системных испытаний. Основной целью UAT является проверка end-to-end business flow. Он не фокусируется на косметических ошибках, орфографических ошибках или тестировании системы. Приемочное тестирование пользователя выполняется в отдельной среде тестирования с настройкой данных, аналогичных производственным. Это своего рода тестирование черного ящика, в котором будут участвовать два или более конечных пользователя. Этапы:

- Анализ бизнес-требований
- Создать плана тестирования UAT
- Определить Test Scenario
- Создать Test case UAT
- Подготовить Test Data (Production like Data)
- Запустить Test case
- Записать результаты
- Подтвердить бизнес-цели

Что такое эксплуатационное приемочное тестирование? (OAT - Operational Acceptance testing)

ИСПЫТАНИЕ НА ЭКСПЛУАТАЦИЮ (OAT) - это тип тестирования программного обеспечения, который оценивает операционную готовность программного приложения до его выпуска в производство. Целью эксплуатационного тестирования является обеспечение бесперебойной работы системы в ее стандартной операционной среде (SOE - standard operating environment). Это также называется Оперативное тестирование (Operational testing). Эксплуатационное приемочное тестирование обеспечивает соответствие системы и компонентов в стандартной операционной среде приложения (SOE). Типы OAT:

- Installation testing
- Load & Performance Test Operation
- Backup and Restore testing

Security testing

- Code Analysis
- Fail over testing
- Recovery testing
- End-to-End• Test Environment Operational testing
- Operational Documentation Review

Примеры Test case:

- Резервные копии, сделанные на одном сайте, могут быть развернуты на тот же сайт
- Резервные копии, сделанные на одном сайте, можно развернуть на другом сайте.
- Внедрение любых новых функций в живую производственную среду не должно отрицательно влиять на целостность текущих производственных услуг.

- Процесс внедрения может быть воспроизведен с использованием действующей документации
- Каждый компонент может быть отключен и успешно запущен в согласованные сроки.
- Для оповещений - все критические оповещения должны идти в TEC и ссылаться на документ правильного разрешения.
- Оповещения созданы и выдаются при превышении согласованных пороговых значений
- Любая документация по восстановлению, созданная или измененная, включая сервисные диаграммы, действительна
- Это должно быть передано в соответствующие области поддержки.
- Любой компонент, на который влияет сбой, должен показывать рекомендуемый порядок перезапуска, время завершения и т. д.

Расскажите об инсталляционном тестировании?

Тестирование инсталляции (установки) направленно на проверку успешной инсталляции и настройки, а также обновления или удаления ПО, как десктопного, так и мобильного.

Тестирование инсталляции в большинстве своем не входит в Веб-тестирование, являясь специализированным тестированием установки приложений на различные операционные системы.

Следующие проверки должны быть выполнены для этапов:

Установка.

- Установка должна начаться при клике по кнопке, подтверждающей данное действие
- Установки во всех поддерживаемых окружениях и на всех поддерживаемых платформах
- Установки в неподдерживаемых окружениях, а также в нужных окружениях с некорректными настройками
- Права, которые требует инсталляция (чаще всего они должны быть админскими), проверить установить приложение как гость
- Установки в clean state (при отсутствии любых возможных связанных файлов и предыдущих версий)
- Подсчитывается ли при установке количество свободного места на диске и выдается ли предупреждение если места недостаточно
- Установки загруженного ранее приложения, а также прямая установка с использованием сети/беспроводного соединения
- Восстановится ли процесс установки при внезапном его прерывании (отключение устройства, отказ сети, отключение беспроводного соединения)
- Установка приложения, его запуск, удаление приложения должны возвращать систему в исходное состояние
- Распознается ли наличие в системе приложений/программ, необходимых для корректной работы устанавливаемого приложения
- Повторный запуск установки приложения при уже текущем должен выдавать корректное сообщение, двойная установка должна быть исключена
- Процесс установки может быть настраиваемый/дефолтный. Убедиться, что оба корректно работают
- Наличие кнопки, которая предложит сохранить приложение в определенную папку, а также указывает дефолтное местоположение (“C:/programs/.”)

- Правильно ли установлены, сохранены ли в корректных папках файлы приложения
- Наличие созданных ярлыков, корректно ли они расположены
- После установки в системной вкладке “ Программы и компоненты” должны быть доступны: название приложения, иконка, имя издателя, размер приложения, дата установки и номер версии
- Настройки переменных сред PATH
- Убедиться, что лицензионный ключ сохраняется в Windows Registry library
- Поддерживает ли приложение функции ‘UnInstall’, ‘Modify’, ‘ReInstall’ и корректно ли они работают
- Работа приложения с уже существующими DLL-файлами, с DLL-файлами приложений, которые необходимы для корректной работы устанавливаемого приложения
- Наличие информации/сообщение о том, когда истекает срок действия установленной пробной версии приложения

Обновление:

- Поддерживает ли приложение функцию обновления/автообновления
- При попытке установить ранее установленную версию приложения система должна ее распознать и выдать корректное сообщение
- Сохраняются ли пользовательские настройки при попытке загрузить новую версию/обновить старую версию
- При попытке обновить версию должны быть доступны функции удалить приложение и восстановить приложение
- Стандартные проверки как при первичной установке приложения
- Убедиться, что номер версии приложения сменился новым
- Запустить приложение и убедиться, что оно работает корректно

Откат до предыдущей версии:

- Попробовать установить старую версию на более новую
- Наличие корректного сообщения при попытке отката
- Убедиться, что приложение работает корректно

Удаление приложения:

- Не остается ли в системе никаких папок/файлов/ярлыков/ключей реестра после полного удаления приложения
- Корректно ли работает система после установки и последующего удаления приложения

Что вы знаете о тестировании безопасности? (Security and Access Control testing)

Это тип тестирования ПО, который выявляет уязвимости, угрозы и риски. Целью тестов безопасности является выявление всех возможных лазеек и слабых мест в ПО, которые могут привести к потере информации, доходов, репутации компании, сотрудников или клиентов. Общая стратегия безопасности основывается на трех основных принципах:

- Конфиденциальность - сокрытие определенных ресурсов или информации
- Целостность – ресурс может быть изменен только в соответствии с полномочиями пользователя
- Доступность - ресурсы должны быть доступны только авторизованному пользователю, внутреннему объекту или устройству

Тестирование безопасности обычно выполняет отдельный специалист по безопасности. В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено все:

- попытки узнать пароль с помощью внешних средств;
- атака системы с помощью специальных утилит, анализирующих защиты;
- подавление, ошеломление системы (в надежде, что она откажется обслуживать других клиентов);
- целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;
- просмотр несекретных данных в надежде найти ключ для входа в систему.

При неограниченном времени и ресурсах хорошее тестирование безопасности взломает любую систему. Задача проектировщика системы — сделать цену проникновения более высокой, чем цена получаемой в результате информации.
Типы тестирования безопасности:

- Сканирование уязвимостей/оценка защищенности (Vulnerability Scanning) выполняется с помощью автоматизированного ПО для сканирования системы на наличие известных сигнатур уязвимостей.
- Сканирование безопасности (Security Scanning) включает в себя выявление слабых сторон сети и системы, а затем предоставляет решения для снижения этих рисков. Это сканирование может быть выполнено как ручным, так и автоматизированным.
- Тестирование на проникновение (Penetration testing) - этот тип тестирования имитирует атаку злоумышленника. Это тестирование включает анализ конкретной системы для проверки потенциальных уязвимостей при попытке внешнего взлома.
- Оценка рисков (Risk Assessment) тестирование включает анализ рисков безопасности, наблюдаемых в организации. Риски классифицируются как Низкие, Средние и Высокие. Это тестирование рекомендует меры по снижению риска.
- Аудит безопасности (Security Auditing) - внутренняя проверка приложений и операционных систем на наличие уязвимостей. Аудит также может быть выполнен путем построчной проверки кода
- Этический взлом (Ethical hacking) - совершается с целью выявления проблем безопасности в системе. Это делается White Hat хакерами - это специалисты по безопасности, которые использует свои навыки законным способом для помощи в выявлении дефектов системы, в отличии от Black Hat (преступников) или Gray Hat (что-то между).
- Оценка состояния (Posture Assessment) объединяет сканирование безопасности, этический взлом и оценки рисков, чтобы показать общее состояние безопасности организации.

SDLC фаза	Security Processes
Requirements	Анализ безопасности для требований и проверка случаев злоупотребления / неправильного использования
Design	Анализ рисков безопасности для проектирования. Разработка плана тестирования с учетом тестирования безопасности
Coding and Unit testing	Статическое и динамическое тестирование безопасности и тестирование белого ящика
Integration testing	Тестирование черного ящика
System testing	Тестирование черного ящика и сканирование уязвимостей

Implementation Тестирование на проникновение, сканирование уязвимостей
Support Анализ воздействия патчей

Доп. материал:
Топ-• 10 • уязвимостей мобильных приложений и способы их устранения
Безопасность веб-приложений: от уязвимостей до мониторинга
Социотехническое тестирование: какое лучше выбрать в • 2021 • году?
Анализ безопасности веб-проектов
Безопасность интернет-приложений
Red Teaming • —• • комплексная имитация атак. Методология и инструменты
cHack
OWASP Top Ten
Santoku Linux
Kali Linux
• https://github.com/FSecureLABS/drozer
SQL-• инъекции' union select null,null,null --
Что такое XSS-уязвимость и как тестировщику не пропустить ее
Что означает оценка уязвимости/защищенности? (Vulnerability Assessment)
Это процесс оценки рисков безопасности в программной системе с целью уменьшения вероятности угрозы. Уязвимость - это любые ошибки или слабости в процедурах безопасности системы, разработке, реализации или любом внутреннем контроле, которые могут привести к нарушению политики безопасности системы. Целью оценки уязвимости является снижение возможности несанкционированного доступа для злоумышленников (хакеров). Анализ проникновения зависит от двух механизмов, а именно от оценки уязвимости и тестирования на проникновение (VAPT - Vulnerability Assessment and Penetration testing).
Методы:
• Активное тестирование
• Inactive testing, тестировщик вводит новые test data и анализирует результаты.
• В процессе тестирования тестировщики создают интеллектуальную модель процесса, и она будет расти и дальше во время взаимодействия с тестируемым программным обеспечением.
• Выполняя тест, тестировщик будет активно вовлекаться в процесс обнаружения новых Test case и новых идей. Вот почему это называется Active testing.
• Пассивное тестирование
• Пассивное тестирование - отслеживание результатов запуска тестируемого программного обеспечения без введения новых Test case или data.
• Тестирование сети
• Тестирование сети - это процесс измерения и записи текущего состояния работы сети за определенный период времени.
• Тестирование в основном проводится для прогнозирования работы сети под нагрузкой или для выявления проблем, создаваемых новыми сервисами.
• Нам нужно проверить следующие характеристики сети:
• Уровни утилизации
• Количество пользователей
• Использование приложения
• Распределенное Тестирование

- Распределенные тесты применяются для тестирования распределенных приложений, то есть приложений, работающих с несколькими клиентами одновременно. По сути, тестирование распределенного приложения означает тестирование его клиентской и серверной частей по отдельности, но с помощью метода распределенного тестирования мы можем протестировать их все вместе.
- Части теста будут взаимодействовать друг с другом во время теста. Это делает их синхронизированными соответствующим образом. Синхронизация является одним из наиболее важных моментов в распределенном тестировании.

Расскажите подробнее о тестировании на проникновение? (Penetration testing)
PENETRATION testing - это тип тестирования безопасности, который выявляет уязвимости, угрозы, риски в программном приложении, сети или веб-приложении, которые может использовать злоумышленник. Тестирование на проникновение показывает реальную картину существующей угрозы в системе безопасности и определяет уязвимости организации к ручным атакам. Проведение пентеста на регулярной основе позволит определить технические ресурсы, инфраструктуру, физические и кадровый арсенал содержащие в себе слабые аспекты, которые требуют развития и усовершенствования. Данный вид тестирования выполняется как вручную, так и автоматически и может быть как Black Box, так и Grey и White. Ввиду необходимости наличия специфических знаний и опыта для выполнения этого вида тестирования привлекается отдельный специалист – пентестер. Примеры кейсов тестирования на проникновение:

- Тест на проникновение в сеть:
- выявлении уязвимостей сетевого и системного уровня;
- определение неправильных конфигураций и настроек;
- выявление уязвимости беспроводной сети;
- мошеннические услуги;
- отсутствие • надежных паролей и наличие слабых протоколов.

Тест на проникновение приложений:

- выявление недостатков прикладного уровня;
- подделка запросов;
- применение злонамеренных скриптов;
- нарушение работы управления сеансами;
- и т.п.
- Тест на физическое проникновение:
- взлом физических барьеров;
- проверка и взлом замков;
- нарушения работы и обход датчиков;
- вывод из строя камер видеонаблюдения;
- и т. д.

Отличия Vulnerability Assessment от Penetration testing?
Оба этих вида отличаются друг от друга по силе и задачам, которые они выполняют. Однако для составления исчерпывающего отчета по тестированию уязвимостей рекомендуется сочетание обеих процедур.

	Vulnerability Assessment	Penetration testing
Working Mechanism	Нахождений уязвимостей Обнаружение и сканирование	Выявление и использование уязвимостей Симуляция

Focus	Ширина	Глубина
Coverage of Completeness	Высокое покрытие	Низкое
Cost	Низкая стоимость	Высокая
Performed By	Внутренний персонал	Атакующий или пентестер
How often to Run	После каждой сборки	Реже, зависит от политики безопасности компании и продукта
Result	Предоставит частичную информацию об уязвимостях	Предоставит полную информацию об уязвимостях

Что такое Fuzz тестирование?
FUZZ testing (fuzzing) – это метод тестирования ПО методом черного ящика, один из типов тестирования безопасности, который вводит недействительные или случайные данные, называемые FUZZ, в систему программного обеспечения для обнаружения ошибок кодирования и лазеек в безопасности. Данные вводятся с использованием автоматических или полуавтоматических методов тестирования, после чего система отслеживается на предмет различных исключений, таких как сбой системы или сбой встроенного кода и т. д.
Обычно fuzzing обнаруживает наиболее серьезные ошибки или дефекты безопасности. Это очень экономически эффективный метод тестирования. Fuzzing - один из самых распространенных методов хакеров, используемых для обнаружения уязвимости системы (сюда относятся популярные SQL- или скриптовые инъекции). Примеры фаззеров:

- Mutation-Based Fuzzers: изменяет существующие образцы данных для создания новых test data. Это очень простой и понятный подход, он начинается с действительных образцов и постоянно корректирует каждый байт или файл.
- Generation-Based Fuzzers: определяет новые данные на основе ввода модели. Он начинает генерировать ввод с нуля на основе спецификации.
- PROTOCOL-BASED-fuzzer: самый успешный фаззер - это детальное знание тестируемого формата протокола. Понимание зависит от спецификации. Это включает в себя запись массива спецификации в инструмент, а затем с помощью метода генерации тестов на основе модели проходится спецификация и добавляется неравномерность в содержимое данных, последовательность и т. д. Это также известно как синтаксическое тестирование, грамматическое тестирование, тестирование надежности, и т. д. Fuzzer может генерировать Test case из существующего или использовать допустимые или недействительные входные данные.

Типы ошибок, обнаруживаемых Fuzz testing:

- Сбои ассертов и утечки памяти (Assertion failures and memory leaks). Эта методология широко используется для больших приложений, где ошибки влияют на безопасность памяти, что является серьезной уязвимостью.
- Некорректный ввод (Invalid input). Фаззеры используются для генерирования неверного ввода, который используется для тестирования процедур обработки ошибок, и это важно для программного обеспечения, которое не контролирует его ввод. Простой фаззинг может быть способом автоматизации отрицательного тестирования.
- Исправление ошибок (Correctness bugs). Fuzzing также может использоваться для обнаружения некоторых типов ошибок «правильности». Например, поврежденная база данных, плохие результаты поиска и т. д.

Доп. материал:
Фаззинг тестирование веб-интерфейса. Расшифровка доклада
Можно ли отнести тестирование безопасности или нагрузочное тестирование к функциональным видам тестирования?
Данные виды во многих источниках относят к нефункциональным видам тестирования, но если это является основной функцией приложения, то можно отнести и к функциональным.

Мнение:
“<..> Есть функциональное требование:
"Пользователь должен иметь возможность перевести деньги со своей карты на другую карту по номеру".
Это функциональное требование (ну, на самом деле это целая тонна требований, но обобщим их до одной user story).
Оно отвечает на вопрос "какие операции должен уметь выполнять сервис".
К этой функциональности может предъявляться еще куча требований - по безопасности, по скорости, по отказоустойчивости, и т.д. Они описывают то, как система должна работать, а не что она должна уметь.
Нефункциональные требования могут быть критичными, могут блокировать выпуск той или иной функциональности. Но это все еще свойство фичи, а не какая-то самостоятельная ее функция.
В то же время, есть, например, функциональные требования безопасности, типа "автоматически блокировать транзакции обладающие характеристиками А, Б, В". © @azshoo
Это снова нас возвращает к тому, что система должна обладать какими-то функциями.”
Что вы знаете о конфигурационном тестировании? (Configuration testing)
Конфигурационное тестирование (Configuration testing) — специальный вид тестирования, направленный на проверку работы ПО при различных аппаратных и программных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т. д.)
В зависимости от типа проекта конфигурационное тестирование может иметь разные цели:

- Проект по профилированию работы системы

Цель Тестирования: определить оптимальную конфигурацию оборудования, обеспечивающую требуемые характеристики производительности и времени реакции тестируемой системы.

- Проект по миграции системы с одной платформы на другую

Цель Тестирования: Проверить объект тестирования на совместимость с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм.
Для клиент-серверных приложений конфигурационное тестирование можно условно разделить на два уровня (для некоторых типов приложений может быть актуален только один):

- Серверный
- Клиентский

На первом (серверном) уровне, тестируется взаимодействие выпускаемого ПО с окружением, в которое оно будет установлено:

- Аппаратные средства (тип и количество процессоров, объем памяти, характеристики сети / сетевых адаптеров и т. д.)
- Программные средства (ОС, драйвера и библиотеки, стороннее ПО, влияющее на работу приложения и т. д.)

Основной упор здесь делается на тестирование с целью определения оптимальной конфигурации оборудования, удовлетворяющего требуемым характеристикам качества (эффективность, портативность, удобство сопровождения, надежность).
На следующем (клиентском) уровне, ПО тестируется с позиции его конечного пользователя и конфигурации его рабочей станции. На этом этапе будут протестированы следующие характеристики: удобство использования, функциональность. Для этого необходимо будет провести ряд тестов с различными конфигурациями рабочих станций:

- Тип, версия и битность операционной системы (подобный вид тестирования называется кроссплатформенное тестирование)
- Тип и версия Web браузера, в случае если тестируется Web приложение (подобный вид тестирования называется кросс-браузерное тестирование)
- Тип и модель видеоадаптера (при тестировании игр это очень важно)
- Работа приложения при различных разрешениях экрана
- Версии драйверов, библиотек и т. д. (для JAVA приложений версия JAVA машины очень важна, тоже можно сказать и для .NET приложений касательно версии .NET библиотеки)

и т. д.
Перед началом проведения конфигурационного тестирования рекомендуется:

- создавать матрицу покрытия (матрица покрытия - это таблица, в которую заносят все возможные конфигурации),
- проводить приоритезацию конфигураций (на практике, скорее всего, все желаемые конфигурации проверить не получится),
- шаг за шагом, в соответствии с расставленными приоритетами, проверять каждую конфигурацию.

Уже на начальном этапе становится очевидно, что чем больше требований к работе приложения при различных конфигурациях рабочих станций, тем больше тестов нам необходимо будет провести. В связи с этим, рекомендуем, по возможности, автоматизировать этот процесс, так как именно при конфигурационном тестировании автоматизация реально помогает сэкономить время и ресурсы. Конечно же автоматизированное тестирование не является панацеей, но в данном случае оно окажется очень эффективным помощником.
В итоге: конфигурационным называется тестирование совместимости выпускаемого продукта (ПО) с различным аппаратным и программным средствами.
Основные цели - определение оптимальной конфигурации и проверка совместимости приложения с требуемым окружением (оборудованием, ОС и т. д.). Автоматизация конфигурационного тестирования позволяет избежать лишних расходов
Примечание: в ISTQB вообще не говорится о таком виде тестирования как конфигурационное. Согласно глоссарию, данный вид тестирования рассматривается там как тестирование портируемости:
configuration testing: See portability testing.
portability testing: The process of testing to determine the portability of a software product.

Что подразумевается под проверкой на дым / дымовым тестированием? (Smoke testing)
Смоук тестирование рассматривается как короткий цикл тестов, выполняемый для каждой новой сборки для подтверждения того, что ПО стартует и выполняет основные функции без критических и блокирующих дефектов. В случае отсутствия таковых дефектов дымовое тестирование объявляется пройденным, и команда QA может начинать дальнейшее тестирование полного цикла, в противном случае, сборка объявляется дефектной, что делает дальнейшее тестирование пустой тратой времени и ресурсов. Сборка возвращается на доработку и исправление.
Аналогами дымового тестирования являются Build Verification testing и Acceptance testing, выполняемые на функциональном уровне командой тестирования, по результатам которых делается вывод о том, принимается или нет установленная версия ПО в тестирование, эксплуатацию или на поставку заказчику.
Если мы говорим про сайт интернет-магазина, то сценарий будет примерно следующим:

- Сайт открывается
- Можно выбрать случайный товар и добавить его в корзину
- Можно оформить и оплатить заказ

Если мы говорим про мобильное приложение, например, messenger, то:

- Приложение устанавливается и запускается
- Можно авторизоваться
- Можно написать сообщение случайном контакту

Синонимом в некоторых источниках указан breath testing.
Небольшая шпаргалка по степени важности:

- smoke - самое важное
- critical path - повседневное
- extended - все

В русском языке термин ошибочно переводят как проверка дыма, корректнее уж говорить “на дым”. История термина: Первое свое применение этот термин получил у печников, которые, собрав печь, закрывали все заглушки, затапливали ее и смотрели, чтобы дым шел только из положенных мест.
Повторное «рождение» термина произошло в радиоэлектронике. Первое включение нового радиоэлектронного устройства, пришедшего из производства, совершается на очень короткое время (меньше секунды). Затем инженер руками ощупывает все микросхемы на предмет перегрева. Сильно нагревшаяся за эту секунду микросхема может свидетельствовать о грубой ошибке в схеме. Если первое включение не выявило перегрева, то прибор включается снова на большее время. Проверка повторяется. И так далее несколько раз. Выражение «smoke-test» используется инженерами в шуточном смысле, так как появления дыма, а значит и порчи частей устройства, стараются избежать.

Доп. материал:
QA Outsourcing: Smoke Testing, Critical Path Testing, Extended Testing
Что такое тестирование встряхиванием? (Shake out testing)
Стандартного определения ISO для теста на встряхивание для ПО не существует. Говорят, что это синоним к Smoke тестированию.

Что подразумевается под санитарным тестированием? (Sanity testing)
Для начала стоит сказать, что санитарным оно в русскоязычной среде назвалось по совершенно непонятным причинам, но гуглится только так. На самом же деле корректно говорить тестирование на вменяемость или разумность.
Санитарное тестирование - это узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Обычно выполняется вручную.

Доп. материалы:
Definition of sanity
What is Sanity testing? Advantages, disadvantages HYPERLINK "http://tryqa.com/what-is-sanity-testing/"& HYPERLINK "http://tryqa.com/what-is-sanity-testing/" differences
Отличие санитарного тестирования от дымового? (Sanity vs Smoke testing)
Эти виды тестирования имеют "вектора движения", направления в разные стороны. В отличии от дымового (Smoke testing), санитарное тестирование (Sanity testing) направлено вглубь проверяемой функции, в то время как дымовое направлено вширь, для покрытия тестами как можно большего функционала в кратчайшие сроки.

Доп. материал:
В чем разница Smoke, Sanity, Regression, Re-test и как их различать?
Что вы знаете про регрессионное тестирование? (Regression testing)
При корректировках программы необходимо гарантировать сохранение качества. Для этого используется регрессионное тестирование - дорогостоящая, но необходимая деятельность в рамках этапа сопровождения, направленная на перепроверку корректности измененной программы. В соответствии со стандартным определением, регрессионное тестирование - это выборочное тестирование, позволяющее убедиться, что изменения не вызвали нежелательных побочных эффектов, или что измененная система по-прежнему соответствует требованиям.
Главной задачей этапа сопровождения является реализация систематического процесса обработки изменений в коде. После каждой модификации программы необходимо удостовериться, что на функциональность программы не оказал влияния модифицированный код. Если такое влияние обнаружено, говорят о регрессионном дефекте. Для регрессионного тестирования функциональных возможностей, изменение которых не планировалось, используются ранее разработанные тесты. Одна из целей регрессионного тестирования состоит в том, чтобы, в соответствии с используемым критерием покрытия кода (например, критерием покрытия потока операторов или потока данных), гарантировать тот же уровень покрытия, что и при полном повторном тестировании программы. Для этого необходимо запускать тесты, относящиеся к измененным областям кода или функциональным возможностям. Другая цель регрессионного тестирования состоит в том, чтобы удостовериться, что программа функционирует в соответствии со своей спецификацией, и что изменения не привели к внесению новых ошибок в ранее протестированный код. Эта цель всегда может быть достигнута повторным выполнением всех тестов регрессионного набора, но более перспективно отсеивать тесты, на которых выходные данные

модифицированной и старой программы не могут различаться. Важной задачей регрессионного тестирования является также уменьшение стоимости и сокращение времени выполнения тестов.
Можно заключить, что регрессионное тестирование выполняется чтобы минимизировать регрессионные риски. То есть, риски того, что при очередном изменении продукт перестанет выполнять свои функции. С регрессионным тестированием плотно связана другая активность - импакт анализ (или иначе, анализ влияния изменений). Обычно под импакт анализом имеют в виду одно из следующих:

- Попытку оценить регрессионные риски еще на этапе планирования изменений (этим определением, по моему опыту, чаще пользуются менеджеры и разработчики);
- Попытку определить объем регрессионного тестирования с учетом изменений, которые уже произошли (это определение чаще используют сами тестировщики). У Пола Джеррарда есть серия статей, где более детально раскрывается понятие импакт анализа, причем не только с позиции тестировщика.

Очевидно, что от эффективности импакт анализа зависит эффективность регрессионного тестирования. Но не всегда тщательно проведенный импакт анализ позволяет сократить затраты на последующее тестирование.
Обоснование корректности метода отбора тестов. Перечислим некоторые особенности реализации регрессионного тестирования:

- Некоторые участки кода программы не получают управление при выполнении некоторых тестов.
- Если участок кода реализует требование, но измененный фрагмент кода не получает управления при выполнении теста, то он и не может воздействовать на значения выходных данных программы при выполнении данного теста.
- Даже если участок кода, реализующий требование, получает управление при выполнении теста, это далеко не всегда отражается на выходных данных программы при выполнении данного теста. Действительно, если изменяется первый блок программы, например, путем добавления инициализации переменной, все пути в программе также изменяются, и, как следствие, требуют повторного тестирования. Однако может так случиться, что только на небольшом подмножестве путей действительно используется эта инициализированная переменная.
- Не каждый тест, проверяющий код, находящийся на одном пути с измененным кодом, обязательно покрывает этот измененный код.
- Код, находящийся на одном пути с измененным кодом, может не воздействовать на значения выходных данных измененных модулей программы.
- Не всегда каждый оператор программы воздействует на каждый элемент ее выходных данных.

Предположим, что изменения в программе ограничиваются одним оператором. Если при выполнении какого-либо теста на исходной программе этот оператор никогда не получает управление, можно с уверенностью сказать, что он не получит управление и в ходе выполнения теста на новой программе, а результаты тестирования новой и старой программ будут совпадать. Следовательно, нет необходимости выполнять этот тест на новой программе. Указанный метод легко можно обобщить для случая нескольких изменений: если тест не задействует ни одного измененного оператора, и его входные данные не изменились, код, выполняемый им в измененной программе, будет в точности таким же, как в первоначальной версии. Такой тест не выявляет различий между двумя версиями системы; следовательно, нет необходимости прогонять его повторно. Если тест не затрагивает ни одного оператора вывода,

поведение которого зависит от измененных операторов, это означает, что, несмотря на изменения в программе, все операторы, которые получают управление при выполнении этого теста, не изменят вывод системы по отношению к предыдущей версии. Таким образом, нет необходимости повторно прогонять и тесты такого рода. Следовательно, необходимо ориентироваться на выбор только тех тестов, которые покрывают измененный код, воздействующий, в свою очередь, на вывод программы. Такой подход гарантирует, что будут выбраны только тесты, обнаруживающие изменения, и метод будет, как говорят, точным.
Классификация тестов при отборе.
Создание наборов регрессионных тестов рекомендуется начинать с множества исходных тестов. При заданном критерии регрессионного тестирования все исходные тесты подразделяются на четыре подмножества:

- Множество тестов, пригодных для повторного использования. Это тесты, которые уже запускались и пригодны к использованию, но затрагивают только покрываемые элементы программы, не претерпевшие изменений. При повторном выполнении выходные данные таких тестов совпадут с выходными данными, полученными на исходной программе. Следовательно, такие тесты не требуют перезапуска.
- Множество тестов, требующих повторного запуска. К ним относятся тесты, которые уже запускались, но требуют перезапуска, поскольку затрагивают, по крайней мере, один измененный покрываемый элемент, подлежащий повторному тестированию. При повторном выполнении такие тесты могут давать результат, отличный от результата, показанного на исходной программе. Множество тестов, требующих повторного запуска, обеспечивает хорошее покрытие структурных элементов даже при наличии новых функциональных возможностей.
- Множество устаревших тестов. Это тесты, более не применимые к измененной программе и непригодные для дальнейшего тестирования, поскольку они затрагивают только покрываемые элементы, которые были удалены при изменении программы. Их можно удалить из набора регрессионных тестов.
- Новые тесты, которые еще не запускались и могут быть использованы для тестирования.

Сразу после создания тест вводится в структуру базы данных как новый. После исполнения новый тест переходит в категорию тестов, пригодных для повторного использования либо устаревших. Если выполнение теста способствовало увеличению текущей степени покрытия кода, тест помечается как пригодный для повторного использования. В противном случае он помечается как устаревший и отбрасывается. Существующие тесты, повторно запущенные после внесения изменения в код, также классифицируются заново как пригодные для повторного использования или устаревшие в зависимости от тестовых траекторий и используемого критерия тестирования.

Классификация тестов по отношению к изменениям в коде требует анализа последствий изменений. Тесты, активирующие код, затронутый изменениями, могут требовать повторного запуска или оказаться устаревшими. Чтобы тест был включен в класс тестов, требующих повторного запуска, он должен быть затронут изменениями в коде, а также должен способствовать увеличению степени покрытия измененного кода по используемому критерию. Затронутым элементом теста может быть траектория, выходные значения, или и то, и другое. Чтобы тест был включен в класс тестов,

пригодных для повторного использования, он должен вносить вклад в увеличение степени покрытия кода и не требовать повторного запуска.
Степень покрытия кода определяется для тестов, пригодных для повторного использования, поскольку к этому классу относятся тесты, не требующие повторного запуска и способствующие увеличению степени покрытия до желаемой величины. Если имеется компонент программы, не задействованный пригодными для повторного использования тестами, то вместо них выбираются и выполняются с целью увеличения степени покрытия тесты, требующие повторного запуска. После запуска такой тест становится пригодным для повторного использования или устаревшим. Если тестов, требующих повторного запуска, больше не осталось, а необходимая степень покрытия кода еще не достигнута, порождаются дополнительные тесты и тестирование повторяется.
Окончательный набор тестов собирается из тестов, пригодных для повторного использования, тестов, требующих повторного запуска, и новых тестов. Наконец, устаревшие и избыточные тесты удаляются из набора тестов, поскольку избыточные тесты не проверяют новые функциональные возможности и не увеличивают покрытие.

Дополнительный материал.
Регрессионное тестирование: разновидности метода отбора тестов
Регрессионное тестирование: методики, не связанные с отбором тестов и методики порождения тестов
Регрессионное тестирование: алгоритм и программная система поддержки
Типы регрессии по Канеру?
Сэм Канер описал 3 основных типа регрессионного тестирования:

- Регрессия багов (Bug regression) — попытка доказать, что исправленная ошибка на самом деле не исправлена
- Регрессия старых багов (Old bugs regression) — попытка доказать, что недавнее изменение кода или данных сломало исправление старых ошибок, т.е. старые баги стали снова воспроизводиться.
- Регрессия побочного эффекта (Side effect regression) — попытка доказать, что недавнее изменение кода или данных сломало другие части разрабатываемого приложения

Объясните, что такое тестирование N+1?
Вариант регрессионного тестирования представлен как N+1. В этом методе тестирование выполняется в несколько циклов, в которых ошибки, обнаруженные в тестовом цикле «N», устраняются и повторно тестируются в тестовом цикле N + 1. Цикл повторяется, пока не будет найдено ни одной ошибки.
Что означает подтверждающее тестирование? (confirmation/re-testing)
Повторное тестирование - это тип тестирования, выполняемый для проверки того, что конкретный дефект устранен после исправления кода.
В чем разница между повторным и регрессионным тестированием?

- Регрессионное тестирование проводится для подтверждения того, что недавнее изменение программы или кода не оказало неблагоприятного воздействия на существующие функции. Повторное тестирование проводится для подтверждения того, что тест-кейсы, которые не прошли, проходят после устранения дефектов.

- Цель регрессионного тестирования подтвердить, что новые изменения кода не должны иметь побочных эффектов для существующих функций. Повторное тестирование проводится на основе исправлений дефектов.
- Проверка дефектов не является частью регрессионного тестирования. Проверка дефекта является частью повторного тестирования
- В зависимости от проекта и наличия ресурсов, регрессионное тестирование может проводиться параллельно с повторным тестированием. Приоритет повторного тестирования выше, чем регрессионное тестирование, поэтому оно проводится перед регрессионным тестированием.
- Вы можете сделать автоматизацию для регрессионного тестирования, ручное тестирование может быть дорогим и трудоемким.
- Регрессионное тестирование называется общим (generic) тестированием. Повторное тестирование - это плановое (planned) тестирование.
- Регрессионное тестирование проводится для пройденных Test case. Повторное тестирование проводится только для неудачных тестов.
- Регрессионное тестирование проверяет наличие неожиданных побочных эффектов. Повторное тестирование гарантирует, что первоначальная ошибка была исправлена.
- Регрессионное тестирование проводится только тогда, когда есть какие-либо изменения или изменения становятся обязательными в существующем проекте. Повторное тестирование выполняет дефект с теми же данными и той же средой с разными входными данными с новой сборкой (build).
- Test case для регрессионного тестирования могут быть получены из функциональной спецификации, user tutorials and manuals, а также defect reports в отношении исправленных проблем. Test case для повторного тестирования не могут быть получены до начала тестирования.

Что вы знаете о тестировании сборки? (Build Verification Test)
Тестирование, направленное на определение соответствия, выпущенной версии, критериям качества для начала тестирования. По своим целям является аналогом Дымового Тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию. Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии.
Что такое тестирование потоков? (Thread testing)
Тестирование потоков определяется как тип тестирования программного обеспечения, который проверяет основные функциональные возможности конкретной задачи (потока). Обычно проводится на ранней стадии фазы интеграционного тестирования. Тестирование на основе потоков является одной из дополнительных стратегий, принятых в ходе тестирования системной интеграции. Поэтому его, вероятно, следует более правильно назвать «тестом взаимодействия потоков» (thread interaction test). Тестирование на основе потоков подразделяется на две категории:
- Однопоточное тестирование включает одну транзакцию приложения за раз
- Многопоточное тестирование включает одновременно несколько активных транзакций

Как производить:
- Тестирование на основе потоков является обобщенной формой тестирования на основе сеансов (session-based testing), в котором сеансы являются формой потока, но поток не обязательно является сеансом.

- Для тестирования потока, поток или программа (небольшая функциональность) интегрируются и тестируются постепенно как подсистема, а затем выполняются для всей системы.
- На самом низком уровне оно предоставляет интеграторам лучшее представление о том, что тестировать.
- Вместо непосредственного тестирования программных компонентов требуется, чтобы интеграторы сосредоточились на тестировании логических путей выполнения в контексте всей системы.

Что такое тестирование документации? (Documentation testing)

Плохая документация может повлиять на качество продукта. Хорошая документация по продукту играет решающую роль в конечном продукте. Тестирование артефактов, разработанных до, во время и после тестирования продукта, называется тестированием документации. Это нефункциональный тип тестирования программного обеспечения. Мы знаем, что дефекты, обнаруженные на этапе тестирования, более дорогостоящие, чем если бы они были обнаружены на этапе требований. Стоимость исправления ошибки увеличивается тем больше, чем позже вы найдете ее. Таким образом, тестирование документации может начаться с самого начала процесса разработки программного обеспечения, чтобы сэкономить большую сумму денег.

Некоторые часто проверяемые артефакты:

- Requirement documents
- Test Plan
- Test case
- Traceability Matrix (RTM)

Какие вы знаете уровни тестирования данных?

Тесты группируются в зависимости от того, где они добавлены в SDLC, или от уровня детализации, который они содержат. В целом, существует четыре уровня или слоя тестирования: модульное тестирование, интеграционное тестирование, системное тестирование и приемочное тестирование. Целью уровней тестирования является систематизация тестирования программного обеспечения и простота выявления всех возможных Test case на определенном уровне.

Здесь работает хорошая аналогия с пирамидой тестирования. Это распределение по количеству тестов на разных уровнях приложения.

- Unit-слой — это когда тестируется один модуль программы, чаще всего это одна функция или метод. Таких тестов должно быть больше всего. Unit-тест для данных — это когда мы определяем требования для каждой ячейки. Нет смысла тестировать дальше, если у нас есть ошибки на уровне ячеек. Если, например, в фамилии содержатся цифры, то какой смысл проверять что-то дальше? Возможно, там должны быть буквы, похожие на эти цифры. И тогда нам нужно все исправлять и проверять следующий уровень, чтобы у нас все было в единственном числе и не было дубликатов, если так сказано в требованиях.
- Integration-слой — это когда несколько кусков программы тестируются вместе. Слой API для данных — это когда мы говорим о всей таблице. Допустим, у нас могут быть дубликаты, но не больше ста штук. Если у нас город-миллионник, то на одной улице не может жить миллион человек. Поэтому если мы сделаем выборку по улице, то количество адресов должно быть десять тысяч или тысяча — это надо определить. А если у нас миллион, то с данными что-то не так.
- System-слой — это когда вся программа тестируется полностью. В случае с данными этот слой означает, что тестируется вся система. Здесь включается

статистика. Например, мы говорим, что у нас не может быть больше 30% мужчин, рожденных после 1985 года. Или мы говорим, что 80% данных должны быть одного типа.

- Приемочное тестирование - это тест, проводимый для определения того, удовлетворяются ли требования спецификации или контракта в соответствии с его поставкой. Приемочное тестирование в основном выполняется пользователем или заказчиком. Тем не менее, другие акционеры могут быть вовлечены в этот процесс.

Что такое подкожный тест? (Subcutaneous test)
Тест, который выполняется не для конечного пользовательского интерфейса, а для интерфейса, расположенного чуть ниже поверхности (пример - API).

Доп. материал:
Subcutaneous Test
Расскажите о локализации, глобализации и интернационализации? (Localization/ globalization/internationalization testing)
Глобализированное ПО - это ПО, функционирующее одинаково независимо от географической, культурной и национальной среды. Тестирование глобализации концентрируется на выявлении потенциальных проблем в дизайне продукта, которые могут испортить глобализацию. Например, разработчик должен заложить в CSS основу для вертикального текста, если в будущем планируется локализовать продукт на язык с вертикальным письмом или обработку почтовых индексов для разных стран (где-то цифры, где-то цифры с буквами и т.п.). Оно гарантирует, что код может обрабатывать желаемую международную поддержку без нарушения какой-либо функциональности. А также, что не будет никакой потери данных и проблем с отображением.

Тестирование глобализации, в свою очередь, можно разделить на два подвида:

- Тестирование интернационализации (118N);
- Тестирование локализации (L10N);

(Цифра означает количество пропущенных букв, типа как k8s - kubernetes)

Интернализация ПО – это особый процесс, при котором веб-софт создается таким образом, чтобы оно было равноудаленным от какой-либо культуры и (или) специфики определенного географического региона.
Например, одна из задач по интернационализации ПО– корректное редактирование логики всех подключенных параметров форматирования (формат даты, времени, цифровое и валютное форматирование).
Также, тестировщики во время проверки на соответствие ПО требованиям 118N тестируют работу продукта на равномерность работы в разных регионах и культурах мира.
Основной задачей тестирования интернациональности является проверка того, может ли программный код работать со всей международной поддержкой без нарушения функциональности, что может привести к потере данных или проблемам целостности информации.
В основном, фокус тестирования интернациональности направлен на:

- Тестирование языковой совместимости. Проводятся проверки того, может ли веб-продукт корректно работать в определенной языковой среде;

- Проверка функциональности: полное выполнение регрессионных тестов в разнообразных языковых средах (корректное отображение специфической информации – даты, времени и валюты);
- Тестирование на корректность отображения графического интерфейса;
- Проверка удобства пользования: тестирование простоты использования ПО на различных операционных системах, разнообразных устройствах и прочее.

Локализация ПО – деятельность по модификации ПО в соответствии с определенными региональными настройками (языком, географической территорией, кодировкой и прочим), которая должна бесперебойно работать.
В данный вид проверки входит необходимость выполнения работ по переводу всего контента программного обеспечения для конечного пользователя. Во время перевода должны учитываться иконки, информационная графика, справочные материалы, техническая документация и иные культурные особенности регионов, где в будущем будет доступно это программное обеспечение.

На что обратить внимание:
- Длина переведенных слов
- Параметры шрифта пользовательского интерфейса
- Ввод текста в разных локализациях
- RTL-языки (справа-налево) или вертикальных
- Перевод сокращений или аббревиатур
- Мета-теги (проблемы с SEO или отображением имени вкладки (title, description, keywords))
- Соответствие мер исчисления, валюты, postal code и т.п.

Доп. материал:
Локализационное тестирование: зачем оно нужно приложению или сайту?
Почему интернационализация и локализация имеют значение
Гайд по тестированию локализации и интернационализации, а также большой и полезный checklist
Accelerate localization from code to delivery
Что такое исследовательское тестирование? (Exploratory testing)
Исследовательское Тестирование — одновременно является и техникой, и видом тестирования. Такое тестирование подразумевает под собой одновременно изучение проекта, функционала, проектирование тест кейсов в уме и тут же их исполнение, не записывая и не создавая тестовую документацию.
Такой вид тестирования обычно не предусматривается в тест плане и тест кейсы выполняются и модифицируются динамически. Эффективность такого тестирования напрямую зависит от опыта тестировщика ранее имевшим дело с этим приложением, платформой, знанием мест скопления возможных багов и рисками которые относятся к конкретному продукту.
Цель данного тестирования — это углубление в познании продукта, приложения и нахождения «на лету» возможных багов. Также такое тестирование помогает в дальнейшем проектировании тест кейсов для покрытия функционала данного приложения. Исследовательское тестирование широко используется в Agile-моделях.

Доп. материал:

Исследовательское тестирование: пустая трата времени или мощный инструмент?
Rapid Software Testing Methodology
Exploratory Testing
Exploratory Testing
Exploratory Testing
Exploratory Testing Dynamics
A Tutorial in Exploratory Testing
Plan your next exploratory testing session
Что вы знаете о турах Виттакера в исследовательском тестировании?
Чтобы систематизировать исследовательское тестирование можно использовать идею туров. Туры – это идеи и инструкции по исследованию программного продукта, объединенные определенной общей темой или целью. Туры, как правило, ограничены по времени – длительность тестовой сессии не должна превышать 4 часа. Идею туров развивали в своих работах Канер, Бах, Хендриксон, Болтон, Кохл и другие. Джеймс Виттакер (James A. Whittaker), хоть и не придумал саму идею туров, но предложил свой подход к исследовательскому тестированию с использованием туров и в своей книге “Exploratory Software Testing” в доступной форме озвучил идею туров и описал сами туры.
Тур – это своего рода план тестирования, он отражает основные цели и задачи, на которых будет сконцентрировано внимание тестировщика во время сессии исследовательского тестирования. При этом Виттакер использует метафору, что тестировщик – это турист, а тестируемое приложение – это город. Обычно у туриста (тестировщика) мало времени, поэтому он выполняет конкретную задачу в рамках выбранного тура, ни на что другое не отвлекаясь. Город (ПО) разбит на районы: деловой центр, исторический район, район развлечений, туристический район, район отелей, неблагополучный район.

Доп. материал:
Туры в исследовательском тестировании. Личный перевод из книги Д. Виттакера • «Исследовательское • тестирование ПО»
Переводы туров для исследовательского тестирования
Исследовательское тестирование и исследовательские туры Виттакера
Что такое Свободное или Интуитивное тестирование? (Adhoc)
Часто его путают с другим видом тестирования «Exploratory testing» – «Исследовательское тестирование».
Свободное тестирование (ad-hoc testing) – это вид тестирования, который выполняется без подготовки к тестированию продукта, без определения ожидаемых результатов, проектирования тестовых сценариев. Это неформальное, импровизационное тестирование. Оно не требует никакой документации, планирования, процессов, которых следует придерживаться при выполнении тестирования. Такой способ тестирования в большинстве случаев дает большее количество заведенных отчетов об ошибке. Это обусловлено тем, что тестировщик на первых шагах приступает к тестированию основной функциональной части продукта и выполняет как позитивные, так и негативные варианты возможных сценариев.
Чаще всего такое тестирование выполняется, когда владелец продукта не обладает конкретными целями, проектной документацией и ранее поставленными задачами. При этом тестировщик полагается на свое общее представление о продукте, сравнение с похожими продуктами, собственный опыт. Однако при тестировании

ad-hoc имеет смысл владеть общей информацией о продукте, особенно если проект очень сложный и большой. Поэтому нужно хорошее представление о целях проекта, его назначении и основных функциях, и возможностях. А дальше уже можно приступать к ad-hoc тестированию.
Виды свободного тестирования (ad-hoc testing):

- Buddy testing – процесс, когда 2 человека, как правило разработчик и тестировщик, работают параллельно и находят дефекты в одном и том же модуле тестируемого продукта. Такой вид тестирования помогает тестировщику выполнять необходимые проверки, а разработчику исправлять множество дефектов на ранних этапах.
- Pair testing – процесс, когда 2 тестировщика проверяют один модуль и помогают друг другу. К примеру, один может искать дефекты, а второй их документировать. Таким образом, у одного тестировщика будет функция, скажем так, обнаружителя, у другого – описателя.
- Monkey testing – произвольное тестирование продукта с целью как можно быстрее, используя различные вариации входных данных, нарушить работу программы или вызвать ее остановку (простыми словами – сломать).

Различия между Buddy testing и Pair testing:

- Buddy testing (Совместное тестирование) – это сочетание модульного тестирования и системного тестирования между разработчиком и тестировщиком.
- Pair testing (Парное тестирование) – выполняется только тестировщиками с разным уровнем знаний и опыта (такое сочетание поможет поделиться взглядами и идеями).

Основные преимущества ad-hoc testing:

- нет необходимости тратить время на подготовку документации;
- самые важные дефекты зачастую обнаруживаются на ранних этапах;
- часто применяется, когда берут нового сотрудника. С помощью этого метода, человек усваивает за 3 дня то, что, разбираясь тестовыми случаями, разбирал бы неделю – это называется форсированное обучение новых сотрудников;
- возможность найти трудновоспроизводимые и трудноуловимые дефекты, которые невозможно было бы найти, используя стандартные сценарии проверок.

Если нам нужно провести ad-hoc тестирование интернет-магазина, то этот краткий список может помочь с тем, что нужно проверить:

- все возможности сайта доступны без регистрации;
- корректность отображения анимаций и картинок;
- все возможности сайта доступны после регистрации;
- процесс регистрации;
- процесс добавления/удаления из корзины;
- процесс оплаты покупок;
- удобство в пользовании для новичков, простота, подсказки, помощь.

Что вы знаете о мутационном тестировании? (Mutation testing)
Mutation testing - это тип тестирования программного обеспечения, в котором мы мутируем (меняем) определенные выражения в исходном коде и проверяем, способны ли Test case найти ошибки. Это тип тестирования белого ящика, который в основном используется для модульного тестирования. Изменения в мутантной программе

сохраняются крайне небольшими, поэтому это не влияет на общую цель программы. Цель Mutation testing - оценить качество Test case, которые должны быть достаточно надежными, чтобы не выполнять мутантный код. Этот метод также называется стратегией тестирования на основе ошибок, так как он включает в себя создание ошибки в программе.

- Шаг 1: Ошибки вводятся в исходный код программы путем создания множества версий, называемых мутантами. Каждый мутант должен содержать одну ошибку, и цель состоит в том, чтобы заставить версию мутанта потерпеть неудачу, что демонстрирует эффективность Test case.
- Шаг 2: Test case применяются к исходной программе, а также к программе мутанта.
- Шаг 3: Сравните результаты оригинальной и мутантной программы.
- Шаг 4: Если исходная программа и программы-мутанты генерируют разные выходные данные, то этот мутант уничтожается by the Test case. Следовательно, Test case достаточно хорош, чтобы обнаружить изменение между оригинальной и мутантной программой.
- Шаг 5: Если исходная программа и программа-мутант генерируют одинаковые выходные данные, мутант остается в живых. В таких случаях необходимо создать более эффективные Test case, которые убивают всех мутантов.

Что изменить в программе мутантов? Есть несколько методов, которые могут быть использованы для создания мутантных программ:

- Операторы замены операндов (Operand replacement operators) – например, в условии if (x> y) поменять местами значения x и y
- Операторы модификации выражений (Expression Modification Operators) – например, в условии if (x == y) Мы можем заменить == на >=
- Операторы модификации операторов (Statement modification Operators) – например, удалить часть else в конструкции if-else или удалить целиком конструкцию if-else, чтобы проверить, как ведет себя программа

Оценка мутации = (убитые мутанты / общее количество мутантов) * 100

Автоматизированные инструменты для разных ЯП: mutmut, Humbug и Infection и т.п.

Доп. материал:
Мутационное тестирование
Что означает механизм тестирования по ключевым словам? (Keyword Driven testing Framework)
Это скриптовая техника, которая использует файлы данных, которые содержат ключевые слова, связанные с тестируемым ПО. Эти ключевые слова описывают набор действий, необходимых для выполнения определенного шага. Тест на основе ключевых слов состоит из ключевых слов высокого и низкого уровня, включая аргументы ключевых слов, которые составлены для описания действия Test case. Это также называется тестированием на основе таблиц (table-driven testing) или тестированием на основе action word (action word based testing). В тестировании по ключевым словам вы сначала идентифицируете набор ключевых слов, а затем связываете действие (или функцию), связанную с этими ключевыми словами. Здесь каждое действие тестирования, такое как открытие или закрытие браузера, щелчок мыши, нажатия клавиш и т. д., описывается ключевым словом, таким как openbrowser,

click, Typtext и т. д. Тестирование на основе ключевых слов обычно выполняется с помощью автоматического тестирования.
Что вы знаете о тестировании интерфейса прикладного программирования (API - Application Programming Interface)?
Каждый день используя любимые мобильные приложения и веб-ресурсы вы незаметно взаимодействуете с API, скрытым под интерфейсом пользователя.
API действует как интерфейс между двумя программными приложениями и позволяет им связываться друг с другом на оговоренных правилах и не залезая в реализацию предоставляемых функций. Простой пример: вы можете встроить на свою главную страницу сайта маленький виджет прогноза погоды, который будет отправлять определенный правилами запрос к API некоего сервиса погоды и получать определенный правилами ответ, содержащий посылку с данными.
Еще более простой пример: примите официанта в качестве API ресторана. В ресторане вы даете заказ на основе блюд, определенных в меню. Официант принимает ваш заказ и на этом ваше участие заканчивается и вам не интересно, что там произойдет дальше. От официанта вы ожидаете только итог – вам приносят заказанное блюдо.
Можете попробовать взаимодействие с API сами: отправляете GET запрос на https://reqres.in/api/users, и получаете в ответ список пользователей. Это очень удобно, когда вы хотите предоставить интерфейс взаимодействия со своим сервисом сторонним лицам. Например, у google, instagram, vk и, в общем-то, всех популярных продуктов есть открытая часть API. То есть у вас есть документ с перечнем того, что и как можно спросить и что вам на это придет в ответ. Взаимодействовать с API можно и с веб-страницы, и с помощью специальных инструментов и напрямую из кода. Помимо этого, API еще чаще используется для внутренних нужд как архитектурное решение для связи между сервисами или вообще представлять собой не веб-взаимодействие, а решение внутри кода одного продукта (таким образом, гипотетически API может быть и у десктопного приложения и практически где угодно).
Тестирование API - это тип тестирования который включает в себя тестирование API напрямую, а также в рамках интеграционного тестирования, чтобы проверить, соответствует ли API ожиданиям с точки зрения функциональности, надежности, производительности и безопасности приложения. В тестировании API наш основной упор будет сделан на уровне бизнес-логики архитектуры программного обеспечения.

Мнение:
“Тестирование API - не какой-то отдельный вид или тип тестирования, это просто еще один способ взаимодействия с системой. В случае с UI у вас есть, условно, страница набором полей, которые вы заполняете, отправляете и ждете реакции от системы. В случае с API у вас есть эндпоинт и набор параметров. Посылаете запрос -> получаете ответ -> валидируете ответ. Часть из этих тестов будет негативными, часть перейдет в "контрактное" тестирование, часть уйдет в тестирование валидации. Вся остальная логика тестирования - про приоритеты, техники тест-дизайна и прочее остается неизменным.

С чего начать.
- Почитать про инструменты, что это такое и как оно работает.

- Потренироваться на любом открытом API посылать/получать/обрабатывать запросы любым удобным вам способом - хоть с помощью python (разобраться можно за 2-3 вечера неспешного изучения с нуля), хоть с помощью GUI инструментов, хоть с помощью CURL.
- Получить список функциональности API:
- - Кто является "клиентом" для этого АПИ (использует его ваше приложение/фронтенд или вы его отдаете во вне).
- - Какие функции доступны через API.
- Для списка функциональности выше составить список функциональных тест-кейсов, начиная с того, что представляет большую бизнес значимость.
- Для того же списка функциональности получить все нужные данные:
- - Эндпоинты (если они есть в вашем случае).
- - Схему авторизации.
- - Документацию (если есть) или описание работы.
- Дополнить список функциональных проверок из п.3 более подробными тестами про схему взаимодействия, авторизацию, тестирование контракта, валидацию значений и пр.
- Начать писать запросы следуя списку тест-кейсов из п.3 и 4 с помощью инструмента, который выбрали в п.1”

Типы тестирования API:

- Unit testing: Для проверки функциональности отдельной операции
- Functional testing: Чтобы проверить функциональность более широких сценариев с помощью блока результатов unit-тестирования, протестированных вместе
- Load testing: Чтобы проверить функциональность и производительность под нагрузкой
- Runtime/Error Detection: Мониторинг приложения для выявления проблем, таких как исключения и утечки ресурсов
- Security testing: Чтобы гарантировать, что реализация API защищена от внешних угроз
- UI testing: Это выполняется как часть end-to-end integration тестов, чтобы убедиться, что каждый аспект пользовательского интерфейса функционирует должным образом.
- Interoperability and WS Compliance testing: Совместимость и WS Compliance testing - это тип тестирования, который применяется к SOAP API. Функциональная совместимость между API-интерфейсами SOAP проверяется путем обеспечения соответствия профилям функциональной совместимости веб-служб. Соответствие WS-* проверено, чтобы гарантировать, что стандарты, такие как WS-Addressing, WS-Discovery, WS-Federation, WS-Policy, WS-Security и WS-Trust, должным образом реализованы и используются
- Penetration testing: Чтобы найти уязвимости при атаках злоумышленников
- Fuzz testing: Для проверки API путем принудительного ввода в систему некорректных данных для попытки принудительного сбоя

Примеры для тестирования API:

- Возвращаемое значение на основе входных условий: его относительно легко проверить, поскольку входные данные могут быть определены и результаты могут быть проверены.

- Ничего не возвращает: при отсутствии возвращаемого значения проверяется поведение API в системе.
- Вызов другого API / события / прерывания: если выход API инициирует какое-либо событие или прерывание, то эти события и прерывания listeners следует отслеживать
- Обновление структуры данных: Обновление структуры данных будет иметь определенный эффект или влияние на систему, и это должно быть проверено
- Изменение определенных ресурсов: если вызов API изменяет некоторые ресурсы, его следует проверить путем доступа к соответствующим ресурсам.

Доп. материал:
Курс Тестирование ПО. Занятие • 29• . Тестирование API | QA START UP
От шока до принятия: пять стадий тестирования API
Тестирование API
Swagger/OpenAPI Specification • как основа для ваших приемочных тестов
История одного сервера и тестировщика Васи
What Is an API?
Тестирование API простыми словами за • 8 • минут / Тестировщик API
Как протестировать API без документации/черным ящиком?
Если Вам по какой-то причине предстоит проделать эту неблагодарную работу, определитесь, насколько все плохо и какая у Вас есть информация об объекте тестирования.

Известно ли какие порты для Вас открыты? Знаете ли Вы нужные endpoints?
Если дело совсем плохо - просканируйте порты, например, с помощью netcat. Открытые порты сохраните в файл.
Эта операция займет довольно много времени. Можете почитать советы по работе с Nmap и Netcat.
Если Вам известен нужный порт и соответствующий endopoint - переберите все возможные HTTP методы. Начните с наиболее очевидных POST, PUT, GET. Для ускорения процесса напишите скрипт, например, на Python.

В худшем случае, когда ни порт ни endpoints неизвестны Вам, скорее всего придется перебирать все открытые порты и генерировать endpoints, которые подходят по смыслу.

Разработчики обычно не особо заморачиваются и закладывают минимально-необходимую информацию. Так что включите воображение и попробуйте придумать endpoints опираясь на бизнес логику и принятые в Вашей компании стандарты.

Если ни endpoints ни бизнес логика Вам неизвестны, то у меня есть подозрение, что Вы тестируете API с не самыми хорошими намерениями.

Тестирование клиентской части и серверной, в чем разница? (Frontend testing Vs. Backend testing?)

Frontend testing - это тип тестирования, который проверяет уровень представления 3-уровневой архитектуры. С точки зрения непрофессионала, вы проверяете GUI - все,

что видно на экране, на стороне клиента. Для веб-приложения интерфейсное тестирование будет включать проверку функциональных возможностей, таких как формы, графики, меню, отчеты и т. д., а также связанный Javascript. Frontend testing - это термин, охватывающий различные стратегии тестирования, включая оценку производительности фронтенда, которая является хорошей практикой перед тестированием приложения с высокими пользовательскими нагрузками. Тестировщик должен хорошо понимать бизнес-требования для выполнения этого типа тестирования. Ранее оптимизация производительности означала оптимизацию на стороне сервера. Это было связано с тем, что большинство веб-сайтов были в основном статичными, а большая часть обработки выполнялась на стороне сервера. Однако сегодня веб-приложения становятся более динамичными и в результате код на стороне клиента нередко становится причиной низкой производительности. Тестирование клиентской части невозможно в некоторых случаях: бэкенд разрабатывают быстрее, чем фронтенд; очевидно, если клиентская часть отсутствует в принципе (самодостаточное приложение, терминальная команда).
Backend testing - это тип тестирования, который проверяет уровень приложений и базы данных 3-уровневой архитектуры. В сложном программном приложении, таком как ERP, внутреннее тестирование повлечет за собой проверку бизнес-логики на уровне приложений. Для более простых приложений бэкэнд-тестирование проверяет серверную часть или базу данных. Это означает, что данные, введенные в интерфейс, будут проверены в базе данных. Базы данных проверяются на наличие свойств ACID, операций CRUD, их схемы, соответствия бизнес-правилам. Базы данных также проверяются на безопасность и производительность. Производится проверка целостности данных, Проверка достоверности данных, Тестирование функций, процедур и триггеров. При внутреннем тестировании нет необходимости использовать графический интерфейс. Вы можете напрямую передавать данные с помощью браузера с параметрами, необходимыми для функции, чтобы получить ответ в некотором формате по умолчанию. Например, XML или JSON. Вы также подключаетесь к базе данных напрямую и проверяете данные с помощью SQL-запросов.

Доп. материал:
Как найти границы на клиенте и сервере
Как Иван ошибку в бэкенде локализовывал
Круглый стол • HYPERLINK "https://www.youtube.com/watch?v=XSeoWpSlcig• &• feature=youtu.be• &• ab_channel=PodlodkaPodcast"• "• HYPERLINK "https://www.youtube.com/watch?v=XSeoWpSlcig• &• feature=youtu.be• &• ab_channel=PodlodkaPodcast"• Почему не стоит тестировать бэкенд руками• HYPERLINK "https://www.youtube.com/watch?v=XSeoWpSlcig• &• feature=youtu.be• &• ab_channel=PodlodkaPodcast"• "
Что подразумевают под эталонным тестированием? (Baseline testing)
Это подход к тестированию, в котором за точку отсчета берется базовая линия - это показатель конкретного ориентира, который служит основой для нового тестирования. В базовом тестировании тесты собирают и сохраняют все результаты, полученные в исходном коде, и сравнивают с эталонным базовым уровнем. Этот базовый уровень относится к последним принятым результатам испытаний. Если в исходном коде есть новые изменения, то для повторного выполнения тестов необходимо сформировать текущий базовый уровень. Если последние результаты будут приняты, то текущая

базовая линия станет эталонной. По большей части Baseline testing относят к тестированию производительности.

В чем разница между Baseline и Benchmark testing?

- Baseline предназначено для оценки производительности приложения. Benchmark сравнивает производительность приложения с отраслевым стандартом.
- Baseline тестирование использует данные, собранные для повышения производительности. Benchmark возвращает информацию о целевом приложении по сравнению с другими приложениями.
- Baseline тестирование сравнивает текущую производительность с предыдущей производительностью приложения, тогда как Benchmark сравнивает производительность нашего приложения с производительностью конкурентов.

Что такое параллельное/многопользовательское тестирование? (Concurrency/Multi-user testing)

Параллельное тестирование определяется как метод тестирования для обнаружения дефектов в приложении, когда в систему вошли несколько пользователей. Другими словами, отслеживание эффекта, когда несколько пользователей выполняют одно и то же действие одновременно. Параллельное тестирование также называется многопользовательским тестированием. Тестирование параллельной программы является более сложной задачей, чем тестирование последовательной программы, из-за недетерминированности и проблем синхронизации. Зачем оно нужно:

- Определяет влияние одновременного доступа к одним и тем же записям базы данных, модулям или коду приложения.
- Определяет и измеряет уровень взаимоблокировки, блокировки и использования однопоточного кода и ограничения доступа к общим ресурсам

Как вы думаете, что такое тестирование на переносимость?

Тестирование переносимости — это тип тестирования программного обеспечения, который проводится для определения степени легкости или сложности, с которой программное приложение может быть эффективно и эффективно перенесено с одного аппаратного обеспечения, программного обеспечения или среды на другое. Результаты тестирования переносимости представляют собой измерения того, насколько легко программный компонент или приложение будут интегрированы в среду, и затем эти результаты будут сравниваться с нефункциональным требованием переносимости программной системы. Измерение основано на сравнении стоимости адаптации программного обеспечения к новой среде и стоимости реконструкции.

Атрибуты тестирования переносимости:

- Адаптивность: Адаптируемость определяется как способность программного приложения адаптироваться к конкретной среде без каких-либо усилий. Общие стандарты связи между несколькими системами помогают повысить адаптивность системы в целом.
- Installability: Устанавливаемость определяется как способность программного приложения быть установленным в желаемой среде без использования дополнительных ресурсов. Устанавливаемость выполняется на программном обеспечении, которое должно быть установлено в целевой среде.
- Заменяемость: Возможность замены определяется как способность программного приложения заменять другое программное обеспечение в конкретной среде. Приложение, которое заменяет предыдущее приложение, должно давать одинаковые результаты во всех целевых средах.

- Сосуществование: Сосуществование определяется как способность программного приложения работать с другим программным приложением в системе, не мешая друг другу и совместно используя один и тот же ресурс. Специально это тестирование используется в больших системах, которые включают в себя несколько подсистем.

Что такое тестирование графического интерфейса/визуальное тестирование? (GUI - Graphical User Interface testing)

Существует два типа интерфейсов для компьютерного приложения. Интерфейс командной строки, где вы вводите текст, и компьютер отвечает на эту команду и GUI - графический интерфейс пользователя, где вы взаимодействуете с компьютером, используя графическое представление, а не текст.

Цель тестирования графического интерфейса пользователя (GUI) - проверить функциональность интерфейса пользователя.

Примеры:

- Тестирование размера, положения, ширины, высоты элементов.
- Тестирование сообщений об ошибках, которые отображаются.
- Тестирование разных разделов экрана.
- Проверка шрифта, читаемый ли он или нет.
- Тестирование экрана в разных разрешениях с помощью увеличения и уменьшения масштаба, например, 640 x 480, 600x800 и т. д.
- Проверка выравнивания текстов и других элементов, таких как значки, кнопки и т. д. , находятся на своем месте или нет.
- Тестирование цветов шрифтов.
- Проверка цветов сообщений об ошибках, предупреждающих сообщений.
- Проверка, имеет ли изображение хорошую четкость или нет.
- Тестирование выравнивания изображений.
- Проверка орфографии.
- Пользователь не должен разочаровываться при использовании системного интерфейса.
- Тестирование, является ли интерфейс привлекательным или нет.
- Тестирование полос прокрутки в соответствии с размером страницы, если таковые имеются.
- Тестирование отключенных полей, если таковые имеются.
- Тестирование размера изображений.
- Проверка заголовков, правильно ли они выровнены или нет.
- Тестирование цвета гиперссылки.

Визуальное тестирование проверяет корректность отображения пользователю web-сайта, мобильного или десктопного приложения, дизайн-системы, PDF-файла или отдельного изображения на наличие расхождений с спецификацией дизайна (рендеринг страниц, шрифтов, изображений и т. д.). Раньше выполнялось вручную, т.к., например, в случае веб-сайта классическое автоматизированное тестирование было бесполезно – большинство инструментов сверялись с DOM и не видели те ошибки, что человек мог увидеть вживую. Сейчас визуальное тестирование выполняется автоматизированными инструментами создания скриншотов и сверки их с эталоном. Там все еще очень много нюансов, но это уже не относится к статье о ручном тестировании.

Доп. материал:
Эффективное тестирование верстки
Что такое A/B тестирование?
A / B-тестирование также называется сплит-тестированием (split). При тестировании AB мы создаем и анализируем два варианта приложения, чтобы найти, какой вариант работает лучше с точки зрения пользовательского опыта, потенциальных клиентов, конверсий или любой другой цели, а затем в конечном итоге сохранить наиболее эффективный вариант.
Давайте попробуем понять это на примере. Предположим, у нас есть интернет магазин и каталог отображается определенным образом. В какой-то момент (новые маркетинговые исследования/пожелания клиента и т. д.) решено изменить дизайн выдачи товаров в каталоге. Независимо от того, сколько проведено анализа, выпуск нового пользовательского интерфейса будет большим изменением и может иметь неприятные последствия.
В этом случае мы можем использовать A / B-тестирование. Мы создадим интерфейс нового варианта и выпустим его для некоторого процента пользователей. Например - мы можем распределить пользователей в соотношении 50:50 или 80:20 между двумя вариантами - A и B. После этого в течение определенного периода времени мы будем наблюдать эффективность обоих вариантов. Таким образом, тестирование A/B помогает принять решение о выборе лучшего варианта.

Доп. материал:
Ошибки в дизайне A/B тестов, которые я думала, что никогда не совершу
Что означает сквозное тестирование? (E2E - End–to–End)
Сквозное тестирование - это стратегия тестирования для выполнения тестов, которые охватывают все возможные потоки приложения от его начала до конца; проверяет программную систему вместе с ее интеграцией с внешними интерфейсами. Целью сквозного тестирования является создание полного производственного сценария, выявление программных зависимостей и утверждение, что между различными программными модулями и подсистемами передается правильный ввод. Сквозное тестирование обычно выполняется после функционального и системного тестирования. Оно использует реальные данные, такие как данные и тестовая среда, для имитации настроек в реальном времени. Сквозное тестирование также называется цепным тестированием (Chain testing).
Для чего оно нужно? Современные программные системы являются сложными и взаимосвязаны с несколькими подсистемами. Подсистема может отличаться от текущей системы или может принадлежать другой организации. Если какая-либо из подсистем выйдет из строя, вся система программного обеспечения может рухнуть. Это серьезный риск, и его можно избежать путем сквозного тестирования.
В чем разница между E2E и системным тестированием?

End to End testing System testing
Проверяет программную систему, а также взаимосвязанные подсистемы Проверяет только программную систему в соответствии со спецификациями требований.
Проверяет весь E2E flow Проверяет функциональные возможности и функции системы.
Все интерфейсы, бэкэнд-системы Функциональное и нефункциональное тестирование

Выполняется после завершения System testing Выполняется после завершения Integration testing
Сквозное тестирование включает проверку внешних интерфейсов, которые могут быть сложными для автоматизации. Следовательно, ручное тестирование является предпочтительным. Как ручное, так и автоматическое могут быть выполнены для тестирования системы

Что такое параллельное тестирование? (Parallel testing)
Это тип тестирования ПО, который одновременно проверяет несколько приложений или подкомпонентов одного приложения, чтобы сократить время выполнения теста. При параллельном тестировании тестировщик запускает две разные версии программного обеспечения одновременно с одним и тем же вводом. Цель состоит в том, чтобы выяснить, ведут ли себя прежняя система и новая система одинаково или по-разному. Это гарантирует, что новая система достаточно способна для эффективной работы программного обеспечения.

Пример: когда какая-либо организация переходит от старой системы к новой, legacy является важной частью. Передача этих данных является сложным процессом. При тестировании программного обеспечения проверка совместимости вновь разработанной системы со старой системой осуществляется посредством «параллельного тестирования».
Это Parallel testing Это НЕ Parallel testing
- Тестирование обновленного приложения по сравнению с предыдущим приложением.
- Запуск старого сценария с новым программным обеспечением с зарезервированными условиями ввода.
- Цель состоит в том, чтобы узнать, соответствует ли результат предыдущей системе.
- Должен иметь знания о старой и недавно разработанной системе • Тестирование только одного ПО
- Кросс-браузерное или кроссплатформенное тестирование.
- Цель состоит в том, чтобы выяснить проблему проектирования.
- Знать разницу не обязательно.

----- Тест дизайн -----
Тест дизайн? (Test Design)
Этап процесса тестирования ПО, на котором проектируются и создаются Test case (тест кейсы), в соответствии с определенными ранее критериями качества и целями тестирования. План работы над тест дизайном:
- анализ имеющихся проектных артефактов: документация (спецификации, требования, планы), модели, исполняемый код и т. д.
- написание спецификации по тест дизайну (• Test Design Specification)
- проектирование и создание • Test case

Роли, ответственные за тест дизайн:
- Тест аналитик - определяет "ЧТО тестировать?"
- Тест дизайнер - определяет "КАК тестировать?"

Попросту говоря, задача тест аналитиков и дизайнеров сводится к тому, чтобы используя различные стратегии и техники тест дизайна, создать набор Test case, обеспечивающий оптимальное тестовое покрытие тестируемого приложения. Однако, на большинстве проектов эти роли не выделяется, а доверяется обычным тестировщикам, что не всегда положительно сказывается на качестве тестов, тестировании и, как из этого следует, на качестве ПО (конечного продукта).
Перечислите известные техники тест-дизайна?

- Статические (Static):
- Review:
- Неформальное ревью (Informal review)
- Прохождение (Walkthrough)
- Техническое ревью (Technical Review)
- Инспекция (Inspection)
- Статический анализ (Static Analysis):
- Поток данных (Data Flow)
- Поток управления (Control Flow)
- Путь (Path)
- Стандарты (Standards)
- Динамические (Dynamic):
- Белый ящик (White-box)
- Выражение (Statement)
- Решение (Decision)
- Ветвь (Branch)
- Условие (Condition)
- Конечный автомат (FSM)
- Основанные на опыте (Experience-based):
- Предугадывание ошибки (Error Guessing - EG)
- Исследовательское тестирование (Exploratory testing)
- Ad-hoc testing
- Черный ящик (Black-box):
- Эквивалентное Разделение (Equivalence Partitioning - EP)
- Анализ Граничных Значений (Boundary Value Analysis - BVA)
- Комбинаторные техники (• Combinatorial Test Techniques)
- Переходы между состояниями (State transition)
- Случаи использования (Use case testing)

Альтернативный источник:
Specification-Based Testing Techniques (or Black Box techniques):
Equivalence Partitioning
Boundary Value Analysis
Combinatorial Test Techniques:
All Combinations
Pairwise Testing
Each Choice Testing
Base Choice Testing
Decision Table Testing
Classification Tree Method
State Transition Testing

Cause-Effect Graphing
Scenario Testing
Random Testing
Syntax Testing
Structure-Based Testing Techniques (or White Box techniques):
Statement Testing
Decision Testing
Condition Testing:
Branch Condition Testing
Branch Condition Combination Testing
Modified Condition Decision Coverage (MCDC) Testing
Data Flow Testing
Experience-Based Testing Techniques:
Error Guessing

Все методы тестирования на основе спецификаций (черного ящика) могут быть удобно описаны и систематизированы с помощью следующей таблицы:

Группа Техника Когда используется
Элементарные методы:
- сфокусированы на анализе входных / выходных параметров - могут быть объединены для обеспечения лучшего покрытия - обычно не используют и не зависят от других приемов Equivalence Partitioning Входные и выходные параметры имеют большое количество возможных значений
Boundary Value Analysis Значения параметров имеют явные (например, четко определенные в документации) границы и диапазоны или неявные (например, известные технические ограничения) границы
Комбинаторные стратегии:
- объединить возможные значения нескольких входных / выходных параметров - можно использовать элементарные приемы, чтобы уменьшить количество возможных значений All Combinations Количество возможных комбинаций входов достаточно мало, или каждая отдельная комбинация входов приводит к определенному выходу
Pairwise Testing Количество входных комбинаций чрезвычайно велико и должно быть сведено к приемлемому набору cases
Each Choice Testing У вас есть функциональность, при которой конкретное значение параметра чаще вызывает ошибку чем комбинация значений
Base Choice Testing Вы можете выделить набор значений параметров, которые имеют наибольшую вероятность использования
Продвинутые техники: - помочь проанализировать Систему с точки зрения бизнес-логики, иерархических отношений, сценариев и т. д. - анализ основан на данных, организованных в виде таблиц, диаграмм и шаблонов - может полагаться на элементарные и комбинаторные методы для разработки Test case Decision Table Testing Существует набор комбинаций параметров и их выводов, описываемых бизнес-правилами или другими правилами.
Classification Tree Method У вас есть иерархически структурированные данные, или данные могут быть представлены в виде иерархического дерева

	State Transition Testing	В функциональности есть очевидные состояния, у которых переходы регулируются правилами (например, потоками)
	Cause-Effect Graphing	Причины (входы) и следствия (выходы) связаны большим количеством сложных логических зависимостей
	Scenario Testing	Есть четкие сценарии в функционале
Другие	Random Testing	Вы должны подражать непредсказуемости реальных входных данных, или функциональность имеет несистематические дефекты
	Syntax Testing	Функциональность имеет сложный синтаксический формат для ввода (например, коды, сложные имена электронной почты и т. д.)

Методы тестирования на основе структуры (Structure-Based Testing Techniques): также известны как методика тестирования White Box, это означает, что мы знакомы с кодом, который собираемся тестировать. Чтобы понять эти методы, мы должны определить, что такое покрытие в контексте разработки теста. Вот хорошее определение из Книги ISTQB: Тестирование покрытия определенным образом измеряет количество тестов, выполненных набором тестов (полученных другим способом, например, с использованием методов, основанных на спецификациях). Везде, где мы можем посчитать вещи и сказать, была ли каждая из этих вещей проверена каким-либо тестом, мы можем измерить охват. Основная мера покрытия:

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

где «coverage item» - это то, что мы смогли подсчитать и посмотреть, выполнил ли тест этот элемент или использовал его.
Для методов, основанных на спецификациях, это могут быть случаи использования, разделы эквивалентности, граничные значения, состояния из диаграммы перехода состояний, процент бизнес-правил из таблицы решений и т. д. Для методов, основанных на структуре, элементы покрытия представлены структурные элементы кода. В принципе, оценка покрытия означает, что мы должны решить, какие структурные элементы мы будем использовать (например, заявления или решения). Затем найдите общее количество этих элементов в коде. Введите дополнительные операторы (например, ведение журнала) рядом с каждым структурным элементом, чтобы выяснить, использовался ли этот элемент во время выполнения Test case. И, наконец, измерьте охват, выполнив тесты и используя формулу, упомянутую выше. Но, как правило, вы не должны думать обо всех этих шагах, потому что есть много автоматизированных инструментов для измерения покрытия. Методы структурного тестирования обычно подразумевают, что вы должны измерить покрытие для существующих наборов тестов (включая наборы черного ящика), а затем разработать дополнительные Test case белого ящика, основанные на элементах структурного кода, для достижения максимально возможного покрытия.

Доп. материал:
Тест дизайн методом Interface • —• • Model • —• • State
Paradigms Paradigms of Black Box Software Software Testing Testing
Test Design: A Survey of Black Box Software Testing Techniques
Что такое статическое тестирование, когда оно начинается и что оно охватывает?

При статическом тестировании код не выполняется. Вы вручную проверяете код, документы требований и проектные документы на наличие ошибок. Отсюда и название «статичный». Основная цель этого тестирования - повысить качество программных продуктов путем выявления ошибок на ранних этапах цикла разработки. Это тестирование также называется Non-execution техникой или verification. Проверки могут также производиться автоматически (например, используя линтеры).
В статическом тестировании проверяются следующее:

- Unit Test cases
- Business Requirements Document (BRD)
- Use Cases
- System/Functional Requirements
- Prototype
- Prototype Specification Document
- DB Fields Dictionary Spreadsheet
- Test Data
- Traceability Matrix Document
- User Manual/Training Guides/Documentation
- Test Plan Strategy Document/Test cases
- Automation/Performance Test Scripts

Что такое динамическое тестирование, когда оно начинается и что оно охватывает?
При динамическом тестировании выполняется код. Оно проверяет функциональное поведение ПО, использование памяти / процессора и общую производительность системы. Основная цель этого тестирования - подтвердить, что программный продукт работает в соответствии с требованиями бизнеса. Это тестирование также называется Execution technique или validation. Динамическое тестирование выполняется на всех уровнях тестирования, и это может быть либо тестирование черного, либо белого ящика.
Виды динамического тестирования: Модульное тестирование. Интеграционное тестирование: Системное тестирование. Кроме того, нефункциональное тестирование, такое как производительность, тестирование безопасности.
На примере: Предположим, мы тестируем страницу входа в систему, где у нас есть два поля с именем «Имя пользователя» и «Пароль», а имя пользователя ограничено буквенно-цифровым форматом. Когда пользователь вводит имя пользователя «VA610», система принимает это. Но когда пользователь вводит «VA610 @ 123», тогда приложение выдает сообщение об ошибке. Этот результат показывает, что код действует динамически на основе пользовательского ввода.
Какие виды Review вы знаете?

- Неофициальные reviews: это один из видов рецензирования, который не сопровождается каким-либо процессом поиска ошибок в документе. При использовании этого метода вы просто просматриваете документ и оставляете неофициальные комментарии к нему.
- Технические reviews: команда, состоящая из ваших коллег, изучает технические характеристики программного продукта и проверяет, подходят ли он для проекта. Они пытаются найти любые несоответствия в спецификациях и стандартах. Этот обзор концентрируется главным образом на технической документации, связанной с программным обеспечением, такой как: Стратегия тестирования, План тестирования и спецификации требований.

- Пошаговое руководство. Автор рабочего продукта объясняет продукт своей команде. Участники могут задавать вопросы, если таковые имеются. Встреча проводится автором.
- Инспекция: Основная цель - найти дефекты, а встречу проводит обученный модератор. Этот обзор является формальным типом обзора, где следует строгий процесс поиска дефектов. У рецензентов есть контрольный список для проверки рабочих продуктов. Они фиксируют дефект и информируют участников об устранении этих ошибок.
- Code Walk Through: Это неформальный анализ исходного кода программы для выявления дефектов и проверки методов кодирования.

Что вы знаете о Data Flow testing?

Тестирование потока данных - это еще один набор методов / стратегий белого ящика, который связан с анализом потока управления, но с точки зрения жизненного цикла переменной. Переменные определяются, используются и уничтожаются, когда в них больше нет необходимости. Аномалии в этом процессе, такие как использование переменной без ее определения или после ее уничтожения, могут привести к ошибке. Существуют условные обозначения, которые могут помочь в описании последовательных во времени пар в жизненном цикле переменной:

- ~ - переменная еще не существует или предыдущий этап был последним
- d - определено, создано, инициализировано
- k - не определено, убито
- u - используется (c - использование вычислений; p - использование предикатов)

Таким образом, ~ d, du, kd, ud, uk, uu, k ~, u ~ являются вполне допустимыми комбинациями, когда ~ u, ~ k, dd, dk, kk, ku, d ~ являются аномалиями, потенциальными или явными ошибками. В настоящее время практически все они эффективно обнаруживаются компиляторами или, по крайней мере, IDE, и нам редко требуется выполнять статический анализ для обнаружения этих аномалий. То же самое относится и к динамическому анализу, который сфокусирован на исследовании / выполнении du пар - современные языки программирования снижают вероятность возникновения проблем, связанных с du. Так что в настоящее время такая проверка в основном не стоит усилий.

Что вы знаете о Control Flow testing?

Тестирование потоков управления (Control Flow Testing) - это одна из двух техник тестирования белого ящика, основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей. Фундаментом для тестирования потоков управления является построение графов потоков управления (Control Flow Graph), основными блоками которых являются:

- блок процесса - одна точка входа и одна точка выхода
- точка альтернативы - одна точка входа, две и более точки выхода
- точка соединения - две и более точек входа, одна точка выхода

При тестировании потока управления определяются различные уровни покрытия тестами. Под «покрытием» мы подразумеваем процент кода, который был протестирован, по сравнению с тем, который есть для тестирования. В тестировании потока управления мы определяем покрытие на нескольких различных уровнях. (Обратите внимание, что эти уровни охвата представлены не по порядку. Это связано с тем, что в некоторых случаях проще определить более высокий уровень охвата, а затем определить более низкий уровень охвата с точки зрения более высокого.):

Уровень Название Краткое описание
Уровень 0 -- “Тестируй все что протестируешь, пользователи протестируют остальное” На английском языке это звучит намного элегантнее: “Test whatever you test, users will test the rest”
Уровень 1 Покрытие операторов Каждый оператор должен быть выполнен как минимум один раз.
Уровень 2 Покрытие альтернатив / Покрытие ветвей Каждый узел с ветвлением (альтернатива) выполнен как минимум один раз.
Уровень 3 Покрытие условий Каждое условие, имеющее TRUE и FALSE на выходе, выполнено как минимум один раз.
Уровень 4 Покрытие условий альтернатив Тестовые случаи создаются для каждого условия и альтернативы
Уровень 5 Покрытие множественных условий Достигается покрытие альтернатив, условий и условий альтернатив (Уровни 2, 3 и 4)
Уровень 6 “Покрытие бесконечного числа путей” Если, в случае зацикливания, количество путей становится бесконечным, допускается существенное их сокращение, ограничивая количество циклов выполнения, для уменьшения числа тестовых случаев.
Уровень 7 Покрытие путей Все пути должны быть проверены

Подробный разбор с примерами доступен в источнике:
flylib.com/books/en/2.156.1/control_flow_testing.html
Что такое Loop coverage?
Loop testing определяется как тип тестирования программного обеспечения методом белого ящика, который полностью фокусируется на валидности конструкций цикла. Это одна из частей тестирования структуры управления (Control Structure testing): тестирование пути, проверка данных, тестирование условий (path testing, data validation testing, condition testing).
Этот показатель сообщает, выполняли ли вы каждое тело цикла ноль раз, ровно один раз и более одного раза (последовательно). Для циклов do-while покрытие цикла сообщает, выполняли ли вы тело ровно один раз и более одного раза. Ценным аспектом этой метрики является определение того, выполняются ли циклы while и for более одного раза, т.к. эта информация не сообщается другими метриками.

Что такое Race coverage?
Этот показатель показывает, выполняет ли несколько потоков один и тот же код одновременно. Это помогает обнаружить сбой при синхронизации доступа к ресурсам. Это полезно для тестирования многопоточных программ, например, в операционной системе.
Тестирование пути и тестирование базового пути? (Path testing & Basis Path testing)
Path testing - это метод структурного тестирования, который включает использование исходного кода программы для нахождения каждого возможного исполняемого пути. Это помогает определить все ошибки, лежащие в части кода. Здесь Test case выполняются таким образом, что каждый путь проходится по крайней мере один раз. Все возможные control paths, включая все loop paths. Test case подготавливаются на основе логической сложности. При таком типе тестирования каждый оператор в программе гарантированно выполняется как минимум один раз. Flow Graph, Cyclomatic Complexity и Graph Metrics используются для достижения basis path.

Любая программа включает в себя несколько точек входа и выхода. Тестирование каждого из этих пунктов является сложным и трудоемким. Для сокращения избыточных тестов и достижения максимального охвата тестов используется Basis Path testing. Basis Path testing такое же, но оно основано на методе White Box testing и определяет Test case на основе потоков или логического пути, которые могут быть пройдены через программу. В программной инженерии Basis Path testing включает в себя выполнение всех возможных блоков в программе и достижение максимального охвата пути с наименьшим количеством Test case. Это гибрид branch testing и path testing. Цель Basis Path testing состоит в том, что оно определяет количество независимых путей, таким образом, число необходимых Test case может быть определено явно (максимизирует охват каждого тестового случая).
Тесты критического пути (Critical path tests) запускаются для проверки функциональности, используемой обычными пользователями во время их повседневной деятельности. Многие пользователи обычно используют определенный набор функций приложения, который необходимо проверить, как только фаза smoke будет успешно завершена. Здесь метрический предел немного ниже, чем при smoke, и он соответствует 70-80-90% в зависимости от цели проекта при ста процентах у smoke. 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.

Что вы знаете о Statement coverage?
Охват операторов - это метод проектирования теста белого ящика, который включает в себя выполнение всех исполняемых операторов (if, for и switch) в исходном коде как минимум один раз. Он используется для вычисления и измерения количества операторов в исходном коде, которые могут быть выполнены с учетом требований. Другими словами, тестировщик будет концентрироваться на внутренней работе исходного кода, касающегося control flow graphs или flow charts. Как правило, в любом программном обеспечении, если мы посмотрим на исходный код, будет множество элементов, таких как операторы, функции, циклы, обработчики исключений и т. д. В зависимости от входных данных программы некоторые части кода могут не выполняться. Цель покрытия Statement - охватить все возможные пути, строки и операторы в коде. Тестируются операторы потока управления, такие как.
Покрытие операторов позволяет найти:

- Неиспользованные выражения (Unused Statements)
- Мертвый код (Dead Code)
- Неиспользуемые ветви (Unused Branches)
- Недостающие операторы (Missing Statements)

Что вы знаете о Decision coverage?
«Решение» - это программная точка, в которой control flow имеет два или более альтернативных маршрута. control flow- это последовательность событий (paths) при выполнении через компонент или систему. Таким образом, если быть точным, «решение» - это узел потока управления (например, оператор if, оператор цикла или оператор case), который имеет две или более ссылок на отдельные ветви выполнения.

100% покрытие решений означает, что все возможные результаты решения были выполнены по крайней мере один раз. Для утверждений if это правда или ложь. Иногда этот метод называют Branch testing.
Что вы знаете о Branch coverage?
В покрытии ветвей проверяется каждый результат из модуля кода. Например, если результаты являются бинарными, вам необходимо протестировать как истинные, так и ложные результаты. Это помогает вам гарантировать, что каждая возможная ветвь из каждого условия решения выполняется по крайней мере один раз. Используя метод покрытия Branch, вы также можете измерить долю независимых сегментов кода. Это также поможет вам узнать, какие разделы кода не имеют ветвей.

Если тесты имеют полный branch coverage, то мы можем сказать, что они также имеют полный statement coverage, но не наоборот. Причина заключается в том, что в branch coverage, кроме выполнения всех statements, мы также должны проверить, выполняют ли тесты все ветви, что можно интерпретировать как охват всех ребер в control flow branch.

Что вы знаете о Condition coverage?
Покрытие условий (Condition/Toggle Coverage) - рассматриваются только выражения с логическими операндами, например, AND, OR, XOR. Условное покрытие обеспечивает лучшую чувствительность к control flow, чем decision coverage. Condition Coverage не дает гарантии о полном decision coverage.
Multiple Condition Coverage: Множественное покрытие условий сообщает, встречается ли каждая возможная комбинация условий. Test Case, необходимые для полного охвата решения несколькими условиями, приведены в таблице истинности логического оператора для решения. Как и в случае покрытия условий, покрытие нескольких условий не включает покрытие решений.
Что вы знаете о FSM coverage?
Покрытие FSM (FSM Coverage) - Покрытие конечного автомата, безусловно, является наиболее сложным методом покрытия кода. В этом методе покрытия вам нужно посмотреть, как много было переходов/посещений определенных по времени состояний (time-specific states). Оно также проверяет, сколько последовательностей включено в конечный автомат.
Что такое Function coverage?
Этот показатель показывает, вызывали ли вы каждую функцию или процедуру. Во время предварительного тестирования полезно обеспечить хотя бы некоторое покрытие во всех областях программного обеспечения. Широкое неглубокое тестирование быстро обнаруживает серьезные недостатки в наборе тестов.
Что такое Call coverage?
Этот показатель показывает, выполняли ли вы каждый вызов функции. Гипотеза состоит в том, что ошибки обычно возникают в интерфейсах между модулями. Также известен как покрытие пары вызовов (call pair coverage).
Что означает LCSAJ coverage?
LCSAJ (linear code sequence and jump) «линейная последовательность кода и переход». Эта вариация path coverage учитывает только подпути, которые могут быть легко представлены в исходном коде программы, не требуя flow graph. LCSAJ - это последовательность строк исходного кода, выполняемых последовательно. Эта

«линейная» последовательность может содержать decisions, пока control flow фактически продолжается от одной строки к следующей во время выполнения. Подпути создаются путем объединения LCSAJ. Исследователи ссылаются на коэффициент покрытия путей длиной n LCSAJ как на коэффициент эффективности теста (TER) n + 2.

Его основное применение при динамическом анализе программного обеспечения, чтобы помочь ответить на вопрос «Сколько тестирования достаточно?». Динамический анализ программного обеспечения используются для измерения качества и эффективности тестовых данных программного обеспечения, где количественное определение выполняются в терминах структурных единиц кода при тестировании. В более узком смысле, LCSAJ является хорошо определенным линейным участком коды программы. При использовании в этом смысле, LCSAJ также называют JJ-путь, стоя для скачка к скачку-пути. 100% LCSAJ означает 100% Statement Coverage, 100% Branch Coverage, 100% procedure или Function call Coverage, 100% Multiple condition Coverage.

Сравнение некоторых метрик

Вы можете сравнить «относительные силы»? (relative strengths), когда более сильная метрика включает более слабую метрику.

- Decision coverage включает в statement coverage, поскольку выполнение каждой ветви должно приводить к выполнению каждого statement. Эта связь сохраняется только тогда, когда control flows непрерывно до конца всех основных блоков. Например, функция C / C ++ может никогда не вернуться для завершения вызывающего базового блока, потому что она использует throw, abort, семейство exec, exit или longjmp.
- Path coverage включает в себя Decision coverage.
- Predicate coverage включает в себя Path coverage и multiple condition coverage, а также большинство других метрик.

Показатели покрытия нельзя сравнивать количественно.

Что такое Equivalence Partitioning?

Хорошо известная техника и одна из самых используемых. Часто вы можете найти такое объяснение: Например, у вас есть диапазон допустимых значений от 1 до 10, поэтому вам просто нужно проверить одно значение из диапазона - «5» и одно вне диапазона - «0». Это очень простое и понятное объяснение, но оно может дать узкий взгляд на эту технику. Что если у нас нет диапазона чисел? Концепция разделения эквивалентности возникла из математики и понимания того, что такое класс эквивалентности и отношение эквивалентности, что может быть трудно быстро понять без математического бекграунда. Но для целей тестирования, я думаю, это можно упростить, определив его следующим образом: эквивалентное разделение - это подмножество элементов из определенного набора, которые обрабатываются Системой (тестируются) одинаково. Таким образом, вам не нужно выполнять тесты для каждого элемента подмножества, и достаточно одной проверки, чтобы охватить все подмножество. Следовательно, методика может быть описана как разделение всего набора данных ввода / вывода на такие разделы. И если у вас есть, например, набор данных, содержащий около 100 элементов, которые можно разделить на 5 разделов, вы можете уменьшить количество кейсов до 5. Хитрость заключается в том, чтобы увидеть и идентифицировать разделы. Их можно найти в наборах без номеров (например, листья деревьев, разделенные по цвету - желтый, зеленый и т. д.), или даже один элемент может быть классом эквивалентности (например, лифт обычно

более полон на первом этаже, чем на других этажах). По мере того, как люди выходят из здания, поднимается лифт, поэтому первый этаж - это отдельный класс эквивалентности). Очень эффективная, экономящая время техника.

Что такое Boundary Value Analysis?

Анализ Граничных Значений (Boundary Value Analysis - BVA). Второй известный метод, который часто используется в паре с предыдущим. Идея этого метода заключается в проверке граничных значений для разделов эквивалентности, когда результат изменяется с действительного на недействительный (для этого раздела). Test case, основанные на этих значениях, чувствительны к ошибкам и имеют высокую вероятность обнаружения ошибок при использовании логических операторов (> вместо >=, < вместо >, || вместо && и т. l.).

Это описание немного неоднозначно. Для проверки границ мы должны проверять допустимые значения границ диапазона, плюс одно значение ниже и одно значение вне диапазона. Таким образом, для диапазона больше 1 и меньше или равного 10, это 1, 2 и 10, 11.

Что такое Error Guessing?

Предугадывание ошибки (Error Guessing - EG). Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку. Например, спецификация говорит: "пользователь должен ввести код". Тест аналитик, будет думать: "Что, если я не введу код?", "Что, если я введу неправильный код? ", и так далее. Это и есть предугадывание ошибки.

Что такое Cause/Effect?

Причина / Следствие (Cause/Effect - CE). Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие), то есть Простая проверка базовых действий и их результата. Например, если нажать крестик в правом верхнем углу окна (причина), оно закроется (следствие). Или вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как "Имя", "Адрес", "Номер Телефона" а затем, нажать кнопку "Добавить" - это "Причина". После нажатия кнопки "Добавить", система добавляет клиента в базу данных и показывает его номер на экране - это "Следствие".

Граф причинно-следственных связей похож на Decision Table и также использует идею объединения условий. И иногда они описываются как один метод. Но если между условиями существует много логических зависимостей, может быть проще их визуализировать на cause-effect graph. Здесь можно выделить три этапа:

- Определить условия и последствия
- Нарисовать график со всеми логическими зависимостями и ограничениями
- Преобразовать график в Decision Table, отслеживая каждую комбинацию причин, которые приводят к эффекту от эффекта (tracing each combination of causes that lead to an effect from the effect).

Что такое Exhaustive testing?

Исчерпывающее тестирование (Exhaustive testing - ET) - это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

Какие вы знаете комбинаторные техники тест-дизайна?

Или, скорее, стоит рассматривать их как комбинаторные стратегии. Их основное назначение - создавать комбинации входных параметров на основе одного из приведенных ниже алгоритмов.
Все комбинации (All combinations): как видно из названия, этот алгоритм подразумевает генерацию всех возможных комбинаций. Это означает исчерпывающее тестирование и имеет смысл только при разумном количестве комбинаций. Например, 3 переменные с 3 значениями для каждой дают нам матрицу параметров 3x3 с 27 возможными комбинациями.
Попарное тестирование (Pairwise testing) — это техника формирования наборов тестовых данных. Сформулировать суть можно, например, вот так: формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров. Смысл метода не в том, чтобы перебрать все возможные пары параметров, а в том, чтобы подобрать пары, обеспечивающие максимально эффективную проверку при минимальном количестве выполняемых тестов. С этой задачей помогают справиться математические методы, называемые ортогональными таблицами (OAT). Также существует ряд инструментов, которые помогают автоматизировать этот процесс.

Тестирование каждого выбора (Each choice testing): эта стратегия означает, что каждое значение каждого конкретного параметра должно использоваться как минимум один раз в тестовом наборе. Таким образом, полученное количество случаев будет равно количеству значений параметра с наибольшим диапазоном. Каждый выбор - это минимальная стратегия покрытия.

param1		param2	param3
a	1	X	
b	2	Y	
c	3		
	4		
	5		

Test Case		param1		param2	param3
1	a	1	X		
2	b	2	Y		
3	c	3	X		
4	a	4	Y		
5	b	5	X		

Тестирование базового выбора (Base choice testing): для этой стратегии мы должны определить наши базовые значения для каждого параметра. Это могут быть самые распространенные, самые маленькие / самые большие, самые простые или значения по умолчанию. После того, как мы сделали наш базовый выбор, мы должны изменять значение каждого параметра по одному, сохраняя при этом значения других параметров фиксированными при базовом выборе. Пусть a, 2 и Y будут нашим базовым выбором. Тогда кейсы будут:

param1	param2	param3

a	1	X
b	2	Y
c	3	Z

Test Case	param1	param2	param3
1	a	2	Y
2	b	2	Y
3	c	2	Y
4	a	1	Y
5	a	3	Y
6	a	2	X
7	a	2	Z

Доп. материал:
Меньше, чем пара. Еще один способ сокращения количества тестов

Что такое тестирование ортогональных массивов? (OAT - Orthogonal Array testing)
Оно является техникой тестирования, которая использует ортогональные массивы для создания Test case. Это особенно полезно, когда тестируемая система имеет огромные входные данные. Например, когда необходимо проверить билет на поезд, необходимо проверить такие факторы, как - количество пассажиров, номер билета, номера мест и номера поездов. Один за другим проверка каждого фактора / ввода является громоздкой. Это более эффективно, когда инженер QA объединяет больше входных данных и проводит тестирование. В таких случаях мы можем использовать метод тестирования Orthogonal Array. Этот тип сопряжения или объединения входов и тестирования системы для экономии времени называется попарным тестированием (pairwise testing). Метод OATS используется для попарного тестирования.
Сформулировать суть pairwise можно, например, вот так: формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров.

В реальном мире у тестировщиков не будет свободного времени для выполнения всех возможных Test case, чтобы выявить дефекты, поскольку существуют другие процессы, такие как документация, предложения и обратная связь с заказчиком, которые необходимо учитывать на этапе тестирования. Следовательно, test managers хотели оптимизировать количество и качество Test case, чтобы обеспечить максимальное покрытие тестами с минимальными усилиями. Это усилие называется Test case Optimisation.

- Систематический и статистический способ тестирования парных взаимодействий
- Точки взаимодействия и интеграции являются основным источником дефектов.
- Выполните четко определенные, краткие Test case, которые могут выявить большинство ошибок.
- Ортогональный подход гарантирует попарное покрытие всех переменных.

Формула для расчета OAT:

Runs (N) - количество строк в массиве, что выражается в количестве тестовых случаев, которые будут сгенерированы. Factors (K) - Количество столбцов в массиве, что выражается в максимальном количестве переменных, которые могут быть обработаны. Levels (V) - максимальное количество значений, которые могут быть приняты для любого отдельного фактора. Один фактор имеет от 2 до 3 входов для тестирования. Это максимальное количество входов определяет уровни.
Что такое Domain analysis/testing?
Доменный анализ: Это тип функционального тестирования, направленный на разбиение диапазона возможных значений переменной (или переменных) на поддиапазоны (или домены), с последующим выбором одного или нескольких значений из каждого домена для тестирования, то есть на анализ показательных значений и взаимосвязи элементов. Основная цель Domain testing: предоставить стратегию по выбору минимального набора показательных тестов. Кроме того, оно проверяет, что система не должна принимать входные данные, условия и индексы за пределами указанного или действительного диапазона. Во многом доменное тестирование пересекается с известными нам техниками разбиения на классы эквивалентности и анализа граничных значений. Но доменное тестирование не ограничивается перечисленными техниками. Оно включает в себя как анализ зависимостей между переменными, так и поиск тех значений переменных, которые несут в себе большой риск (не только на границах).

Доп. материал:
Introduction to Domain Testing
Master of Your Domain
Что такое Cyclomatic Complexity в тестировании ПО?
Это метрика ПО, используемая для измерения сложности программы. Это количественная мера независимых путей в исходном коде программы. Независимый путь определяется как путь, имеющий хотя бы одно ребро, которое ранее не проходило ни в одном другом пути. Цикломатическая сложность может быть рассчитана относительно функций, модулей, методов или классов в программе. Эта метрика основана на представлении программы как control flow. Поток управления изображает программу в виде графа, который состоит из вершин и ребер. На графе вершины представляют обработку задачи, а ребра представляют поток управления между вершинами.
Тестирование базового пути является одним из методов «белого ящика» и гарантирует выполнение хотя бы одного оператора во время тестирования. Он проверяет каждый линейно независимый путь в программе, что означает, что число тестовых примеров будет эквивалентно цикломатической сложности программы.

V(G) = E - N + 2 где E – количество ребер, N –количество вершин
или
V(G) = P + 1 где P - Количество предикатных узлов (узел, содержащий условие)

Сложность 1-10 говорит о хорошо написанном коде, легкой тестируемости и экономичности. 10-20 и 20-30 говорят об усложненном коде и трудностях с

тестированием такого кода вкупе с высокими затратами. Сложность больше 40 практически не поддается проверке и стоит очень дорого. Подсчитывается вручную в случае небольшого ПО или с помощью автоматизированного ПО для крупного. Цикломатическая Сложность может оказаться очень полезной:

- Помогает разработчикам и тестировщикам определять независимые пути выполнения
- Разработчики могут быть уверены, что все пути были протестированы хотя бы раз
- Помогает нам больше сосредоточиться на открытых (uncovered) путях
- Улучшение покрытия кода
- Оценка рисков
- Использование этих метрик в начале цикла снижает риски

Что такое State Transition testing?

Тестирование переходов между состояниями определяется как метод тестирования ПО, при котором изменения входных условий вызывают изменения состояния в тестируемом приложении (AUT). Это метод тестирования черного ящика, в котором тестировщик анализирует поведение тестируемого приложения для различных входных условий в последовательности. В этом методе тестировщик предоставляет как положительные, так и негативные входные значения теста и записывает поведение системы. Это модель, на которой основаны система и тесты. Любая система, в которой вы получаете разные выходные данные для одного и того же ввода, в зависимости от того, что произошло раньше, является системой конечных состояний. Техника тестирования переходов между состояниями полезна, когда вам нужно протестировать различные системные переходы. Этот подход лучше всего подходит там, где есть возможность рассматривать всю систему как конечный автомат

- Состояние (state, представленное в виде круга на диаграмме) – это состояние приложения, в котором оно ожидает одно или более событий. Состояние помнит входные данные, полученные до этого, и показывает, как приложение будет реагировать на полученные события. События могут вызывать смену состояния и/или инициировать действия.
- Переход (transition, представлено в виде стрелки на диаграмме) – это преобразование одного состояния в другое, происходящее по событию.
- Событие (event, представленное ярлыком над стрелкой) – это что-то, что заставляет приложение поменять свое состояние. События могут поступать извне приложения, через интерфейс самого приложения. Само приложение также может генерировать события (например, событие «истек таймер»). Когда происходит событие, приложение может поменять (или не поменять) состояние и выполнить (или не выполнить) действие. События могут иметь параметры (например, событие «Оплата» может иметь параметры «Наличные деньги», «Чек», «Приходная карта» или «Кредитная карта»).
- Действие (action, представлено после «/» в ярлыке над переходом) инициируется сменой состояния («напечатать билет», «показать на экране» и др.). Обычно действия создают что-то, что является выходными/возвращаемыми данными системы. Действия возникают при переходах, сами по себе состояния пассивны.
- Точка входа обозначается черным кружком.
- Точка выхода показывается на диаграмме в виде мишени.

Для наглядности возьмем классический пример покупки авиабилетов. Все начинается с точки входа. Мы (клиенты) предоставляем авиакомпании информацию для бронирования. Служащий авиакомпании является интерфейсом между нами и системой бронирования авиабилетов. Он использует предоставленную нами информацию для создания бронирования. После этого наше бронирование находится в состоянии «Создано». После создания бронирования система также запускает таймер. Если время таймера истекает, а забронированный билет еще не оплачен, то система автоматически снимает бронь.
Каждое действие, выполненное над билетом, и соответствующее состояние (отмена бронирования пользователем, оплата билета, получение билета на руки, и т. д.) отображаются в блок-схеме. На основании полученной схемы составляется набор тестов, в котором хотя бы раз проверяются все переходы.
Некоторым исследователям представляется более удобным свести весь процесс в таблицу состояний и переходов. Конечно, таблица не так наглядна, как схема, но зато она получается более полной и систематизированной, так как определяет все возможные State-Transition варианты, а не только валидные.

Еще пример с банкоматом. После того, как мы визуализировали все переходы, мы просто выполняем определенные пути из диаграммы в качестве Test case:

Что такое Scenario (use case) testing?
Use Case описывает сценарий взаимодействия двух и более участников (как правило – пользователя и системы). Пользователем может выступать как человек, так и другая система. Юзкейсы могут быть позитивными (Sunny day use case) – в приоритете, и негативными (Rainy day use case).
Целью этого тестирования является нахождение дефектов, которые трудно найти при тестировании частей/модулей отдельно. Но нужно понимать, что это не аналог приемочного тестирования (но может быть его частью) и оно не охватывает все возможные варианты путей.
Как правило, Test case представлен очень небольшим количеством действий и одним результатом. Сценарий, с другой стороны, представляет собой последовательность действий (с промежуточными результатами), которые приводят к достижению какой-то конкретной цели. Сценарии могут быть частью документации разработчика (scenario diagrams). Довольно часто они задокументированы в требованиях как «use cases» - сценарии, которые обычно описывают взаимодействие пользователя с Системой. Но часто эти сценарии не очень подробны. Кроме того, прежде чем использовать их для создания Test case, их необходимо подробно описать с помощью шаблона. Шаблоны могут варьироваться от проекта к проекту. Но среди таких обычных полей, как имя, цель, предварительные условия, актер (ы) и т. д., всегда есть основной успешный сценарий и так называемые расширения (плюс иногда подвариации). Расширения - это условия, которые влияют на основной сценарий успеха. А подвариации - это условия, которые не влияют на основной flow, но все же должны быть рассмотрены. После того, как шаблон заполнен данными, мы создаем конкретные Test case, используя методы эквивалентного разделения и граничных значений. Для минимального охвата нам нужен как минимум один тестовый сценарий для основного сценария успеха и как минимум один Test case для каждого расширения. Опять же, этот метод соответствует общей формуле «получите условия, которые меняют наш результат, и проверьте

комбинации». Но способ получить это - проанализировать поведение Системы с помощью сценариев.
Пример: Рассмотрим сценарий, когда пользователь покупает товар с сайта онлайн-покупок. Пользователь сначала войдет в систему и начнет поиск. Пользователь выберет один или несколько товаров, отображаемых в результатах поиска, и добавит их в корзину. После всего этого он проверит. Так что это пример логически связанного ряда шагов, которые пользователь выполнит в системе для выполнения задачи. В этом тестировании проверяется поток транзакций во всей системе от конца до конца. Варианты использования - это, как правило, путь, который пользователи чаще всего используют для достижения конкретной задачи. Таким образом, это упрощает использование прецедентов при обнаружении дефектов, поскольку включает в себя путь, с которым пользователи чаще всего сталкиваются при первом использовании приложения.

Как создать хорошие сценарии (Сэм Канер):
- Напишите истории жизни для объектов в системе.
- Перечислите возможных пользователей, проанализируйте их интересы и цели.
- Подумайте об отрицательных пользователях: как они хотят злоупотреблять вашей системой?
- Перечислите «системные события». Как система справляется с ними?
- Перечислите «особые события». Какие приспособления система делает для них?
- Перечислите преимущества и создайте сквозные задачи, чтобы проверить их.
- Интервью пользователей об известных проблемах и сбоях старой системы.
- Работайте вместе с пользователями, чтобы увидеть, как они работают и что они делают.
- Читайте о том, что должны делать подобные системы.
- Изучите жалобы на предшественника этой системы или ее конкурентов.
- Создать фиктивный бизнес. Относитесь к нему как к реальному и обрабатывайте его данные.
- Попробуйте преобразовать реальные данные из конкурирующего или предшествующего приложения.

Что такое Decision Table testing?
Этот простой метод заключается в документировании бизнес-логики в таблице как наборов условий и действий. Например, если у вас есть набор переменных, которые трудно запомнить и управлять ими, таблица решений поможет упорядочить их, чтобы упростить идентификацию правильных случаев.

Пример: если пользователь вводит правильное имя пользователя и пароль, он будет перенаправлен на домашнюю страницу. Если какой-либо из вводимых данных неправильный, появится сообщение об ошибке.

Условие	1	2	3	4
Имя пользователя (+/-)		-	+	-	+
Пароль (+/-)	-	-	+	+
Итог (Е/Н)	Е	Е	Е	Н

Условные обозначения: «+» - правильное имя пользователя / пароль «-» - Неверное имя пользователя / пароль E - Сообщение об ошибке отображается H - отображается главный экран

- Случай 1 - имя пользователя и пароль были неверны. Пользователю показывается сообщение об ошибке.
- Случай 2 - Имя пользователя было правильным, но пароль был неправильным. Пользователю показывается сообщение об ошибке.
- Случай 3 - Имя пользователя было неверным, но пароль был правильным. Пользователю показывается сообщение об ошибке.
- Случай 4 - имя пользователя и пароль были правильными, и пользователь перешел на домашнюю страницу Преобразуя это в тестовый пример, мы можем создать 2 сценария:
- Введите правильное имя пользователя и правильный пароль и нажмите на кнопку входа, и ожидаемый результат будет пользователь должен перейти на домашнюю страницу
- И один из сценария ниже:
- Введите неправильное имя пользователя и неправильный пароль и нажмите на кнопку входа, и ожидаемый результат будет, пользователь должен получить сообщение об ошибке
- Введите правильное имя пользователя и неправильный пароль и нажмите на кнопку входа, и ожидаемый результат будет, пользователь должен получить сообщение об ошибке
- Введите неправильное имя пользователя и правильный пароль и нажмите на кнопку входа, и ожидаемый результат будет, пользователь должен получить сообщение об ошибке

Что такое Random testing?
Техника функционального тестирования (Black box), также известный как тестирование с произвольным вводом, это, вероятно, самый недооцененный метод, основная идея которого заключается в выборе случайных входных данных из возможных значений для определенной функциональности. Так что нет никакой системы в выборе входных данных. Этот метод также называется monkey testing, и если мы говорим о ручном тестировании, он может быть менее эффективным, чем другие методы черного ящика. Но если мы добавим автоматизацию, она станет мощным инструментом. Просто представьте, что Test case (с различными наборами входных данных) генерируются, выполняются и оцениваются автоматически в непрерывном цикле, что позволяет вам запускать тысячи и миллионы случаев в течение разумного времени. Создание «Случайного тестера» - довольно интересная тема, но также довольно сложная и требует более глубокого изучения. Здесь я опишу это только концептуально.
В Monkey testing тестировщиком (иногда и разработчиком) считается «Обезьяна». Если обезьяна использует компьютер, она будет произвольно выполнять любую задачу в системе из своего понимания. Точно так же, как тестировщик будет применять случайные Test case в тестируемой системе, чтобы находить bugs/errors без предварительного определения тестового примера. В некоторых случаях Monkey testing также посвящен модульному тестированию или GUI-тестированию. Основная задача: попытаться сломать систему.
Gorilla testing - это метод тестирования ПО, при котором модуль программы многократно тестируется, чтобы убедиться, что он работает правильно и в этом модуле

нет ошибок. Модуль может быть проверен более ста раз одним и тем же способом. Gorilla testing также известен как «раздражающее тестирование».

Monkey testing Gorilla testing
Тестирование выполняется случайным образом без каких-либо заранее определенных Test case Тоже
Тестирование выполняется на всей системе может иметь несколько Test case
Тестирование выполняется на нескольких отдельных модулях с небольшим количеством Test case.
Целью Monkey testing является проверка на сбой системы Целью тестирования Gorilla является проверка правильности работы модуля.

Monkey testing Ad-hoc testing
Тестирование выполняется случайным образом без каких-либо заранее определенных Test case Тестирование выполняется без планирования и документации (контрольные примеры и SRS)
В Monkey testing тестировщики могут не знать, что за ПО и для чего оно предназначено. В Ad-hoc testing тестировщик должен понять ПО перед выполнением тестирования
Целью Monkey testing является проверка на сбой системы Целью специального тестирования является случайное разделение системы на части и проверка их функциональности.
Типы Обезьян:

- Тупая обезьяна: тестировщики не имеют представления о системе и ее функциональных возможностях, а также не имеют никаких гарантий относительно достоверности Test case.
- Умная обезьяна: тестировщик имеет четкое представление о системе, ее назначении и функциональности. Тестировщик перемещается по системе и предоставляет действительные данные для выполнения тестирования.
- Выдающаяся обезьяна: тестировщики выполняют тестирование в соответствии с поведением пользователя и могут указать некоторые вероятности возникновения ошибок.

Что такое Syntax testing?
Синтаксическое тестирование - это метод проверки «черного ящика». Этот метод помогает при разработке Test case для входных форматов. Конечно, если наши правила синтаксиса звучат как «должны быть только цифры или буквы», нам не нужен этот метод. Но если у нас есть какой-то сложный формат (например, почтовый индекс, телефон, конкретный адрес электронной почты), это может быть удобно. Прежде всего, мы должны определить наш формат и описать его формально, используя форму Бэкуса-Наура (или расширенный BNF). Это очень важный момент, поэтому я бы посоветовал вам ознакомиться с BNF, прежде чем читать дальше. После того, как мы создали определение BNF для нашего формата, пришло время генерировать положительные и отрицательные кейсы:
Для положительных случаев мы находим возможные варианты значений, допускаемые отдельными элементами определения BNF, а затем разрабатываем варианты, чтобы просто охватить эти варианты.

Для случаев с недопустимым синтаксисом мы определяем и применяем возможные мутации (например, отсутствующий элемент, нежелательный дополнительный элемент, недопустимое значение для элемента и т. д.) к отдельным элементам определения BNF. Затем мы разрабатываем наши кейсы, применяя мутации, которые могут давать отличительные результаты (случаи, которые приводят к действительным комбинациям, исключаются).

Что вы знаете о Classification tree method?
Большинство ресурсов выделяют два основных шага для этой техники:

- Определение относящихся к тесту аспектов (аспектов, которые влияют на функциональность - так называемые классификации) и их соответствующих значений (называемых классами, это могут быть точные значения или классы эквивалентности).
- Объединение разных классов из всех классификаций в Test case.

Оба эти шага справедливы для большинства методов тестирования и могут быть перефразированы следующим образом: - определить параметры ввода / вывода, а затем объединить их, чтобы получить ваши кейсы.
Теперь давайте добавим некоторые детали. Классификационное дерево - это графический метод, который помогает нам визуализировать относящиеся к тесту аспекты (аспекты, которые влияют на поведение тестового объекта во время теста) в форме иерархического дерева.
Как вырастить это дерево? Это похоже на интеллектуальные карты с небольшими отличиями, если это вам о чем-то говорит. У нас есть тестовый объект (целое приложение, определенная функция, абстрактная идея и т. д.) вверху как корень. Мы рисуем ответвления от корня как классификации (проверяем соответствующие аспекты, которые мы определили). Затем, используя распределение по эквивалентности и анализ граничных значений, мы определяем наши листья как классы из диапазона всех возможных значений для конкретной классификации. И если некоторые из классов могут быть классифицированы далее, мы рисуем под-ветку / классификацию с собственными листьями / классами. Когда наше дерево завершено, мы делаем проекции листьев на горизонтальной линии (Test case), используя одну из комбинаторных стратегий (все комбинации, каждый выбор и т. д.), и создаем все необходимые комбинации.

В приведенном выше примере использовалась комбинаторная стратегия «Каждый выбор». Наиболее очевидные преимущества здесь - это большая наглядность и ясность объема тестового объекта и идей Test case. Если у вас есть сложные, иерархически структурированные данные, и вы можете позволить себе тратить время на создание и поддержку дерева, этот метод будет чрезвычайно удобен. И для эффективного применения метода вы можете, например, рассмотреть возможность использования специального инструмента, такого как Classification Tree Editor.
Как мы узнаем, что код соответствует спецификациям?
Матрица прослеживаемости - это интуитивно понятный инструмент, который обеспечивает соответствие требований тестовым примерам. Когда выполнение всех Test case заканчивается успешно, это указывает на то, что код соответствует требованиям.
Что включает в себя матрица отслеживания требований? (RTM - Requirement Traceability Matrix)

Матрица прослеживаемости - это документ, который связывает любые два базовых документа, которые требуют отношения «многие ко многим» для проверки полноты отношений. В случае тестирования это матрица покрытия функциональных требований тест-кейсами.
Есть даже такое понятие как Requirement based testing, которое имеет место быть, когда есть требования к продукту, на их основе составляются тест-сценарии и выполняется тестирование.
Зачем нужна эта матрица?
Например, для того чтобы:
- при разработке тестов четко ориентироваться какие из требований уже покрыты тестами, а какие еще нет;
- при выполнении тестирования ориентироваться какие из требований прошли все написанные для них тесты успешно, а какие - еще нет.
Матрица трассировки может служить одновременно в качестве матрицы покрытия. Наличие такой матрицы позволяет объективно оценить, какая часть продукта покрыта тестами, а какая нет. Это необходимое условие, чтобы оценить, какой объем работы мы уже выполнили и что еще осталось сделать - и по части создания, и по части выполнения тестов.
Еще одно преимущество traceability matrix – ее наглядность. Если она поддерживается в актуальном состоянии, то можно сразу увидеть "белые пятна" и сосредоточиться на них.
Traceability matrix также позволяет сравнивать тесты между собой по критерию количества требований, которые они покрывают. Одни тесты могут покрывать несколько требований, другие – только одно.
В чем разница между Test matrix и Traceability matrix?
Тестовая матрица: тестовая матрица используется для определения фактического качества, усилий, плана, ресурсов и времени, необходимых для охвата всех этапов тестирования программного обеспечения.
Матрица прослеживаемости: сопоставление между Test case и требованиями.
Что такое анализ GAP?
Анализ пробелов выявляет любые отклонения между функциями, доступными для тестирования, и восприятием их клиентом. Матрица прослеживаемости - это инструмент тестирования, который тестировщики могут использовать для отслеживания пробелов.
Что такое граф причинно-следственных связей? (Cause Effect Graph)
Это графическое представление входных данных и связанных с ними выходных эффектов, которые помогают в разработке Test case.
В чем разница между предугадыванием ошибок и посевом? (Error guessing and error seeding)
Предугадывание - методика разработки Test Suite, в которой тестировщики должны предполагать возможные дефекты и писать тестовые наборы для их представления.
Seeding. Это процесс добавления известных ошибок в программу для отслеживания скорости обнаружения и удаления. Это также помогает оценить количество неисправностей, оставшихся в программе.
Стили тестов?
*источник утерян, аналогов не нашел

- Output based verification (смотрим что возвращает)
- State based verification (смотрим состояние)

- Communication based verification (общение и зависимости)

Техники тестирования требований?

- Взаимный просмотр:
- беглый просмотр — автор показывает свою работу коллегам, они в свою очередь дают свои рекомендации, высказывают свои вопросы и замечания;
- технический просмотр — выполняется группой специалистов;
- формальная инспекция — привлекается большое количество специалистов, представляет собой структурированный, систематизированный и документированный подход. Минус такого варианта — тратится много времени.
- Вопросы — если возникают вопросы, то можно спрашивать у представителей заказчика, более опытных коллег.
- Тест-кейсы и чек-листы — хорошее требование должно быть проверяемым, чтобы это определить можно использовать чек-листы или полноценные тест-кейсы. Если можно быстро придумать несколько пунктов чек-листа — это уже хороший знак.
- Исследование поведения системы — необходимо мысленно смоделировать процесс работы пользователя с системой, созданной по тестируемым требованиям, после этого определить неоднозначные варианты определения системы.
- Рисунки — графическое представление дает наглядное представление приложения, на рисунке проще увидеть, что какие-то элементы не стыкуются, где-то чего-то не хватает и т. д.
- Прототипирование — сделав наброски пользовательского интерфейса, легко оценить применение тех или иных пользовательских решений

Что такое эвристики?

Эвристики – это быстрые, недорогие способы решения проблемы или принятия решения. Эвристики подвержены ошибкам, то есть они могут как сработать, так и не сработать. Эвристики недостаточно абстрактны, они могут перекрываться и пересекаться друг с другом. Также эвристики зависят от контекста

Процесс тестирования на основе эвристик – это такая технология тестирования алгоритмов, приложений и программ, при использовании которой стратегия тестирования основывается на предыдущем опыте и данных о вероятности наступления различных событий.

- Эвристика «Время вышло!» мы останавливаем тестирование, когда заканчивается выделенное на него время.
- Эвристика • пиньяты (The Piñata Heuristic). Мы прекращаем ломать программу, когда начинают выпадать конфеты – мы останавливаем тестирование, когда видим первую достаточно серьезную проблему.
- Эвристика «мертвой лошади». В программе слишком много ошибок, так что продолжение тестирования не имеет смысла. Мы знаем, что все изменится настолько, что сведет на нет результаты текущего тестирования.
- Эвристика «Задание выполнено» останавливаем тестирование, когда найдены ответы на все поставленные вопросы.
- Эвристика «Отмена задания» Наш клиент сказал нам: «пожалуйста, прекратите тестирование». Это может произойти по причине перерасхода бюджета, или вследствие отмены проекта, и по любой другой причине. Какова бы ни была причина, нам поручили остановить тестирование. (На самом деле эвристика «Время вышло!» может быть частным случаем более общей «Отмены задания», в том случае, если предпочтительнее, чтобы не мы сами, а заказчик принял решение о том, что время вышло.)

- Эвристика «Я зашел в тупик!» По какой бы то ни было причине мы останавливаемся, поскольку обнаруживаем некое препятствие. У нас нет информации, которая нам требуется (например, многие люди заявляют, что не могут тестировать без достаточного количества спецификаций). Имеется блокирующая ошибка, и таким образом мы не можем перейти в ту область продукта, которую необходимо протестировать, у нас нет необходимого оборудования или инструментария, у команды нет квалификации, требуемой для выполнения некоторых специальных тестов.
- Эвристика «освежающей паузы» Вместо прекращения тестирования мы приостанавливаем его на некоторое время. Мы можем остановить тестирование и сделать перерыв, когда мы устали, когда нам стало скучно или пропало вдохновение. Мы можем сделать паузу на то, чтобы выполнить некоторые исследования, разработать планы, поразмыслить над тем, что мы делали в прошлом и понять, что делать дальше. Идея заключается в том, что нам требуется определенный перерыв, после которого мы сможем вернуться к продукту со свежим взглядом или свежими мыслями. Также есть и другой вид паузы: мы можем остановить тестирование какой-либо функции, поскольку в настоящий момент другая имеет более высокий приоритет.
- Эвристика «Отсутствие продвижения» Что бы мы ни делали, мы получаем тот же самый результат. Это может происходить в случае, когда программа падает определенным способом или перестает отвечать, но также мы можем не продвигаться, когда программа в основном ведет себя стабильно: "выглядит хорошо!"
- Эвристика Привычного завершения. Мы останавливаем тестирование тогда, когда мы обычно останавливаем тестирование. Имеется протокол, задающий определенное количество идей для тестирования, или тест-кейсов, или циклов тестирования, или как вариант – имеется определенный объем работ по тестированию, который мы выполняем и после этого останавливаемся. Agile-команды, например, часто применяют такой подход: «когда выполнены все приемочные тесты, мы знаем, что продукт готов к поставке. Отличие от эвристики «Время вышло!» в том, что временные ограничения могут изменяться более гибко, чем некоторые другие.
- Больше нет интересных вопросов В этот момент мы решаем, что не осталось вопросов, ответы на которые были бы достаточно ценными, чтобы оправдать стоимость продолжения тестирования, и поэтому мы останавливаемся. Эта эвристика используется в основном как дополнение к другим эвристикам, помогая принять решение о том, есть ли какие-то вопросы или риски, которые отменяют действие этих эвристик (примеры таких вопросов я привожу после каждой эвристики). Кроме того, если одна эвристика советует нам прекратить тестирование, следует проверить, нет ли интересных вопросов или серьезных рисков в других областях, и если они есть, то мы скорее продолжим тестирование, чем остановимся.
- Эвристика уклонения/безразличия. Иногда людей не интересует дополнительная информация, либо они не хотят знать, что происходит в программе. Тестируемое приложение может быть первой версией, которую, как мы знаем, скоро заменят. Некоторые люди прекращают тестирование по причине лени, злого умысла или отсутствия мотивации. Иногда бизнес-критичность выпуска нового релиза настолько высока, что никакая мыслимая проблема не остановит выход программы, и поэтому никакие новые результаты тестирования не будут иметь значения.

Дополнение: Кем Канер (Cem Kaner) предложил еще одну эвристику: «Отказ от выполнения задания» (Mission Rejected), в которой тестировщик сам отказывается от продолжения тестирования. Подробнее читайте здесь.

Процесс тестирования на основе эвристик – это такая технология тестирования алгоритмов, приложений и программ, при использовании которой стратегия тестирования основывается на предыдущем опыте и данных о вероятности наступления различных событий.

Мнемоника – это набор правил и приемов, которые помогают эффективно запоминать необходимые сведения (информацию).
Большинство экспертов с мировым именем в сфере тестирования настоятельно рекомендуют использовать проверенную схему SFDPOT, автором которой является Джеймс Бах. По их словам, это самый качественный и правильный инструмент для быстрой генерации тестовых идей и наработок.
Итак, что такое SFDPOT?
SFDPOT – Structures, Functions, Data, Platforms, Operations, Times:

- Structures – архитектура приложения (продукта), которая проверяется по частям. На этом этапе создаются тестовые идеи и шаги, неразрывно связанные со структурой веб-продукта;
- Function – производительность приложения (продукта). Проверка того, что может выполнять приложение (продукт). На этом этапе проводится функциональное тестирование ПО;
- Data – взаимодействие с данными. Проверка приложения на взаимодействие с данными. QA специалисты должны определить по какой логике продукт взаимодействует с данными, как проходит их получение, тип обработки и виды информации;
- Platform – экосистема, платформа. Выполнение проверки того, как именно приложение (продукт) потенциально взаимодействует с платформой, на которой оно создано и запущенно. Тестировщик должен определить, на каких именно платформах и внутри каких систем необходимо провести процедуру ручного и автоматизированного тестирования;
- Operations – проверка созданных сценариев для разработанного приложения. На этом этапе перед тестировщиками постает задача по выяснению потенциального круга будущих пользователей, которые будут взаимодействовать с создаваемым продуктом;
- Time – период времени, проверка того, как продукт ведет себя в зависимости от наступления или завершения каких-либо временных промежутков.

Что же касается CRUCSPIC STMP, то этот метод может рассматриваться как составная часть Operations внутри методики SFDPOT.
Если вкратце сказать о функциональности CRUCSPIC STMP, то можно отметить, что это своего рода атрибуты системы. Наработки CRUCSPIC STMP позволяют выделить основные объекты тестирования, его целей, а также построить карту эффективности взаимодействия между собой.

----- Тестовые артефакты и документация (Test Deliverables/TestWare/test artifacts) -----
Виды тестовой документации?
Внешняя документация:

Замечание – короткая записка, комментарий о небольшой неточности в реализации продукта.
Дефект-репорт – описание выявленного случая несоответствия производимого продукта требованиям, к нему выдвигаемым – ошибки или ее проявления. Он обязательно должен содержать следующие элементы:

- Идею Test case, вызвавшего ошибку.
- Описание исходного состояния системы для выполнения кейса.
- Шаги, необходимые для того, чтобы выявить ошибку или ее проявление.
- Ожидаемый результат, т. е. то, что должно было произойти в соответствии с требованиями.
- Фактический результат, т. е. то, что произошло на самом деле.
- Входные данные, которые использовались во время воспроизведения кейса.
- Прочую информацию, без которой повторить кейс не получится.
- Критичность и/или приоритет.
- Экранный снимок (скрин).
- Версию, сборку, ресурс и другие данные об окружении.

Запрос на изменение (улучшение) – описание неявных/некритичных косвенных требований, которые не были учтены при планировании/реализации продукта, но несоблюдение, которых может вызвать неприятие у конечного потребителя. И пути/рекомендации по модификации продукта для соответствия им.
Сводный отчет об испытаниях (Test summary report) представляет собой документ высокого уровня, в котором обобщаются проведенные испытания и результаты испытаний.
Политика тестирования (Test policy) – Общая цель организации при выполнении тестовых заданий описана в документе «Политика тестирования». Он создается Lead’ами и Senior’ами в группе управления тестированием совместно с руководителями групп заинтересованных сторон. Иногда тестовая политика является частью более широкой политики качества, принятой организацией. В таких случаях политика качества разъяснит общую цель менеджмента в отношении качества.
Политика тестирования — это краткий документ, обобщенный на высоком уровне, который содержит следующее:

- преимущества тестирования, ценность для бизнеса, предоставляемую организации, которая оправдывает стоимость качества
- цели тестирования, такие как укрепление доверия, выявление дефектов и снижение рисков для качества
- методы измерения эффективности и результативности тестирования при выполнении целей теста
- способы для организации улучшить свои процессы тестирования

Стратегия тестирования (Test strategy) — это подход к проведению тестирования (STLC). Это относительно небольшой статический документ, который предшествует плану тестирования. Прежде чем писать объемный и подробный план, стоит формализовать некоторые базовые подходы к тестированию и убедиться, что все заинтересованные лица понимают одинаково, что и как будет тестироваться. Вероятность пропустить какую-либо тестовую активность очень мала, если существует правильная стратегия тестирования. Это самый важный документ для любой команды QA. Написание эффективного Стратегического документа - это навык, который тестировщик развивает с опытом. Стратегия включает:

- Роли и обязанности в команде тестирования

- Область тестирования
- Тестовые инструменты
- Тестовая среда
- График тестирования
- Сопутствующие риски

Внутренняя документация:

- Тест-план (Test plan, план тестирования) – документ, описывающий весь объем работ по тестированию, начиная с описания объекта, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения. Ответственность за создание документа плана тестирования лежит на Test Lead или менеджере по тестированию. Содержание:
- Что надо тестировать? — описание объекта тестирования: системы, приложения, оборудования
- Что будете тестировать? — список функций и описание тестируемой системы и ее компонент в отдельности
- Как будете тестировать? — стратегия тестирования, а именно: • виды тестирования и их применение по отношению к объекту тестирования
- Когда будете тестировать? — последовательность проведения работ: подготовка (Test Preparation), тестирование (Testing), анализ результатов (Test Result Analisys) в разрезе запланированных фаз разработки
- Критерии начала тестирования:
- готовность тестовой платформы (тестового стенда)
- законченность разработки требуемого функционала
- наличие всей необходимой документации
- Критерии окончания тестирования:
- результаты тестирования удовлетворяют критериям качества продукта: требования к количеству открытых багов выполнены
- выдержка определенного периода без изменения исходного кода приложения Code Freeze (CF)
- выдержка определенного периода без открытия новых багов Zero Bug Bounce (ZBB)
- Хорошие дополнения:
- Окружение тестируемой системы (описание программно-аппаратных средств)
- Необходимое для тестирования оборудование и программные средства (тестовый стенд и его конфигурация, программы для автоматизированного тестирования и т. д.)
- Риски и пути их разрешения
- Тестовый сценарий (Test scenario) – последовательность действий над продуктом, которые связаны единым ограниченным бизнес-процессом использования, и сообразных им проверок корректности поведения продукта в ходе этих действий. Фактически при успешном прохождении всего тестового сценария мы можем сделать заключение о том, что продукт может выполнять ту или иную возложенную на него функцию.
- Создание тестовых сценариев обеспечивает полное покрытие тестами
- Сценарии тестирования могут быть одобрены различными заинтересованными сторонами, такими как бизнес-аналитик, разработчики, клиенты, для обеспечения

тщательного тестирования ПО. Это гарантирует, что ПО работает для наиболее распространенных юзкейсов (use cases – сценарии использования).

- Они служат быстрым инструментом для определения трудозатрат на тестирование и, соответственно, создают предложение для клиента или организуют рабочую силу.
- Они помогают определить наиболее важные end-to-end транзакции или реальное использование ПО.
- Для изучения end-to-end функционирования программы, тестовый сценарий имеет решающее значение.
- Тестовый набор/комплект (Test Suite) — Некоторый набор формализованных Test case, объединенных между собой по общему логическому признаку.
- Test case (тест-кейс) – Это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или ее части. Более строго — формализованное описание одной показательной проверки на соответствие требованиям прямым или косвенным. Обязательно должен содержать следующую информацию:
- Идея проверки.
- Описание проверяемого требования или проверяемой части требования.
- Используемое для проверки тестовое окружение.
- Исходное состояние продукта перед началом проверки.
- Шаги для приведения продукта в состояние, подлежащее проверке.
- Входные данные для использования при воспроизведении шагов.
- Ожидаемый результат.
- Прочую информацию, необходимую для проведения проверки.
- Чек-лист (лист проверок) – перечень формализованных Test case в упрощенном виде удобном для проведения проверок. Test case в чек-листе не должны быть зависимыми друг от друга. Обязательно должен содержать в себе информацию о: идеях проверок, наборах входных данных, ожидаемых результатах, булевую отметку о прохождении/непрохождении тестового случая, булевую отметку о совпадении/несовпадении фактического и ожидаемого результата по каждой проверке. Может также содержать шаги для проведения проверки, данные об особенностях окружения и прочую информацию необходимую для проведения проверок. Цель – обеспечить стабильность покрытия требований проверками необходимыми и достаточными для заключения о соответствии им продукта. Особенностью является то, что чек-листы компонуются теми Test case, которые показательны для определенного требования.
- Матрица покрытия требований (RTM — Requirements Traceability Matrix) - это документ, который связывает требования с тест-кейсами.
- Тестовые данные (Test Data) — это данные, которые нужны для выполнения Test case.

Доп. материал:
Какая бывает документация
Стратегия тестирования в условиях Scrum: зачем она нужна и как построить
Интеллект-карта при составлении тест-плана
Какие отличия у тест-кейса высокого и низкого уровня?
Высокоуровневый тест-кейс (high level test case или logical test case) — тест-кейс без конкретных входных данных и ожидаемых результатов. Как правило, ограничивается

общими идеями и операциями, схож по своей сути с подробно описанным пунктом чек-листа. Достаточно часто встречается в интеграционном тестировании и системном тестировании, а также на уровне дымового тестирования. Может служить отправной точкой для проведения исследовательского тестирования или для создания низкоуровневых тест-кейсов. Низкоуровневый тест-кейс (low level test case) — тест-кейс с конкретными входными данными и ожидаемыми результатами. Представляет собой «полностью готовый к выполнению» тест-кейс и вообще является наиболее классическим видом тест-кейсов. Начинающих тестировщиков чаще всего учат писать именно такие тесты, т.к. прописать все данные подробно — намного проще, чем понять, какой информацией можно пренебречь, при этом не снизив ценность тест-кейса.

Отличия Test Suite от Test Scenario?

Test Suite - это серия логически связанных Test case, которые позволяют проверить один из вариантов сценария. Test Scenario - представляет собой некий пользовательский сценарий по тестированию некой функциональности. Что-то, что пользователь может захотеть сделать с вашей системой, и вы хотите это проверить. Сценарий может иметь один или несколько Test Suite.

Какие отличия у плана тестирования и стратегии тестирования?

- План тестирования - это документ, в котором описываются объем, цель и значение задачи тестирования ПО, тогда как стратегия тестирования описывает, как необходимо проводить тестирование.
- План тестирования используется на уровне проекта, тогда как стратегия тестирования используется на уровне организации.
- План тестирования имеет первостепенную цель, состоящую в том, как тестировать, когда тестировать, и кто будет проверять, в то время как стратегия тестирования имеет первостепенную цель - какой техники придерживаться и что проверять.
- План тестирования может быть изменен, тогда как стратегия тестирования не может быть изменена.
- План тестирования выполняется тест менеджером (Test Manager), а стратегия тестирования - менеджером проекта (Project Manager).

В чем разница между тест-кейсом и чек-листом?

Сила тест-кейса в том, что в нем все расписано очень детально, и с помощью тест-кейсов тестировать сможет даже человек, который ни разу не видел тестируемое им приложение. Но все это сложно обновлять или изменять.

Сила чек-листа в том, что он простой. Там нет глубокой детализации, это просто памятка. Но тестировать приложение по чек-листу сразу, без подготовки, не понимая, что подразумевается под «Зачарджить ордер на бэкофисе» (это где? это как? это что? это откуда и куда?) — практически невозможно.

Доп. материал:

12 • характеристик высокоэффективных тестов

Ожидаемый результат

Чем Test case отличается от сценария тестирования?

Test case - это артефакт тестирования для проверки определенного flow с определенными входными значениями, предварительными условиями тестирования, ожидаемым выходным сигналом и постусловиями, подготовленными для охвата определенного поведения. Тестовый сценарий может иметь одну или несколько

ассоциаций с тестовым набором, что означает, что он может включать несколько тестовых наборов.
Виды тест планов?
Чаще всего на практике приходится сталкиваться со следующими видами тест планов:
- Мастер Тест План (Master Plan or Master Test Plan)
- Тест План (Test Plan), назовем его детальный тест план
- План Приемочных Испытаний (Product Acceptance Plan) - документ, описывающий набор действий, связанных с • приемочным тестированием (стратегия, дата проведения, ответственные работники и т. д.)

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.
В повседневной жизни на проекте может быть один Мастер Тест План и несколько детальных тест планов, описывающих отдельные модули одного приложения.
Для увеличения ценности вашего тест плана рекомендуется проводить его периодическое рецензирование со стороны участников проектной группы:
- Ведущий тестировщик
- Тест менеджер (менеджер по качеству)
- Руководитель разработки
- Менеджер Проекта

Каждый из перечисленных участников проекта, перед утверждением, проведет рецензию и внесет свои комментарии и предложения, которые помогут сделать Ваш тест план более полным и качественным.

Доп. материал:
Тест-план не для галочки, или 8 вопросов к заказчику на старте проекта
Что является основой для подготовки плана приемки? (PAP - Product Acceptance Plan)
Прежде всего, вам нужны данные из следующих областей, чтобы подготовиться к приемке. Они могут варьироваться от компании к компании и от проекта к проекту.
- Документ с требованиями: Этот документ определяет, что именно требуется в проекте с точки зрения клиента.
- Вклад клиента: это могут быть обсуждения, неформальные чаты, письма электронной почты и т. д.
- План проекта: план проекта от менеджера проекта (PM) также служит хорошим вкладом в завершение вашего приемочного теста.

Что такое тест-анализ/основа/база тестирования и условия тестирования ? (Test Analysis/Test Basis/Test conditions)
База тестирования определяется как источник информации или документ, необходимый для написания кейсов, а также для анализа тестов. База тестирования должна быть четко определена и должным образом структурирована, чтобы можно было легко определить условия тестирования, из которых можно получить тестовые примеры. Типичная основа тестирования:

- Документ с требованиями
- План тестирования
- Репозиторий кодов
- Бизнес-требования

Т. е. Test conditions - тестируемый аспект в test basis - элемент или событие компонента или системы, которые могут быть проверены одним или несколькими кейсами: функция, транзакция, фича, атрибут качества или структурный элемент и т.п.

Доп. материал:
ISTQB Glossary - Test Basis
ISTQB Glossary - Test Condition
Что такое документ бизнес-требований (BRD)?
BRD предоставляет полное бизнес-решение для проекта, включая документацию о потребностях и ожиданиях клиентов. BRD выполняет следующие задачи.
- Получить соглашение с заинтересованными сторонами.
- Обеспечить ясность бизнес-требований.
- Описывает решение, которое соответствует потребностям клиента / бизнеса.
- Определяет входные данные для следующей фазы проекта.

Что вы знаете о требованиях (уровни/виды и т. д.)?
Виды требований по уровню:
- Бизнес-требования — определяют назначение ПО, описываются в документе о видении (vision) и границах проекта (scope).
- 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).
- Функциональный уровень (функции).

Виды требований по характеру:
1. Функциональные требования - что система должна делать. К функциональным требованиям относят (из первого вытекает следующее):
- Бизнес-требования. Что система должна делать с точки зрения бизнеса. Слово "бизнес" в данном контексте ближе к слову "заказчик". Пример бизнес-требования: промо-сайт, привлекающий внимание определенной аудитории к определенной продукции компании.
- Пользовательские требования – описывают цели/задачи пользователей системы, которые должны достигаться/выполняться пользователями при помощи создаваемой программной системы. Эти требования часто представляют в виде вариантов использования (Use Cases). Иначе говоря, пользовательские требования - это что может сделать пользователь: зарегистрироваться, посмотреть определенную информацию, пересчитать данные по определенному алгоритму и прочее.
- Функциональные требования – определяют функциональность (поведение) программной системы, которая должна быть создана разработчиками для предоставления возможности выполнения пользователями своих обязанностей в рамках бизнес-требований и в контексте пользовательских требований. Другими

словами, что будут делать разработчики, чтобы выполнить пользовательские требования.
В группу функциональных требований относят и системные требования. Эти характеристики могут описывать требования как к аппаратному обеспечению (тип и частота процессора, объем оперативной памяти, объем жесткого диска), так и к программному окружению (операционная система, наличие установленных системных компонентов и сервисов и т. п.). Обычно такие требования составляются производителем или автором ПО. Например, для игры это могут быть требования такого типа: видеокарта - объем памяти от 64 Мб, совместимость cDirectX 9.0b и новейшие драйвера. Для сайта: ОС - Windows не ниже XP, браузеры IE не ниже 7.0 и так далее.
Почему важно указывать системные требования и утверждать их у заказчика? Если не указать, например, что важно обеспечить просмотр сайта в IE 6, то разработчики вполне могут выбрать такое архитектурное решение, которое не позволит корректно отображать сайт. Системные требования напрямую зависят от целевой аудитории проекта.

2. Нефункциональные требования. Иначе говоря, как будет работать система и почему именно так.

- Бизнес-правила. Они определяют почему система работать должна именно так, как написано. Это могут быть ссылки на законодательство, внутренние правила заказчика и прочие причины. Часто упускают этот раздел и получается, что некоторые системные решения выглядят нетипичным и совсем неочевидными. Например, многие табачные компании и компании, производящие алкоголь, требуют постоянного доказательства того, что промо-сайтами пользуются люди, достигшие определенного возраста. Это бизнес-правило (подтверждение возраста) возникает по требованию этических комитетов заказчика, хотя и несколько противоречит маркетинговым целям и требованиям по usability.
- Внешние интерфейсы. Это не только интерфейсы пользователя, но и протоколы взаимодействия с другими системами. Например, часто сайты связаны с CRM системами. Особенности протокола взаимодействия «сайт-CRM» также относятся к нефункциональным требованиям.
- Атрибуты качества. Атрибуты касаются вопросов прозрачности взаимодействия с другими системами, целостности, устойчивости и т.п. К таким характеристикам относятся:
- легкость и простота использования (usability)
- производительность (performance)
- удобство эксплуатации и технического обслуживания (maintainability)
- надежность и устойчивость к сбоям (reliability)
- взаимодействия системы с внешним миром (interfaces)
- расширяемость (scalability)
- требования к пользовательским и программным интерфейсам (user and software interface).

Еще одна классификация:

- Явные требования: все, что вы написали. Чаще всего встречаются в документах, переданных заинтересованными сторонами команде разработчиков. Они

могут принимать форму сложной спецификации дизайна, набора критериев приема или описание каркаса ПО.

- Неявные требования - это второй тип. Это все то, что ожидают пользователи и что не было прописано в явных требованиях. Примеры включают производительность, удобство использования, доступность и безопасность.

Рассмотрим облачный продукт хранения, который позволяет хранить ваши файлы в Интернете. Продукт получает новое явное требование: пользователи должны иметь доступ к частному контенту других пользователей через URL, используя кнопку совместного доступа.

- Скрытые требования: все, что будет приятным сюрпризом для клиента.

Скрытые требования представляют собой функции, которые пользователи не ожидают увидеть в используемом продукте, основываясь на своем предыдущем опыте, но при их наличии данное ПО будет выигрывать в сравнении с конкурентами.

Источники требований:

- Федеральное и муниципальное отраслевое законодательство (конституция, законы, распоряжения)
- Нормативное обеспечение организации (регламенты, положения, уставы, приказы)
- Текущая организация деятельности объекта автоматизации
- Модели деятельности (диаграммы бизнес-процессов)
- Представления и ожидания потребителей и пользователей системы
- Журналы использования существующих программно-аппаратных систем
- Конкурирующие программные продукты

Методы выявления требований:

- Интервью, опросы, анкетирование
- Мозговой штурм, семинар
- Наблюдение за производственной деятельностью, «фотографирование» рабочего дня
- Анализ нормативной документации
- Анализ моделей деятельности
- Анализ конкурентных продуктов
- Анализ статистики использования предыдущих версий системы

Требования начала для производства ПО:

- необходимое тестовое окружение,
- билд/ресурс/предмет тестирования,
- код, БД, прочие компоненты объекта тестирования «заморожены», т. е. не изменяются в период всей сессии тестирования,
- модификация требований (хотя бы прямых) «заморожена»,
- известно направление тестирования,
- известны сроки на сессию тестирования.

Есть и другие условия, но они менее значимы и сильно зависят от конкретного процесса в компании.

Расскажите, какие есть требования к самим требованиям?

- Понятность. Если требования понятны кому-то одному, но не понятны всем остальным участникам, или наоборот понятны всем кроме одного — свидетельство об ошибке в их составлении. Как понять, что требование понятно?

- Реализуемость. Требования должны быть реализуемы в рамках заявленных платформ вообще, а также реализуемыми в заявленные сроки. Разработчик должен в первую очередь смотреть на этот атрибут при проведении ревью.
- Обращайте внимание на очень большие и очень маленькие числа в требованиях. Реализовать отклик в 1 миллисекунду или закачку файлов в 100ГБ, наверное, можно, только зачем, и будет ли это соответствовать основным функциональным задачам сайта?
- Полнота. Зависит от назначения и формата требований. В общем случае требования должны полностью описывать то, что в них изначально подразумевается (не должно быть недосказанности или «и т. д. »)
- Непротиворечивость. Требования должны быть поняты однозначно. Это «тест на внимательность» для тестировщика. Хорошо, если противоречие содержится в двух требованиях, расположенных рядом. Но в длинных документах требования могут занимать несколько страниц, и требования на 15 странице могут противоречить требованиям со 2 страницы. Так же требования должны не противоречить законам физики, геометрии, математики и прочим обусловленными внешними законами и обстоятельствами
- Измеримость. Требования можно посчитать и измерить (нет - «большая база», «быстрый отклик»)
- Актуальность. Требования не устарели.
- Проверяемость. Реализованность требования может быть определена через один из четырех возможных методов: осмотр, демонстрация, тест или анализ.

Доп. материал:
Чек-лист тестирования требований
Пример требований
- Так же хорошо раскрыта эта тема в книге Куликова
-

----- Дефекты и ошибки -----
Что такое дефект?
Разница между ожидаемым и фактическим результатом.

Доп. материал:
Баг или фича? Вот в чем вопрос!
Классы дефектов?
- Функциональные
- Дефекты требований
- Дефекты функций
- Функциональные дефекты самого процесса тестирования
- Системные
- Внутренние взаимодействия (интерфейсы)
- Аппаратная часть
- ОС
- Архитектура ПО
- Дефекты, связанные с процессом
- Арифметика (например, нюансы с округлениями в коде)
- Инициализация
- Последовательность

- Статическая логика (например, валидация форм)
- Дефекты, связанные с данными
- Дефекты, связанные со стандартами
- Дефекты пользовательского интерфейса (UI)

Какие есть категории дефектов?

Существует три основных категории дефектов:

- Wrong: это указывает на несоответствие в требовании и реализации. Это подразумевает отклонение от данной спецификации.
- Missing: конечный продукт не имеет функции, соответствующей требованию. Это отличается от спецификаций и означает, что вы неправильно задокументировали требование.
- Extra: вы добавили функцию, которую клиент не запрашивал. Это снова отклонение от спецификации. И пользователям продукта может понравиться эта функция. Но это все еще дефект, потому что это не часть спецификаций.

Error/Mistake/Defect/Bug/Failure/Fault?

В русском языке переводятся практически одинаково, но это разные термины. В некоторых источниках данные термины следуют за категориями, описанными выше.

- Ошибка (Error) возникает из-за ошибки (Mistake) в написании кода разработчиком.
- Дефект (Defect) это скрытый недостаток в ПО, возникший из-за ошибки в написании кода.
- Когда дефект (Defect) обнаруживается тестировщиком, он называется багом (Bug).
- Если тестировщики упустили дефект и его нашел пользователь, то это сбой (Failure).
- Если программа в итоге не выполняет свою функцию, то это отказ (Fault).

Каково содержание эффективного сообщения об ошибке?

- Project
- Subject
- Description
- Summary
- Detected By (Name of the Software Tester)
- Assigned To (Name of the Developer of the feature)
- Test Lead (Name)
- Detected in Version
- Closed in Version
- Date Detected
- Expected Date of Closure
- Actual Date of Closure
- Priority (Medium, Low, High, or Urgent)
- Severity (Range => 1 to 5)
- Status.
- Bug ID
- Attachment
- Test case Failed (Total no. of Test cases which are failing for a Bug)

Доп. материал:

О записи багов, или Найди кота

Несколько ключевых моментов, которые следует учитывать при написании отчета об ошибке?

- Воспроизведите ошибку 2-3 раза.
- Используйте некоторые ключевые слова, связанные с вашей ошибкой, и выполните поиск в инструменте отслеживания дефектов.
- Проверьте в аналогичных модулях.
- Сообщите о проблеме немедленно.
- Напишите подробные шаги для воспроизведения ошибки.
- Напишите хорошее summary дефекта. Следите за словами в процессе написания сообщения об ошибке, они не должны оскорблять людей. Никогда не используйте капс, объясняя проблему.
- Желательно проиллюстрировать проблему с помощью правильных скриншотов.
- Перед публикацией перепроверьте два или три раза ваш отчет об ошибке.

Серьезность и Приоритет Дефекта (Severity & Priority)
Разные системы баг трекинга предлагают нам разные пути описания серьезности и приоритета баг репорта, неизменным остается лишь смысл, вкладываемый в эти поля. Все знают такой баг-трекер, как Atlassian JIRA. В нем, начиная с какой-то версии вместо одновременного использования полей Severity и Priority, оставили только Priority, которое собрало в себе свойства обоих полей. Но, все же, разделение этих понятий может быть очень важно, а точнее использование обоих полей Severity и Priority, так как смысл, вкладываемый в них, различный:
Серьезность - представляет серьезность / глубину ошибки в контексте работоспособности самого ПО. Приоритет - указывает на очередность выполнения задачи или устранения дефекта. Серьезность - Описывает точку зрения приложения. Приоритет - точку зрения бизнеса. Приоритет выставляет менеджер, разработчик берет таски исходя из приоритета.

пример:
я в ходе тестирования нашел дефект, довольно критичный для приложения на мой взгляд т.к. этот дефект закрывает доступ к 20% функционала. ставлю такому багу Severity "критикал". ПМ видит багрепорт, анализирует ситуацию (поджимающие сроки, этот функционал будет рефакториться или вообще выкидываться в следующей итерации и т.п.) и ставит приорити - "медиум" или ниже.
а девелопер при работе с баг- тасктреккером руководствуется исключительно приорити, т.к. приорити и существует для регулирования очередности выполнения тасков.

Градация Серьезности дефекта (Severity):

- Блокирующий (S1 – Blocker) – тестирование значительной части функциональности вообще недоступно. Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна.
- Критический (S2 – 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, то есть не работает важная часть одной какой-либо функции либо не работает значительная часть, но имеется workaround (обходной путь/другие входные точки), позволяющий продолжить тестирование.

- Значительный (S3 – Major)– не работает важная часть одной какой-либо функции/бизнес-логики, но при выполнении специфических условий, либо есть workaround, позволяющий продолжить ее тестирование либо не работает не очень значительная часть какой-либо функции. Также относится к дефектам с высокими visibility – обычно не сильно влияющие на функциональность дефекты дизайна, которые, однако, сразу бросаются в глаза.
- Незначительный (S4 – Minor) – часто ошибки GUI, которые не влияют на функциональность, но портят юзабилити или внешний вид. Также незначительные функциональные дефекты, либо которые воспроизводятся на определенном устройстве.
- Тривиальный (S5 – Trivial) – почти всегда дефекты на GUI - опечатки в тексте, несоответствие шрифта и оттенка и т.п., либо плохо воспроизводимая ошибка, не касающаяся бизнес-логики, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Серьезность ошибки может быть низкой, средней или высокой, в зависимости от контекста.
- Дефект интерфейса пользователя - Низкая
- Пограничные дефекты - Средняя
- Обработка ошибок - Средняя
- Дефекты расчета - Высокая
- Неверно истолкованные данные - Высокая
- Аппаратные сбои - Высокий
- Проблемы совместимости - Высокая
- Дефекты потока управления - Высокая
- Условия нагрузки (утечки памяти при нагрузочном тестировании) – Высокая

Градация Приоритета дефекта (Priority):
- P1 Высокий (High)

Ошибка должна быть исправлена как можно быстрее, т.к. ее наличие является критичным для проекта.
- P2 Средний (Medium)

Ошибка должна быть исправлена, ее наличие не является критичным, но требует обязательного решения.
- P3 Низкий (Low)

Ошибка должна быть исправлена, но ее наличие не является критичным и не требует срочного решения.
Может ли быть высокий severity и низкий priority? А наоборот?
Да, такое может быть при некоторых стечениях обстоятельств.
- Очень низкая серьезность с высоким приоритетом: ошибка логотипа для любого продающего веб-сайта или крупной компании может иметь низкую серьезность, поскольку она не повлияет на функциональность веб-сайта, но может иметь высокий приоритет, поскольку вы не хотите, чтобы дальнейшая продажа продолжалась с неправильным логотипом. Или, например, сайт-визитка, показывающий только основную краткую информацию об организации может содержать грамматические ошибки, которые в иных случаях были бы с минимальным приоритетом, но в данном приведут к репутационным потерям.
- Очень высокая серьезность с низким приоритетом:

- Workaround, скорое обновление/удаление функционала: для веб-сайта авиакомпании дефект функциональности бронирования может быть серьезным, но может иметь низкий приоритет, так как исправление уже может быть запланировано в следующем цикле и возможно бронирование по телефону.
- Наоборот, обновление запланировано на потом: бухгалтерское ПО. Не работает функция генерации годового отчета. Только вот сейчас июль. И эту функцию до декабря трогать не будут. Поэтому, приоритет можно понизить.
- Редкость проявления дефекта/сложность воспроизведения для юзеров: например, если нажать на кнопку 50 раз подряд, то приложение упадет с ошибкой и будет ее показывать при попытке запуска. Или такое происходит на одной редкой модели телефона

Жизненный цикл дефекта?

Жизненный цикл дефекта - это представление различных состояний дефекта, в которых он пребывает от начального до конечного этапа своего существования. Он может варьироваться от компании к компании или даже настраиваться для некоторых проектов.
Допустим вы нашли баг и зарегистрировали его в баг трекинг системе. Согласно нашей блок-схеме он получит статус “Новый”. Тестировщик, ответственный за валидацию новых баг репортов, или координатор проекта (в зависимости от распределения ролей в вашей команде) может перевести его в один из следующих статусов:
“Отклонен”, если данный баг невалидный или повторный, или же его просто не смогли воспроизвести
“Отсрочен”, если данный баг не нужно исправлять в данной итерации
“Открыт”, если исправление бага необходимо
Рассмотрим теперь по порядку каждый из вариантов.
Отклонен. В этом случае вы можете либо поспорить о судьбе вашего багрепорта, изменив статус на “Переоткрыт” либо закрыть его - статус “Закрыт”
Отсрочен. Баг репорт в статусе “Отсрочен” можно перевести в статус “Открыт”, когда потребуется исправление либо в статус “Закрыт”, если уже не потребуется.
Открыт. Именно в таком состоянии разработчик получает баг репорт для исправления. Он может отклонить (дальнейшие действия смотрите в пункте 1) или исправить баг. Баг репорт в статусе “Исправлен” переводится на тестировщика для проверки. В случае если проблема все еще воспроизводится, выставляется статус “Переоткрыт” и баг репорт направляется назад на доработку к разработчику. Если же исправление было успешным, то баг репорт переводится в статус “Закрыт”.
* * *
Хотим отметить, что данная схема сильно упрощена. Для большей наглядности и, возможно, удобства работы на проекте, вы можете добавить дополнительные статусы и переходы, тем более, что современные баг трекинговые системы позволяют это делать. Правда имейте в виду, что излишне запутанные схемы переходов и лишние статусы могут значительно усложнить жизнь.

Примечание 1: в некоторых системах баг трекинга, созданный баг репорт сразу получает статус “Открыт” без дополнительной валидации

Примечание 2: многие баг трекинговые системы позволяют переоткрывать закрытые баги, однако лично я против такой практики, поэтому и не описывал подобный переход в выше представленном жизненном цикле
Примечание 3: Рассмотренный выше жизненный цикл основан на том, что в команде есть кто-то, ответственный за назначение баг репортов. В случае, если такой роли на проекте нет, то баги назначаются разработчиками самостоятельно, и тогда во избежании путанницы, есть смысл ввести еще один промежуточный статус “В разработке” (In progress), показывающий, что данный баг репорт уже назначен и находится на стадии исправления.
Что такое утечка дефектов и релиз бага? (Bug Leakage & Bug Release)
Релиз бага - это когда программное обеспечение или приложение передается группе тестирования, зная, что дефект присутствует в выпуске. При этом приоритет и серьезность ошибки низки, поскольку ошибка может быть удалена до окончательной передачи обслуживания.
Утечка бага - когда баг обнаруживается конечными пользователями или заказчиком, а не обнаруживается группой тестирования во время тестирования программного обеспечения.
Что означает плотность дефектов при тестировании ПО?
Плотность дефектов - это количество дефектов, подтвержденных в программном обеспечении / модуле в течение определенного периода эксплуатации или разработки, деленное на размер ПО / модуля. Это позволяет решить, готова ли часть ПО к выпуску. Плотность дефектов рассчитывается на тысячу строк кода, и обозначается KLOC. Однако не существует фиксированного стандарта для плотности ошибок, исследования показывают, что один дефект на тысячу строк кода обычно считается признаком хорошего качества проекта.
Что означает процент обнаружения дефектов при тестировании ПО?
Процент обнаружения дефектов (DDP) - это тип метрики тестирования. Он показывает эффективность процесса тестирования путем измерения соотношения дефектов, обнаруженных до выпуска и сообщенных после выпуска клиентами. Например, скажем, QA зарегистрировало 70 дефектов во время цикла тестирования, а клиент сообщил еще 20 после выпуска. DDP составит 72,1% после расчета 70 / (70 + 20) = 72,1%.
Что означает эффективность устранения дефектов при тестировании ПО? (DRP)
Эффективность устранения дефектов (DRP) - это тип метрики тестирования. Это показатель эффективности команды разработчиков для устранения проблем перед выпуском. Он измеряется как отношение зафиксированных дефектов к общему количеству обнаруженных проблем. Например, допустим, что во время цикла тестирования было обнаружено 75 дефектов, в то время как 62 из них были устранены командой разработчиков во время измерения. DRE достигнет 82,6% после расчета 62/75 = 82,6%.
Что означает эффективность Test case в тестировании ПО? (TCE)
Эффективность тестирования (TCE) - это тип метрики тестирования. Это четкий показатель эффективности выполнения Test case на этапе выполнения теста в выпуске. Это помогает в обеспечении и измерении качества Test case. Эффективность тестового набора (TCE) => (Количество обнаруженных дефектов / Количество выполненных Test case) * 100
Возраст дефекта в тестировании ПО?

Возраст дефекта - это время, прошедшее между днем обнаружения тестировщиком и днем, когда разработчик исправил его.

Что такое принцип Парето в тестировании ПО?

В тестировании ПО принцип Парето говорит о том, что 80% всех ошибок приходится на 20% кода.

Каковы различные способы применения принципа Парето в тестировании ПО?

1. Расставьте дефекты по их причинам, а не по последствиям. Не клубите ошибки, которые дают тот же результат. Группируйте проблемы в зависимости от того, в каком модуле они возникают.
2. Сотрудничайте с командой разработчиков, чтобы найти новые способы классификации проблем. Например, используйте ту же статическую библиотеку для компонентов, которые учитывают большинство ошибок.
3. Больше энергии вкладывайте в поиск проблемных областей в исходном коде, а не в случайный поиск.
4. Переупорядочьте Test case и выберите наиболее важные для начала.
5. Обратите внимание на реакцию конечного пользователя и оцените зоны риска.

В чем основное отличие отладки от тестирования? (Debugging Vs. Testing)

Тестирование заключается в обнаружении дефектов при использовании продукта, а отладка - в части кода, вызывающей сбой. Отладка изолирует проблемную область в коде, сделанном разработчиком, тогда как Тестирование идентифицирует ошибку в приложении и выполняется тестировщиком.

Почему в программном обеспечении есть ошибки?

- Недопонимание.
- Ошибки программирования.
- Сжатые сроки.
- Изменение в требованиях.
- Сложность ПО.

Что вы будете делать, если во время тестирования появится ошибка?

Когда обнаруживается ошибка, запустите больше тестов, чтобы убедиться, что проблема имеет четкое описание. Запустите еще несколько тестов, чтобы убедиться, что одна и та же проблема не существует с разными входами. Как только мы полностью уверены в ошибке, мы можем добавить подробности и сообщить о ней.

Как вы справляетесь с невоспроизводимой ошибкой?

Следующие типы ошибок относятся к категории невоспроизводимых.

- Наблюдаемые дефекты из-за недостатка памяти
- Проблемы, возникающие из-за адреса, указывающего на область памяти, которая не существует.
- Состояние гонки - ошибка проектирования • многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода

Тестировщик может предпринять следующие действия для устранения невоспроизводимых ошибок.

- Выполните шаги теста, близкие к описанию ошибки.
- Оцените тестовую среду.
- Изучите и оцените результаты выполнения теста.
- Держите ресурсы и временные ограничения под контролем.

Если продукт находится в производстве и один из его модулей обновляется, то необходимо ли провести повторную проверку?

Повторная проверка относится только к исправленным дефектам. После обновления модуля желательно выполнить регрессионное тестирование и запустить интеграционные тесты и прочие тесты для всех других модулей. После, QA следует провести Системное тестирование. Все зависит от компании и бюджета, стратегии/политики тестирования или политики качества.

Что такое анализ рисков?

Анализ рисков - это метод выявления вещей, которые могут пойти не так в проекте разработки ПО. Они могут негативно повлиять на объем, качество, своевременность и стоимость проекта. Тем не менее, все участники проекта участвуют в минимизации риска. Но именно лидер гарантирует, что вся команда понимает индивидуальную роль в управлении риском.

Как выполнять анализ рисков во время тестирования ПО? Анализ рисков - это процесс выявления скрытых проблем, которые могут помешать успешной доставке приложения. Он также устанавливает приоритетность последовательности устранения выявленных рисков для целей тестирования. Ниже приведены некоторые из рисков, которые представляют интерес для обеспечения качества.

- Новое оборудование
- Новые технологии
- Новый инструмент автоматизации
- Последовательность доставки кода
- Наличие тестовых ресурсов для приложения

Мы выделяем их в три категории, которые заключаются в следующем.

- Высокая важность: влияние ошибки значительно на другие функциональные возможности приложения
- Средний: это несколько терпимо, но не желательно.
- Низкий: терпимо. Этот вид риска не влияет на бизнес компании.

Что такое скрытый дефект? (Latent defect)

Этот дефект является существующим дефектом в системе, но он не вызывает никаких сбоев, поскольку точный набор условий еще никогда не выполнялся.

Что такое маскировка ошибок, объясните примером?

Когда наличие одного дефекта скрывает наличие другого дефекта в системе, это называется маскированием неисправностей. Пример: если «отрицательное значение» вызывает исключение необработанного системного исключения, разработчик предотвратит ввод отрицательных значений. Это решит проблему и скроет дефект обработки необработанных исключений.

Категории/подходы к отладке? (Debugging approaches)

Отладка - этап, следующий после разработки и тестирования. Тестирование предназначено для поиска ошибок, а отладка - для поиска причины конкретной ошибки.

В целом мы можем разделить методы отладки на две основные категории:

- Методы, которые анализируют производительность программы путем проверки основной памяти после сбоя программы (посмертная отладка - post-mortem debugging).
- Методы пошагового анализа программы путем ее запуска под отладчиком («динамическая отладка» или «отслеживание отладки» - “dynamic debugging” or “tracing debugging”).

Подробнее:

- Метод грубой силы (Brute Force Method): это наиболее распространенный метод отладки, однако он является наименее экономичным. Во время этого подхода программа запускается с расставленными операторами print для печати промежуточных значений с надеждой, что ряд записанных значений может облегчить обнаружение оператора с ошибкой.
- Возврат (Backtracking): это также довольно распространенный подход. Во время этого подхода, начиная с оператора, в котором был обнаружен симптом ошибки, исходный код извлекается в обратном порядке, пока не будет обнаружена ошибка. К сожалению, из-за того, что количество возможных обратных линий поставок будет увеличиваться, количество потенциальных обратных методов возрастет и должно стать невообразимо большим, что ограничивает использование этого подхода.
- Метод устранения причины (Cause Elimination Method): при таком подходе составляется список причин, которые предположительно могли способствовать появлению симптомов ошибки, и проводятся тесты для устранения каждой ошибки. Связанный метод идентификации ошибки из симптома ошибки - это анализ дерева ошибок пакета (fault tree analysis).
- Нарезка программы (Program Slicing): этот метод аналогичен поиску с возвратом. Здесь поиск сокращен срезами процесса. Слайс (нарезка) программы для конкретной переменной в конкретном операторе это набор строк, предшествующих этому оператору, которые будут влиять на значение этой переменной
- Анализ дерева отказов (FTA - Fault tree analysis) - это нисходящий дедуктивный анализ отказов, в котором нежелательное состояние системы анализируется с использованием булевой логики для объединения серии событий нижнего уровня

Что такое Эффективность удаления дефектов? (DRE - Defect Removal Efficiency)
Этот показатель используется для измерения эффективности теста. Из этого показателя мы узнаем, сколько ошибок мы обнаружили в наборе Test case. Формула для расчета DRE имеет вид
DRE = Количество ошибок во время тестирования / (количество ошибок во время тестирования + количество ошибок, обнаруженных пользователем)
Что такое сортировка дефектов? (Bug triage)
Это формальный процесс, позволяющий определить, какие ошибки являются важными, путем определения их приоритетов на основе их серьезности, частоты, риска и других важных параметров. Тестировщики назначают приоритет (высокий, средний, низкий) каждой ошибке на bug triage meeting и в зависимости от приоритета эти ошибки будут исправлены в соответствующем порядке. Делая это, мы экономим ресурсы организации.

----- SDLC и STLC -----
Что вы знаете о жизненном цикле разработки ПО? (SDLC - Software Development Lifecycle)
SDLC определяет все стандартные фазы, которые участвуют в процессе разработки ПО. Жизненный цикл SDLC - это процесс поэтапной разработки ПО, которому следуют в программных проектах. Каждый этап SDLC производит результаты, необходимые для следующего этапа жизненного цикла. Требования транслируются в дизайн. Код пишется в соответствии с дизайном. Тестирование должно проводиться на разработанном продукте в соответствии с требованиями. Развертывание должно быть сделано после завершения тестирования.

- Этап требований (Requirement Phase): Сбор и анализ требований является наиболее важной фазой в жизненном цикле разработки программного обеспечения. Бизнес-аналитик получает требование от Заказчика / Клиента в соответствии с бизнес-потребностями и документирует требования в Спецификации бизнес-требований (Business Requirement Specification, название документа зависит от Организации. Некоторые примеры: Спецификация требований клиента (CRS - Customer Requirement Specification), Бизнес-спецификация (BS - Business Specification), и т. д.), и предоставляет то же самое для команды разработчиков.
- Этап анализа (Analysis Phase): После сбора и анализа требований следующим шагом является определение и документирование требований к продукту и их утверждение заказчиком. Это делается с помощью документа Спецификация требований к программному обеспечению (SRS - Software Requirement Specification). SRS состоит из всех требований к продукту, которые должны быть спроектированы и разработаны в течение жизненного цикла проекта. Ключевыми людьми, вовлеченными в этот этап, являются Project Manager, Business Analyst и Senior члены команды. Результатом этого этапа является спецификация требований к программному обеспечению.
- Этап проектирования (Design Phase): высокоуровневый дизайн (HLD - High-Level Design) – это про архитектуру программного продукта, который должен быть разработан, и выполняется архитекторами и старшими разработчиками. Низкоуровневый дизайн (LLD - Low-Level Design) - это делается старшими разработчиками. Он описывает, как должна работать каждая функция продукта и как должен работать каждый компонент. Здесь будет только дизайн, а не код. Результатом этого этапа является документ высокого уровня и документ низкого уровня, который служит входными данными для следующего этапа.
- Этап разработки (Development Phase): Разработчики всех уровней участвуют в этом этапе. На этом этапе мы начинаем создавать программное обеспечение и начинаем писать код для продукта. Результатом этого этапа является исходный код документа (SCD - Source Code Document) и разработанный продукт.
- Этап тестирования (Testing Phase): Когда программное обеспечение готово, оно отправляется в отдел тестирования, где группа тестирования тщательно тестирует его на наличие различных дефектов. Они либо тестируют программное обеспечение вручную, либо используют инструменты автоматического тестирования, в зависимости от процесса, определенного в STLC (жизненный цикл тестирования программного обеспечения), и гарантируют, что каждый компонент программного обеспечения работает нормально. Как только QA удостоверится, что программное обеспечение не содержит серьезных ошибок, оно переходит к следующему этапу, который является реализацией. Результатом этого этапа является качество продукта и артефакты тестирования.
- Этап развертывания и обслуживания (Deployment and Maintenance Phase): После успешного тестирования продукт доставляется / развертывается заказчику для использования. Развертывание выполняется Deployment/Implementation engineers. Однажды, когда клиенты начнут использовать разработанную систему, возникнут реальные проблемы, и время от времени их придется решать. Устранение проблем, обнаруженных заказчиком, происходит на этапе обслуживания. Техническое

обслуживание должно выполняться в соответствии Соглашением об уровне обслуживания (SLA - Service Level Agreement)
Что такое цикл/колесо Деминга? (Deming circle/cycle/wheel)
Цикл PDCA (Plan – Do – Check – Act) - это непрерывный (до получения конечного результата) итеративный четырехступенчатый метод управления, используемый в бизнесе для постоянного улучшения процессов. Это одна из ключевых концепций качества, и она также называется кругом/ циклом/ колесом Деминга.

- Plan: Запланируйте изменения (либо для решения проблемы, либо для улучшения некоторых областей) и решите, какую цель достичь. Здесь мы определяем цель, стратегию и вспомогательные методы для достижения цели нашего плана.
- Do: Разработать или пересмотреть бизнес-требования в соответствии с планом. Здесь мы реализуем план (с точки зрения реализации плана) и проверяем его эффективность
- Check: Оцените результаты, чтобы убедиться, что мы достигнем целей, как планировалось. Здесь мы составляем контрольный список для записи того, что прошло хорошо, а что не сработало (извлеченные уроки)
- Act: Если изменения не соответствуют запланированным, продолжайте цикл для достижения цели с другим планом. Здесь мы принимаем меры по факту того, что не работает, как планировалось. Задача состоит в том, чтобы продолжать пытаться улучшить процесс с другим планом.

Модели разработки ПО?
Waterfall (каскадная модель, или «водопад»). В этой модели разработка осуществляется поэтапно: каждая следующая стадия начинается только после того, как заканчивается предыдущая. Если все делать правильно, «водопад» будет наиболее быстрой и простой моделью. Применяется уже почти полвека, с 1970-х.

Преимущества «водопада»: Разработку просто контролировать. Заказчик всегда знает, чем сейчас заняты программисты, может управлять сроками и стоимостью. Стоимость проекта определяется на начальном этапе. Все шаги запланированы уже на этапе согласования договора, ПО пишется непрерывно «от и до».
Не нужно нанимать тестировщиков с серьезной технической подготовкой. Тестировщики смогут опираться на подробную техническую документацию.
Недостатки каскадной модели: Тестирование начинается на последних этапах разработки. Если в требованиях к продукту была допущена ошибка, то исправить ее будет стоить дорого. Тестировщики обнаружат ее, когда разработчик уже написал код, а технические писатели — документацию.
Заказчик видит готовый продукт в конце разработки и только тогда может дать обратную связь. Велика вероятность, что результат его не устроит. Разработчики пишут много технической документации, что задерживает работы. Чем обширнее документация у проекта, тем больше изменений нужно вносить и дольше их согласовывать.
«Водопад» подходит для разработки проектов в медицинской и космической отрасли, где уже сформирована обширная база документов (СНиПов и спецификаций), на основе которых можно написать требования к новому ПО. При работе с каскадной моделью основная задача — написать подробные требования к разработке. На этапе тестирования не должно выясниться, что в них есть ошибка, которая влияет на весь продукт.

V-образная модель (разработка через тестирование)
Это усовершенствованная каскадная модель, в которой заказчик с командой программистов одновременно составляют требования к системе и описывают, как будут тестировать ее на каждом этапе. История этой модели начинается в 1980-х.

Преимущества V-образной модели: Количество ошибок в архитектуре ПО сводится к минимуму.
Недостатки V-образной модели: Если при разработке архитектуры была допущена ошибка, то вернуться и исправить ее будет стоить дорого, как и в «водопаде».
V-модель подходит для проектов, в которых важна надежность и цена ошибки очень высока. Например, при разработке подушек безопасности для автомобилей или систем наблюдения за пациентами в клиниках.

Incremental Model (инкрементная модель)
Это модель разработки по частям (increment в переводе с англ. — приращение) уходит корнями в 1930-е. Рассмотрим ее на примере создания социальной сети.
Заказчик решил, что хочет запустить соцсеть, и написал подробное техническое задание. Программисты предложили реализовать основные функции — страницу с личной информацией и чат. А затем протестировать на пользователях, «взлетит или нет».
Команда разработки показывает продукт заказчику и выпускает его на рынок. Если и заказчику, и пользователям социальная сеть нравится, работа над ней продолжается, но уже по частям.
Программисты параллельно создают функциональность для загрузки фотографий, обмена документами, прослушивания музыки и других действий, согласованных с заказчиком. Инкремент за инкрементом они совершенствуют продукт, приближаясь к описанному в техническом задании.

Преимущества инкрементной модели
Не нужно вкладывать много денег на начальном этапе. Заказчик оплачивает создание основных функций, получает продукт, «выкатывает» его на рынок — и по итогам обратной связи решает, продолжать ли разработку. Можно быстро получить фидбэк от пользователей и оперативно обновить техническое задание. Так снижается риск создать продукт, который никому не нужен. Ошибка обходится дешевле. Если при разработке архитектуры была допущена ошибка, то исправить ее будет стоить не так дорого, как в «водопаде» или V-образной модели.
Недостатки инкрементной модели: Каждая команда программистов разрабатывает свою функциональность и может реализовать интерфейс продукта по-своему. Чтобы этого не произошло, важно на этапе обсуждения техзадания объяснить, каким он будет, чтобы у всех участников проекта сложилось единое понимание.
Разработчики будут оттягивать доработку основной функциональности и «пилить мелочевку». Чтобы этого не случилось, менеджер проекта должен контролировать, чем занимается каждая команда.
Инкрементная модель подходит для проектов, в которых точное техзадание прописано уже на старте, а продукт должен быстро выйти на рынок.

Iterative Model (итеративная модель)

Это модель, при которой заказчик не обязан понимать, какой продукт хочет получить в итоге, и может не прописывать сразу подробное техзадание.

Рассмотрим на примере создания мессенджера, как эта модель работает.
Заказчик решил, что хочет создать мессенджер. Разработчики сделали приложение, в котором можно добавить друга и запустить чат на двоих.
Мессенджер «выкатили» в магазин приложений, пользователи начали его скачивать и активно использовать. Заказчик понял, что продукт пользуется популярностью, и решил его доработать.
Программисты добавили в мессенджер возможность просмотра видео, загрузки фотографий, записи аудиосообщений. Они постепенно улучшают функциональность приложения, адаптируют его к требованиям рынка.
Преимущества итеративной модели: Быстрый выпуск минимального продукта дает возможность оперативно получать обратную связь от заказчика и пользователей. А значит, фокусироваться на наиболее важных функциях ПО и улучшать их в соответствии с требованиями рынка и пожеланиями клиента. Постоянное тестирование пользователями позволяет быстро обнаруживать и устранять ошибки.
Недостатки итеративной модели: Использование на начальном этапе баз данных или серверов — первые сложно масштабировать, а вторые не выдерживают нагрузку. Возможно, придется переписывать большую часть приложения. Отсутствие фиксированного бюджета и сроков. Заказчик не знает, как выглядит конечная цель и когда закончится разработка.
Итеративная модель подходит для работы над большими проектами с неопределенными требованиями, либо для задач с инновационным подходом, когда заказчик не уверен в результате.

Spiral Model (спиральная модель)
Используя эту модель, заказчик и команда разработчиков серьезно анализируют риски проекта и выполняют его итерациями. Последующая стадия основывается на предыдущей, а в конце каждого витка — цикла итераций — принимается решение на основе рисков, продолжать ли развивать проект.

Рассмотрим, как функционирует эта модель, на примере разработки системы «Умный дом».
Заказчик решил, что хочет сделать такую систему, и заказал программистам реализовать управление чайником с телефона. Они начали действовать по модели «водопад»: выслушали идею, провели анализ предложений на рынке, обсудили с заказчиком архитектуру системы, решили, как будут ее реализовывать, разработали, протестировали и «выкатили» конечный продукт.
Заказчик оценил результат и риски: насколько нужна пользователям следующая версия продукта — уже с управлением телевизором. Рассчитал сроки, бюджет и заказал разработку. Программисты действовали по каскадной модели и представили заказчику более сложный продукт, разработанный на базе первого.
Заказчик подумал, что пора создать функциональность для управления холодильником с телефона. Но, анализируя риски, понял, что в холодильник сложно встроить Wi-Fi-модуль, да и производители не заинтересованы в сотрудничестве по этому вопросу. Следовательно, риски превышают потенциальную выгоду. На основе полученных данных заказчик решил прекратить разработку и совершенствовать

имеющуюся функциональность, чтобы со временем понять, как развивать систему «Умный дом».
Спиральная модель похожа на инкрементную, но здесь гораздо больше времени уделяется оценке рисков. С каждым новым витком спирали процесс усложняется. Эта модель часто используется в исследовательских проектах и там, где высоки риски.
Преимущества спиральной модели: Большое внимание уделяется проработке рисков.
Недостатки спиральной модели: Есть риск застрять на начальном этапе — бесконечно совершенствовать первую версию продукта и не продвинуться к следующим. Разработка длится долго и стоит дорого.

На основе итеративной модели была создана Agile — не модель и не методология, а скорее подход к разработке.

Что такое Agile?
В классической модели waterfall каждая стадия начиналась после предыдущей без возврата назад и только в самом конце начиналось тестирование. Можно ли обеспечить качество, когда уже все готово? В книге «Как тестируют в Google» говорится, что QA не отвечает единолично за качество продукта, за это отвечает вся команда и в первую очередь разработчики.
В результате post-development тестирования создавалась иллюзия качественного продукта, но это не обеспечение качества, а скорее QC. Еще одним следствием следования каскадной модели являлось то, что команда старалась реализовать сразу все требования и к этапу тестирования выяснялось, что требуется много доделок/переделок и в результате релиз откладывался. Помимо прочего, пока разработчики писали код, тестировщики бездействовали. Безусловно, что-то писалось по требованиям и интуитивно, но смысл понятен. Нередко из-за срыва срока релизов сокращался срок, отводимый на тестирование, что также неблагоприятно сказывалось на итоговом качестве продукта.
Далее вместе с прогрессом пошла эволюция процессов SDLC и пришло понимание необходимости встраивания процессов обеспечения качества в жизненный цикл разработки продукта. Таким образом появлялись новые модели разработки и однажды группой энтузиастов был придуман Agile-манифест — основной документ, содержащий описание ценностей и принципов гибкой разработки программного обеспечения (4 ценности, 12 принципов):
Ценности:
- Люди и взаимодействие важнее процессов и инструментов.
- Работающий продукт важнее исчерпывающей документации.
- Сотрудничество с клиентом важнее согласования условий контракта.
- Готовность к изменениям важнее следования первоначальному плану.

Основные принципы:
- Наивысшим приоритетом является удовлетворение потребностей клиента, благодаря регулярной и ранней поставке ценного программного обеспечения.
- Изменение требований приветствуется, даже на поздних стадиях разработки.
- Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.
- На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.

- Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.
- Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.
- Работающий продукт — основной показатель прогресса.
- Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно.
- Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
- Простота — искусство минимизации лишней работы — крайне необходима.
- Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
- Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

Agile включает в себя практики, подходы и методологии, которые помогают создавать продукт более эффективно:
- экстремальное программирование (Extreme Programming, XP);
- бережливую разработку программного обеспечения (Lean);
- фреймворк для управления проектами Scrum;
- разработку, управляемую функциональностью (Feature-driven development, FDD);
- разработку через тестирование (Test-driven development, TDD);
- методологию «чистой комнаты» (Cleanroom Software Engineering);
- итеративно-инкрементальный метод разработки (OpenUP);
- методологию разработки Microsoft Solutions Framework (MSF);
- метод разработки динамических систем (Dynamic Systems Development Method, DSDM);
- метод управления разработкой Kanban.

Не все перечисленное в списке — методологии. Например, Scrum чаще называют не методологией, а фреймворком. В чем разница? Фреймворк — это более сформированная методология со строгими правилами. В скраме все роли и процессы четко прописаны. Помимо Scrum, часто используют Kanban. Сегодня это одна из наиболее популярных методологий разработки ПО. Команда ведет работу с помощью виртуальной доски, которая разбита на этапы проекта. Каждый участник видит, какие задачи находятся в работе, какие — застряли на одном из этапов, а какие уже дошли до его столбца и требуют внимания.
В отличие от скрама, в канбане можно взять срочные задачи в разработку сразу, не дожидаясь начала следующего спринта. Канбан удобно использовать не только в работе, но и в личных целях — распределять собственные планы или задачи семьи на выходные, наглядно отслеживать прогресс.

(Красный – waterfall, синий – разработка по гибкой методологии)
Впоследствии был разработан отдельный Манифест тестирования в Agile:
- постоянное тестирование, а не только в конце разработки
- предотвращение багов более значимо, чем их поиск
- понимание тестируемого продукта выше проверки функционала

- построение лучшей системы в связке с командой выше поиска методов ее сломать
- вся команда отвечает за качество, а не только тестировщик

Доп. материал:
10 • примеров эффективного общения для тестировщиков
• • A Brief Guide to Testing in DevOps
Как выживать тестировщику в Agile среде
Стратегия тестирования краткосрочного проекта
Что такое Scrum?
Scrum – наиболее популярный Agile-подход, для многих людей согласно статистике эти термины являются синонимами. В целом о Scrum:

- итеративно-инкрементальная разработка. Слово «итеративная» означает, что разработка разбивается на равные по длительности промежутки времени — спринты. Один спринт занимает от одной до четырех недель. Слово «инкрементальная» подразумевает, что в результате итерации получается новый, потенциально рабочий продукт, решающий бизнес-проблему. Такой продукт называется инкрементом продукта;
- самоорганизующаяся команда, в которой нет проектного менеджера;
- в команде присутствует SCRUM-master;
- в команде есть человек со стороны заказчиков — product owner, или владелец продукта;
- весь период разработки разбит на промежутки времени — спринты. Длина спринта устанавливается в начале проекта и меняется только в том случае, если всплывают неучтенные детали, мешающие уложиться в заданные рамки;
- задачи (функциональные требования, баги, правки заказчика и т.п.) формируют пул работ — бэклог. Изначально в него входят только требования заказчика;
- в начале спринта (и в любом его месте, если это нужно) проводится Backlog Grooming — обработка задач из бэклога. В результате получается проработанный бэклог на 2-3 будущих спринта. Затем PO и SCRUM-команда формулирует цель спринта и ожидаемый результат, и команда составляет бэклог спринта;
- после планирования спринта его состав стараются не менять. Если добавление новых задач все же происходит, то из спринта исключаются старые и сопоставимые по ценности задачи. Если эти изменения привели к смене цели спринта, то спринт отменяется и планируется заново;
- ежедневные короткие SCRUM-митинги. Они дают понять, как движется процесс, а команда в курсе того, идут ли они к цели спринта или нет;
- в конце спринта выполненные задачи либо подтверждаются, либо отклоняются и возвращаются в бэклог;
- по результатам спринта команда получает инкремент продукта.

Роли:
Product Owner Scrum Master Команда
определяет особенности продукта.управляет командой и заботится о продуктивности команды команда обычно состоит из 5-9 человек.
определяет дату релиза и соответствующие функции поддерживает block список и устраняет барьеры в разработке в нее входят разработчики, дизайнер, а иногда и тестировщики и т. д.

устанавливает приоритеты функций в соответствии с рыночной стоимостью и прибыльностью продукта. координирует все роли и функции команда организует и планирует свою работу самостоятельно
несет ответственность за прибыльность продукта защищает команду от внешних помех имеет право делать все в рамках проекта для достижения цели спринта
может принять или отклонить результат задания приглашает на ежедневные разборы, обзор спринта и встречи по планированию активно участвуют в ежедневных церемониях

На практике в скрам-тестировании успешность спринта - это когда все задачи, которые были добавлены в Scrum, оказались в статусе Done. Хотя много зависит от определения, что такое Done в вашем проекте. Скрам по своей сути это просто таймбокс для работ и если не уложились в сроки – то это проблема планирования и на ретроспективе нужно разобраться почему это произошло и зафиксировать на будущее.

Доп. материал:
Мини-справочник и руководство по Scrum
Scrum-• мем на злобу дня
Какие вообще особенности у тестирования в Scrum?
Классика типа Савина или Канера повествует скорее о традиционных моделях (именно поэтому там уделено столько внимания дизайну, документированию, видам тестирования и т.п., т.к. на все это там есть время) но в гибких все значительно отличается. В гибкой методологии времени на разработку тест-кейсов низкого уровня и прочей документации обычно не бывает, поэтому подготавливаются чек-листы. Наборы проверок могут определяться как по не формализованным требованиям, так и на основе рисков (risk based). Ну и тестирование на основе экспертизы – самый простой подход к тестированию, но в то же время и самый рискованный, потому что все тестирование завязывается на экспертизу специалиста, выполняющего тестирование. В некоторых случаях мы все же можем формализовать Стратегию тестирования (порядок тестирования и подход к выполнению работ по тестированию ПО) и Тест-план (в кратком содержании для спринта не менее 2-х недель, при меньших сроках спринта ведение не актуально). Требования к документации в аджайл: она должна служить потребностям команды, живая документация ценнее архивной.

3+1 принципа тестирования в аджайл: предотвращение, автоматизация (авто QA - функциональное, приемочное. Devs - юнит, интеграция), гибкость, здравый смысл.
В agile user stories пишет Product Owner или Business Analyst. Кодеры по ним кодят, а тестировщики по ним тестят. ПО или БА к каждой ЮС будут писать критерии приемки, а тестировщики на основании их будут писать кейсы. При планировании спринта тестировщик должен выбрать пользовательскую историю из журнала невыполненных работ, которую следует протестировать. Как тестировщик, он / она должен решить, сколько часов (оценка усилий) потребуется, чтобы завершить тестирование для каждой из выбранных пользовательских историй. Как тестировщик , он / она должен знать, каковы цели спринта. В качестве тестировщика, внести свой вклад в процесс расстановки приоритетов. Тестировщик отвечает за разработку сценариев автоматизации. Он планирует автоматизированное тестирование с помощью системы непрерывной интеграции (CI). Автоматизация получает важность из-за коротких сроков

доставки.Выполнение нефункционального тестирования для утвержденных пользовательских историй. Тестировщик взаимодействует с клиентом и владельцем продукта, чтобы определить критерии приемки для приемочных испытаний. В конце спринта тестировщик также проводит приемочное тестирование (UAT) в некоторых случаях и подтверждает полноту тестирования для текущего спринта. Узнать больше можно по ссылкам в теме "Ты – единственный тестировщик на проекте. Что делать?".

Доп. материал:
Тестирование в рамках SCRUM. Тернии, грабли и успехи
QA-• процесс в Miro: отказ от водопада и ручного тестирования, передача ответственности за качество всей команде
Тесты должна писать разработка (?)
Как провести ретроспективу
The Role of QA in Sprint Planning
В чем отличие Kanban от Scrum?
В отличие от Scrum, в команде канбан отсутствуют роли владельца продукта и модератора, а процесс разработки делится не на универсальные спринты, а на стадии выполнения задач («Планируется», «Разрабатывается», «Тестируется», «Завершено»). Жизненный цикл задачи отображается на канбан-доске, физической или электронной. Такая визуализация делает рабочий процесс открытым и понятным для всех участников, что особенно важно в Agile, когда у команды нет одного формального руководителя.
Канбан, как и другие практики бережливого производства, пришедшие из Японии, направлен на достижение баланса и выравнивание нагрузки исполнителей. Эффективность работы оценивается по среднему времени жизни задачи, от начальной до конечной стадии. Если задача прошла весь путь быстро, то команда проекта работала продуктивно и слаженно. Иначе – необходимо решать проблему: искать, где и почему возникли задержки и чью работу надо оптимизировать
Что знаете о User stories в гибких подходах к разработке?
Пользовательские истории (англ. User Story) - способ описания требований к разрабатываемой системе, сформулированных как одно или несколько предложений на повседневном или деловом языке пользователя. Пользовательские истории – это один из самых быстрых способов документирования требований клиента (цель документирования состоит в том, чтобы оперативно и без затрат реагировать на возникающие изменения).
Главное действующее лицо User story – это некий персонаж, который будет совершать какие-либо действия с нашим тестируемым продуктом с учетом его потребностей. Персонаж сопровождается описанием проблем, которые он может (и хочет) решить с помощью нашего продукта. Потребность представляет собой тезис в 1-2 предложения. Для одного пользователя может быть разработано несколько (например, 4-6) User Story.
Для того, чтобы персонажи стали эффективными инструментами проектирования сайта, потребуется не только провести исследование, но и выявить закономерности в поведении пользователей. Как правило, принято создавать и детально прорабатывать одного основного (как показано на mind-map ниже) и несколько второстепенных персонажей.

Что значит жизненный цикл тестирования ПО? (STLC – Software Testing Lifecycle)

STLC это модель тестирования, которая предлагает выполнять тестирование систематическим и запланированным способом.

Модель STLC устанавливает следующие этапы:

- Инициация,
- Выявление и анализ требований прямых и косвенных
- Планирование испытаний
- Генерация Test case,
- Отбор показательных Test case,
- Настройка среды
- Проведение проверок,
- Фиксация результатов,
- Анализ результатов,
- Передача информации о соответствии проверенного продукта требованиям.

- Инициация – событие, которое извещает команду тестирования о необходимости сессии тестирования

- Выявление требований (RA) – пожалуй, один из главных шагов в процессе тестирования. Необходимо собрать всю доступную информацию о предмете тестирования, вариантах использования и т. п. Первый источник – техническая документация и юзер-стори – это прямые требования. Если некоторые спецификации не являются точными или имеют разногласия, то заинтересованные стороны, такие как бизнес-аналитик (BA), архитекторы, клиенты, дают ясность. Качество же косвенных требований во многом зависят от добросовестности, ответственности, квалификации тестировщика и всей команды проекта.
- Планирование тестирования (TP) - Как правило, на этом этапе старший QA определяет усилия и смету расходов по проекту, а также готовит и завершает план тестирования. На этом этапе также определяется стратегия тестирования. Команда тестирования выполняет следующие задачи на этапе TP:
- Подготовка плана тестирования / стратегии для различных типов тестирования
- Выбор тестовых инструментов
- Оценка усилий
- Планирование ресурсов и определение ролей и обязанностей.
- Требования к обучению
- Расписание, критерии начала и окончания
- Оценка рисков

- Генерация Test case – выявление всех возможных случаев использования продукта, его характеристик и особенностей в процессе эксплуатации. Это значит: всех случаев, которые тестировщик может «придумать» на основе прямых и косвенных требований, известных ему. Этот этап требует высокой квалификации специалиста по тестированию. При выполнении этой задачи также необходимо подготовить входные данные, необходимые для тестирования. Как только план тестирования будет готов, он должен быть рассмотрен Senior или Lead. Один из документов, который должна подготовить группа, - это матрица отслеживания требований (RTM). Это общеотраслевой стандарт, обеспечивающий правильное сопоставление тест-кейсов с требованиями.

- Отбор Test case – отбор наиболее показательных, значимых и воспроизводимых Test case. От этого этапа зависит, насколько тестирование будет полезным, эффективным и анализируемым. Количество косвенных требований стремится к бесконечности, и проверять их все подряд – полный абсурд, но подобные кейсы должны быть сгенерированы хотя бы в голове проверяющего.
- Настройка тестовой среды в STLC - Среда тестирования определяет условия программного и аппаратного обеспечения, при которых тестируется рабочий продукт. Настройка тестовой среды является одним из важнейших аспектов процесса тестирования и может выполняться параллельно с этапом разработки тестового набора. Команда тестирования может быть не вовлечена в это действие, если клиент / команда разработчиков предоставляет среду тестирования, и в этом случае команда тестирования должна выполнить проверку готовности (тестирование дыма) данной среды. Какие задачи выполняются на этапе настройки среды тестирования STLC?
- Ознакомление с необходимой архитектурой, настройка среды и подготовка требований к оборудованию и ПО для среды тестирования.
- Настройка тестовой среды и тестовых данных
- Выполнение Smoke тестирования
- Проведение проверок – тут все понятно. На этом этапе команда запускает тестовые наборы в соответствии с планом тестирования, определенным в предыдущих шагах, либо adhoc (интуитивно, свободный поиск, без документации). В любом случае это проводится согласно списку отобранных проверок.
- Фиксация результатов – создание внутренней и внешней тестовой документации в формализованном виде или в виде записей и т. п. На данном этапе отчет о тестировании даже если и создается, то не считается законченным.
- Анализ результатов – вынесение решения о соответствии проверенного продукта требованиям. Формализация данного решения и его обоснование в виде отчета о тестировании. Сюда также входят процедуры по оценке покрытия требований проверками, тайм-шитинг и пр. Таким образом, проводится анализ не только результатов, но и самой сессии тестирования.
- Передача информации о соответствии продукта требованиям. Формально: передача внешней тестовой документации заинтересованным в ней сторонам, зачастую инициатору сессии тестирования. В общем случае: помимо документации предоставляется информация о рисках, которые были выявлены в продукте, требованиях, процессах, передаются рекомендации по отработке этих рисков и т. п.

Что вы знаете о техниках оценки теста? (Test Estimation)
Оценка теста - это управленческая деятельность, которая приблизительно показывает, сколько времени потребуется для выполнения Задачи. Оценка усилий для теста является одной из основных и важных задач в управлении тестированием (Test Management). Что оценивается?
- Ресурсы необходимы для выполнения любых задач проекта. Это могут быть люди, оборудование, средства, финансирование или что-то еще, что может быть определено для завершения деятельности по проекту. Время - самый ценный ресурс в проекте. Каждый проект имеет срок доставки.
- Человеческие навыки означают знания и опыт членов Команды. Они влияют на вашу оценку. Например, команде, члены которой имеют низкие навыки тестирования, потребуется больше времени для завершения проекта, чем команде, обладающей высокими навыками тестирования.

- Стоимость - это бюджет проекта. Вообще говоря, это означает, сколько денег нужно, чтобы закончить проект.

В чем разница между SDLC и STLC?

SDLC определяет все стандартные фазы, которые участвуют в процессе разработки ПО, тогда как процесс STLC определяет различные действия для улучшения качества продукта. SDLC - это жизненный цикл разработки, тогда как STLC - это жизненный цикл тестирования. В SDLC команда разработчиков создает планы проектирования высокого и низкого уровня, в то время как в STLC аналитик тестов создает Систему, План тестирования интеграции. В SDLC разрабатывается реальный код, и фактическая работа выполняется в соответствии с проектной документацией, тогда как в STLC команда тестирования готовит среду тестирования и выполняет тестовые примеры. Жизненный цикл SDLC помогает команде завершить успешную разработку ПО, в то время как фазы STLC охватывают только тестирование ПО.

Что такое быстрая разработка приложений? (RAD - Rapid Application Development)

Быстрой разработкой приложений формально является параллельное развитие функций и последующей интеграции. Компоненты/функции развиваются параллельно, как если бы они были мини-проекты, события, time-boxed, доставлены и собраны в работающий прототип. Это может очень быстро дать клиенту что-то увидеть и использовать, и обеспечить обратную связь по поводу доставки и их требований. Быстрое изменение и развитие продукта возможно с помощью этой методики, однако спецификация продукта должна быть разработана для продукта в какой-то момент, и проект должен быть размещен под более формальный контроль перед запуском в производство.

Что такое разработка через тестирование (TDD - Test Driven Development)?

В традиционном подходе сначала пишут код, который затем покрывают тестами; в этом подходе сначала разрабатываются тесты, которые в будущем должны быть пройдены кодом. Разработчик будет писать код, пока тесты не начнут проходить. Разработка через тестирование начинается с проектирования и разработки тестов для каждой небольшой функциональности приложения. Также очевидно, в TDD вы получаете 100% покрытия кода тестами.

Есть два уровня TDD:

- Acceptance TDD (ATDD): вы пишете один приемочный тест. Этот тест удовлетворяет требованиям спецификации или удовлетворяет поведению системы. После этого пишете достаточно производственного / функционального кода, чтобы выполнить этот приемочный тест. Приемочный тест фокусируется на общем поведении системы. ATDD также был известен как BDD - Behavioral Driven Development.
- Developer TDD: вы пишете один тест разработчика, то есть модульный тест, а затем просто достаточно производственного кода для выполнения этого теста. Модульное тестирование фокусируется на каждой небольшой функциональности системы. Это называется просто TDD. Основная цель ATDD и TDD - определить подробные, выполнимые требования для вашего решения точно в срок (JIT). JIT означает принятие во внимание только тех требований, которые необходимы в системе, что повышает эффективность.

Что такое Value Driven Testing (тестирование на основе ценности)?

Чтобы понять, как обстоят дела с тестированием на проекте, нужно проанализировать его эффективность с точки зрения качества создаваемого продукта и процессов. Тут можно рассчитывать плотность дефектов, разрывы, утечки, эффективность тест-кейсов, RC, FDP, DDP, PTC, MTTD, TDE и десятки других метрик тестирования. Но, чтобы определить рентабельность такого тестирования, необходимо считать деньги. Деньги и их возрастающий поток — основная цель заказчика в большинстве случаев разработки ПО.
Чтобы правильно принимать управленческие решения, тест-менеджеру необходимо в полной мере ориентироваться в себестоимости активностей по тестированию, видеть зоны развития и пути оптимизации процессов. Заказчику также важно понимать, за что он платит и почему, где он теряет, а где зарабатывает. Один в поле не воин, и задача так называемого архитектора постараться заставить пчел реально осознать, сколько денег они приносят заказчику, сколько помогли сэкономить. Сэкономленные деньги не обязательно, но могут формировать фонд для потенциального увеличения оплаты труда тех же пчел.
Цена и ценность
Любое качество имеет свою цену: Cost of Quality = Cost of Poor Quality + Cost of Good Quality:

Общая стоимость тестирования достаточно велика, но стоит лишь оценить стоимость плохого тестирования, как она уже кажется вполне приемлемой. Уоррен Баффет как-то сказал, что цена — это то, что вы платите, а ценность — то, что получаете. И не всегда они совпадают. Качество еще не означает ценность. Попробуйте сегодня продать очень качественную печатную машинку либо убедить заказчика, что для него ценнее будет зарелизить фичу не послезавтра, а через год, ведь за это время вы еще лучше все протестите и качество будет выше. Не получится. Дорога ложка к обеду, и time to market никто не отменял.
Задача в том, чтобы достичь оптимального соотношение цена/качество для заказчика. Почему оптимального? Потому что по мере увеличения затрат на поиск дефектов и их устранения стоимость поломки будет снижаться до тех пор, пока не будет достигнута оптимальная точка, после которой дальнейшее увеличение активностей по тестированию станет экономически нецелесообразным.

Источник: Что нужно знать о Value Driven Testing. Анализируем ценность и экономическую целесообразность тестирования
TDD в Agile Model Driven Development (AMDD)
TDD очень хорош в детальной спецификации и валидации. Он не может продумать более важные вопросы, такие как общий дизайн, использование системы или пользовательский интерфейс. AMDD решает проблемы гибкого масштабирования, которых нет в TDD. Таким образом, AMDD используется для больших проблем.
Жизненный цикл AMDD:

- Iteration 0: Envisioning
- Initial requirements envisioning.
- Initial Architectural envisioning.
- Iteration modeling.
- Model storming.
- Test Driven Development (TDD).
- Reviews.

TDD сокращает цикл обратной связи программирования, AMDD сокращает цикл обратной связи моделирования. TDD - детальная спецификация, AMDD работает для больших проблем. TDD способствует разработке качественного кода, AMDD способствует качественному общению с заинтересованными сторонами и разработчиками. TDD общается с программистами, AMDD общается с бизнес-аналитиками, заинтересованными сторонами и специалистами по данным. TDD не визуально ориентированный, AMDD визуально ориентированный. TDD имеет ограниченную область применения, AMDD имеет широкий охват. Это предполагает работу по достижению общего понимания. Оба поддерживают эволюционное развитие.
Тестирование на основе моделей (MDD - Model-driven Development)
Это метод тестирования черного ящика, при котором поведение тестируемого программного обеспечения во время выполнения проверяется на основе прогнозов, сделанных моделями. Модель - это описание поведения системы. Поведение может быть описано в виде наглядной схемы, Data Flow, Control Flow, Dependency Graphs, Decision Tables, State transition machines или mind map. Простой аналогией модели в тестировании является электрическая схема при разработке электроприбора. Этот подход к тестированию требуется, когда высока цена ошибки в большом продукте и нужно как можно раньше попытаться ее предотвратить.

Доп. материал:
Тестирование на основе моделей
Тестирование на основе данных (DDT - Data Driven testing)
DATA DRIVEN testing - это инфраструктура автоматизации тестирования, которая хранит тестовые данные в виде таблицы. Это позволяет инженерам по автоматизации иметь единый сценарий тестирования, который может выполнять тесты для всех тестовых данных в таблице. В этой структуре входные значения считываются из файлов данных и сохраняются в переменную в тестовых сценариях. Ddt (тестирование на основе данных) позволяет объединять как положительные, так и отрицательные Test case в один тест. В среде автоматизации тестирования на основе данных входные данные могут храниться в одном или нескольких источниках данных, таких как xls, XML, csv и базы данных.
Часто у нас есть несколько наборов данных, на которых мы должны запускать одни и те же тесты. Создание отдельного теста для каждого набора данных - длительный и трудоемкий процесс. Data Driven testing Framework решает эту проблему, отделяя данные от функциональных тестов. Один и тот же сценарий тестирования может выполняться для различных комбинаций входных данных теста и генерировать результаты теста. Например, мы хотим протестировать вход в систему с несколькими полями ввода с 1000 различными наборами данных. Чтобы проверить это, вы можете использовать следующие разные подходы: Подход 1) Создайте 1000 сценариев по одному для каждого набора данных и запускайте каждый тест отдельно по одному. Подход 2) Вручную измените значение в тестовом скрипте и запустите его несколько раз. Подход 3) Импортируйте данные из листа Excel. Извлеките данные теста из строк Excel по очереди и выполните скрипт. В приведенных трех сценариях первые два являются слишком трудоемкими. Таким образом, идеально следовать третьему подходу и это не что иное, как Data-Driven Framework.
Тестирование на основе риска (RBT - Risk Based Testing)

ТЕСТИРОВАНИЕ НА ОСНОВЕ РИСКА (RBT) - это тип тестирования, основанный на вероятности риска. Он включает в себя оценку риска на основе сложности, критичности бизнеса, частоты использования, видимых областей, областей, подверженных дефектам, и т. д. Он включает определение приоритетов тестирования модулей и функций тестируемого приложения на основе влияния и вероятности отказов.
Риск - это возникновение неопределенного события, которое положительно или отрицательно влияет на измеримые критерии успеха проекта. Это могут быть события, которые произошли в прошлом или текущие события, или что-то, что может произойти в будущем. Эти неопределенные события могут повлиять на стоимость, бизнес, технические и качественные цели проекта. Позитивные риски упоминаются как возможности и помощь в устойчивости бизнеса. Например, инвестирование в новый проект, изменение бизнес-процессов, разработка новых продуктов. Отрицательные риски называются угрозами, и для успеха проекта должны быть реализованы рекомендации по их минимизации или устранению.
Примерный чеклист:

- Важные функциональные возможности в проекте.
- Видимая пользователю функциональность в проекте
- Функциональность, оказывающая наибольшее влияние на безопасность
- Функциональные возможности, которые оказывают наибольшее финансовое влияние на пользователей
- Сложные области исходного кода и кодов, подверженных ошибкам
- Функции, которые могут быть проверены в начале цикла разработки.
- Особенности или функциональные возможности, которые были добавлены в дизайн продукта в последнюю минуту.
- Критические факторы подобных / связанных предыдущих проектов, которые вызвали проблемы.
- Основные факторы или проблемы аналогичных / связанных проектов, которые оказали огромное влияние на эксплуатационные расходы.
- Плохие требования, которые приводят к плохим проектам и тестам, которые могут повлиять на цели и результаты проекта.
- В худшем случае продукт может быть настолько дефектным, что его невозможно переработать, и его необходимо полностью утилизировать, что может нанести серьезный ущерб репутации компании. Определите, какие проблемы имеют решающее значение для целей продукта.
- Ситуации или проблемы, которые могут привести к постоянным жалобам на обслуживание клиентов. Сквозные тесты могут легко сфокусироваться на нескольких функциональных возможностях системы. Оптимальный набор тестов, которые могут максимизировать покрытие риска
- Какие тесты будут иметь лучшее соотношение риска и требуемого времени?

Что вы знаете о потоковом тестировании? (BFT — Business Flow Testing)
BFT – это подход к тестированию, где вы смотрите на продукт не с точки зрения конкретных кейсов, а с точки зрения поведения системы в каждой ее точке.
Подход является объединением таких концепций, как Data Driven testing и Behaviour Driven testing, примененным к бизнес-моделям, хорошо описываемым графами движения данных.

Вы не тестируете отдельные тест-кейсы, а проверяете работоспособность системы, тест-кейсы получаются сами по себе.
Образно говоря, у вас есть:

- Набор разнообразных входных бизнес данных
- Система, разбитая на важные для тестирования состояния бизнес-сущностей, которые надо проверить
- Правила перехода между состояниями

У вас нет тест-кейсов. Тест-кейсы генерируются на основании входных данных сами. Больше не надо спорить, как назвать тест-кейсы правильно. Название тест-кейса – это набор определяющих его входных данных и путь, по которым они пройдут. Название получается длинное, но полностью описывающее данный тест. И к тому же вы не тратите на него ни секунды — оно создается само.
В чем разница между coupling и cohesion?
Это термины из принципов разработки ПО:

- Связанность, сопряжение (coupling)— способ и степень взаимозависимости между программными модулями; сила взаимосвязей между модулями; мера того, насколько взаимозависимы разные подпрограммы или модули. Сильная связанность (High coupling) рассматривается как серьезный недостаток, поскольку затрудняет понимание логики модулей, их модификацию, автономное тестирование, а также переиспользование по отдельности.
- Связность, или прочность (cohesion) — мера силы взаимосвязанности элементов внутри модуля; способ и степень, в которой задачи, выполняемые некоторым программным модулем, связаны друг с другом.Считается, что объект (подсистема) обладает высокой связностью (High cohesion), если его обязанности хорошо согласованы между собой и он не выполняет огромных объемов работы.

Доп. материал:
Low Coupling • и High Cohesion
ООП для ООП: GRASP

----- Тестирование в разных сферах/областях (testing different domains) -----
Что такое веб-тестирование и как его производить?
ВЕБ-ТЕСТИРОВАНИЕ, или тестирование веб-сайта, проверяет ваше веб-приложение или веб-сайт на наличие потенциальных ошибок, прежде чем оно будет опубликовано и доступно для широкой публики.

1. Функциональное тестирование: используется для проверки того, соответствует ли ваш продукт спецификациям, а также функциональным требованиям, которые вы наметили для него в документации по разработке. Включает в себя:

- Проверка, что все ссылки на ваших веб-страницах работают правильно и что нет битых ссылок.
- Исходящие ссылки
- Внутренние ссылки
- Якорные ссылки (слово или словосочетание, на котором поставлена ссылка)
- MailTo Ссылки

- Текстовые формы работают как положено.
- Проверка скриптов в форме работает как положено. Например, если пользователь не заполняет обязательное поле в форме, отображается сообщение об ошибке.
- Проверьте значения по умолчанию
- После отправки данные в формах отправляются в базу данных или связываются с рабочим адресом электронной почты.
- Формы оптимально отформатированы для лучшей читаемости
- Тестовые куки работают как положено. Файлы cookie - это небольшие файлы, используемые веб-сайтами для запоминания активных пользовательских сессий, поэтому вам не нужно входить в систему каждый раз, когда вы посещаете веб-сайт. Тестирование файлов cookie будет включать
- Тестовые файлы cookie (сеансы) удаляются либо после очистки кэша, либо по истечении срока их действия. Удалите файлы cookie (сеансы) и проверьте, запрашиваются ли учетные данные при следующем посещении сайта.
- Протестируйте HTML и CSS, чтобы поисковые системы могли легко сканировать ваш сайт. Это будет включать
- Проверка на синтаксические ошибки
- Удобочитаемые цветовые схемы
- Стандартное соответствие. Убедитесь, что соблюдаются такие стандарты, как W3C, OASIS, IETF, ISO, ECMA или WS-I.
- Тест бизнес-воркфлоу - это будет включать в себя
- Тестирование вашего end-to-end workflow / бизнес-сценариев
- Также проверьте отрицательные сценарии, чтобы при выполнении пользователем неожиданного шага в веб-приложении отображалось соответствующее сообщение об ошибке или справка.
- Примеры функциональных тест-кейсов:
- Все обязательные поля должны быть валидированы.
- Звездочка должна отображаться для всех обязательных полей.
- Не должно отображаться сообщение об ошибке для дополнительных полей.
- Проверьте, что високосные годы проверены правильно и не вызывают ошибок.
- Числовые поля не должны принимать буквы и должно отображаться соответствующее сообщение об ошибке.
- Проверьте наличие отрицательных чисел, если это разрешено для числовых полей.
- Тестовое деление на ноль должно быть правильно обработано.
- Проверьте максимальную длину каждого поля, чтобы убедиться, что данные не усекаются.
- Тест всплывающего сообщения («Это поле ограничено 500 символами») должно отображаться, если данные достигают максимального размера поля.
- Проверьте, должно ли отображаться подтверждающее сообщение для операций обновления и удаления.
- Величины должны быть в подходящем формате.
- Проверьте все поля ввода на ввод специальных символов.
- Проверьте функциональность тайм-аута.
- Проверьте функциональность сортировок.
- Проверьте, что FAQ и Политика конфиденциальности четко определены и доступны для пользователей.

- Проверьте, все ли работает и не перенаправляется ли пользователь на страницу ошибки.
- Все загруженные документы открываются правильно.
- Пользователь должен иметь возможность скачать загруженные файлы.
- Проверьте функциональность электронной почты системы. Тестируемый скрипт корректно работает в разных браузерах (IE, Firefox, Chrome, Safari и Opera).
- Проверьте, что произойдет, если пользователь удалит файлы cookie, находясь на сайте.
- Проверьте, что произойдет, если пользователь удалит файлы cookie после посещения сайта.

2. Юзабилити-тестирование стало важной частью любого веб-проекта. Его могут провести тестировщики или небольшая фокус-группа, похожая на целевую аудиторию веб-приложения.
- Навигация:
- Меню, кнопки или ссылки на разные страницы вашего сайта должны быть легко видны и согласованы на всех веб-страницах.
- Проверьте содержимое:
- Содержание должно быть разборчивым, без орфографических или грамматических ошибок.
- Изображения, если они присутствуют, должны содержать «альтернативный» текст
- Примеры тестов юзабилити:
- Содержание веб-страницы должно быть правильным без каких-либо орфографических или грамматических ошибок
- Все шрифты должны быть в соответствии с требованиями.
- Весь текст должен быть правильно выровнен.
- Все сообщения об ошибках должны быть правильными без каких-либо орфографических или грамматических ошибок, а сообщение об ошибке должно соответствовать метке поля.
- Текст подсказки должен быть там для каждого поля.
- Все поля должны быть правильно выровнены.
- Должно быть достаточно места между метками полей, столбцами, строками и сообщениями об ошибках.
- Все кнопки должны быть в стандартном формате и размере.
- Домашняя ссылка должна быть на каждой странице.
- Отключенные поля должны быть недоступны.
- Проверьте наличие битых ссылок и изображений.
- Сообщение о подтверждении должно отображаться для любого вида операции обновления и удаления. Проверить сайт на разных разрешениях (640 x 480, 600x800 и т. д.)
- Убедитесь, что вкладка должна работать правильно.
- Полоса прокрутки должна появляться только при необходимости.
- Если при отправке появляется сообщение об ошибке, информация, заполненная пользователем, должна быть там.
- Название должно отображаться на каждой веб-странице

- Все поля (текстовое поле, раскрывающийся список, переключатель и т. д.) И кнопки должны быть доступны с помощью сочетаний клавиш, и пользователь должен иметь возможность выполнять все операции с помощью клавиатуры.
- Проверьте, не усекаются ли выпадающие данные из-за размера поля.
- Также проверьте, жестко ли закодированы или управляются данные через администратора.

3. Тестирование интерфейсов: Здесь тестируются три области: приложение, веб-сервер и сервер базы данных.
- Приложение: тестовые запросы правильно отправляются в базу данных и вывод на стороне клиента отображается правильно. Ошибки, если таковые имеются, должны быть обнаружены приложением и должны отображаться только администратору, а не конечному пользователю.
- Веб-сервер: тестовый веб-сервер обрабатывает все запросы приложений без какого-либо отказа в обслуживании.
- Сервер базы данных: убедитесь, что запросы, отправленные в базу данных, дают ожидаемые результаты. Проверьте реакцию системы, когда невозможно установить соединение между тремя уровнями (Приложение, Интернет и База данных) и соответствующее сообщение отображается конечному пользователю.

4. Тестирование базы данных: База данных является одним из важнейших компонентов вашего веб-приложения, и необходимо тщательно провести тестирование. Тестирование будет включать в себя:
- Проверьте, отображаются ли какие-либо ошибки при выполнении запросов
- Целостность данных поддерживается при создании, обновлении или удалении данных в базе данных.
- Проверьте время ответа на запросы.
- Тестовые данные, полученные из вашей базы данных, точно отображаются в вашем веб-приложении.

- Примеры тест-кейсов для тестирования базы данных:
- Проверьте имя базы данных: имя базы данных должно соответствовать спецификациям.
- Проверьте таблицы, столбцы, типы столбцов и значения по умолчанию: все должно соответствовать спецификациям.
- Проверьте, допускает ли столбец null значение или нет.
- Проверьте первичный и внешний ключ каждой таблицы.
- Проверьте хранимую процедуру:
- Проверьте, установлена ли сохраненная процедура или нет.
- Проверьте имя хранимой процедуры
- Проверьте имена параметров, типы и количество параметров.
- Проверьте требуемые параметры.
- Проверьте хранимую процедуру, удалив некоторые параметры
- Проверьте, когда выход равен нулю, это должно повлиять на нулевые записи.
- Проверьте хранимую процедуру, написав простые запросы SQL.
- Проверьте, возвращает ли хранимая процедура значения
- Проверьте хранимую процедуру с образцами входных данных.
- Проверьте поведение каждого флага в таблице.

- Убедитесь, что данные правильно сохраняются в базе данных после каждой отправки страницы.
- Проверьте данные, если выполняются операции DML (Обновить, удалить и вставить).
- Проверьте длину каждого поля: длина поля на Frontend и backend должна быть одинаковой.
- Проверьте имена баз данных QA, UAT и production. Имена должны быть уникальными.
- Проверьте зашифрованные данные в базе данных.
- Проверьте размер базы данных.
- Также проверьте время ответа каждого выполненного запроса.
- Проверьте данные, отображаемые на Frontend, и убедитесь, что они совпадают с backend.
- Проверьте достоверность данных, вставив неверные данные в базу данных.
- Проверьте триггеры.

5. Тестирование на совместимость. Тесты на совместимость гарантируют, что ваше веб-приложение правильно отображается на разных устройствах.
- Вам нужно проверить, правильно ли отображается ваше веб-приложение в браузерах, работает ли JavaScript, AJAX и аутентификация нормально. Вы также можете проверить совместимость мобильного браузера. Рендеринг веб-элементов, таких как кнопки, текстовые поля и т. д. , изменяется с изменением в операционной системе. Убедитесь, что ваш сайт работает нормально для различных комбинаций операционных систем, таких как Windows, Linux, Mac и браузеров, таких как Firefox, Internet Explorer, Safari и т. д.
- Примеры тестов на совместимость:
- Протестируйте сайт в разных браузерах (IE, Firefox, Chrome, Safari и Opera) и убедитесь, что сайт отображается правильно.
- Используемая версия HTML совместима с соответствующими версиями браузера.
- Проверьте правильность отображения изображений в разных браузерах.
- Протестируйте шрифты, которые можно использовать в разных браузерах.
- Протестируйте код Javascript в разных браузерах.
- Проверьте анимированные GIF-файлы в разных браузерах.

6. Тестирование производительности: Это нужно, чтобы обеспечить работу вашего сайта при любых нагрузках. Деятельность по тестированию ПО будет включать, но не ограничиваться:
- Время отклика приложения сайта на разных скоростях соединения
- Нагрузочное тестирование вашего веб-приложения, чтобы определить его поведение при нормальной и пиковой нагрузке.
- Стресс-тест вашего веб-сайта, чтобы определить его точку остановки при превышении нормальных нагрузок в пиковое время.
- Проверьте, происходит ли сбой из-за пиковой нагрузки, как сайт восстанавливается после такого события, убедитесь, что методы оптимизации, такие как сжатие gzip и кэш включены, чтобы сократить время загрузки

7. Тестирование безопасности жизненно важно для сайта электронной коммерции, который хранит конфиденциальную информацию о клиентах, например, кредитные карты. Деятельность по тестированию будет включать:

- Проверка несанкционированного доступа к защищенным страницам
- Запрещенные файлы не должны быть загружаемыми без соответствующего доступа
- Сессии автоматически прекращаются после длительного отсутствия активности пользователя
- При использовании SSL-сертификатов веб-сайт должен перенаправить на зашифрованные SSL-страницы.

- Примеры тестовых сценариев для тестирования безопасности:
- Убедитесь, что веб-страница, содержащая важные данные, такие как пароль, номера кредитных карт, секретные ответы на секретный вопрос и т. д. , Должна быть отправлена через HTTPS (SSL).
- Убедитесь, что важная информация, такая как пароль, номера кредитных карт и т. д. , Должна отображаться в зашифрованном виде.
- Правила проверки пароля применяются на всех страницах аутентификации, таких как Регистрация, забытый пароль, смена пароля.
- Убедитесь, что, если пароль изменен, пользователь не должен иметь возможность войти со старым паролем. Убедитесь, что сообщения об ошибках не должны отображать важную информацию.
- Убедитесь, что, если пользователь вышел из системы или сеанс пользователя истек, пользователь не должен перемещаться по сайту авторизованным.
- Проверьте доступ к защищенным и незащищенным веб-страницам напрямую без входа в систему.
- Убедитесь, что опция «Просмотр исходного кода» отключена и не должна быть видна пользователю. Убедитесь, что учетная запись пользователя заблокирована, если пользователь вводит неправильный пароль несколько раз.
- Убедитесь, что куки не должны хранить пароли.
- Убедитесь, что, если какая-либо функция не работает, система не должна отображать информацию о приложении, сервере или базе данных. Вместо этого она должна отображать пользовательскую страницу ошибки.
- Проверьте атаки SQL-инъекций.
- Проверьте роли пользователей и их права. Например, запрашивающая сторона не должна иметь доступа к странице администратора.
- Убедитесь, что важные операции записаны в файлы журналов, и эта информация должна быть отслеживаемой.
- Убедитесь, что значения сеанса находятся в зашифрованном формате в адресной строке.
- Убедитесь, что информация о файлах cookie хранится в зашифрованном формате.
- Проверьте приложение на брутфорс-атаки

8. Тестирование толпы (Crowd Testing): Вы берете большое количество людей (толпу) для выполнения тестов, которые в противном случае были бы выполнены выбранной группой людей в компании. Краудсорсинговое тестирование представляет собой интересную и перспективную концепцию и помогает выявить многие незамеченные

дефекты. Оно включает в себя практически все типы тестирования, применимые к вашему веб-приложению.

Тестирование банковского ПО

Сектор BFSI (банковские, финансовые услуги и страхование) является крупнейшим потребителем ИТ-услуг. Банковские приложения напрямую связаны с конфиденциальными финансовыми данными. Обязательно, чтобы все операции, выполняемые банковским программным обеспечением, выполнялись без сбоев и ошибок. Банковское ПО выполняет различные функции, такие как перевод и внесение средств, запрос баланса, история транзакций, вывод средств и так далее. Тестирование банковского приложения гарантирует, что эти действия не только хорошо выполняются, но и остаются защищенными от хакеров.

Важно отметить стандартные функции, ожидаемые от любого банковского приложения:

- Оно должно поддерживать тысячи одновременных пользовательских сессий
- Банковское приложение должно интегрироваться с другими многочисленными приложениями, такими как торговые счета, утилита оплаты счетов, кредитные карты и т. д.
- Должно обрабатывать быстрые и безопасные транзакции
- Оно должно включать в себя массивную систему хранения.
- Для устранения проблем с клиентами у него должна быть высокая возможность аудита
- Оно должен обрабатывать сложные бизнес-процессы
- Нужно поддерживать пользователей на разных платформах (Mac, Linux, Unix, Windows)
- Оно должно поддерживать пользователей из разных мест и на разных языках
- Поддерживать пользователей в различных платежных системах (VISA, AMEX, MasterCard)
- Оно должно поддерживать несколько секторов обслуживания (кредиты, розничные банковские операции и т. д.)
- Иметь механизм защиты от катастрофических сбоев

Тестирование электронной коммерции (eCommerce)

Тестирование электронной коммерции помогает в предотвращении ошибок и повышает ценность продукта, обеспечивая соответствие требованиям клиента.

Целями тестирования являются:

- Обеспечение надежности и качества ПО
- Уверенность в системе
- Оптимальная производительность

Настройка системы электронной коммерции является сложным процессом и зависит от множества рыночных переменных. Для поддержания целостности системы электронной коммерции тестирование становится обязательным. Что проверяется:

- Совместимость браузера:
- Поддержка для старых браузеров
- Специальные браузерные расширения
- Тестирование браузера должно охватывать основные платформы (Linux, Windows, Mac и т. д.)
- Отображение страниц:
- Некорректное отображение страниц
- Сообщения об ошибках во время выполнения

- Плохое время загрузки страницы
- Битые ссылки, зависимость от плагина, размер шрифта и т. д.
- Управление сессиями
- Истечение сессии
- Хранение сессии
- Удобство и простота
- Неинтуитивный дизайн
- Плохая навигация по сайту
- Навигация по каталогам
- Отсутствие помощи-поддержки
- Анализ содержимого
- Вводящий в заблуждение, оскорбительный или незаконный контент
- Роялти-фри изображения и нарушение авторских прав
- Функциональность персонализации
- Доступность 24/7
- Доступность
- Атаки отказа в обслуживании
- Неприемлемые уровни недоступности
- Резервное копирование и восстановление
- Сбой или отказ восстановления
- Ошибка резервного копирования
- Отказоустойчивость
- Операции
- Целостность транзакции
- Пропускная способность
- Аудит
- Обработка заказов на покупку и покупка
- Функциональность корзины
- Обработка заказов
- Процесс оплаты
- Отслеживание заказа
- Интернационализация
- Языковая поддержка
- Отображение языков
- Культурная чувствительность
- Региональный учет
- Оперативные бизнес-процедуры
- Насколько хорошо справляется электронная процедура
- Наблюдение за узкими местами
- Системная интеграция
- Формат интерфейса данных
- Обновления
- Величина нагрузки интерфейса
- Интегрированная производительность
- Производительность
- Узкие места производительности
- Обработка нагрузки
- Анализ масштабируемости

- Логин и безопасность
- Возможность входа
- Проникновение и контроль доступа
- Небезопасная передача информации
- Веб-атаки
- Компьютерные вирусы
- Цифровые подписи

Тестирование производительности - главный приоритет в электронной коммерции. Просто задержите около 250 миллисекунд времени загрузки страницы – и это заставляет вашего клиента идти к вашему конкуренту. Гигант розничной торговли Walmart пересмотрел скорость своего сайта и заметил увеличение на 2% коэффициента конверсии посетителей и доходов на 1%. Эффективность вашего сайта зависит от этих факторов:

- Пропускная способность (Throughput):
- Запросов в секунду
- Транзакций в минуту
- Выполнений за клик
- Время отклика (Response Time):
- Длительность задачи
- Секунд на клик
- Загрузка страницы
- DNS Lookup
- Продолжительность времени между кликом и просмотром страницы

Тестирование платежного шлюза (Payment Gateway)
Платежный шлюз - это сервис приложений электронной коммерции, который принимает оплату кредитной картой для покупок в Интернете. Платежные шлюзы защищают данные кредитной карты, шифруя конфиденциальную информацию, такую как номера кредитных карт, данные владельца счета и так далее. Эта информация безопасно передается между покупателем и продавцом, и наоборот. Современные платежные шлюзы также надежно подтверждают платежи с помощью дебетовых карт, электронных банковских переводов, банковских карт, бонусных баллов и т. д.
Типы платежных систем:

- Собственный местный платежный шлюз: Хостинговая система шлюзов платежей направляет клиента от сайта электронной коммерции к шлюзу во время процесса оплаты. Как только платеж будет выполнен, он вернет клиента на сайт электронной коммерции. Для такого типа оплаты вам не нужен идентификатор продавца, например, хостинговый платежный шлюз - PayPal, Noche и WorldPay.
- Shared (встраиваемый у партнеров) платежный шлюз: В разделяемом платежном шлюзе при обработке платежа клиент направляется на страницу оплаты и остается на сайте электронной коммерции. Как только реквизиты платежа заполнены, процесс оплаты продолжается. Поскольку он не покидает сайт электронной коммерции во время обработки платежа, этот режим прост и, более предпочтителен, примером шлюза с общими платежами является eWay, Stripe.

Тестирование для Платежного шлюза должно включать:

- Функциональное тестирование: это тестирование базовой функциональности платежного шлюза. Оно предназначено для проверки того, ведет ли себя приложение ожидаемым образом при обработке заказов, расчетах, добавлении НДС в зависимости от страны и т. д.
- Интеграция: Проверьте интеграцию с вашей кредитной картой.
- Производительность. Определите различные показатели производительности, такие как максимально возможное количество пользователей, проходящих через шлюзы в течение определенного дня, и конвертирующих их в одновременных пользователей.
- Безопасность: вам необходимо выполнить глубокую проверку безопасности для Платежного шлюза

Примеры тест-кейсов для тестирования платежного шлюза:
- В процессе оплаты попробуйте изменить язык платежного шлюза.
- После успешной оплаты проверьте все необходимые компоненты
- Проверьте, что произойдет, если платежный шлюз перестанет отвечать во время оплаты
- В процессе оплаты проверьте, что произойдет, если сессия заканчивается
- В процессе оплаты проверьте, что происходит в бэкэнде
- Проверьте, что произойдет, если процесс оплаты не удастся
- Проверьте записи базы данных, хранят ли они данные кредитной карты или нет
- В процессе оплаты проверяйте страницы ошибок и страницы безопасности
- Проверьте при наличии блокировщика всплывающих окон
- Между платежным шлюзом и страницами проверьте буферные страницы.
- Проверка успешной оплаты, код успеха отправляется в приложение и пользователю отображается страница подтверждения
- Убедитесь, что транзакция обрабатывается немедленно или обработка передана вашему банку.
- После успешной транзакции проверьте, возвращается ли платежный шлюз в ваше приложение.
- Проверьте все форматы и сообщения при успешном процессе оплаты
- Если у вас нет квитанции об авторизации от платежного шлюза, товар не должен быть отправлен
- Сообщите владельцу о любой транзакции, обработанной по электронной почте. Шифровать содержимое почты.
- Проверьте формат суммы с форматом валюты
- Проверьте, доступен ли каждый из вариантов оплаты
- Проверьте, открывает ли каждый перечисленный способ оплаты соответствующий способ оплаты в соответствии со спецификацией.
- Убедитесь, что в платежном шлюзе по умолчанию выбран нужный вариант дебетовой / кредитной карты.
- Проверьте опцию по умолчанию для дебетовых карт - показывает выпадающее меню выбора карты

Тестирование систем розничной торговли (POS - Point Of Sale)

POS-тестирование определяется как тестирование приложения в точках продаж. ПО POS или Point Of Sale - это жизненно важное решение для предприятий розничной торговли, позволяющее легко совершать розничные транзакции из любого места. Вы, наверное, видели терминал торговой точки в своем любимом торговом центре.

Система является более сложной, чем вы думаете, и тесно интегрирована с другими программными системами, такими как Склад, Инвентарь, Заказ на поставку, Цепочка поставок, Маркетинг, Планирование товаров и т. д. Знание предметной области POS важно для тестирования.
Оценка POS-системы может быть разбита на два уровня:

- Уровень применения (Application Level)
- Уровень предприятия (Enterprise Level)

Сценарий Кейсы
Деятельность кассира • Проверьте правильность ввода товаров, приобретенных клиентом

- Тестовые скидки применяются правильно
- Убедитесь, что платежные карты магазина (value cards) могут быть использованы
- Управление мелочью работает как положено
- Проверьте соответствие итогов и закрытия
- Убедитесь, что денежный ящик кассы работает правильно
- Проверьте, что система POS совместима с периферийными устройствами, такими как считыватель RFID, сканер штрих-кода и т. д.

Обработка платежного шлюза • Проверка правильности CVV кредитных карт

- Тест на использование карт с обеих сторон
- Убедитесь, что данные карты правильно зашифрованы и расшифрованы

Продажи • Проверьте для обычного процесса продажи

- Продажи могут быть обработаны дебетовой / кредитной картой
- Проверить покупку по карте лояльности
- Проверка правильности отображаемой цены для покупаемых товаров
- Тест для "0" или нулевой транзакции
- Привязка UPC или штрих-кодов с поставщиками
- Проверка платежных данных или данных о доставке в диспетчере платежей
- Тест для reference транзакции
- Проверьте формат печати сгенерированной квитанции
- Убедитесь, что для утвержденных, удержанных или отклоненных транзакций создан правильный код

Сценарии возврата и обмена • Убедитесь, что внутренняя опись хорошо интегрирована с другими торговыми точками или цепочкой поставок

- Чек на обмен или возврат товара наличными
- Проверьте, работает ли система при обмене или возврате товара с помощью кредитной карты
- Проверка системы обработки продажи с чеком или без чека
- Убедитесь, что система должна позволять вводить штрих-код вручную, если сканер не работает
- Убедитесь, что система отображает как текущую сумму, так и сумму скидки при обмене товара, если применимо

Производительность• Проверьте скорость или время, необходимое для получения ответа или отправки запроса

- Проверьте, применяются ли правила транзакций (скидки / налоги и т. д.)
- Убедитесь, что для утвержденных, удержанных или отклоненных транзакций создан правильный код

Негативные сценарии • Тестовая система с просроченными данными карты
• Тест с неверным PIN-кодом для кредитной карты
• Проверьте инвентарь, введя неправильный код товара
• Проверьте, как система реагирует при вводе неверного номера счета
• Тест на отрицательную транзакцию
• Проверьте ответ системы при вводе недопустимой даты для рекламных предложений онлайн-товаров
Управление акциями и скидками • Проверка: система для различных скидок, таких как ветеранская скидка, сезонная скидка, скидка на покупку или перелет и т. д.
• Проверка: система для различных рекламных предложений по отдельным позициям
• Проверка: система оповещений, которая уведомляет об окончании или начале сезонных предложений
• Проверьте, распечатывает ли квитанция точную скидку или предложения, которые используются
• Проверка: система для определения неправильных предложений или скидок на товары онлайн
• Протестируйте процесс управления заказами
• Убедитесь, что данные продукта, полученные после сканирования штрих-кода, являются точными
Отслеживание данных клиента • Проверка ответа системы с неверным вводом данных клиента
• Проверка: система для разрешения авторизованного доступа к конфиденциальным данным клиента
• Протестируйте базу данных для записи истории покупок клиента (что он покупает, как часто он покупает и т. д.)
Безопасность и соответствие нормативным требованиям • Проверка системы POS на соответствие нормативным требованиям
• Проверка: система оповещения -> security defenders
• Убедитесь, что вы можете аннулировать платеж перед отправкой
• Тестируйте профили пользователей и уровни доступа в POS Software
• Проверка согласованности базы данных
• Проверьте конкретную информацию о каждой наличности, идентификатор купона, номер чека и т. д.
Тестирование отчетности • Тестирование отчета по анализу трендов
• Тестовая информация, связанная с транзакцией по кредитной карте, должна быть отражена в отчетах
• Тест для отдельных, а также сводные отчеты истории покупок клиентов
• Тест для генерации онлайн отчетов

Тестирование в сфере страхования (Insurance)
Страховые компании в значительной степени полагаются на ПО для ведения своего бизнеса. Программные системы помогают им заниматься различными видами страховой деятельности, такими как разработка стандартных форм полисов, обработка процесса выставления счетов, управление данными клиента, оказание качественных услуг клиенту, координация между филиалами и так далее. Хотя это ПО разработано с учетом ожиданий заказчика, его надежность и согласованность должны

быть проверены перед его фактическим внедрением. Тестирование ПО гарантирует качество страхового ПО, выявляя ошибки перед запуском.

Страхование определяется как справедливая передача риска убытков от одного субъекта другому в обмен на платеж. Страховая компания, которая продает полис, называется СТРАХОВОЙ, а лицо или компания, которая использует полис, называется ЗАСТРАХОВАННЫЙ. Страховые полисы обычно делятся на две категории, и страховщик покупает эти полисы в соответствии с их требованиями и бюджетом. Тем не менее, есть другие виды страхования, которые подпадают под эти категории:

- Страхование по безработице (Unemployment insurance)
- Социальное обеспечение (Social Security)
- Компенсация рабочим (Workers Compensation)

Есть много ветвей в страховой компании, которые требуют тестирования:

- Системы администрирования политики (Policy Administration Systems)
- Системы управления претензиями (Claim Management Systems)
- Системы управления распределением (Distribution Management Systems)
- Системы управления инвестициями (Investment Management Systems)
- Сторонние системы администрирования (Third party Administration Systems)
- Решения по управлению рисками (Risk Management Solutions)
- Регулирование и соответствие (Regulatory and Compliance)
- Актуарные системы (Оценка и ценообразование) (Actuarial Systems (Valuation & Pricing))

Сектор страхования представляет собой сеть небольших подразделений, которая прямо или косвенно занимается обработкой требований. Для бесперебойного функционирования страховой компании необходимо, чтобы каждое из этих подразделений было тщательно проверено для достижения желаемого результата. Тестирование включает в себя:

- Колл-центр (Call Center):
- Интеграционное тестирование IVR (IVR • Integration testing)
- Маршрутизация и назначение вызовов (Call routing and assignment)
- Безопасность и доступ (Security and access)
- Рефлексивные вопросы (Reflexive Questions)
- Политика обслуживания (Policy Serving):
- Тестирование жизненного цикла политики (Policy life cycle testing)
- Изменения в финансовой и нефинансовой политике (Financial and Non-financial policy changes)
- Политика недействительности и восстановления (Policy lapse and Re-instatement)
- Оповещения о страховых выплатах (Premium due alerts)
- Оценка NPV/NAV (Valuation of NPV/NAV)
- Претензии (Claims):
- Сортировка и уступка требований (Claims triage and assignment)
- Тестирование жизненного цикла претензий (Testing claims life cycle)
- Учет требований / резервирование (Claims accounting/reserving)
- EDI / обмен сообщениями от третьих лиц (Third party EDI/messaging)
- Прямой канал (Direct channel):
- Мобильный доступ (Mobile access)
- Кросс-браузерность / кроссплатформенность (Cross browser/cross platform accessibility)

- Производительность приложения (Application performance)
- Удобство использования приложения (Usability of application)
- Отчеты / BI (Reports/BI):
- Соблюдение нормативных требований (Behaving to regulatory requirements)
- Генерация качественных данных для отчетности (Generate quality data for reporting)
- Создание массовых данных для сводных отчетов (Create bulk data for roll-up reports)
- Тестирование полей на основе формул в отчетах (Testing formula based fields in reports)
- Андеррайтинг (Underwriting):
- Качество андеррайтинга (Underwriting quality)
- Ручная и прямая обработка (Manual and Straight through processing)
- Сложные бизнес-правила (Complex business rules)
- Рейтинг эффективности (Rating efficiency)
- Управление требованиями (Vendor Interfacing) (Requirements Management)
- Интеграция (Integration):
- Интеграция данных (Data integration)
- Комплексная интеграция интерфейса (Complex interface integration)
- Форматы источника / назначения (Source/Destination formats)
- Производственный интерфейс (Production like interface)
- Эффективность пулла/пуша веб-сервиса (Web service pull/push efficiency)
- Новый бизнес (New Business):
- Проверить комбинации коэффициентов (Validate rates-factor combinations)
- Расписания и запуски заданий (Batch job schedules and runs)
- Ввод в эксплуатацию расчетов урегулирований (Commissioning calculations settlements)
- Быстрое и подробное назначение цен (Quick and detailed quote)
- Иллюстрация преимущества (Benefit illustration)
- Валидация суммарной выгоды (Benefit summary validation)

Образцы тестов для страховой заявки:

- Проверка правил претензий (Validate claims rule)
- Убедитесь, что претензия может возникнуть на максимальный и минимальный платеж (Ensure that claim can occur to the maximum and minimum payment)
- Убедитесь, что данные передаются точно во все подсистемы, включая учетные записи и отчетность (Verify data is transferred accurately to all sub-systems including accounts and reporting)
- Убедитесь, что претензии могут быть обработаны по всем каналам, например, через Интернет, мобильный телефон, звонки и т. Д (Check that the claims can be processed via all channels example web, mobile, calls, etc)
- Тест на 100% покрытие и точность в расчетах, определяющих ставки премии (Test for 100% coverage and accuracy in calculations determining premium rates)
- Убедитесь, что формула для расчета дивидендов и выплаченных значений дает правильное значение (Make sure formula for calculating dividend and paid up values gives correct value)
- Убедитесь, что значения выдачи рассчитываются в соответствии с требованиями политики (Verify surrender values are calculated as per the policy requirement)

- Проверьте фидуциарные детали и требования бухгалтерского учета (Verify fiduciary details and bookkeeping requirements)
- Тестирование сложных сценариев для отклонения политики провала и восстановления (Test complex scenarios for policy lapse and revivals)
- Испытайте различные условия для стоимости без конфискации (Test various conditions for non-forfeiture value)
- Тестовые сценарии для прекращения действия политики (Test scenarios for policy termination)
- Убедитесь, что учетная запись главной книги ведет себя так же, как и для сверки с дополнительной книгой (Verify general ledger account behave same as to reconcile with subsidiary ledger)
- Тестовый расчет чистого обязательства для оценки (Test calculation of net liability for valuation)
- Условия тестирования для длительного страхования (Test conditions for extended term insurance)
- Проверка политики для варианта без конфискации (Verify policy for a non-forfeiture option)
- Проверьте, что другой страховой продукт ведет себя как ожидалось (Check different insurance product term behaves as expected)
- Проверьте сумму выплаты согласно плану продукта (Verify premium value as per product plan)
- Тестирование автоматической системы обмена сообщениями для информирования клиентов о новых продуктах (Test automatic messaging system to inform customer about new products)
- Проверяйте все данные, введенные пользователями, по мере их прохождения через рабочий процесс, чтобы инициировать предупреждения, соответствие, уведомления и другие события рабочего процесса (Validate all the data entered by users as it progresses through the workflow to trigger warnings, compliance, notification and other workflow events)
- Убедитесь, что шаблон страхового документа поддерживает такой формат документа, как MS-Word (Verify insurance document template supports the document format like MS-Word)
- Тестовая система для автоматического выставления счета и отправки его клиенту по электронной почте (Test system for generating invoice automatically and send it to customer through e-mail)

Тестирование в сфере телекоммуникаций (Telecom)

После перехода сектора телекоммуникаций на цифровые и компьютерные сети, телекоммуникационная отрасль повсеместно использует ПО. Сектор телекоммуникаций зависит от различных видов компонентов ПО, чтобы доставить множество услуг, таких как маршрутизация и коммутация, VoIP и широкополосный доступ, и т. д. Таким образом, тестирование ПО Telecom является неизбежным.

Для предоставления телекоммуникационных услуг требуется наличие IVR, колл-центров, выставление счетов и т. д. и системы, которые включают в себя маршрутизаторы, коммутаторы, сотовые вышки и т. д.

Примеры тест-кейсов:

- Биллинговая Система:
- Телефонный номер клиента зарегистрирован на данного оператора связи

- Продолжает ли номер работать
- Введенный номер является валидным, а это 10-значный номер
- Отображение неоплаченных счетов
- Проверьте, что все предыдущие аккаунты номера стерты
- Убедитесь, что система зафиксировала количество звонков точно
- Проверьте, что тариф, выбранный клиентом, отображается в биллинговой системе
- Общая суммы расходов является точной
- Тестирование Приложения
- Протоколы, подача сигнала, полевые испытания для IoT
- Функциональное тестирование для базового применения мобильных телефонов как звонок, SMS, перевод/удержание и т. д.
- Тестирование различных приложений, таких как финансы, спорт и на основе определения местоположения, и т. д. OCC-БСС тестирования
- OCC-БСС тестирования (OSS-BSS testing)
- Выставление счетов клиентам, партнерам, правопорядка и борьбы с мошенничеством, обеспечения доходов
- Сетевое управление, посредничество, обеспечение, и т. д.
- ЦОВ, CRM и ERP-систем, хранилищ данных и т. д.
- Тестирование соответствия
- Совместимость электрических интерфейсов
- Соответствие протокола
- Соответствие транспортных слоев
- Тестирование IVR
- Интерактивные тестовые сценарии
- Распознавание голоса
- Голосовое меню и ветвление
- Ввод тонового сигнала DTMF

Тестирование протокола: L2 и L3 OSI

Когда компьютеры общаются друг с другом, существует общий набор правил и условий, которым должен следовать каждый компьютер. Другими словами, протоколы определяют, как данные передаются между вычислительными устройствами и по сетям.

PROTOCOL testing проверяет протоколы связи в областях коммутации, беспроводной связи, VoIP, маршрутизации и т. д. Цель состоит в том, чтобы проверить структуру пакетов, которые отправляются по сети, с помощью инструментов тестирования протоколов.

Протоколы делятся на две категории: Routed и routing. Routed могут использоваться для отправки пользовательских данных из одной сети в другую. Он переносит пользовательский трафик, такой как электронная почта, веб-трафик, передача файлов и т. д. Routed являются IP, IPX и AppleTalk.

Routing это сетевые протоколы, которые определяют маршруты для маршрутизаторов. Они используется только между маршрутизаторами. Например, RIP, IGRP, EIGRP и т. д.

Модель OSI имеет в общей сложности 7 уровней сетевого взаимодействия, в которых уровень 2 и уровень 3 очень важны.

- Уровень 2: это уровень канала передачи данных. Mac-адрес, Ethernet, Token Ring и Frame Relay являются примерами канального уровня.

- Уровень 3: это сетевой уровень, который определяет наилучший доступный путь в сети для связи. IP-адрес является примером layer3.

Для тестирования протокола вам понадобится анализатор протокола и симулятор. Анализатор протокола обеспечивает правильное декодирование наряду с анализом вызовов и сеансов. В то время как симулятор имитирует различные сущности сетевого элемента.

Обычно тестирование протокола выполняется DUT (тестируемым устройством) для других устройств, таких как коммутаторы и маршрутизаторы, и для настройки протокола в нем. После этого проверяется структура пакетов, отправленных устройствами. Он проверяет масштабируемость, производительность, алгоритм протокола и т. д. устройства с помощью таких инструментов, как IxNetworks, Scapy и Wireshark.

Тестирование протокола включает тестирование функциональности, производительности, стека протоколов, функциональной совместимости и т. д. Во время тестирования протокола, в основном, выполняется три проверки:

- Корректность: получаем ли мы пакет X как ожидали
- Задержка: сколько времени занимает доставка пакета
- Пропускная способность: сколько пакетов мы можем отправить в секунду

Тестирование протокола может быть разделено на две категории. Стресс и тесты надежности и функциональные тесты. Стресс-тесты и тесты надежности охватывают нагрузочное тестирование, стресс-тестирование, тестирование производительности и т. д. В то время как функциональное тестирование включает в себя негативное тестирование, тестирование на соответствие, тестирование на совместимость и т. д. Тестирование соответствия: протоколы, реализованные в продуктах, тестируются на соответствие, например, IEEE, RFC и т. д. Тестирование совместимости: проверяется совместимость для разных поставщиков. Это тестирование проводится после тестирования соответствия на соответствующей платформе. Проверка функциональности сети: функциональность сетевых продуктов проверена на функциональность со ссылкой на проектную документацию. Например, функциями могут быть защита портов на коммутаторе, ACL на маршрутизаторе и т. д.

Вот примеры Test case для роутеров:

- One VLAN on One Switch: Создайте две разные VLAN. Проверьте видимость между хостами в разных VLAN
- Three Symmetric VLANs on One switch: Создайте три разных VLAN. Проверьте видимость между хостами
- Spanning Tree: Root Path Cost Variation: Проверьте, как изменяется стоимость маршрута корневого пути после изменения топологии
- Spanning Tree: Port Blocking: Проверьте, как протокол связующего дерева предотвращает образование циклов в сети, блокируя избыточные каналы, также при наличии VLAN
- Spanning Tree: Port Blocking: Покажите, что каждый MSTI может иметь разные корневые мосты
- Visibility between different STP Regions: с теми же VLAN проверить видимость между различными регионами STP
- Telephone switch Performance: Создайте 1000 телефонных звонков и проверьте, нормально ли работает телефонный коммутатор или его производительность снижается

- Negative test for device: Введите неверный ключ и проверьте пользователя на аутентификацию.
- Line speed: Проверьте устройство, работающее на скорости 10 Гбит / с, используя всю доступную пропускную способность для обработки входящего трафика.
- Protocol conversation rate: Отслеживайте диалог TCP между двумя устройствами и убедитесь, что каждое устройство работает правильно
- Response time for session initiation: Измерьте время отклика устройства на запрос приглашения для инициации сеанса

Тестирование интернета вещей (IoT - Internet of Things)

Интернет вещей - это сеть, состоящая из устройств в транспортных средствах, зданиях или любых других подключенных электронных устройств. Эта взаимосвязь облегчает сбор и обмен данными. 4 общих компонента системы IoT:

- Sensor
- Application
- Network
- Backend (Data Center)

IOT - это соединение идентифицируемых встроенных устройств с существующей интернет-инфраструктурой. Проще говоря, мы можем сказать, что IOT - это эра «умных», связанных продуктов, которые обмениваются данными и передают большой объем данных и загружают их в облако.

IOT elements

Testing Types	Sensor	Application	Network	Backend (Data Center)
Functional testing	True	True	False	False
Usability testing	True	True	False	False
Security testing	True	True	True	True
Performance testing	False	True	True	True
Compatibility testing	True	True	False	False
Services testing	False	True	True	True
Operational testing	True	True	False	False

Категории тестов с примерами Test Conditions:

- Проверка компонентов (Components Validation):
- Аппаратное обеспечение устройства (Device Hardware)
- Встроенное программное обеспечение (Embedded Software)
- Облачная инфраструктура (Cloud infrastructure)
- Подключение к сети (Network Connectivity)
- Стороннее программное обеспечение (Third-party software)
- Тестирование датчиков (Sensor testing)
- Тестирование команд (Command testing)
- Тестирование формата данных (Data format testing)
- Испытание на прочность (Robustness testing)
- Тестирование безопасности (Safety testing)
- Проверка функций (Function Validation):
- Базовое тестирование устройства (Basic device testing)
- Тестирование между устройствами IOT (Testing between IOT devices)
- Обработка ошибок (Error Handling)
- Правильность расчета (Valid Calculation)

- Проверка соответствия (Conditioning Validation):
- Ручная (Manual Conditioning)
- Автоматическая (Automated Conditioning)
- Профили (Conditioning profiles)
- Проверка производительности (Performance Validation):
- Частота передачи данных (Data transmit Frequency)
- Обработка многократнах запросов (Multiple request handing)
- Синхронизация (Synchronization)
- Тестирование прерываний (Interrupt testing)
- Производительность устройства (Device performance)
- Проверка согласованности (Consistency validation)
- Безопасность и проверка данных (Security and Data Validation):
- Проверка пакетов данных (Validate data packets)
- Проверка на потерю или повреждение пакетов (Verify data loses or corrupt packets)
- Шифрование / дешифрование данных (Data encryption/decryption)
- Значения данных (Data values)
- Роли и ответственность пользователей и их модель использования (Users Roles and Responsibility & its Usage Pattern)
- Проверка шлюза:
- Тестирование облачного интерфейса (Cloud interface testing)
- Тестирование протокола от устройства к облаку (Device to cloud protocol testing)
- Тестирование задержек (Latency testing)
- Проверка аналитики (Analytics Validation):
- Проверка аналитики данных датчика (Sensor data analytics checking)
- Операционная аналитика системы IOT (IOT system operational analytics)
- Аналитика системного фильтра (System filter analytics)
- Проверка правил (Rules verification)
- Проверка связи (Communication Validation):
- Совместимость (Interoperability)
- M2M или от устройства к устройству (M2M or Device to Device)
- Тестирование трансляции (Broadcast testing)
- Тестирование прерываний (Interrupt testing)
- Протокол (Protocol)

Что такое облачное тестирование? (Cloud testing)

CLOUD testing - это тип тестирования программного обеспечения, который проверяет услуги облачных вычислений. Облачные вычисления - это интернет-платформа, предоставляющая различные компьютерные сервисы, такие как оборудование, программное обеспечение и другие компьютерные сервисы, удаленно. Существует три модели облачных вычислений:

- SaaS- Software as a service
- PaaS- Platform as a service
- IaaS- Infrastructure as a service

Все облачное тестирование разделено на четыре основные категории:

- Тестирование всего облака (Testing of the whole cloud). Облако рассматривается как единое целое и на основе его возможностей проводится тестирование. SaaS и облачные вендоры, а также конечные пользователи заинтересованы в проведении такого типа тестирования.

- Тестирование в пределах облака (Testing within a cloud). Проверяя каждую из его внутренних функций, проводится тестирование. Только поставщики облачных услуг могут выполнять этот тип тестирования.
- Тестирование через облако (Testing across cloud). Тестирование проводится в облачных, частных, публичных и гибридных облаках различных типов.
- SaaS-тестирование в облаке (SaaS testing in cloud): функциональное и нефункциональное тестирование проводится на основе требований приложений.

Облачное тестирование фокусируется на основных компонентах, таких как:

- Приложение (Application): охватывает тестирование функций, сквозные бизнес-процессы (end-to-end business workflows), безопасность данных, совместимость с браузерами и т. д.
- Сеть (Network): включает в себя тестирование различной пропускной способности сети, протоколов и успешную передачу данных через сети.
- Инфраструктура (Infrastructure): включает в себя тестирование аварийного восстановления, резервное копирование, безопасное соединение и политики хранения. Инфраструктура должна быть проверена на соответствие нормативным требованиям.

Другие типы тестирования в облаке включают:

- Performance
- Availability
- Compliance
- Security
- Scalability
- Multi-tenancy
- Live upgrade testing

Как выполнять облачное тестирование:

- SaaS или облачное тестирование: Этот тип тестирования обычно выполняется поставщиками облачных или SaaS-приложений. Основной задачей является обеспечение качества предоставляемых сервисных функций, предлагаемых в облачной или SaaS-программе. Тестирование, выполняемое в этой среде, - это проверка интеграции, функциональности, безопасности, функциональности модулей, системных функций и регрессионного тестирования, а также оценка производительности и масштабируемости.
- Онлайн тестирование приложений в облаке: Производители онлайн-приложений проводят это тестирование, которое проверяет производительность и функциональное тестирование облачных сервисов. Когда приложения связаны с legacy системами, проверяется качество связи между legacy системой и тестируемым приложением в облаке.
- Тестирование облачных приложений над облаками: Для проверки качества облачного приложения в разных облаках выполняется этот тип тестирования.

Примеры Test Scenario и несколько Test case для каждого из них:

- Тестирование производительности (• Performance testing):
- Сбой из-за одного действия пользователя в облаке не должен влиять на других пользователей
- Ручное или автоматическое масштабирование не должно вызывать сбоев
- На всех типах устройств производительность приложения должна оставаться неизменной

- Повторное бронирование на стороне поставщика не должно снижать производительность приложения
- Тестирование безопасности (Security testing):
- Только авторизованный клиент должен получать доступ к данным
- Данные должны быть хорошо зашифрованы
- Данные должны быть полностью удалены, если они не используются клиентом
- Администрация поставщиков не должна получать доступ к данным клиентов.
- Проверьте наличие различных настроек безопасности, таких как брандмауэр, VPN, антивирус и т. д.
- Функциональное тестирование (Functional testing):
- Валидный ввод должен давать ожидаемые результаты
- Сервис должен должным образом интегрироваться с другими приложениями
- Система должна отображать тип учетной записи клиента при успешном входе в облако
- Когда клиент решил переключиться на другие службы, работающая служба должна автоматически закрыться
- Тестирование совместимости (Interoperability & Compatibility testing):
- Проверка требований совместимости тестируемой системы и приложения
- Проверьте совместимость браузера в облачной среде
- Определите дефект, который может возникнуть при подключении к облаку
- Любые неполные данные в облаке не должны быть переданы
- Убедитесь, что приложение работает на другой платформе облака
- Протестируйте приложение в собственной среде, а затем разверните его в облачной среде.
- Тестирование сети (Network testing):
- Тестовый протокол, отвечающий за подключение к облаку
- Проверка целостности данных при передаче данных
- Проверьте правильность подключения к сети
- Проверьте, отбрасываются ли пакеты брандмауэром с обеих сторон
- Нагрузка и стресс-тестирование (Load and Stress testing):
- Проверьте сервисы, когда несколько пользователей получают к ним доступ
- Определите дефект, ответственный за сбой оборудования или среды
- Проверьте, отказывает ли система при увеличении удельной нагрузки
- Проверьте, как система изменяется со временем при определенной нагрузке

Что такое тестирование сервис-ориентированной архитектуры? (SOA - Service Oriented Architecture)

Это тестирование архитектурного стиля SOA, в котором компоненты приложения предназначены для сообщения по протоколам связи, обычно через сеть. SOA - это метод интеграции бизнес-приложений и процессов для удовлетворения потребностей бизнеса. В разработке программного обеспечения SOA обеспечивает гибкость бизнес-процессов. Изменения в процессе или приложении могут быть направлены на конкретный компонент, не затрагивая всю систему. Разработчики программного обеспечения в SOA либо разрабатывают, либо покупают куски программ под названием SERVICES. Что такое Service?

- Service могут быть функциональной единицей приложения или бизнес-процесса, которая может быть повторно использована или повторена любым другим приложением или процессом. (Например, Платежный шлюз - это сервис, который может быть повторно использован любым сайтом электронной коммерции.

Каждый раз, когда необходимо сделать платеж, сайт электронной коммерции вызывает / запрашивает сервис Платежного шлюза. После оплаты через шлюз, ответ отправляется на сайт электронной коммерции)

- Service просты в сборке и легко переконфигурируют компоненты.
- Service можно сравнить со строительными блоками. Они могут построить любое необходимое приложение. Добавить и удалить их из приложения или бизнес-процесса очень просто.
- Service больше определяются бизнес-функциями, которые они выполняют, а не кусками кода.

Пример: на домашней странице веб-сайта и в поисковой системе отображается ежедневный прогноз погоды. Вместо того, чтобы писать код для виджета прогноза погоды, у продавца можно купить Службу прогноза погоды и встроить ее в страницу.
Тестирование SOA должно быть сосредоточено на 3 уровнях:

- Уровень сервисов (Services Layer): Этот уровень состоит из сервисов, полученных из бизнес-функций. Например - Рассмотрим оздоровительный сайт, который состоит из: Трекер веса, Отслеживание уровня сахара в крови и трекера артериального давления. Трекеры отображают соответствующие данные и дату их ввода. Уровень сервисов состоит из сервисов, которые получают соответствующие данные из базы данных – Сервис трекера веса, сервис отслеживания уровня сахара в крови, сервис отслеживания артериального давления и Сервис логина.

- Уровень процесса (Process Layer): Уровень процесса состоит из процессов, набора сервисов, которые являются частью единой функциональности. Процессы могут быть частью пользовательского интерфейса (например, поисковая система), частью инструмента ETL (для получения данных из базы данных). Основное внимание на этом уровне будет уделяться пользовательским интерфейсам и процессам. Пользовательский интерфейс весового трекера и его интеграция с базой данных является основным направлением.
- Потребительский уровень (Consumer Layer): Этот уровень в основном состоит из пользовательских интерфейсов. Тестирование приложения SOA распределяется на три уровня: Уровень обслуживания, Уровень интерфейса, Уровень end-to-end. Подход сверху вниз используется для проектирования тестов. Подход снизу-вверх используется для выполнения теста.

Методы тестирования SOA:

- Data based testing на основе бизнес-сценариев:
- Различные аспекты бизнеса, связанные с системой, должны быть проанализированы.
- Сценарии должны быть разработаны на основе интеграции
- Различные веб-сервисы приложения
- Веб-сервисы и приложения
- Настройка данных должна быть выполнена на основе вышеуказанных сценариев.
- Настройка данных должна быть выполнена так, чтобы охватить также сквозные сценарии
- Заглушки (Stubs):
- Будут созданы фиктивные интерфейсы для тестирования сервисов.

- Через эти интерфейсы могут быть предоставлены различные входные данные, а выходные данные могут быть проверены.
- Когда приложение использует интерфейс с внешней службой, которая не тестируется (сторонняя служба), во время тестирования интеграции можно создать заглушку.
- Regression testing:
- Регрессионное тестирование приложения должно проводиться при наличии нескольких релизов, чтобы обеспечить стабильность и доступность систем.
- Будет создан комплексный набор регрессионных тестов, охватывающий сервисы, которые составляют важную часть приложения.
- Этот набор тестов может быть повторно использован в нескольких релизах проекта.
- Тестирование уровня сервиса (Service Level testing):
- Тестирование уровня сервиса включает тестирование компонента на функциональность, безопасность, производительность и функциональную совместимость. Каждую услугу необходимо сначала протестировать независимо.
- Functional testing:
- Убедитесь, что служба отправляет правильный ответ на каждый запрос.
- Правильные ошибки получены для запросов с неверными данными.
- Проверьте каждый запрос и ответ для каждой операции, которую служба должна выполнять во время выполнения.
- Проверяйте сообщения об ошибках при возникновении ошибки на уровне сервера, клиента или сети.
- Убедитесь, что полученные ответы имеют правильный формат.
- Подтвердите, что данные, полученные в ответе, соответствуют запрашиваемым данным.
- Security testing:
- Отраслевой стандарт, определенный тестированием WS-Security, должен соблюдаться веб-службой.
- Меры безопасности должны работать без нареканий.
- Шифрование данных и Цифровые подписи на документах
- Аутентификация и авторизация
- SQL-инъекции, вредоносные программы, XSS, CSRF и другие уязвимости должны быть проверены на XML. Атаки отказа в обслуживании
- Performance testing:
- Производительность и функциональность сервиса необходимо тестировать при большой нагрузке.
- Производительность службы необходимо сравнивать при работе индивидуально и в приложении, с которым она связана.
- Нагрузочное тестирование сервиса: проверить время отклика, проверить наличие узких мест, проверить использование процессора и памяти, прогнозировать масштабируемость
- Тестирование уровня интеграции (Integration level testing):
- Интеграционное тестирование проводится с упором в основном на интерфейсы.
- Этот этап охватывает все возможные бизнес-сценарии.
- Нефункциональное тестирование приложения должно быть сделано еще раз на этом этапе. Security, compliance, and Performance testing обеспечивают доступность и стабильность системы во всех аспектах.

- Коммуникационные и сетевые протоколы должны быть протестированы для проверки согласованности обмена данными между сервисами.
- End to End testing:
- Все сервисы работают должным образом после интеграции
- Обработка исключений
- Пользовательский интерфейс приложения
- Правильный data flow через все компоненты
- Бизнес-процесс

Что такое тестирование планирования ресурсов предприятия? (ERP - Enterprise Resource Planning)

Планирование ресурсов предприятия, также известное как ERP, представляет собой комплексное программное обеспечение, которое объединяет различные функции организации в единую систему. Программное обеспечение имеет общую базу данных, содержащую всю информацию, относящуюся к различным функциям или подразделениям организации. Система ERP помогает оптимизировать процессы и доступ к информации по всей организации 24 × 7.

Приложения ERP стали критически важными для бесперебойной работы предприятий. Поскольку они включают в себя множество модулей, функций и процессов, необходимость их проверки становится критической. Предприятия осознают необходимость использования модели SMAC (Social, Mobile, Analytics и Cloud) для ускорения роста. Однако капитальный ремонт основных процессов, администрируемых устаревшими приложениями ERP, также важен. Приложения ERP помогают предприятиям управлять различными функциями, отделами и процессами, включая генерируемые в них данные. Эти приложения помогают предприятиям работать как единое целое и в процессе генерировать такие результаты, как повышение производительности, повышение эффективности, сокращение отходов, повышение качества обслуживания клиентов и повышение рентабельности инвестиций. Ввиду важности приложений ERP для организаций, они должны быть протестированы и утверждены. Тестирование приложений ERP может обеспечить бесперебойную работу множества задач в организациях. Они могут включать в себя отслеживание инвентаризации и операций с клиентами, управление финансами и человеческими ресурсами, среди многих других.

Каждое программное обеспечение ERP поставляется с несколькими версиями и требует настройки в соответствии с конкретными бизнес-требованиями. Более того, поскольку каждый элемент приложения связан с каким-либо другим модулем, их обновление может быть сложной задачей. Например, для создания заказа на продажу потребуется доступ к модулю управления запасами. Если какой-либо из модулей не функционирует оптимально, это может повлиять на все приложение ERP. Это может оказать каскадное влияние на производительность компании, а также создать плохой опыт клиентов. Следовательно, тестирование приложений ERP должно обеспечить правильную реализацию программного обеспечения и предотвратить сбои.

Тестирование программного обеспечения ERP, помимо проверки функциональности программного обеспечения, должно обеспечивать точное формирование отчетов и форм. Выявляя и устраняя ошибки на этапе тестирования, тестировщики могут избежать столкновения с проблемами после внедрения. Более того, это может привести к скорейшему внедрению программного обеспечения и обеспечить его бесперебойную работу. Службы тестирования приложений ERP проверяют

бизнес-процессы, функции и регулирующие их правила. Они помогают снизить операционные риски в условиях ограниченности имеющихся ресурсов и времени. Поскольку система ERP содержит огромные объемы данных, тестирование ручных процедур может потребовать много времени и средств. Автоматизированное тестирование может помочь проверить все функции и возможности при минимальных затратах времени и средств. Кроме того, поскольку несколько бизнес-подразделений организации могут иметь разные процессы или процедуры, автоматическое тестирование может проверять точность их результатов по конкретным параметрам. Кроме того, ERP-систему необходимо периодически обновлять с появлением новых технологий, таких как Cloud, Big Data и Mobility. Такие обновления помогают организации проверять транзакции в режиме реального времени, что невозможно вручную.
Системы ERP доступны в нескольких версиях, предназначенных для нескольких доменов, подразделений и клиентов, лучшие доступные инструменты:

- Microsoft Dynamics NAV - для малых и средних предприятий
- SAP Insurance - Для страховых компаний
- Microsoft Dynamics AX - для крупных предприятий
- SAP Banking - Для банковского сектора

Доп. материал:
Тестування CRM-систем на прикладі Salesforce
Тестирование качества видеосвязи WebRTC-based сервиса видеоконференций
WebRTC (англ. real-time communications — коммуникации в реальном времени) — проект с открытым исходным кодом, предназначенный для организации передачи потоковых данных между браузерами или другими поддерживающими его приложениями по технологии точка-точка. Стандарт поддерживается в браузерах, поэтому низкий порог вхождения – не нужно писать клиент. Помимо браузеров известны такие гиганты в сфере видеоконференций, как: Skype, Google Meets/hangouts, Discord.
В чем выражается качество видеосвязи? В подавляющем большинстве случаев речь о:

- Разрешение
- Количество кадров в секунду

Как обычно пытаются тестировать? С помощью плохой сети. Например, отойти с планшетом подальше от wi-fi точки. Вообще плохая сеть подразумевает большой пинг, низкую пропускную способность канала, потерю пакетов.
К сожалению, ручное тестирование видеосвязи (впрочем, как и аудио-) не даст достоверных и точных результатов. На следующем этапе команда приходит к идее писать автотесты (по большей части unit) и лишь некоторые доходят до написания бенчмарков. Возможно, в комментариях опытные коллеги поделятся своим опытом.
Что такое тестирование ETL?
ETL расшифровывается как Extract, Transform, Load. ETL - это процесс, объединяющий три этапа: извлечение, преобразование и загрузка данных из одного источника в другой. Проще говоря, операции ETL выполняются с данными, чтобы вытащить их из одной базы данных в другую. Процесс ETL часто используется в хранилищах данных.

- Первым этапом процесса ETL является извлечение данных. На этом этапе данные извлекаются из исходной базы данных; может быть более одного источника данных.

- На втором этапе, Преобразование данных, извлеченные данные преобразуются путем применения различных правил и функций, которые должны храниться в целевой базе данных в надлежащем формате. Данные извлекаются из разных источников, и вполне вероятно, что у них будет много проблем, например, одному и тому же объекту присваиваются разные имена или одно и то же имя присваивается разным объектам.
- На последнем этапе загрузка данных, преобразованные и однородно отформатированные данные загружаются в базу данных назначения.

ETL-тестирование - это тип тестирования, выполняемый для гарантии того, что данные, перенесенные из исходной в целевую базу данных, являются точными и соответствуют действующим правилам преобразования.

Пример:
Давайте рассмотрим пример слияния двух компаний - компании A и компании B. После слияния их операции будут объединены, а их клиенты, сотрудники и другие данные будут храниться в единой централизованной базе данных. Предположим, что компания A использует базу данных Oracle для хранения всей информации, а компания B использует MySQL. Теперь для объединения своей информации обе компании могут использовать процесс ETL для переноса данных из своих отдельных баз данных в одну согласованную базу данных. В процессе ETL, поскольку две базы данных различны, данные обеих компаний будут в другом формате, будут использоваться разные соглашения об именах, будут использоваться разные структуры таблиц и так далее. Из-за этих различий компаниям необходимо удостовериться, что перед загрузкой данных в целевую базу данных она была должным образом очищена и может сформировать нужный формат. При тестировании ETL тестировщики должны убедиться, что данные обеих баз данных были преобразованы в формат целевой базы данных; необходимые функции преобразования были выполнены; в процессе не было потеряно никаких данных, и данные являются точными.

Типы тестирования ETL:
- Новое тестирование хранилища данных (New Data Warehouse Testing) - в этом типе тестирования ETL все делается с нуля. Информация для ввода данных собирается от клиента. Исходные и целевые базы данных заново создаются и проверяются с использованием инструментов ETL.
- Миграционное тестирование (Migration Testing) - в этом типе тестирования ETL у клиента есть существующее хранилище рабочих данных; у клиента также есть существующий инструмент ETL. Процесс тестирования миграции требуется, когда данные загружаются из существующей базы данных в новую базу данных. Старая база данных называется устаревшей или исходной базой данных, а новая база данных называется целевой базой данных.
- Запрос на изменение (Change Request) - в этом процессе данные выбираются из разных источников и загружаются в существующее хранилище, при этом не используются никакие новые базы данных. Помимо загрузки новых данных, клиенту может потребоваться изменить существующее бизнес-правило или добавить новое бизнес-правило.
- Тестирование отчетов (Report Testing). После создания хранилища данных система позволяет пользователям создавать различные отчеты. Это тестирование проверяет макет, точность данных и ограничения доступа пользователей к отчетам.

Доп. материал:

«Как QA в управлении хранилища данных эволюционировал»
----- Мобильное тестирование -----
Каковы особенности в тестировании мобильных приложений?

- Высокая фрагментация устройств (где посмотреть: лучше аналитика с ваших пользователей, но есть и источники статистики по странам, версиям ОС и вендорам)
- Размеры дисплеев, разрешение и сам по себе тач-интерфейс (ландшафтная и портретная ориентация, все элементы должны быть такого размера, чтобы можно было однозначно по ним попасть; сценарии не должны вести на пустые экраны. Всегда нужно проверять несколько нажатий на одну и ту же кнопку, мультитач (если поддерживается, если нет то аппа не должна на него реагировать), нативные жесты.
- Постоянная обратная связь с пользователем (реакция кнопок на нажатие - у каждого элемента должно быть нажатое состояние, должны быть сообщения при загрузке контента/прогресс, сообщения об успешном исполнении сценария, звук/вибрация должны быть корректно синхронизированы по событию, сообщения при ошибке доступа к сети, наличие сообщений при попытке удалить важную информацию, наличие экрана/сообщения при окончании процесса/игры (экран Game over)).
- Ограничения ресурсов (энергопотребление, утечки памяти (особенно может проявляться на окнах, с большим количеством информации (длинные списки как пример), во время задач с длительным workflow (когда пользователь долго не выходит из приложения), при некорректно работающем кэшировании изображений), не хватает места для установки и работы, обновления, перенос на SD карту.
- Реакция на внешние прерывания (выключение или перезагрузка, входящий звонок или сообщение, обновления приложений, уведомления, разрядка, переход в энергосберегающий режим и режим ожидания, смена ориентации + в режиме ожидания, переход wi-fi -> 3G и обратно, включение и отключение функций необходимых устройству (геопозиции, блютус, режим полета), запрет на использование аппаратных ресурсов, функции с камерой, кейсы с потерей связи, зарядкой устройства, подключением отключением сд карты/симки/АКБ, подключение кабеля или гарнитуры, биометрические (напр для банковского приложения), платежи NFC или просто разные фичи, сворачивание приложения, принудительная остановка).
- Если приложение работает с какими-то форматами файлов, нужно проверять корректность работы с каждым из них
- Учитывать шторку и челку или вырез под фронталки
- Бэкап и восстановление из него
- Работа в режиме разделенного экрана
- Датчики, температура окружающей среды
- Тестирование смартфона начинается с тестирования самого устройства в заводском состоянии.

В чем отличия тестирования мобильного приложения от десктопного?

- В первую очередь это разные операционные системы и разная архитектура «железа», хотя сейчас прогресс нацелен на унификацию (например, такие гиганты, как Microsoft и Apple. У MS это планшеты-ноутбуки Surface на базе ARM и Windows 10, Apple в июне 2020 года заявила о переходе на ARM-архитектуру в компьютерах).
- Пока еще актуальное различие – аппаратные ресурсы. Мощность начинки и количество памяти. Хотя, опять же, в 2020 году этот пункт уже утрачивает актуальность. Мощности новых флагманских мобильных устройств сопоставимы с бюджетными ноутбуками на архитектуре x86/64, а начинка бюджетных и средних по

цене мобильных аппаратов обеспечивает достаточный для большинства нужд уровень производительности.

- Самое очевидное различие в аппаратной части помимо мощности – наличие разнообразных датчиков и модулей связи в мобильном устройстве, а также нескольких камер, вибромотора, сканера отпечатков и т. д.
- Наличие датчика ориентации уже предполагает тестирование в портретной и ландшафтной ориентациях. Добавьте к этому множество разрешений дисплея, их различные типы матриц со своими особенностями отображения и т.п.
- Помимо этого, в мобильном устройстве очень большое внимание уделяется обработке разнообразных прерываний (входящий звонок, уведомление, нажатие кнопки блокировки, выгрузка из ОЗУ и т. д.)
- Основная функция мобильных устройств – по-прежнему связь. Голосовая, но также и через мобильный интернет, что усложняет тестирование по сравнению с десктопами.
- Связь также в контексте взаимодействия с носимой электроникой (беспроводные наушники, фитнес-браслеты, смарт-часы, очки и т.п.), бесконтактной оплатой и т.п.
- Прогресс в обновлениях ОС. В мобильных устройствах это происходит гораздо чаще и имеет большее значение в связи с большой конкуренцией.
- Различия в гайдлайнах для ОС.
- Десктопные приложения чаще всего загружаются с официального веб-сайта производителя. Мобильное приложения почти всегда загружается из соответствующего ОС магазина приложений.
- Ожидания. Т.к. приложения десктопных устройств созданы в основном для осуществления некой функциональной деятельности и являются рабочими инструментами, им может быть иногда простительно то, что в мобильном приложении приведет к немедленному удалению его пользователем и негативным отзывам. Любая мелкая проблема с интуитивностью, интерфейсом, локализацией, функциональностью, производительностью, расходом батареи может моментально отпугнуть пользователя и тот отдаст предпочтение конкурентам.

В чем отличия тестирования мобильного приложения от web?

- Большее количество вариантов и комбинаций ОС/железа и т.п. в мобильных устройствах (сюда же вытекающее следствие необходимости использования эмуляторов)
- Браузеры «стационарны», в то время как при тестировании мобильных приложений нужно учитывать ориентацию, прерывания, связь, наличие дополнительных модулей.
- С т.з. связи веб приложение фактически становится бесполезным при потере интернет-соединения (хотя в последнее время это иногда не совсем так), для нативного мобильного приложения ничего не изменится*.
- Аппаратные ограничения. Веб браузеру обычно доступно куда больше ресурсов.
- Публикация и распространение. Для того, чтобы люди начали пользоваться мобильным приложением, необходим аккаунт разработчика и пройденная модерация в магазине приложений.

Из этих основных различий следуют и различия в тестировании – уровни, виды, типы и т. д.

Мнение:
“Ваши усилия должны скорее быть направлены на выявление узких мест такие как ограничения на доступ к ресурсам выходящие с каждой новой версией мобильных операционных систем - шифрование хранилищ. Глубокая модернизация ядра (Что касается Android) устройств. Нюансы работы оффлайн воркеров для PWA. С другой стороны в "Старших" операционных системах вы столкнетесь с взаимодействием с другими программными продуктами в рамках одной операционной системы - поскольку в них приложения не запускаются в своем собственном Sandbox но очень не слабо влияют друг на друга. Самая яркая иллюстрация этого существование InstallShield в рамках WIndows в течении уже многих лет.” (c) @Havy_Man
Как работает Android, какая у него архитектура?

Applications. Android поставляется с набором основных приложений, включающим календарь, карты, браузер, менеджер контактов и другие. Все перечисленные приложения написаны на Java.
Application Framework. Предоставляя открытую платформу разработки, Android дает разработчикам возможность создавать гибкие и инновационные приложения. Разработчики могут использовать аппаратные возможности устройства, получать информацию о местоположении, выполнять задачи в фоновом режиме, устанавливать оповещения и многое другое. Разработчики имеют полный доступ к тем же API, что используются в основных приложениях.
Архитектура приложений разработана с целью упрощения повторного использования компонентов; любое приложение может "публиковать" свои возможности и любое другое приложение может затем использовать эти возможности (с учетом ограничений безопасности). Этот же механизм позволяет заменять стандартные компоненты на пользовательские.
Libraries. Android включает в себя набор C/C++ библиотек, используемых различными компонентами системы. Эти возможности доступны разработчикам в контексте применения Android Application Framework. Некоторые основные библиотеки, перечислены ниже:
Медиа библиотеки – эти библиотеки предоставляют поддержку воспроизведения и записи многих популярных аудио, видео форматов и форматов изображений, в том числе MPEG4, MP3, AAC, AMR, JPG, PNG и других;
Surface Manager – управляет доступом к подсистеме отображения 2D и 3D графических слоев;
LibWebCore – современный веб-движок, на котором построен браузер Android;
SGL – основной графический движок 2D;
3D библиотеки – реализованы на основе OpenGL; библиотеки используют либо аппаратное 3D-ускорение (при его наличии), либо включены программно;
FreeType – поддержка растровых и векторных шрифтов
SQLite – механизм базы данных, доступной для всех приложений.
Android Runtime. Android включает в себя набор основных библиотек, которые обеспечивают большинство функций, доступных в библиотеках Java. Каждое приложение Android работает в своем собственном процессе, со своим собственным

экземпляром виртуальной машины Dalvik. Dalvik была написана так, что устройство может работать эффективно с несколькими виртуальными машинами одновременно. Dalvik проектировалась специально под платформу Android. Виртуальная машина оптимизирована для низкого потребления памяти и работы на мобильном аппаратном обеспечении. Dalvik использует собственный байт-код. Android-приложения переводятся компилятором в специальный машинно-независимый низкоуровневый код. И именно Dalvik интерпретирует и выполняет такую программу при выполнении на платформе. Кроме того, с помощью специальной утилиты, входящей в состав Android SDK, Dalvik способна переводить байт-коды Java в коды собственного формата и также исполнять их в своей виртуальной среде.
Linux Kernel. Android основан на Linux версии 2.6 с основными системными службами – безопасность, управление памятью, управление процессами и модель драйверов. Разработчики Android модифицировали ядро Linux, добавив поддержку аппаратного обеспечения, используемого в мобильных устройствах и, чаще всего, недоступного на компьютерах.
Источник: https://intuit.ru/studies/courses/4462/988/lecture/14988
Доп. материал: Android's HIDL: Treble in the HAL
Что такое тестирование прерываний (Interrupt Testing)? Причем тут Activity Lifecycle?
У приложения есть жизненный цикл со строго определенными в системе активностями. То есть при любом типе прерывания приложение должно вести себя корректно. Это важно проверять, так как это происходит буквально постоянно, а разработчики могут упустить такие кейсы в коде. Примеры можно посмотреть в разделе полезных ресурсов, там множество чек-листов и мнемоник.

Как устроена iOS?
Все в курсе, что мобильные девайсы Apple работают под управлением iOS. Многие знают, что iOS представляет собой облегченную версию настольной Mac OS X. Некоторые догадываются, что в основе Mac OS X лежит POSIX-совместимая ОС Darwin, а те, кто всерьез интересуется IT, в курсе, что основа Darwin — это ядро XNU, появившееся на свет в результате слияния микроядра Mach и компонентов ядра FreeBSD. Однако все это голые факты, которые ничего не скажут нам о том, как же на самом деле работает iOS и в чем ее отличия от настольного собрата.
Условно начинку OS X / iOS можно разделить на три логических уровня: ядро XNU, слой совместимости со стандартом POSIX (плюс различные системные демоны/сервисы) и слой NeXTSTEP, реализующий графический стек, фреймворк и API приложений. Darwin включает в себя первые два слоя и распространяется свободно, но только в версии для OS X. iOS-вариант, портированный на архитектуру ARM и включающий в себя некоторые доработки, полностью закрыт и распространяется только в составе прошивок для айдевайсов (судя по всему, это защита от портирования iOS на другие устройства).
По своей сути Darwin — это «голая» UNIX-подобная ОС, которая включает в себя POSIX API, шелл, набор команд и сервисов, минимально необходимых для работы системы в консольном режиме и запуска UNIX-софта. В этом плане он похож на базовую систему FreeBSD или минимальную установку какого-нибудь Arch Linux, которые позволяют запустить консольный UNIX-софт, но не имеют ни графической оболочки, ни всего необходимого для запуска серьезных графических приложений из сред GNOME или KDE.

Ключевой компонент Darwin — гибридное ядро XNU, основанное, как уже было сказано выше, на ядре Mach и компонентах ядра FreeBSD, таких как планировщик процессов, сетевой стек и виртуальная файловая система (слой VFS). В отличие от Mach и FreeBSD, ядро OS X использует собственный API драйверов, названный I/O Kit и позволяющий писать драйверы на C++, используя объектно-ориентированный подход, сильно упрощающий разработку.
iOS использует несколько измененную версию XNU, однако в силу того, что ядро iOS закрыто, сказать, что именно изменила Apple, затруднительно. Известно только, что оно собрано с другими опциями компилятора и модифицированным менеджером памяти, который учитывает небольшие объемы оперативки в мобильных устройствах. Во всем остальном это все то же XNU, которое можно найти в виде зашифрованного кеша (ядро + все драйверы/модули) в каталоге /System/Library/Caches/com.apple.kernelcaches/kernelcache на самом устройстве.
Уровнем выше ядра в Darwin располагается слой UNIX/BSD, включающий в себя набор стандартных библиотек языка си (libc, libmatch, libpthread и так далее), а также инструменты командной строки, набор шеллов (bash, tcsh и ksh) и демонов, таких как launchd и стандартный SSH-сервер. Последний, кстати, можно активировать путем правки файла /System/Library/LaunchDaemons/ssh.plist. Если, конечно, джейлбрейкнуть девайс.
На этом открытая часть ОС под названием Darwin заканчивается, и начинается слой фреймворков, которые как раз и образуют то, что мы привыкли считать OS X / iOS.

Darwin реализует лишь базовую часть Mac OS / iOS, которая отвечает только за низкоуровневые функции (драйверы, запуск/остановка системы, управление сетью, изоляция приложений и так далее). Та часть системы, которая видна пользователю и приложениям, в его состав не входит и реализована в так называемых фреймворках — наборах библиотек и сервисов, которые отвечают в том числе за формирование графического окружения и высокоуровневый API для сторонних и стоковых приложений
Как и во многих других ОС, API Mac OS и iOS разделен на публичный и приватный. Сторонним приложениям доступен исключительно публичный и сильно урезанный API, однако jailbreak-приложения могут использовать и приватный.
В стандартной поставке Mac OS и iOS можно найти десятки различных фреймворков, которые отвечают за доступ к самым разным функциям ОС — от реализации адресной книги (фреймворк AddressBook) до библиотеки OpenGL (GLKit). Набор базовых фреймворков для разработки графических приложений объединен в так называемый Cocoa API, своего рода метафреймворк, позволяющий получить доступ к основным возможностям ОС. В iOS он носит имя Cocoa Touch и отличается от настольной версии ориентацией на сенсорные дисплеи.
<...>
Источник: Как устроена iOS
Жизненный цикл iOS-приложения?
В любой момент времени ваши приложение находятся в каком либо из перечисленных ниже состояний. Система меняет состояния вашего приложения в ответ на происходящие события. Например, когда пользователь нажимает кнопку Home, или поступает входящий вызов, или что либо еще — приложения в ответ на все это меняют свое состояние.
Not Running Приложение не запущено, либо запущенно но прервано системой.

Inactive Приложение работает, но в настоящий момент ничего не делает (это может быть связано с выполнением другого кода). Приложение, как правило, остается в этом состоянии очень мало времени и переходит в другое состояние
Active Нормальный обычный режим работы приложения на переднем плане.
Background Приложение находится в фоне, но работает. Большинство приложений входят в это состояние на короткое время и позже приостанавливаются. Но если необходимо дополнительное время для работы в бекграунде, приложение может оставаться в этом состоянии
Suspended Приложение работает в фоне, но не выполняет никакой код. Система перемещает приложение в это состояние автоматически и не предупреждает об этом. При условии малого количества памяти, система может не предупреждая закрыть приложения в этом состоянии для освобождения памяти.

Приложение должно быть готово к завершению в любое время. Завершение — это нормальная часть жизненного цикла. Система обычно выключает приложения, для очищения памяти и подготовки к запуску других приложений, которые запущены пользователем, но система также может выключить приложения , которые некорректно или не отвечающим на события своевременно.
Suspended приложения не получают уведомления о завершении. Система убивает процесс и восстанавливает соответствующую память. Если приложение запущено в фоне и не отвечает, система вызовет applicationWillTerminate: чтобы приложение подготовилось к выключению. Система не вызывает метод когда устройство перезагружается.

Доп. материал:
PREWORKING • 4• : • ЖИЗНЕННЫЙ ЦИКЛ IOS ПРИЛОЖЕНИЯ (источник) iOS Application Lifecycle, • или жизненный цикл iOS приложения
Что вы знаете о симуляторах и эмуляторах?
Всегда идет спор о том, что тестирование мобильных приложений более экономично и быстрее на реальных устройствах. Но прежде чем разрушить этот миф, давайте взглянем на базовое определение эмуляторов, симуляторов и реальных устройств:
• Тестирование на реальном устройстве позволяет запускать мобильные приложения и проверять его функциональность. Тестирование реального устройства гарантирует, что ваше приложение будет работать без проблем на клиентских телефонах.
• эмулятор пытается дублировать внутреннее устройство устройства, то есть представляет собой полную повторную реализацию конкретного устройства или платформы. Эмулятор действует точно так же, как и реальное устройство.
• симулятор пытается дублировать поведение устройства. Как правило, симулятор — это имитация лишь отдельных свойств, возможностей или функций симулируемой системы, причем не в полном объеме, а только в том, в каком это необходимо в рамках тех задач, которые были поставлены перед симулятором. Вы как будто бы работаете в настоящей ОС, но при этом под капотом оно полностью или частично "фальшивое"
Тестирование на симуляторах или эмуляторах имеет много преимуществ, так как они бесплатны, имеют высокую скорость выполнения, не имеют проблем при автоматическом тестировании, просты в отладке и просты в настройке. Но эмулятор / симулятор не всегда является лучшим решением для таких сценариев, как те, в

которых команде тестирования необходимо проверять производительность приложения в течение более длительного периода времени. Реальные устройства стоят дороже по сравнению с эмулятором / симуляторами. Но, как следует из названия, результаты реальных устройств всегда более точны по сравнению с эмуляторами или симуляторами. Для тестирования мобильных приложений всегда требуется большое количество мобильных устройств, поскольку тестирование мобильных приложений - это все о комбинациях ПО и железа. Следовательно, в таких случаях эффективность может быть достигнута только с помощью реальных устройств.

Доп. материал:
Эмулятор и симулятор мобильного устройства против реального устройства
Типы мобильных приложений?

- Нативные приложения: Эти приложения называют нативными оттого, что они написаны на родном (с англ. native – родной) для определенной платформы языке программирования. Для Android этим языком является Kotlin/Java, тогда как для iOS – objective-C или Swift. Нативные приложения находятся на самом устройстве, доступ к ним можно получить, нажав на иконку. Они устанавливаются через магазин приложений (Play Market на Android, App Store на iOS и др.). Они разработаны специально для конкретной платформы и могут использовать все возможности устройства – камеру, GPS-датчик, акселерометр, компас, список контактов и все остальное. Также они могут распознавать стандартные жесты, предустановленные операционной системой или совершенно новые жесты, которые используются в конкретном приложении. В силу того, что нативные приложения оптимизированы под конкретную ОС, они органично вписываются в любой смартфон, отличаясь высокой скоростью работы и производительностью. Нативные приложения могут получить доступ к системе оповещений устройства, а также, в зависимости от предназначения нативного приложения, оно может всецело или частично обходиться без наличия интернет-соединения.

- Мобильные веб-приложения: На самом деле мобильные веб-приложения не являются приложениями как таковыми. Ведь дело в том, что веб-приложение, в сущности, представляет собой сайт, который адаптирован и оптимизирован под любой смартфон. И для того, чтобы воспользоваться им, достаточно иметь на устройстве браузер, знать его адрес и располагать интернет-соединением (благодаря ему происходит обновление информации в данном виде приложений). Запуская мобильные веб-приложения, пользователь выполняет все те действия, которые он выполняет при переходе на любой веб-сайт, а также получает возможность «установить» их на свой рабочий стол, создав закладку страницы веб-сайта. Веб-приложения отличаются кроссплатформенностью, то есть способны функционировать, независимо от платформы девайса. Козырем в их рукаве выступает и то, что они не используют его программное обеспечение. А по причине того, что являются мобильной версией сайта с расширенным интерактивом, веб-приложения не отбирают драгоценное место в памяти смартфона. Веб-приложения стали широко популярны в то время, когда начал развиваться HTML5 и люди осознали, что могут получить доступ к множеству функций нативных приложений, просто зайдя на веб-сайт через обычный браузер. На сегодняшний день сложно сказать, где именно располагается четкая граница между веб-приложениями и обычными веб-страницами,

поскольку функционал HTML5 растет с каждым днем и все больше и больше сайтов его используют. В то же время камень в огород веб-приложений следует бросить за неспособность работать с ними без Интернета. Причем из этого выплывает и другой минус – их производительность, которая находится на среднем уровне, в сравнении с другими видами приложений. Более того, она зависит от возможностей интернет-соединения провайдера услуг. Также очевидно, что веб-приложения не могут получить доступ к функциям системы и самого устройства.

- Гибридные приложения представляют собой сочетание веб и нативных приложений: доступ к функционалу смартфона (API системы, пуш и т.п.), размещение в маркетах, простота обновления контента, кроссплатформенность веб-части. Фактически это можно объяснить как нативное приложение, в обертке которого загружается веб-приложение. Это может быть основным контентом или лишь одной из функций. Очевидно, без интернет-соединения соответствующая часть приложения потеряет работоспособность.

Что основное проверить при тестировании мобильного приложения?
Проверяем в начале тестирования:

- достаточно ли устройств для тестирования (по целевым рынкам) и готово ли покрытие
- согласованы все процедуры на проекте (понять что все готово, выстроили регламент цикл разработки, договорились кто когда вступает в работу, набрана команда)
- аппа соответствует гайдлайнам
- аппа работает в различных окружениях
- аппа должна корректно работать с внешними и внутренними запросами
- Проверка энергопотребления (как типичный юзер)
- Выбор тестов для автоматизации (когда уже написано достаточно ручных)

Список кейсов под iPhone из утерянного источника, довольно неплохой:

- Перепроверка функциональности, где ранее были обнаружены наиболее критические дефекты (регрессионное тестирование)
- Проверка функциональности на корректных данных (текущая дата, короткие имена и т.д.)
- Проверка на некорректных значениях (например: пустые поля, длинные имена, установка на телефоне даты в прошлом и т.д.)
- Проверка интерфейса приложения на соответствие требованиям Apple (Human interface guidelines for iPhone/iPad)
- Производительность приложения и скорость ответа интерфейса (используется iPhone 2g)
- Тесты на удобство пользования приложением (Usability tests)
- Тест на совместимость с другими приложениями/функциональностью iPhone (будильник/таймер/напоминания/входящий звонок/смс)
- Проверка настроек приложения и корректность их применения
- Поиск возможных мест «падения» приложения (crash) и причин их возникновения
- Корректность работы приложения при использовании wi-fi/gprs (включая обрывы связи/ее отсутствие)
- Проверка на корректность работы приложения с памятью iPhone (memory leaks)
- проверка того, что звук не пропадает при подключении наушников

- Поведение приложения при переходе iPhone в спящий режим
- Работа приложения с акселерометром (поворот экрана в соответствии с положением iPhone, использование функции акселерометра для получения данных приложением (шагомер))
- Тестирование локализации (при поддержке приложением)
- Проверка корректности работы приложения с камерой iPhone (если такая функциональность поддерживается), а также корректность работы приложения с iPod.
- быстрые «клики» по элементам интерфейса (переход по категориям, переход по записям внутри категории)
- если есть длительный workflow – проводить его весь (вроде длинных программ в Yoga) в реальном времени
- если есть готовый список и поле для вбивания параметров, то проверить поведение, когда в поле появляется подсказка из словаря и одновременно кликаешь по записи в списке <> подсказке. возможны конфликты между подсказкой айфона и реальным выбором.
- проверка контента: адекватный размер изображений (до 1МБ) и достаточное качество. Дополнительно смотреть на iPhone4 (большее разрешение) + см. MobileHIG.pdf chapter 11 для требований к разрешению изображений.
- GUI: иконки соответствуют тому, к чему относятся (хелп – знак вопроса, настройки – шестеренка и т.д.), новые окна плавно открываются справа, присутствует значок загрузки если происходит длительный процесс)
- Наличие экрана Game Over и корректные ссылки на нем – для игровых проектов (+ корректная отработка попадания на этот экран)

Мультиплеерные игры.

- корректность подключения игроков (напр., списывание баланса только после подключения)
- временные лаги
- подключение через различные сети
- корректное поведение при отключении игроков
- подключение ботов (если используются)

Conformance testing:

- Protocol testing
- Safety/• Security testing
- SIM card testing
- Radio Frequency(RF) testing
- Audio Tests
- Specific Absorption Tests

И еще: Физическая/виртуальная клавиатура, проверка обновления и чистой установки. Платный контент: соответствие цен и содержимого, покупки 2 типов (восстанавливаемые и невосстанавливаемые (кредиты)) - проверка восстановления покупок привязанных к учетке (переустановка/обновление/другое устройство)+должен быть выбор из текущего прогресса и сохраненного в учетке. Реклама: не должна перекрывать элементы, должна иметь доступную кнопку закрытия (А если кнопка еще не появилась, то каунтдаун до этого). Глобализация - меняется все что нужно и это происходит корректно. Защита от получения преимуществ при манипуляциях с датой и временем. Копирование и вставка из/в приложение.

Больше чек-листов и идей можно найти в разделе полезных ресурсов.
Как быть с покрытием девайсов?
Большинство багов обнаруживается на покрытии около 30% девайсов - разрешение, мощность, версия ос+нижняя граница для данной аппы. Варианты: физическая ферма устройств, эмулятор и симулятор (BrowserStack (облачный), родной в Android Studio, BlueStack, Genymotion и т.п.). Вообще физический устройств желательно иметь хотя бы штук 6 - два отличающихся айфона, по планшету на iOS и Android, 2 разных андроида. Хуавей сейчас тестится отдельно из-за гугл сервисов.
Доп. материал:
Выбор мобильных устройств: пошаговая инструкция для начинающих QA. Часть II
https://habr.com/ru/post/• 464433• /
Последнее обновление Android/iOS, что нового?
Актуализируется перед собеседованием, т.к. написанное здесь будет слишком быстро устаревать.
Основные различия iOS и Android?

- В первую очередь, очевидно, это разные операционные системы от разных компаний
- Различная целевая аудитория, в т.ч. разный ее размер (У Андроид сегодня ориентировочно 80% всех гаджетов в мире)
- Различная монетизация, в т.ч. ее размер (Согласно статистике, iOS пользуются более платежеспособные люди, которые делают покупки в 3 раза чаще).
- Сложность вхождения в разработку (Для разработки желательно иметь реальный девайс на соответствующей ОС и ПК с соответствующей ОС для разработки. Технику для Android-разработки можно купить значительно дешевле. Даже подписка для разработчика iOS стоит в несколько раз дороже)
- Разное количество разработчиков (Java/Kotlin у Android намного распространеннее и чаще встречается в других сферах, нежели Swift у iOS)
- Отличия в модерации приложений для публикации в магазине приложений - у Android процедура значительно быстрее и проще, в Apple в первую очередь проверяют оплату через App Store, безопасность и очень большое внимание уделяют юзабилити и доступности;
- Отличия с точки зрения дизайна и всего связанного с ним можно узнать, прочитав гайдлайны к каждой системе (Гайдлайны (Guidelines) — набор рекомендаций, правил, принципов от создателей платформы, операционной системы, благодаря которым приложения под эти платформы и ОС от разных разработчиков выглядят единообразно). Вот некоторые из нескольких десятков отличий:
- Human Interface Guidelines vs Material Design
- Единицы измерения: pt vs dp
- Размер экрана: 320 pt x 568 pt vs 360 dp x 640 dp
- Системный шрифт: San Francisco vs Roboto
- Android Navigation Bar (в отличие от iOS, у Android есть встроенный инструмент навигации назад)
- Важность теней в Material (в iOS они почти не используются)
- Отличия в нейминге в разных местах
- Отличия в навигации и паттернах (UX)
- Разные Status Bar
- Разные Control
- Разный вид стрелки «Назад» и положение заголовка

- Action View/Activity View vs Modal Bottom Sheet
- Разные требования к размеру зоны нажатия

Доп. материал:
Google play Launch checklist
App Store Review Guidelines
Что такое Middleware?

Связу́ющее програ́ммное обеспе́чение (англ. middleware; также переводится как промежу́точное программное обеспечение, программное обеспечение среднего слоя, подпрогра́ммное обеспечение, межплатфо́рменное программное обеспечение) — широко используемый термин, означающий слой или комплекс технологического программного обеспечения для обеспечения взаимодействия между различными приложениями, системами, компонентами. (с) Вики. В данном разделе нас интересует применение в мобильной разработке.
Особенностью заказной разработки мобильных приложений является то, что основной сервер с API обычно предоставляет заказчик. Ниже список, с чем в этом случае придется иметь дело мобильным разработчикам:
- API не дописано — разработчики заказчика загружены текущей работой и не мотивированы сдать его в срок, при том, что у маркетинга планы и сроки запуска мобильного приложения горят;
- API сильно не совпадает со структурой мобильного приложения (данные для экранов приходится дергать с 3-4 методов и обрабатывать локально);
- Нет документации, либо она сильно разрознена и не актуальна;
- Несколько точек входа (разросшаяся инфраструктура находится на нескольких серверах с разными адресами);
- Уже существующее API меняется после апдейтов основного сайта;Отсутствие тестовых серверов;
- Баги (много багов).

И добавим сюда понимание, что со всем этим придется бороться сразу на двух платформах, которые хоть и являются мобильными, часто используют разные архитектурные подходы и, как следствие, разные сроки старта и готовности этапов. Все это приводит к увеличению сроков разработки и, соответственно, стоимости разработки.
Для минимизации всех этих проблем предлагается использовать промежуточный сервер — шину данных.
Middleware - чаще всего простой быстро настраиваемый сервер, не хранящий каких-либо данных, кроме логов. Он позволяет использовать для общения мобильных приложений с собой простой REST API, строго подогнанный под логику экранов, а сам уже обращается к целевому API необходимым образом.
Бонусом мы имеем возможность разрабатывать приложения со своими наработками по авторизации, обработкам ошибок и прочим мелочам, протестированными и проверенными.
Плюсы такого подхода:
- Отсутствуют простои из-за неготовности API заказчика — в худшем случае на шине отдаются тестовые данные;
- Упрощается реализация мобильного приложения практически до тонкого клиента;

- Единственная точка входа позволяет упростить архитектуру работы с сетью в МП;
- Возможность выкладывать обновления для сервера шины (меняющую взаимодействие с сервером заказчиком) без обновления МП, в кратчайшие сроки, без модерации со стороны третьих фирм;
- Простота и отсутствие полноценной БД позволяет легко разворачивать любое количество тестовых серверов;
- Удобное логирование всех сетевых ошибок и оповещение о них;
- Изменения в работе API заказчика необходимо вносить только в одном месте — на сервере шины;
- Документация ведется принятым и привычным в компании способом;
- Использование наработок снижает сроки и стоимость разработки.

Все это позволяет снизить сроки и стоимость, напрямую и косвенно, за счет снижения рисков простоя, сложности реализации серверной части и тестирования. Шикарный бонус разработчику — возможность предложить более низкую цену и выиграть конкурс, а заказчику - сэкономить и уменьшить сроки внедрения МП.
Источник: Middleware: необходимость в мире разработки мобильных приложений

Виды жестов и т.п.?
Основное, остальное см. в полезных материалах (сборники терминов).

- Скроллинг (scrolling) – прокрутка экрана
- Драг-энд-дроп, дрэг-энд-дроп (drag-and-drop) — «тащить и бросить» — нажать в одном месте, затем двигать и отпустить в другом месте
- Гестуры, жесты (gestures) — разные формы движения мыши, пальца или другого указывающего устройства. Могут быть запрограммированы для выполнения определенных действий. Например, движение пальца сверху вниз по экрану мобильного браузера перезагружает страницу
- Тап, тэп (tap) — короткое нажатие пальцем, сродни клику
- Двойной тап, Дабл-тап, дабл-тэп (double tap) — два коротких нажатия пальцем, сродни дабл-клику
- Длинный тап, Тач (touch) — нажатие дольше, чем Тап
- Тач-энд-холд (touch and hold) — нажать и держать
- Флик (flick) — щелчок наискось по экрану в сторону будущего движения изображения экрана, после флика изображение продолжает двигаться по инерции
- Свайп (swipe), Слайд (slide) — продолжительное скольжение пальцем по экрану
- Пинч (pinch) — щипок, сжимающее движение одновременно двумя пальцами по экрану для уменьшения изображения
- Спред / Спрэд (spread), Стретч (stretch: для Microsoft), Пинч-ит-опен (pinch it open: для Apple) — растягивающее движение одновременно двумя пальцами по экрану для увеличения изображения
- Пан, пэн (pan) — медленное движение пальца по экрану для перемещения и разглядывания увеличенной картинки

Доп. материал:
UI-элементы и жесты в мобильных приложениях
Как проверить использование процессора на мобильных устройствах?
В Google Play или App Store доступны различные инструменты,, из которых можно установить приложения, такие как CPU Monitor, Usemon, CPU Stats, CPU-Z и т. д. Это

расширенный инструмент, который записывает информацию о процессах, запущенных на вашем устройстве. Не все показатели могут быть доступны, это зависит от конкретного устройства. Также в инструментах разработчика доступны инструменты профилирования. Другой вариант - профилировщик в IDE (Android Studio).
Что вы знаете о PWA?
Перспектива разработки мобильного приложения, которое не потребуется скачивать и ждать review из App Store, очень заманчива, ведь аналогов привычного ПО существует несколько: Progressive Web Apps (PWA), Android Instant Apps (AIA) и Accelerated Mobile Pages (AMP).
Доп. материал:
Тестирование аналогов инсталлируемых приложений. Диана Пинчук. Comaqa Spring • 2019
How to Test PWA. In this article we will look at PWA and• … • | by Slava Kirzhak | Effective Developers

----- Сети и около них -----

Некоторая база в одной статье: Сети для начинающего IT-специалиста. Обязательная база
Что такое http?
HTTP — широко распространенный протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов (то есть документов, которые могут содержать ссылки, позволяющие организовать переход к другим документам), сейчас же для любых данных.
Протокол HTTP предполагает использование клиент-серверной структуры передачи данных. Клиентское приложение формирует запрос и отправляет его на сервер, после чего серверное ПО обрабатывает данный запрос, формирует ответ и передает его обратно клиенту.
То есть этот протокол не только устанавливает правила обмена информацией, но и служит транспортом для передачи данных — с его помощью браузер загружает содержимое сайта на ваш компьютер или смартфон.

У HTTP есть один недостаток: данные передаются в открытом виде и никак не защищены. На пути из точки А в точку Б информация в интернете проходит через десятки промежуточных узлов, и, если хоть один из них находится под контролем

злоумышленника, данные могут перехватить. То же самое может произойти, когда вы пользуетесь незащищенной сетью Wi-Fi, например, в кафе. Для установки безопасного соединения используется протокол HTTPS с поддержкой шифрования.
Защиту данных в HTTPS обеспечивает криптографический протокол SSL/TLS, который шифрует передаваемую информацию. По сути этот протокол является оберткой для HTTP. Он обеспечивает шифрование данных и делает их недоступными для просмотра посторонними. Протокол SSL/TLS хорош тем, что позволяет двум незнакомым между собой участникам сети установить защищенное соединение через незащищенный канал. При установке безопасного соединения по HTTPS ваш компьютер и сервер сначала выбирают общий секретный ключ, а затем обмениваются информацией, шифруя ее с помощью этого ключа. Общий секретный ключ генерируется заново для каждого сеанса связи. Его нельзя перехватить и практически невозможно подобрать — обычно это число длиной более 100 знаков. Этот одноразовый секретный ключ и используется для шифрования всего общения браузера и сервера. Казалось бы, идеальная система, гарантирующая абсолютную безопасность соединения. Однако для полной надежности ей кое-чего не хватает: гарантии того, что ваш собеседник именно тот, за кого себя выдает. Для этой гарантии существует сертификат.
Вам как пользователю сертификат не нужен, но любой сервер (сайт), который хочет установить безопасное соединение с вами, должен его иметь. Сертификат подтверждает две вещи: 1) Лицо, которому он выдан, действительно существует и 2) Оно управляет сервером, который указан в сертификате. Выдачей сертификатов занимаются центры сертификации — что-то вроде паспортных столов. Как и в паспорте, в сертификате содержатся данные о его владельце, в том числе имя (или название организации), а также подпись, удостоверяющая подлинность сертификата. Проверка подлинности сертификата — первое, что делает браузер при установке безопасного HTTPS-соединения. Обмен данными начинается только в том случае, если проверка прошла успешно.

Доп. материал:
HTTP • для тестировщиков
Официальная документация
Обзор протокола HTTP
Компоненты HTTP?
HTTP определяет следующую структуру запроса (request):
- стартовая строка (starting line) — определяет тип сообщения, имеет вид Метод URI HTTP/Версия протокола, например GET /web-programming/index.html HTTP/1.1
- заголовки запроса (header fields) — характеризуют тело сообщения, параметры передачи и прочие сведения
- тело сообщения (body) — необязательное

HTTP определяет следующую структуру ответного сообщения (response):
- строка состояния (status line), включающая код состояния и сообщение о причине
- поля заголовка ответа (header fields)
- дополнительное тело сообщения (body)

Доп. материал: https://iit-web-lectures.readthedocs.io/ru/latest/www/http.html
Методы HTTP-запроса?

Метод, используемый в HTTP-запросе, указывает, какое действие вы хотите выполнить с этим запросом. Раньше хватало только GET, т.к. считалось, что вы можете хотеть от сервера только получить ответ. Но сейчас вам может понадобиться отредактировать профиль, удалить пост в соц. сети и т.п. Тогда для удобства были созданы различные методы. Вот основные:

- GET: получить подробную информацию о ресурсе
- POST: создать новый ресурс
- PUT: обновить существующий ресурс
- DELETE: Удалить ресурс

Вообще по спецификации HTTP из всех методов сервер должен уметь понимать только GET, а остальные на усмотрение. Но при этом и не задано строго, что сервер должен делать при получении запроса. То есть гипотетически вы с помощью одного метода можете делать вообще любую операцию. Однако в этом нет никакого практического смысла. В дальнейшем было введено соглашение REST, определившее структуру построения веб-приложений, в том числе и работу с методами.
Доп. материал:
Parameter Binding
HTTP POST with URL query parameters • —• • good idea or not?
Что такое endpoint, ресурс? URI, URL, URN?
Смысл в том, что сайт, написанный на любом языке, поддерживающем HTTP запросы, не посылает на сервер никаких PHP/C/Python команд, а общается ним с помощью запросов, описанных в API.
Адрес, на который посылаются сообщения называется Endpoint. Обычно это URL (например, название сайта) и порт. Если я хочу создать веб сервис на порту 8080, Endpoint будет выглядеть так: http://vladislaveremeev.ru:8080
Если моему Web сервису нужно будет отвечать на различные сообщения я создам сразу несколько URL (interfaces) по которым к сервису можно будет обратиться.
Например:
https://vladislaveremeev.ru:• 8080 /resource1/status
https://vladislaveremeev.ru:• 8080 /resource1/getserviceInfo
https://vladislaveremeev.ru:• 8080 /resource1/putID
http://vladislaveremeev.ru:• 8080 /resource1/eventslist
https://vladislaveremeev.ru:• 8080 /resource2/putID
Как видите у моих эндпойнтов (Endpoints) различные окончания. Такое окончание в Endpoint называются Resource, а начало Base URL.
Такое определение Endpoint и Resource используется, например, в SOAP UI для RESTful интерфейсов
https://vladislaveremeev.ru:8080 - это Base URL
/resource1/status - это Resource
Endpoint = Base URL + Resource
Понятие Endpoint может использоваться в более широком смысле. Можно сказать, что какой-то определенный роутер или компьютер является Endpoint. Обычно это понятно из контекста.
Также следует обратить внимание на то, что понятие Endpoint выходит за рамки RESTful и может использовать как в SOAP так и в других протоколах.
Термин Resource также связан с RESTful, но в более широком смысле может означать что-то другое.

Итак, простейший запрос состоит из метода и Endpoint
Request = Method + Endpoint
Ресурс — это ключевая абстракция, на которой концентрируется протокол HTTP.
Ресурс — это все, что вы хотите показать внешнему миру через ваше приложение.
Например, если мы пишем приложение для управления задачами, экземпляры ресурсов будут следующие:

- Конкретный пользователь
- Конкретная задача
- Список задач

Когда вы разрабатываете RESTful сервисы, вы должны сосредоточить свое внимание на ресурсах приложения. Способ, которым мы идентифицируем ресурс для предоставления, состоит в том, чтобы назначить ему URI — универсальный идентификатор ресурса. Например:

- Создать пользователя: POST /users
- Удалить пользователя: DELETE /users/1
- Получить всех пользователей: GET /users
- Получить одного пользователя: GET /users/1

Расшифруем аббревиатуры:

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

Рассмотрим примеры:

- URI – •https://wiki.merionet.ru/images/vse-chto-vam-nuzhno-znat-pro-devops/• 1• .png
- URL - • https://wiki.merionet.ru
- URN - images/vse-chto-vam-nuzhno-znat-pro-devops/1.png

URI содержит в себе следующие части:

- Схема (scheme) - показывает на то, как обращаться к ресурсу, чаще всего это сетевой протокол (http, ftp, ldap)
- Иерархическая часть (hier-part) - данные, необходимые для идентификации ресурса (например, адрес сайта)
- Запрос (query) - необязательные дополнительные данные ресурса (например, поисковой запрос)
- Фрагмент (fragment) – необязательный компонент для идентификации вторичного ресурса ресурса (например, место на странице)

Общий синтаксис URI выглядит так:
URI = scheme ":" hier-part ["?" query] ["#" fragment]
Теперь, когда мы знаем, что такое URI, URL тоже должен быть достаточно понятным.
Всегда помните - URI может содержать URL, но URL указывает только адрес ресурса.
URL содержит следующую информацию:

- Протокол, который используется для доступа к ресурсу – http, https, ftp
- Расположение сервера с использованием IP-адреса или имени домена - например, wiki.merionet.ru - это имя домена. • https://• 192.168.1.17 - здесь ресурс расположен по указанному IP-адресу

• Номер порта на сервере. Например, • http://localhost: 8080, где 8080 - это порт.
• Точное местоположение в структуре каталогов сервера. Например - https://wiki.merionet.ru/ip-telephoniya/ - это точное местоположение, если пользователь хочет перейти в раздел про телефонию на сайте.
• Необязательный идентификатор фрагмента. Например, • https://www.google.com/search?ei=qw• 3• eqwe• 12• e• 1• w• &• q=URL, где q = URL - это строка запроса, введенная пользователем.

Синтаксис:
[protocol]://www.[domain_name]:[port 80]/[path or exaction resource location]?[query]#[fragment]

Источник: url и uri - в чем различие?
Что такое веб-сервис/веб-служба? (WS - Web service)
Web Service - программная система, предназначенная поддерживать взаимодействие между интераперабельными устройствами через сеть. Веб сервис обладает интерфейсом, описанным в WSDL формате. Другие системы, взаимодействуют с веб сервисом через SOAP-сообщения, которые обычно передаются с помощью HTTP с XML сериализацией в связке с другими веб-стандартами.
• Сервис доступен по сети, может располагаться и выполняться на разных компьютерах.
• Передача сообщений между сервисом и клиентом происходит в независимом формате.
• Web Service может быть создан из существующего Web приложения.
• Сервис использует стандартизированную XML messaging систему.
• Не привязан к операционной системе или языку программирования

Доп. материал:
Веб-сервисы
Что такое веб-сервисы?
What is Microservices?
Microservices vs Monolith: which architecture is the best choice for your business?
Отличие сервиса от сервера?
В контексте архитектуры программного обеспечения, сервис-ориентированности и сервис-ориентированной архитектуры термин «сервис» относится к программным функциям или набору программных функций (таких как получение указанной информации или выполнение набора операций) или «механизм, обеспечивающий доступ к одной или нескольким возможностям, где доступ предоставляется с использованием предписанного интерфейса и осуществляется в соответствии с ограничениями и политиками, указанными в описании службы».
В вычислительной технике сервер - это часть компьютерного оборудования или программного обеспечения (компьютерная программа), которая обеспечивает функциональные возможности для других программ или устройств, называемых «клиентами». Эта архитектура называется клиент-серверной. Серверы могут предоставлять различные функции, часто называемые «сервисами», такие как совместное использование данных или ресурсов между несколькими клиентами или выполнение вычислений для клиента. Один сервер может обслуживать несколько клиентов, а один клиент может использовать несколько серверов. Клиентский процесс

может работать на том же устройстве или может подключаться по сети к серверу на другом устройстве. Типичными серверами являются серверы баз данных, файловые серверы, почтовые серверы, серверы печати, веб-серверы, игровые серверы и серверы приложений.
Отличие сервиса от веб-сайта?

- Веб-сервис не имеет пользовательского интерфейса. Веб-сайт имеет пользовательский интерфейс или графический интерфейс.
- Веб-сервисы предназначены для взаимодействия других приложений через Интернет. Веб-сайты предназначены для использования людьми.
- Веб-сервисы не зависят от платформы, так как используют открытые протоколы. Веб-сайты являются кроссплатформенными, так как требуют настройки для работы в разных браузерах, операционных системах и т. д.
- Доступ к веб-сервисам осуществляется с помощью HTTP-методов - GET, POST, PUT, DELETE и т. д. Доступ к веб-сайтам осуществляется с помощью компонентов GUI - кнопок, текстовых полей, форм и т. д.
- Например, Google maps API - это веб-сервис, который может использоваться веб-сайтами для отображения Карт путем передачи ему координат. Например, ArtOfTesting.com - это веб-сайт, на котором есть коллекция связанных веб-страниц, содержащих учебные пособия.

Что такое REST, SOAP? В чем отличия?

- SOAP (Simple Object Access Protocol) — стандартный протокол обмена структурированными сообщениями в распределенной вычислительной среде. Данные передаются в XML.
- REST (Representational State Transfer) — архитектурный стиль взаимодействия компьютерных систем в сети основанный на методах протокола HTTP. Данные по умолчанию передаются в JSON.

SOAP и REST нельзя сравнивать напрямую, поскольку первый - это протокол (или, по крайней мере, пытается им быть), а второй - архитектурный стиль.
Основное различие между SOAP и REST заключается в степени связи между реализациями клиента и сервера. Клиент SOAP работает как пользовательское настольное приложение, тесно связанное с сервером. Между клиентом и сервером существует жесткое соглашение, и ожидается, что все сломается, если какая-либо из сторон что-то изменит. Вам нужно постоянное обновление после любого изменения, но легче определить, выполняется ли контракт.
REST-клиент больше похож на браузер. Это универсальный клиент, который знает, как использовать протокол и стандартизированные методы, и приложение должно соответствовать этому. Вы не нарушаете стандарты протокола, создавая дополнительные методы, вы используете стандартные методы и создаете с ними действия для своего типа медиа. Если все сделано правильно, связности будет меньше, и с изменениями можно справиться более изящно.

То есть SOAP более применим в сложных архитектурах, где взаимодействие с объектами выходит за рамки теории CRUD, а вот в тех приложениях, которые не покидают рамки данной теории, вполне применимым может оказаться именно REST ввиду своей простоты и прозрачности. Действительно, если любым объектам вашего сервиса не нужны более сложные взаимоотношения, кроме: «Создать», «Прочитать», «Изменить», «Удалить» (как правило — в 99% случаев этого достаточно), возможно,

именно REST станет правильным выбором. Кроме того, REST по сравнению с SOAP, может оказаться и более производительным, так как не требует затрат на разбор сложных XML команд на сервере (выполняются обычные HTTP запросы — PUT, GET, POST, DELETE). Хотя SOAP, в свою очередь, более надежен и безопасен.

Доп. материал:
REST v SOAP - A few perspectives
Что такое JSON, XML?
JSON (JavaScript Object Notation - обозначение объектов JavaScript) - текстовый формат обмена данными, основанный на JavaScript (но он не зависит от языка).
XML (eXtensible Markup Language — расширяемый язык разметки) - это язык разметки. Является выбором по умолчанию для обмена данными, остается легко читаемым, даже при больших массивах информации.
JSON благодаря популярности технологии API REST, получил импульс развития в программировании API и веб-сервисов. Это текстовый, легкий и простой в разборе формат данных, не требующий дополнительного кода для анализа. Таким образом, JSON помогает ускорить обмен данными и для веб-сервисов, которые должны просто возвращать много данных и отображать их.
Пример JSON:

```
{
        “title”:”bananas”,
        “count”:”1000”,
        “description”:[“500 green”, ”500 yellow”]
}
```

В python аналогичная структура данных – словари. То есть это просто набор ключ: значение. При этом ключ должен быть уникальным, значений может быть любое количество. Допускается вложенность (значением может быть другой json или список).

Пример XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<companies>
   <company>
      <company-id>12345</company-id>
      <name lang="ru">Абракадабра</name>
            <country lang="ru">Россия</country>
            <phone>
         <number>+7 (999) 999-99-99</number>
         <ext>777</ext>
         <info>Приемная</info>
         <type>phone</type>
      </phone>
   </company>
</companies>
```

Как видно, XML похож на HTML, однако здесь теги не предопределены.

Если на собеседовании по какой-то причине захотят углубленно проверить эту тему, то могут спросить про разницу между well-formed и valid XML, развить в валидацию и

XSD, через признаки Well-formed документа можно выйти на вложенность, а дальше на префиксы и namespace.

Доп. материал:
Что такое XML
JSONLint
XML Formatting • HYPERLINK "https://xmllint.com/"• &• HYPERLINK "https://xmllint.com/"• Validation Tool
Коды ответов/состояния сервера с примерами? (HTTP status codes)
Несколько из них могут спросить чуть конкретнее, чем просто название, обычно на Ваш же выбор.
Иногда на собеседовании можно услышать вопрос: «Что дают эти коды ответа и что с ними можно делать?». На него настолько обширный ответ, что в рамках данной статьи это было бы не уместно, но конкретно для тестировщика чаще всего это просто удобное понимание, как именно отреагировал сервер на web или API запрос.

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

Код ответа Название Описание
100 Continue "Продолжить". Этот промежуточный ответ указывает, что запрос успешно принят и клиент может продолжать присылать запросы либо проигнорировать этот ответ, если запрос был завершен.
101 Switching Protocol "Переключение протокола". Этот код присылается в ответ на запрос клиента, содержащий заголовок Upgrade, и указывает, что сервер переключился на протокол, который был указан в заголовке. Эта возможность позволяет перейти на несовместимую версию протокола и обычно не используется.
102 Processing "В обработке". Этот код указывает, что сервер получил запрос и обрабатывает его, но обработка еще не завершена.
103 Early Hints "Ранние подсказки". В ответе сообщаются ресурсы, которые могут быть загружены заранее, пока сервер будет подготавливать основной ответ.
200 OK "Успешно". Запрос успешно обработан. Что значит "успешно", зависит от метода HTTP, который был запрошен:
GET: "ПОЛУЧИТЬ". Запрошенный ресурс был найден и передан в теле ответа.
HEAD: "ЗАГОЛОВОК". Заголовки переданы в ответе.
POST: "ПОСЫЛКА". Ресурс, описывающий результат действия сервера на запрос, передан в теле ответа.
TRACE: "ОТСЛЕЖИВАТЬ". Тело ответа содержит тело запроса полученного сервером.
201 Created "Создано". Запрос успешно выполнен и в результате был создан ресурс. Этот код обычно присылается в ответ на запрос PUT "ПОМЕСТИТЬ".
202 Accepted "Принято". Запрос принят, но еще не обработан. Не поддерживаемо, т.е., нет способа с помощью HTTP отправить асинхронный ответ позже, который будет показывать итог обработки запроса. Это предназначено для

случаев, когда запрос обрабатывается другим процессом или сервером, либо для пакетной обработки.
203 Non-Authoritative Information "Информация не авторитетна". Этот код ответа означает, что информация, которая возвращена, была предоставлена не от исходного сервера, а из какого-нибудь другого источника. Во всех остальных ситуациях более предпочтителен код ответа 200 OK.
204 No Content "Нет содержимого". Нет содержимого для ответа на запрос, но заголовки ответа, которые могут быть полезны, присылаются. Клиент может использовать их для обновления кешированных заголовков полученных ранее для этого ресурса.
205 Reset Content "Сбросить содержимое". Этот код присылается, когда запрос обработан, чтобы сообщить клиенту, что необходимо сбросить отображение документа, который прислал этот запрос.
206 Partial Content "Частичное содержимое". Этот код ответа используется, когда клиент присылает заголовок диапазона, чтобы выполнить загрузку отдельно, в несколько потоков.
300 Multiple Choice "Множественный выбор". Этот код ответа присылается, когда запрос имеет более чем один из возможных ответов. И User-agent или пользователь должен выбрать один из ответов. Не существует стандартизированного способа выбора одного из полученных ответов.
301 Moved Permanently "Перемещен на постоянной основе". Этот код ответа значит, что URI запрашиваемого ресурса был изменен. Возможно, новый URI будет предоставлен в ответе.
302 Found "Найдено". Этот код ответа значит, что запрошенный ресурс временно изменен. Новые изменения в URI могут быть доступны в будущем. Таким образом, этот URI, должен быть использован клиентом в будущих запросах.
303 See Other "Просмотр других ресурсов". Этот код ответа присылается, чтобы направлять клиента для получения запрашиваемого ресурса в другой URI с запросом GET.
304 Not Modified "Не модифицировано". Используется для кэширования. Это код ответа значит, что запрошенный ресурс не был изменен. Таким образом, клиент может продолжать использовать кэшированную версию ответа.
305 Use Proxy "Использовать прокси". Это означает, что запрошенный ресурс должен быть доступен через прокси. Этот код ответа в основном не поддерживается из соображений безопасности.
306 Switch Proxy Больше не использовать. Изначально подразумевалось, что " последующие запросы должны использовать указанный прокси."
307 Temporary Redirect "Временное перенаправление". Сервер отправил этот ответ, чтобы клиент получил запрошенный ресурс на другой URL-адрес с тем же методом, который использовал предыдущий запрос. Данный код имеет ту же семантику, что код ответа 302 Found, за исключением того, что агент пользователя не должен изменять используемый метод HTTP: если в первом запросе использовался POST, то во втором запросе также должен использоваться POST.
308 Permanent Redirect "Перенаправление на постоянной основе". Это означает, что ресурс теперь постоянно находится в другом URI, указанном в заголовке Location: HTTP Response. Данный код ответа имеет ту же семантику, что и код ответа 301 Moved Permanently, за исключением того, что агент пользователя не должен изменять

используемый метод HTTP: если POST использовался в первом запросе, POST должен использоваться и во втором запросе.
Примечание: Это экспериментальный код ответа, Спецификация которого в настоящее время находится в черновом виде.
400 Bad Request "Плохой запрос". Этот ответ означает, что сервер не понимает запрос из-за неверного синтаксиса.
401 Unauthorized "Неавторизовано". Для получения запрашиваемого ответа нужна аутентификация. Статус похож на статус 403, но, в этом случае, аутентификация возможна.
402 Payment Required "Необходима оплата". Этот код ответа зарезервирован для будущего использования. Первоначальная цель для создания этого когда была в использовании его для цифровых платежных систем(на данный момент не используется).
403 Forbidden "Запрещено". У клиента нет прав доступа к содержимому, поэтому сервер отказывается дать надлежащий ответ.
404 Not Found "Не найден". Сервер не может найти запрашиваемый ресурс. Код этого ответа, наверно, самый известный из-за частоты его появления в вебе.
405 Method Not Allowed "Метод не разрешен". Сервер знает о запрашиваемом методе, но он был деактивирован и не может быть использован. Два обязательных метода, GET и HEAD, никогда не должны быть деактивированы и не должны возвращать этот код ошибки.
406 Not Acceptable Этот ответ отсылается, когда веб сервер после выполнения server-driven content negotiation, не нашел контента, отвечающего критериям, полученным из user agent.
407 Proxy Authentication Required Этот код ответа аналогичен коду 401, только аутентификация требуется для прокси сервера.
408 Request Timeout Ответ с таким кодом может прийти, даже без предшествующего запроса. Он означает, что сервер хотел бы отключить это неиспользуемое соединение. Этот метод используется все чаще с тех пор, как некоторые браузеры, вроде Chrome и IE9, стали использовать HTTP механизмы предварительного соединения для ускорения серфинга (смотрите баг 634278, будущей реализации этого механизма в Firefox). Также учитывайте, что некоторые серверы прерывают соединения не отправляя подобных сообщений.
409 Conflict Этот ответ отсылается, когда запрос конфликтует с текущим состоянием сервера.
410 Gone Этот ответ отсылается, когда запрашиваемый контент удален с сервера.
411 Length Required Запрос отклонен, потому что сервер требует указание заголовка Content-Length, но он не указан.
412 Precondition Failed Клиент указал в своих заголовках условия, которые сервер не может выполнить
413 Request Entity Too Large Размер запроса превышает лимит, объявленный сервером. Сервер может закрыть соединение, вернув заголовок Retry-After
414 Request-URI Too Long URI запрашиваемый клиентом слишком длинный для того, чтобы сервер смог его обработать
415 Unsupported Media Type Медиа формат запрашиваемых данных не поддерживается сервером, поэтому запрос отклонен
416 Requested Range Not Satisfiable Диапазон указанный заголовком запроса Range не может быть выполнен; возможно, он выходит за пределы переданного URI

417 Expectation Failed Этот код ответа означает, что ожидание, полученное из заголовка запроса Expect, не может быть выполнено сервером.
500 Internal Server Error "Внутренняя ошибка сервера". Сервер столкнулся с ситуацией, которую он не знает, как обработать.
501 Not Implemented "Не выполнено". Метод запроса не поддерживается сервером и не может быть обработан. Единственные методы, которые сервера должны поддерживать (и, соответственно, не должны возвращать этот код) - GET и HEAD.
502 Bad Gateway "Плохой шлюз". Эта ошибка означает что сервер, во время работы в качестве шлюза для получения ответа, нужного для обработки запроса, получил недействительный (недопустимый) ответ.
503 Service Unavailable "Сервис недоступен". Сервер не готов обрабатывать запрос. Зачастую причинами являются отключение сервера или то, что он перегружен. Обратите внимание, что вместе с этим ответом удобная для пользователей(user-friendly) страница должна отправлять объяснение проблемы. Этот ответ должен использоваться для временных условий и Retry-After: HTTP-заголовок должен, если возможно, содержать предполагаемое время до восстановления сервиса. Веб-мастер также должен позаботиться о заголовках, связанных с кэшем, которые отправляются вместе с этим ответом, так как эти ответы, связанные с временными условиями, обычно не должны кэшироваться.
504 Gateway Timeout Этот ответ об ошибке предоставляется, когда сервер действует как шлюз и не может получить ответ вовремя.
505 HTTP Version Not Supported "HTTP-версия не поддерживается". HTTP-версия, используемая в запросе, не поддерживается сервером.
Почему ошибка 404 относится к 4** - клиентской, если по идее должна быть 5**?
Хотя интуитивно можно подумать, что данная ошибка должна относиться к ошибкам со стороны сервера, 404 по задумке является клиентской ошибкой, то есть подразумевается, что клиент (Вы) должен был знать, что URL страницы был перемещен или удален и Вы пытаетесь открыть несуществующую страницу.
Какие еще бывают протоколы?
FTP (File Transfer Protocol) — это протокол передачи файлов со специального файлового сервера на компьютер пользователя. FTP дает возможность абоненту обмениваться двоичными и текстовыми файлами с любым компьютером сети. Установив связь с удаленным компьютером, пользователь может скопировать файл с удаленного компьютера на свой или скопировать файл со своего компьютера на удаленный.
POP• 3 (Post Office Protocol) — это стандартный протокол почтового соединения. Серверы POP обрабатывают входящую почту, а протокол POP предназначен для обработки запросов на получение почты от клиентских почтовых программ.
SMTP (Simple Mail Transfer Protocol) — протокол, который задает набор правил для передачи почты. Сервер SMTP возвращает либо подтверждение о приеме, либо сообщение об ошибке, либо запрашивает дополнительную информацию.
TELNET — это протокол удаленного доступа. TELNET дает возможность абоненту работать на любой ЭВМ находящейся с ним в одной сети, как на своей собственной, то есть запускать программы, менять режим работы и так далее. На практике возможности ограничиваются тем уровнем доступа, который задан администратором удаленной машины.
- TCP — сетевой протокол, отвечающий за передачу данных в сети Интернет.

- UDP - это тоже транспортный протокол передачи данных, но без подтверждения доставки
- Ethernet — протокол, определяющий стандарты сети на физическом и канальном уровнях.

TCP/IP это?

TCP/IP — сетевая модель передачи данных, представленных в цифровом виде. Модель описывает способ передачи данных от источника информации к получателю. В модели предполагается прохождение информации через четыре уровня, каждый из которых описывается правилом (протоколом передачи). Наборы правил, решающих задачу по передаче данных, составляют стек протоколов передачи данных, на которых базируется Интернет.

Набор интернет-протоколов — это концептуальная модель и набор коммуникационных протоколов, используемых в Интернете и подобных компьютерных сетях. Он широко известен как TCP/IP, поскольку базовые протоколы в пакете — это протокол управления передачей (TCP) и интернет-протокол (IP).

Набор интернет-протоколов обеспечивает сквозную передачу данных, определяющую, как данные должны пакетироваться, обрабатываться, передаваться, маршрутизироваться и приниматься. Эта функциональность организована в четыре слоя абстракции, которые классифицируют все связанные протоколы в соответствии с объемом задействованных сетей.

От самого низкого до самого высокого уровня:

- уровень связи, содержащий методы связи для данных, которые остаются в пределах одного сегмента сети;
- интернет-уровень, обеспечивающий межсетевое взаимодействие между независимыми сетями;
- транспортный уровень, обрабатывающий связь между хостами;
- прикладной уровень, который обеспечивает обмен данными между процессами для приложений.

Доп. материал:

TCP/IP: что это и зачем это тестировщику

Что такое куки (cookies)? Как их тестировать?

Файл cookie HTTP (файл cookie Интернета, файл cookie браузера) представляет собой небольшой фрагмент данных (часть http заголовка), который веб-сервер хранит в текстовом файле на жестком диске пользователя (клиента). Эта часть информации затем отправляется обратно на сервер каждый раз, когда браузер запрашивает страницу с сервера. Обычно cookie-файлы содержат персонализированные пользовательские данные или информацию, которые используются для определения того, поступили ли два запроса от одного и того же браузера - например, для входа пользователя в систему или для связи между различными веб-страницами. Он запоминает информацию stateful для stateless протокола HTTP.

Куки в основном используются для трех целей:

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

Куки состоят в основном из трех вещей:

- Имя сервера, с которого был отправлен куки

- Время жизни (Cookies Lifetime)
- Случайно сгенерированный уникальный номер

Максимальный размер кук = 4 килобайт (4096 байт), в некоторых источниках 4093 байт
Виды кук:

- Сессионные cookie, также известные как временные cookie, существуют только во временной памяти, пока пользователь находится на странице веб-сайта. Браузеры обычно удаляют сессионные cookie после того, как пользователь закрывает окно браузер. В отличие от других типов cookie, сессионные cookie не имеют истечения срока действия, и поэтому браузеры понимают их как сессионные.
- Вместо того, чтобы удаляться после закрытия браузера, как это делают сессионные cookie, постоянные cookie-файлы удаляются в определенную дату или через определенный промежуток времени. Это означает, что информация о cookie будет передаваться на сервер каждый раз, когда пользователь посещает веб-сайт, которому эти cookie принадлежат. По этой причине постоянные cookie иногда называются следящие cookie, поскольку они могут использоваться рекламодателями для записи о предпочтениях пользователя в течение длительного периода времени. Однако, они также могут использоваться и в «мирных» целях, например, чтобы избежать повторного ввода данных при каждом посещении сайта.
- Обычно атрибут домена cookie совпадает с доменом, который отображается в адресной строке веб-браузера. Это называется первый файл cookie. Однако сторонний файл cookie принадлежит домену, отличному от того, который указан в адресной строке. Этот тип файлов cookie обычно появляется, когда веб-страницы содержат контент с внешних веб-сайтов, например, рекламные баннеры. Это открывает возможности для отслеживания истории посещений пользователя и часто используется рекламодателями для предоставления релевантной рекламы каждому пользователю.
- Супер-cookie — это cookie-файл с источником домена верхнего уровня (например, .ru) или общедоступным суффиксом (например, .co.uk). Обычные cookie, напротив, имеют происхождение от конкретного доменного имени, например, example.com. Супер-cookie могут быть потенциальной проблемой безопасности и поэтому часто блокируются веб-браузерами. Если браузер разблокирует вредоносный веб-сайт, злоумышленник может установить супер-cookie и потенциально нарушить или выдать себя за законные запросы пользователей на другой веб-сайт, который использует тот же домен верхнего уровня или общедоступный суффикс, что и вредоносный веб-сайт. Например, супер-cookie с происхождением .com может злонамеренно повлиять на запрос к example.com, даже если файл cookie не был создан с сайта example.com. Это может быть использовано для подделки логинов или изменения информации пользователя.
- Поскольку cookie можно очень легко удалить из браузера, программисты ищут способы идентифицировать пользователей даже после полной очистки истории браузера. Одним из таких решений являются зомби-cookie (или evercookie, или persistent cookie) — не удаляемые или трудно удаляемые cookie, которые можно восстановить в браузере с помощью JavaScript. Это возможно потому, что для хранения куков сайт одновременно использует все доступные хранилища браузера (HTTP ETag, Session Storage, Local Storage, Indexed DB), в том числе и хранилища приложений, таких как Flash Player (Local Shared Objects), Microsoft Silverlight (Isolated Storage) и Java (Java persistence API). Когда программа обнаруживает отсутствие в браузере cookie-файла, информация о котором присутствует в других хранилищах —

она тут же восстанавливает его на место и, тем самым, идентифицирует пользователя для сайта.
Примеры Test case для Cookie testing:

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

Доп. материал: Что такое файлы cookie и как их тестировать
Что такое Web Storage?
Почти всем настольным и мобильным приложениям нужно где-то хранить пользовательские данные. Но как быть веб-сайтам? В прошлом, мы использовали для этой цели файлы cookie, но у них есть серьезные ограничения. HTML5 предоставляет более подходящие инструменты для решения этой проблемы. Первый инструмент – это IndexedDB, который является излишним, говоря о замене cookie, а второй – Web Storage, являющееся комбинацией двух очень простых интерфейсов API.
Интернет-хранилище или DOM-хранилище — это программные методы и протоколы веб-приложения, используемые для хранения данных в веб-браузере.
Интернет-хранилище представляет собой постоянное хранилище данных, похожее на куки, но со значительно расширенной емкостью и без хранения информации в заголовке запроса HTTP. Существуют два основных типа веб-хранилища: локальное хранилище (localStorage) и сессионное хранилище (sessionStorage), ведущие себя аналогично постоянным и сессионным кукам соответственно.

Доп. материал:
Client-side storage
HTML• 5 • Web Storage - • обзор веб-хранилища
В чем отличие статических и динамических веб-сайтов?
Статические сайты состоят из неизменяемых страниц. Это значит, что сайт имеет один и тот же внешний вид, а также одно и то же наполнение для всех посетителей. При запросе такого сайта в браузере сервер сразу предоставляет готовый HTML-документ

в исходном виде, в котором он и был создан. Кроме HTML, в коде таких страниц используется разве что CSS и JavaScript, что обеспечивает их легкость и быструю загрузку.
Чаще всего статическими бывают сайты с минимальным количеством страниц или с контентом, который не нужно регулярно обновлять, а именно сайты-визитки, каталоги продукции, справочники технической документации. Однако с помощью сторонних инструментов существует возможность добавить на такие страницы отдельные динамические элементы (комментарии, личный кабинет для пользователей, поиск).
Динамические сайты, в свою очередь, имеют изменяемые страницы, адаптирующиеся под конкретного пользователя. Такие страницы не размещены на сервере в готовом виде, а собираются заново по каждому новому запросу. Сначала сервер находит нужный документ и отправляет его интерпретатору, который выполняет код из HTML-документа и сверяется с файлами и базой данных. После этого документ возвращается на сервер и затем отображается в браузере. Для интерпретации страниц на серверной стороне используются языки программирования Java, PHP, ASP и другие.
Самыми яркими примерами динамических сайтов являются интернет-магазины, социальные сети и т.п.
Источники:
- https://yandex.ru/turbo/jino.ru/s/journal/articles/staticheskie-dinamicheskie-sayty/
- https://wp-system.ru/sozdanie-sayta/staticheskie-i-dinamicheskie-sajty/

Отличие stateless и stateful?
stateful — модель, при которой объект содержит информацию о своем состоянии, все методы работают в контексте его состояния
stateless не предоставляют эту информацию. Все методы объекта работают вне какого-либо контекста или локального состояния объекта, которого в этом случае просто нет. Не делается предположений о состоянии сессии, все изменения атомарны, нет каких-то сессионных переменных на сервере, помнящих результат предыдущего запроса. Они каждый раз дают один и тот же неизменный ответ на один и тот же запрос, функцию или вызов метода. HTTP не имеет состояния в необработанном виде - если вы выполняете GET для определенного URL, вы получаете (теоретически) один и тот же ответ каждый раз.
Различия методов GET и POST?
Основное состоит в способе передачи данных веб-формы обрабатывающему скрипту, а именно:
- Метод GET отправляет скрипту всю собранную информацию формы как часть URL: • http://www.komtet.ru/script.php?login=admin• &• name=komtet
- Метод POST передает данные таким образом, что пользователь сайта уже не видит передаваемые скрипту данные: http://www.komtet.ru/script.php

Оба метода успешно передают необходимую информацию из веб-формы скрипту, поэтому при выборе того или иного метода, который будет наиболее подходить сайту, нужно учитывать следующие факторы:
- Принцип работы метода GET ограничивает объем передаваемой скрипту информации;
- Так как метод GET отправляет скрипту всю собранную информацию формы как часть URL (то есть в открытом виде), то это может пагубно повлиять на безопасность сайта;

- Страницу, сгенерированную методом GET, можно пометить закладкой (адрес страницы будет всегда уникальный), а страницу, сгенерированную метод POST нельзя (адрес страницы остается неизменным, так как данные в URL не подставляются);
- Используя метод GET можно передавать данные не через веб-форму, а через URL страницы, введя необходимые значения через знак &: • http://www.komtet.ru/script.php?login=admin• &• name=komtet
- Метод POST в отличие от метода GET позволяет передавать запросу файлы;
- При использовании метода GET существует риск того, что поисковый робот может выполнить тот или иной открытый запрос.

Клиент - серверная архитектура?

- Сервер – логический процесс, который обеспечивает некоторый сервис по запросу от клиента. Обычно сервер не только выполняет запрос, но и управляет очередностью запросов, буферами обмена, извещает своих клиентов о выполнении запроса и т. д.
- Клиент – процесс, который запрашивает обслуживание от сервера. Процесс не является клиентом по каким-то параметрам своей структуры, он является клиентом только по отношению к серверу.
- Сеть, протоколы – третий компонент, который обеспечивает обмен информацией между клиентом и сервером.

При взаимодействии клиента и сервера инициатором диалога с сервером, как правило, является клиент, сервер сам не инициирует совместную работу.

Технология клиент-сервер - шаблон проектирования, основа для создания веб-приложений, взаимодействие, при котором одна программа (клиент) запрашивает услугу (выполнение какой либо совокупности действий), а другая программа (сервер) ее выполняет.

Двухзвенная архитектура клиент-сервер:

- «Толстый клиент»: на сервере реализованы главным образом функции доступа к базам данных, а основные прикладные вычисления выполняются на стороне клиента.
- «Тонкий клиент»: на сервере выполняется основная часть прикладной обработки данных, а на клиентские рабочие станции передаются уже результаты обработки данных для просмотра и анализа пользователем с возможностью их последующей обработки (в минимальном объеме).

Многоуровневая архитектура «клиент-сервер»:
Разновидность архитектуры клиент-сервер, в которой функция обработки данных вынесена на один или несколько отдельных серверов. Это позволяет разделить функции хранения, обработки и представления данных для более эффективного использования возможностей серверов и клиентов.

Уровни OSI?
Модель OSI

Уровень (layer)	Тип данных (PDU)	Функции	Примеры
Host			

layers 7. Прикладной (application) Данные Доступ к сетевым службам HTTP, FTP, POP3, WebSocket
6. Представления (presentation) Представление и шифрование данных ASCII, EBCDIC
5. Сеансовый (session) Управление сеансом связи RPC, PAP, L2TP
4. Транспортный (transport) Сегменты (segment) /Дейтаграммы (datagram) Прямая связь между конечными пунктами и надежность TCP, UDP, SCTP, PORTS
Media layers 3. Сетевой (network) Пакеты (packet) Определение маршрута и логическая адресация IPv4, IPv6, IPsec, AppleTalk
2. Канальный (data link) Биты (bit)/ Кадры (frame)Физическая адресация PPP, IEEE 802.22, Ethernet, DSL, ARP, сетевая карта.
1. Физический (physical) Биты (bit) Работа со средой передачи, сигналами и двоичными данными USB, кабель («витая пара», коаксиальный, оптоволоконный), радиоканал

Вместо OSI в реальности актуальнее знать TCP/IP. Тестировщики и разработчики почти всегда работают на прикладном уровне. Ниже может быть только тестирование VoIP и т.п.
Что вы подразумеваете под потоковыми медиа? (Streaming media)
Потоковая передача - это процесс загрузки данных с сервера.
Потоковое мультимедиа - мультимедиа, которое непрерывно получает пользователь от провайдера потокового вещания. Это понятие применимо как к информации, распространяемой через телекоммуникации, так и к информации, которая изначально распространялась посредством потокового вещания (например, радио, телевидение)
Основные команды Linux?

- pwd - когда вы впервые открываете терминал, вы попадаете в домашний каталог вашего пользователя. Чтобы узнать, в каком каталоге вы находитесь, вы можете использовать команду «pwd». Это команда выводит полный путь от корневого каталога к текущему рабочему каталогу: в контексте которого (по умолчанию) будут исполняться вводимые команды. Корень является основой файловой системы Linux. Обозначается косой чертой (/). Каталог пользователя обычно выглядит как "/ home / username".
- ls - используйте команду "ls", чтобы узнать, какие файлы находятся в каталоге, в котором вы находитесь. Вы можете увидеть все скрытые файлы, используя команду "ls -a".
- cd - используйте команду "cd", чтобы перейти в каталог. Например, если вы находитесь в домашней папке и хотите перейти в папку загрузок, вы можете ввести «cd Downloads». Помните, что эта команда чувствительна к регистру, и вы должны ввести имя папки в точности так, как оно есть. Но есть один нюанс. Представьте, что у вас есть папка с именем «Raspberry Pi». В этом случае, когда вы вводите «cd Raspberry Pi», оболочка примет второй аргумент команды как другой, поэтому вы получите сообщение об ошибке, говорящее о том, что каталог не существует. Здесь вы можете использовать обратную косую черту, то есть: «cd Raspberry/ Pi». Пробелы работают так: если вы просто наберете «cd» и нажмете клавишу ввода, вы попадете в домашний

каталог. Чтобы вернуться из папки в папку до этого, вы можете набрать «cd ..». Две точки возвращают в предыдущий каталог.

- mkdir и rmdir - используйте команду mkdir, когда вам нужно создать папку или каталог. Например, если вы хотите создать каталог под названием «DIY», вы можете ввести «mkdir DIY». Помните, как уже было сказано, если вы хотите создать каталог с именем «DIY Hacking», вы можете ввести «mkdir DIY/ Hacking». Используйте rmdir для удаления каталога. Но rmdir можно использовать только для удаления пустой директории. Чтобы удалить каталог, содержащий файлы, используйте команду rm.
- rm - используйте команду rm для удаления файлов и каталогов. Используйте «rm -r», чтобы удалить только каталог. Он удаляет как папку, так и содержащиеся в ней файлы при использовании только команды rm.
- touch - команда touch используется для создания файла. Это может быть что угодно, от пустого txt-файла до пустого zip-файла. Например, «touch new.txt».
- man и --help - Чтобы узнать больше о команде и о том, как ее использовать, используйте команду man. Показывает справочные страницы команды. Например, «man ls» показывает справочные страницы команды ls. Ввод имени команды и аргумента помогает показать, каким образом можно использовать команду (например, cd --help).
- cp - используйте команду cp для копирования файлов через командную строку. Он принимает два аргумента: первый - это местоположение файла, который нужно скопировать, второй - куда копировать.
- mv - используйте команду mv для перемещения файлов через командную строку. Мы также можем использовать команду mv для переименования файла. Например, если мы хотим переименовать файл «text» в «new», мы можем использовать «mv text new». Он принимает два аргумента, как и команда cp.
- locate - команда locate используется для поиска файла в системе Linux, так же, как команда поиска в Windows. Эта команда полезна, когда вы не знаете, где файл сохранен или фактическое имя файла. Использование аргумента -i с командой помогает игнорировать регистр (не имеет значения, является ли он прописным или строчным). Итак, если вам нужен файл со словом «hello», он дает список всех файлов в вашей системе Linux, содержащих слово «hello», когда вы вводите «locate -i hello». Если вы помните два слова, вы можете разделить их звездочкой (*). Например, чтобы найти файл, содержащий слова «hello» и «this», вы можете использовать команду «locate -i * hello * this».

Промежуточные команды:

- echo - команда "echo" помогает нам перемещать некоторые данные, обычно текст, в файл. Например, если вы хотите создать новый текстовый файл или добавить в уже созданный текстовый файл, вам просто нужно ввести «echo hello, меня зовут hich >> new.txt». Вам не нужно разделять пробелы с помощью обратной косой черты здесь, потому что мы заключаем в две треугольные скобки, когда мы заканчиваем то, что нам нужно написать.
- cat - Используйте команду cat для отображения содержимого файла. Обычно используется для удобного просмотра программ.
- nano, vi, jed - nano и vi уже установлены текстовые редакторы в командной строке Linux. Команда nano - хороший текстовый редактор, который помечает ключевые слова цветом и может распознавать большинство языков. И vi проще, чем nano. Вы можете создать новый файл или изменить файл с помощью этого редактора.

Например, если вам нужно создать новый файл с именем «check.txt», вы можете создать его с помощью команды «nano check.txt». Вы можете сохранить ваши файлы после редактирования, используя последовательность Ctrl + X, затем Y (или N для no). По моему опыту, использование nano для редактирования HTML выглядит не очень хорошо из-за его цвета, поэтому я рекомендую jed текстовый редактор. Мы скоро приступим к установке пакетов.

- sudo - широко используемая команда в командной строке Linux, sudo означает «SuperUser Do». Поэтому, если вы хотите, чтобы любая команда выполнялась с правами администратора или root, вы можете использовать команду sudo. Например, если вы хотите отредактировать файл, такой как viz. alsa-base.conf, для которого требуются права root, вы можете использовать команду - sudo nano alsa-base.conf. Вы можете ввести корневую командную строку с помощью команды «sudo bash», а затем ввести свой пароль пользователя. Вы также можете использовать команду «su», но перед этим вам нужно установить пароль root. Для этого вы можете использовать команду «sudo passwd» (не с орфографической ошибкой, это passwd). Затем введите новый пароль root.
- df - используйте команду df, чтобы увидеть доступное дисковое пространство в каждом из разделов вашей системы. Вы можете просто ввести df в командной строке и увидеть каждый смонтированный раздел и его использованное / доступное пространство в % и в килобайтах. Если вы хотите, чтобы оно отображалось в мегабайтах, вы можете использовать команду «df -m».
- du - Используйте du, чтобы узнать, как файл используется в вашей системе. Если вы хотите узнать размер занимаемого места на диске для конкретной папки или файла в Linux, вы можете ввести команду df и имя папки или файла. Например, если вы хотите узнать размер дискового пространства, используемое папкой документов в Linux, вы можете использовать команду «du Documents». Вы также можете использовать команду «ls -lah», чтобы просмотреть размеры всех файлов в папке.
- tar - Используйте tar для работы с tarballs (или файлами, сжатыми в архиве tarball) в командной строке Linux. У него длинный список применений. Он может использоваться для сжатия и распаковки различных типов архивов tar, таких как .tar, .tar.gz, .tar.bz2 и т. д. Это работает на основе аргументов, данных ему. К примеру, "tar -cvf" для создания .tar архива, -xvf для распаковки .tar архива, -tvf для просмотра содержимого архива и т. д.
- zip, unzip - используйте zip для сжатия файлов в zip-архив и unzip для извлечения файлов из zip-архива.
- uname - используйте uname, чтобы показать информацию о системе, в которой работает ваш дистрибутив Linux. Использование команды «uname -a» выводит большую часть информации о системе: дату выпуска ядра, версию, тип процессора и т. д.
- apt-get - используйте apt для работы с пакетами в командной строке Linux. Используйте apt-get для установки пакетов. Это команда требует прав суперпользователя, поэтому используйте команду sudo с ним. Например, если вы хотите установить текстовый редактор jed (как я упоминал ранее), мы можем ввести команду «sudo apt-get install jed». Точно так же любые пакеты могут быть установлены следующим образом. Рекомендуется обновлять ваш репозиторий каждый раз, когда вы пытаетесь установить новый пакет. Вы можете сделать это, набрав «sudo apt-get update». Вы можете обновить систему, набрав «sudo apt-get upgrade». Мы также можем обновить дистрибутив, набрав «sudo apt-get dist-upgrade». Команда «apt-cache search»

используется для поиска пакета. Если вы хотите найти его, вы можете ввести «apt-cache search jed» (для этого не требуется root).

- chmod - используйте chmod, чтобы сделать файл исполняемым и изменить разрешения, предоставленные ему в Linux. Представьте, что на вашем компьютере есть код Python с именем numbers.py. Вам нужно будет запускать «python numbers.py» каждый раз, когда вам нужно его запустить. Вместо этого, когда вы делаете его исполняемым, вам просто нужно запустить «numbers.py» в терминале, чтобы запустить файл. Чтобы сделать файл исполняемым, вы можете использовать команду «chmod + x numbers.py» в этом случае. Вы можете использовать «chmod 755 numbers.py», чтобы дать ему права root, или «sudo chmod + x numbers.py» для исполняемого файла root.
- hostname - Используйте команду hostname, чтобы узнать ваше имя в вашем хосте или сети. По сути, он отображает ваше имя хоста и IP-адрес. Просто набрав «hostname», вы получите имя хоста. Набрав «hostname -I», вы получите свой IP-адрес в сети.
- ping - используйте ping для проверки вашего соединения с сервером. Википедия говорит: «Ping - это утилита для администрирования компьютерной сети, используемая для проверки доступности хоста в сети Интернет-протокола (IP)». Например, когда вы набираете, , «ping • google.com», он проверяет, может ли он подключиться к серверу и вернуться обратно. Он измеряет это время в оба конца и дает вам подробную информацию о нем. Использовать эту команду можно и для проверки интернет-соединения. Если он пингует сервер Google (в данном случае) - интернет-соединение активно!

Доп. материал:
Linux Command Line Cheat Sheet by DaveChild
Почему важно тестировать в разных браузерах?
Приложения и сайты в разных браузерах могут вести себя по-разному. Это связано с тем, что любой из браузеров имеет собственные движки, надстройки, плагины, а также различия в десктопной и мобильной версиях. Кроссбраузерное тестирование призвано сгладить эти различия, сделав разработку более или менее универсальной.
Почему возникают кросс-браузерные ошибки:

- Иногда сами браузеры содержат баги или по-разному внедряют функции. Часто это происходит из-за попытки заполучить конкурентное преимущество.
- Некоторые браузеры могут иметь разные уровни поддержки технологических функций для других браузеров. JavaScript фичи, скорее всего, не будут работать на старых браузерах.
- Некоторые девайсы могут содержать ограничения, которые могут заставить веб-сайт работать медленнее или некорректно отображаться. Например, если сайт был спроектирован под десктоп, то вполне вероятно, что его контент будет сложно читать на мобильном устройстве.
- Приступать к тестированию сайта в популярных браузерах следует уже после того как он проверен на дефекты другими видами тестирования. Только в этом случае можно будет сказать, что выявленные некорректные сценарии имеют отношение именно к особенностям браузера, а не были пропущены на других стадиях. Разумеется, при этом ошибка должна проявляться не во всех браузерах. Внимание нужно также уделить сочетанию операционной системы и браузера, выбрав наиболее распространенные из них.

Адаптивный веб-дизайн vs. Отзывчивый веб-дизайн, в чем разница? (Adaptive Vs. Responsive)

Responsive Design (RWD) — отзывчивый дизайн — проектирование сайта с определенными значениями свойств, например, гибкая сетка макета, которые позволяют одному макету работать на разных устройствах;
Помимо своей изменяющейся структуры, у респонсив дизайна есть несколько других преимуществ:
1. Одинаковый внешний вид ресурса в разных браузерах и на различных платформах
2. Наличие у сайта одинакового URL, что способствует SEO-оптимизации
3. Разработчикам необходимо обслуживать лишь один сайт, что позволяет сократить время, затрачиваемое на дизайн и контент
И хотя положительные стороны респонсивного дизайна очевидны, у этого метода существует ряд недостатков. Самым большим из них является скорость загрузки, которая значительно снижается из-за высокого разрешения изображений и других визуальных элементов, необходимых для оформления внешнего вида ресурса.

Adaptive Design (AWD) — адаптивный дизайн, или динамический показ — проектирование сайта с условиями, которые изменяются в зависимости от устройства, базируясь на нескольких макетах фиксированной ширины.
Для создания отзывчивых макетов используются медиазапросы и относительные размеры элементов сетки, заданные с помощью %. В адаптивном дизайне серверные скрипты сначала определяют тип устройства, с помощью которого пользователь пытается получить доступ к сайту (настольный ПК, телефон или планшет), затем загружает именно ту версию страницы, которая наиболее оптимизирована для него. Для элементов сетки задаются фиксированные в пикселях (px) размеры.
Поэтому основное отличие между этими приемами — отзывчивый дизайн — один макет для всех устройств, адаптивный дизайн — один макет для каждого вида устройства. Иными словами, сервер берет на себя всю «тяжелую» работу, вместо того, чтобы заставлять сайт оптимизировать самого себя. Среди достоинств адаптивного дизайна можно выделить следующие:

- Изображения загружаются намного быстрее, так как они сжимаются и адаптируются под устройство пользователя
- Загрузка сайта происходит быстрее, так как сервер определяет тип устройства пользователя и загружает соответствующий ему программный код
- Разработчики пользуются свободой творчества, ведь они могут создавать различные версии сайтов и подгонять их под соответствующие типы устройств, чтобы сделать их более удобными для мобильных пользователей.

Привлекательность этого метода омрачается тем, что создать адаптивный сайт не так-то просто. Из-за адаптации дизайна к различным устройствам, время, затрачиваемое на разработку, значительно увеличивается. Более того, если вам потребуется сделать какие-либо доработки на сайте, придется вносить изменения во все его версии.
Как сервер узнает, с какого типа устройства/браузера/ОС/языка вы открываете веб-сайт? (Например, для Adaptive design)
Когда вы отправляете HTTP-запрос, он содержит в себе заголовки (headers) с различной информацией. Одним из них является User-Agent. Он сообщает: браузер,

его версию и язык, движок браузера, версию движка, операционную систему. Данные могут быть написаны как угодно, однако примерный формат таков:
Браузер/Версия (Платформа; Шифрование; Система, Язык[; Что-нибудь еще]) [Дополнения].

Как еще можно определить, если не из хедеров? Определить версию и тип браузера можно при помощи JavaScript.
Чем отличается авторизация от аутентификации?
Аутентификация Авторизация
Процедура проверки подлинности субъекта Процедура присвоения и проверки прав на совершение определенных действий субъектом
Зависит от предоставляемой пользователем информации Не зависит от действий клиента
Запускается один раз для текущей сессии Происходит при попытке совершения любых действий пользователем

Как работает авторизация/аутентификация? Как сайт понимает, что ты залогинен? Классический вариант – регистрация по логину/почте и паролю. При входе и введении правильных данных, если данные совпадают с таковыми в базе, вы получаете доступ на сайт.

1. Т.к. протокол HTTP не отслеживает состояния, нельзя достоверно знать, что человек, залогинившийся до этого по почте и паролю остается тем же человеком. И тогда изобрели аутентификацию на основе сессий/кук, на основе которых реализовано отслеживание состояний (stateful), то есть аутентификационная запись или сессия хранятся как на сервере, так и на клиенте. Сервер отслеживает открытые сессии в базе данных или в оперативной памяти, в свою очередь на фронтенде создаются cookies, в которых хранится идентификатор сессии. Процедура аутентификации на основе сессий:
- Пользователь вводит в браузере свое имя и пароль, после чего клиентское приложение отправляет на сервер запрос.
- Сервер проверяет пользователя, аутентифицирует его, шлет приложению уникальный пользовательский токен (сохранив его в памяти или базе данных).
- Клиентское приложение сохраняет токены в куках и отправляет их при каждом последующем запросе.
- Сервер получает каждый запрос, требующий аутентификации, с помощью токена аутентифицирует пользователя и возвращает запрошенные данные клиентскому приложению.
- Когда пользователь выходит, клиентское приложение удаляет его токен, поэтому все последующие запросы от этого клиента становятся неаутентифицированными.

У этого метода есть и недостатки:
- При каждой аутентификации пользователя сервер должен создавать у себя запись. Обычно она хранится в памяти, и при большом количестве пользователей есть вероятность слишком высокой нагрузки на сервер.
- Поскольку сессии хранятся в памяти, масштабировать не так просто. Если вы многократно реплицируете сервер, то на все новые серверы придется реплицировать и все пользовательские сессии. Это усложняет масштабирование.

2. Аутентификация на основе токенов в последние годы стала очень популярна из-за распространения одностраничных приложений (SPA), веб-API и интернета вещей. Чаще всего в качестве токенов используются Json Web Tokens (JWT). Хотя реализации бывают разные, но токены JWT превратились в стандарт де-факто.
При аутентификации на основе токенов состояния не отслеживаются (stateless). Мы не будем хранить информацию о пользователе на сервере или в сессии и даже не будем хранить JWT, использованные для клиентов.
Процедура аутентификации на основе токенов:

- Пользователь вводит имя и пароль.
- Сервер проверяет их и возвращает токен (JWT), который может содержать метаданные вроде user_id, разрешений и т. д.
- Токен хранится на клиентской стороне, чаще всего в локальном хранилище, но может лежать и в хранилище сессий или кук.
- Последующие запросы к серверу обычно содержат этот токен в качестве дополнительного заголовка авторизации в виде Bearer {JWT}. Еще токен может пересылаться в теле POST-запроса и даже как параметр запроса.
- Сервер расшифровывает JWT, если токен верный, сервер обрабатывает запрос.
- Когда пользователь выходит из системы, токен на клиентской стороне уничтожается, с сервером взаимодействовать не нужно.

У метода есть ряд преимуществ:
Главное преимущество: поскольку метод никак не оперирует состояниями, серверу не нужно хранить записи с пользовательскими токенами или сессиями. Каждый токен самодостаточен, содержит все необходимые для проверки данные, а также передает затребованную пользовательскую информацию. Поэтому токены не усложняют масштабирование.
В куках вы просто храните ID пользовательских сессий, а JWT позволяет хранить метаданные любого типа, если это корректный JSON.
При использовании кук бэкенд должен выполнять поиск по традиционной SQL-базе или NoSQL-альтернативе, и обмен данными наверняка длится дольше, чем расшифровка токена. Кроме того, раз вы можете хранить внутри JWT дополнительные данные вроде пользовательских разрешений, то можете сэкономить и дополнительные обращения поисковые запросы на получение и обработку данных.
Допустим, у вас есть API-ресурс /api/orders, который возвращает последние созданные приложением заказы, но просматривать их могут только пользователи категории админов. Если вы используете куки, то, сделав запрос, вы генерируете одно обращение к базе данных для проверки сессии, еще одно обращение — для получения пользовательских данных и проверки, относится ли пользователь к админам, и третье обращение — для получения данных.
А если вы применяете JWT, то можете хранить пользовательскую категорию уже в токене. Когда сервер запросит его и расшифрует, вы можете сделать одно обращение к базе данных, чтобы получить нужные заказы.
У использования кук на мобильных платформах есть много ограничений и особенностей. А токены сильно проще реализовать на iOS и Android. К тому же токены проще реализовать для приложений и сервисов интернета вещей, в которых не предусмотрено хранение кук.

Благодаря всему этому аутентификация на основе токенов сегодня набирает популярность.
Примечание: в целях безопасности в некоторых случаях в дополнение к токену применяется сравнение user agent и подобного. В случае различия вас разлогинит. Так же, например, в банковских системах нельзя одновременно залогиниться в одну учетную запись с нескольких устройств одновременно.

3. Беспарольная аутентификация
Первой реакцией на термин «беспарольная аутентификация» может быть «Как аутентифицировать кого-то без пароля? Разве такое возможно?»
В наши головы внедрено убеждение, что пароли — абсолютный источник защиты наших аккаунтов. Но если изучить вопрос глубже, то выяснится, что беспарольная аутентификация может быть не просто безопасной, но и безопаснее традиционного входа по имени и паролю. Возможно, вы даже слышали мнение, что пароли устарели.
Беспарольная аутентификация — это способ конфигурирования процедуры входа и аутентификации пользователей без ввода паролей. Идея такая:
Вместо ввода почты/имени и пароля пользователи вводят только свою почту. Ваше приложение отправляет на этот адрес одноразовую ссылку, пользователь по ней кликает и автоматически входит на ваш сайт / в приложение. При беспарольной аутентификации приложение считает, что в ваш ящик пришло письмо со ссылкой, если вы написали свой, а не чужой адрес.
Есть похожий метод, при котором вместо одноразовой ссылки по SMS отправляется код или одноразовый пароль. Но тогда придется объединить ваше приложение с SMS-сервисом вроде twilio (и сервис не бесплатен). Код или одноразовый пароль тоже можно отправлять по почте.
И еще один, менее (пока) популярный (и доступный только на устройствах Apple) метод беспарольной аутентификации: использовать Touch ID для аутентификации по отпечаткам пальцев. Если вы пользуетесь Slack, то уже могли столкнуться с беспарольной аутентификацией.
Medium предоставляет доступ к своему сайту только по почте. Auth0, или Facebook AccountKit, — это отличный вариант для реализации беспарольной системы для вашего приложения.
Что может пойти не так?
Если кто-то получит доступ к пользовательским почтам, он получит и доступ к приложениям и сайтам. Но это не ваша головная боль — беспокоиться о безопасности почтовых аккаунтов пользователей. Кроме того, если кто-то получит доступ к чужой почте, то сможет перехватить аккаунты в приложениях с беспарольной аутентификацией, воспользовавшись функцией восстановления пароля. Но мы ничего не можем поделать с почтой наших пользователей. Пойдем дальше.
В чем преимущества?
Как часто вы пользуетесь ссылкой «забыли пароль» для сброса пароля, который так и не смогли вспомнить после нескольких неудачных попыток входа на сайт / в приложение? Все мы бываем в такой ситуации. Все пароли не упомнишь, особенно если вы заботитесь о безопасности и для каждого сайта делаете отдельный пароль (соблюдая все эти «должен состоять не менее чем из восьми символов, содержать хотя бы одну цифру, строчную букву и специальный символ»). От всего этого вас избавит беспарольная аутентификация. Знаю, вы думаете сейчас: «Я использую

менеджер паролей». Но не забывайте, что подавляющее большинство пользователей не такие техногики, как вы. Это нужно учитывать.
Если вы думаете, что какие-то пользователи предпочтут старомодные логин/пароль, то предоставьте им оба варианта, чтобы они могли выбирать.
Сегодня беспарольная аутентификация быстро набирает популярность.

4. Единая точка входа (Single Sign On, SSO)
Обращали внимание, что, когда логинишься в браузере в каком-нибудь Google-сервисе, например, Gmail, а потом идешь на Youtube или иной Google-сервис, там не приходится логиниться? Ты автоматически получаешь доступ ко всем сервисам компании. Впечатляет, верно? Ведь хотя Gmail и Youtube — это сервисы Google, но все же раздельные продукты. Как они аутентифицируют пользователя во всех продуктах после единственного входа?
Этот метод называется единой точкой входа (Single Sign On, SSO).
Реализовать его можно по-разному. Например, использовать центральный сервис для оркестрации единого входа между несколькими клиентами. В случае с Google этот сервис называется Google Accounts. Когда пользователь логинится, Google Accounts создает куку, которая сохраняется за пользователем, когда тот ходит по принадлежащим компании сервисам. Как это работает:

- Пользователь входит в один из сервисов Google.
- Пользователь получает сгенерированную в Google Accounts куку.
- Пользователь идет в другой продукт Google.
- Пользователь снова перенаправляется в Google Accounts.
- Google Accounts видит, что пользователю уже присвоена кука, и перенаправляет пользователя в запрошенный продукт.

Очень простое описание единой точки входа: пользователь входит один раз и получает доступ ко всем системам без необходимости входить в каждую из них. В этой процедуре используется три сущности, доверяющие другу прямо и косвенно. Пользователь вводит пароль (или аутентифицируется иначе) у поставщика идентификационной информации (identity provider, IDP), чтобы получить доступ к поставщику услуги (service provider (SP). Пользователь доверяет IDP, и SP доверяет IDP, так что SP может доверять пользователю.
Выглядит очень просто, но конкретные реализации бывают очень сложными.

5. Аутентификация в соцсетях (Social sign-in) или социальным логином (Social Login).
Вы можете аутентифицировать пользователей по их аккаунтам в соцсетях. Тогда пользователям не придется регистрироваться отдельно в вашем приложении. Формально социальный логин — это не отдельный метод аутентификации. Это разновидность единой точки входа с упрощением процесса регистрации/входа пользователя в ваше приложение.
Пользователи могут войти в ваше приложение одним кликом, если у них есть аккаунт в одной из соцсетей. Им не нужно помнить логины и пароли. Это сильно улучшает опыт использования вашего приложения. Вам не нужно волноваться о безопасности пользовательских данных и думать о проверке адресов почты — они уже проверены соцсетями. Кроме того, в соцсетях уже есть механизмы восстановления пароля.
Большинство соцсетей в качестве механизма аутентификации используют авторизацию через OAuth2 (некоторые используют OAuth1, например Twitter).
Разберемся, что такое OAuth. Соцсеть — это сервер ресурсов, ваше приложение —

клиент, а пытающийся войти в ваше приложение пользователь — владелец ресурса. Ресурсом называется пользовательский профиль / информация для аутентификации. Когда пользователь хочет войти в ваше приложение, оно перенаправляет пользователя в соцсеть для аутентификации (обычно это всплывающее окно с URL'ом соцсети). После успешной аутентификации пользователь должен дать вашему приложению разрешение на доступ к своему профилю из соцсети. Затем соцсеть возвращает пользователя обратно в ваше приложение, но уже с токеном доступа. В следующий раз приложение возьмет этот токен и запросит у соцсети информацию из пользовательского профиля. Так работает OAuth (ради простоты я опустил технические подробности).
Для реализации такого механизма вам может понадобиться зарегистрировать свое приложение в разных соцсетях. Вам дадут app_id и другие ключи для конфигурирования подключения к соцсетям. Также есть несколько популярных библиотек/пакетов (вроде Passport, Laravel Socialite и т. д.), которые помогут упростить процедуру и избавят от излишней возни.

6. Двухфакторная аутентификация (2FA) улучшает безопасность доступа за счет использования двух методов (также называемых факторами) проверки личности пользователя. Это разновидность многофакторной аутентификации. Наверное, вам не приходило в голову, но в банкоматах вы проходите двухфакторную аутентификацию: на вашей банковской карте должна быть записана правильная информация, и в дополнение к этому вы вводите PIN. Если кто-то украдет вашу карту, то без кода он не сможет ею воспользоваться. (Не факт! — Примеч. пер.) То есть в системе двухфакторной аутентификации пользователь получает доступ только после того, как предоставит несколько отдельных частей информации.
Другой знакомый пример — двухфакторная аутентификация Mail.Ru, Google, Facebook и т. д. Если включен этот метод входа, то сначала вам нужно ввести логин и пароль, а затем одноразовый пароль (код проверки), отправляемый по SMS. Если ваш обычный пароль был скомпрометирован, аккаунт останется защищенным, потому что на втором шаге входа злоумышленник не сможет ввести нужный код проверки.
Вместо одноразового пароля в качестве второго фактора могут использоваться отпечатки пальцев или снимок сетчатки.
При двухфакторной аутентификации пользователь должен предоставить два из трех:
То, что вы знаете: пароль или PIN.
То, что у вас есть: физическое устройство (смартфон) или приложение, генерирующее одноразовые пароли.
Часть вас: биологически уникальное свойство вроде ваших отпечатков пальцев, голоса или снимка сетчатки.
Большинство хакеров охотятся за паролями и PIN-кодами. Гораздо труднее получить доступ к генератору токенов или биологическим свойствам, поэтому сегодня двухфакторка обеспечивает высокую безопасность аккаунтов.
И все же двухфакторка поможет усилить безопасность аутентификации в вашем приложении. Как реализовать? Возможно, стоит не велосипедить, а воспользоваться существующими решениями вроде Auth0 или Duo.

Доп. материал:
Про токены, JSON Web Tokens (JWT), аутентификацию и авторизацию. Token-Based Authentication

HTTP • авторизация

Почему важно делать подтверждение e-mail при регистрации?

- Основная причина — это провести double opt-in, т.е. убедиться, что был введён валидный адрес пользователя и этот пользователь действительно дает свое согласие на регистрацию на данном ресурсе и получение дополнительных рассылок/писем. Это позволяет отсеивать "мусорные" регистрации, когда подписывают на что-нибудь посторонних людей (намеренно или нет), уменьшать количество спама (за что очень больно бьют по рукам) и т. д.
- Для защиты страницы. Если кто-нибудь попытается получить доступ к аккаунту пользователя, то на почту пользователя придет соответствующее уведомление.
- Для быстрого и самостоятельного восстановления доступа (логина и пароля). Многие пользователи испытывают затруднения при восстановлении доступа, если не указали email при регистрации.
- Для отправки электронного чека после оплаты услуг сервиса.
- Для защиты от ботов.
- Если не подтверждать емайл, то в этом поле можно будет написать все что угодно (в рамках проверки). Соответственно один и тот же пользователь будет регистрироваться множество раз, забывая и свой предыдущий пароль, и логин.

Что такое кэш и зачем его очищать при тестировании?

Кэш — это временное хранилище для данных (перечень определен создателем сайта) с посещенного сайта. Во-первых, многие элементы на страницах сайта одинаковы: изображения, HTML, CSS, JavaScript и нет смысла каждый раз загружать их заново. Во-вторых, при повторном открытии той же самой страницы действует та же логика – эти элементы уже были загружены, ни к чему грузить их каждый раз по новой. Сохранение данных веб-страниц на компьютере, вместо их повторной загрузки, помогает экономить время открытия веб-сайтов в браузере и трафик, но с другой стороны снижает срок службы SSD-накопителей, так что вы всегда можете полностью отключить кеширование в своем браузере.

Виды кеширования:

- Кэширование в браузере. Устройство пользователя создает и сохраняет копию различных элементов сайта. Это могут быть: скрипты, текст, изображения и т. д. При открытии страницы кэш браузера помогает загрузить все это в разы быстрее.
- Кэширование на сервере. Все данные хранятся на сервере. Он сохраняет результаты запросов, что помогает избежать повторной обработки одной и той же информации от пользователя.

В тестировании практически во всех случаях правило – очищать кэш после каждого прохода теста (если только это не целенаправленной тестирование самого кэша или требуется наличие кэша по каким-либо причинам). Дело в том, что кэш очевидно искажает показатели performance testing, а также может быть причиной ошибочного дефект-репорта из-за устаревания и/или несогласованности актуальных и сохраненных данных. В некоторых ситуациях без очистки кеша не обойтись даже просто из-за огромного количества кешируемых данных.

Что такое AJAX в вебе?

Ajax (Asynchronous Javascript and XML — «асинхронный JavaScript и XML») — подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером. В результате при обновлении данных веб-страница не перезагружается полностью, и

веб-приложения становятся быстрее и удобнее. По-русски иногда произносится транслитом как «аякс». У аббревиатуры AJAX нет устоявшегося аналога на кириллице.

В классической модели веб-приложения:

- Пользователь заходит на веб-страницу и нажимает на какой-нибудь ее элемент. Браузер формирует и отправляет запрос серверу.
- В ответ сервер генерирует совершенно новую веб-страницу и отправляет ее браузеру и т. д. , после чего браузер полностью перезагружает всю страницу.

При использовании AJAX:

- Пользователь заходит на веб-страницу и нажимает на какой-нибудь ее элемент. JavaScript определяет, какая информация необходима для обновления страницы. Браузер отправляет соответствующий запрос на сервер.
- Сервер возвращает только ту часть документа, на которую пришел запрос. Скрипт вносит изменения с учетом полученной информации (без полной перезагрузки страницы).

Как работает браузер (коротко)?
Типичный сценарий использования предполагает отправку на некий сервер GET запроса и отображение полученного ответа. При этом происходит много всего: резолв домена в DNS -> получение целевого IP -> TCP/IP финты -> на запрос отдается запрашиваемая страница. Отображение страницы тоже не такой простой процесс:

Модуль отображения выполняет синтаксический анализ полученного HTML-документа и переводит теги в узлы DOM(DOM – объектная модель документа (Document Object Model) – служит для представления HTML-документа и интерфейса элементов HTML таким внешним объектам, как код JavaScript.) в дереве содержания. Информация о стилях извлекается как из внешних CSS-файлов, так и из элементов style. Эта информация и инструкции по отображению в HTML-файле используются для создания еще одного дерева – дерева отображения.
Оно содержит прямоугольники с визуальными атрибутами, такими как цвет и размер. Прямоугольники располагаются в том порядке, в каком они должны быть выведены на экран.
После создания дерева отображения начинается компоновка элементов, в ходе которой каждому узлу присваиваются координаты точки на экране, где он должен появиться. Затем выполняется отрисовка, при которой узлы дерева отображения последовательно отрисовываются с помощью исполнительной части пользовательского интерфейса.

Доп. материал:
Что на самом деле происходит, когда пользователь вбивает в браузер адрес google.com
Что происходит, когда пользователь набирает в браузере адрес сайта
What happens when you type a URL in the browser and press enter?
Как работает сотовая связь?
Почему связь «сотовая»? Если посмотреть сверху на схему сети базовых станций, то их пересекающиеся краями круги покрытия похожи на пчелиные соты.
Сотовая связь потому и называется сотовой, что в основе любой сети — ячейки (соты), каждая сота представляет собой участок территории, который покрывает (обслуживает) базовая станция. Форма и размеры сот зависят от множества факторов,

в том числе от мощности излучения базовой станции, стандарта, рабочих частот, направления антенн и т.п. Соты обязательно перекрывают друг друга, это необходимо для того, чтобы мобильное устройство (терминал) не теряло связь при перемещении из одной соты в другую. Особенно это важно для владельца сотового телефона, который разговаривает во время движения.
В условиях городской застройки невозможно разбить карту города на квадратики и поставить базовые станции через равные расстояния, чтобы добиться качественного покрытия. Начинают играть роль этажность застройки, препятствия в виде памятников, возможность установить базовые станции в том или ином месте. Не зря наши города назвали каменными джунглями, планирование в них радиосетей – это та еще задачка. Поэтому все операторы стараются резервировать дополнительные мощности в крупных городах, создавать перекрывающиеся зоны для базовых станций. И этому есть и другая причина.
Для эффективной работы сети одного покрытия мало, базовые станции должны обслуживать одновременно много пользователей. А в городах — очень много одновременно разговаривающих и пользующихся мобильным интернетом. Полосы частот, на которых передаются голос и данные, — ограниченный и крайне ценный ресурс, за их лицензирование операторы во всем мире платят государству большие деньги.

Когда вы набираете номер и начинаете звонить, ну, или вам кто-нибудь звонит, то ваш мобильный телефон по радиоканалу связывается с одной из антенн ближайшей базовой станции. От антенны сигнал по кабелю передается непосредственно в управляющий блок станции. Базовая станция должна выделить вам свободный голосовой канал. Вместе они и образуют базовую станцию [антенны и управляющий блок]. Несколько базовых станций, чьи антенны обслуживают отдельную территорию, например, район города или небольшой населенный пункт, подсоединены к специальному блоку – контроллеру. К одному контроллеру обычно подключается до 15 базовых станций. В свою очередь, контроллеры, которых также может быть несколько, кабелями подключены к «мозговому центру» – коммутатору. Коммутатор обеспечивает выход и вход сигналов на городские телефонные линии, на других операторов сотовой связи, а также операторов междугородней и международной связи.
В небольших сетях используется только один коммутатор, в более крупных, обслуживающих сразу более миллиона абонентов, могут использоваться два, три и более коммутаторов, объединенных между собой опять-таки проводами.
Когда человек передвигается по улице пешком или идет на автомобиле, поезде и т. д. и при этом еще и разговаривает по телефону, важно обеспечить непрерывность связи. Связисты процесс эстафетной передачи обслуживания в мобильных сетях называют термином «handover». Необходимо вовремя переключать телефон абонента из одной базовой станции на другую, от одного контроллера к другому и так далее.
Если бы базовые станции были напрямую подключены к коммутатору, то всеми этими переключениями пришлось бы управлять коммутатору. А ему «бедному» и так есть, чем заняться. Многоуровневая схема сети дает возможность равномерно распределить нагрузку на технические средства. Это снижает вероятность отказа оборудования и, как следствие, потери связи. Итак, достигнув коммутатора, наш звонок переводится далее – на сеть другого оператора мобильной, городской междугородной и международной связи. Конечно же, это происходит по высокоскоростным кабельным каналам связи. Звонок поступает на коммутатор другого

оператора. При этом последний «знает», на какой территории [в области действия, какого контроллера] сейчас находится нужный абонент. Коммутатор передает телефонный вызов конкретному контроллеру, в котором содержится информация, в зоне действия какой базовой станции находится адресат звонка. Контроллер посылает сигнал этой единственной базовой станции, а она в свою очередь «опрашивает», то есть вызывает мобильный телефон. Точно также происходят телефонные звонки в разные города России, Европы и мира. Для связи коммутаторов различных операторов связи используются высокоскоростные оптоволоконные каналы связи. Благодаря им сотни тысяч километров телефонный сигнал преодолевает за считанные секунды или даже доли секунд.

Как работает подключение к Wi-Fi?

- Начинает процесс подключения клиент, отправляя широковещательное сообщение Обнаружения DHCP (DHCP DISCOVER), в качестве обязательных полей передается номер транзакции - xid, MAC-адрес устройства - chaddr, также в опциях передается последний присвоенный клиенту IP-адрес, однако данная опция может быть проигнорирована сервером.
- Запрос обнаружения рассылается для всех узлов сети, но отвечают на него только DHCP-сервера, формируя сообщение Предложения DHCP (DHCP OFFER), которое содержит предлагаемую сервером сетевую конфигурацию. Если серверов несколько, то предложений клиент получит несколько. Из предложенных конфигураций клиент выбирает одну, как правило полученную первой.
- Так как MAC-адрес отправителя известен, то сервер направляет ответ непосредственно клиенту (unicast), хотя в некоторых случаях может ответить и широковещательным пакетом.
- Приняв предложение, клиент официально запрашивает у сервера данную конфигурацию, для чего отправляет широковещательно Запрос DHCP (DHCP REQUEST), он полностью повторяет по структуре сообщение обнаружения (Discover), только добавляет к нему опцию 54 с адресом сервера, конфигурацию которого клиент принял. Опция 50 содержит предложенный сервером IP-адрес. Несмотря на то, что MAC-адрес DHCP-сервера известен, запрос (Request) рассылается широковещательно, это нужно для того, чтобы остальные DHCP-сервера понимали, что их предложение отвергнуто.
- Получив запрос сервер направляет клиенту в ответ Подтверждение DHCP (DHCP ACK), которое отправляется на MAC-адрес клиента (хотя может и широковещательно) и, получив которое, клиент должен настроить свой сетевой адаптер согласно указанному адресу и опциям.
- Получив адрес, клиент может проверить его на предмет использования при помощи широковещательного ARP-запроса (в большинстве реализаций так и происходит) и если будет обнаружено, что выделенный адрес уже используется (скажем, назначен вручную), то клиент посылает широковещательное сообщение Отказа DHCP (DHCP DECLINE) и начинает процесс получения адреса заново. Сервер, получив сообщение отказа, должен пометить указанный адрес как недоступный и уведомить администратора о возможной проблеме в конфигурации (например, записью в логе).

После этого, все устройства, находящиеся во внутренней сети, будут выходить в Интернет через роутер под одним внешним IP-адресом, но в локальной сети они будут иметь разный IP.

----- Базы данных -----
Базовые понятия?

- Информация - любые сведения о каком-либо событии, процессе, объекте.
- Данные — это информация, представленная в определенном виде, позволяющем автоматизировать ее сбор, хранение и дальнейшую обработку человеком или информационным средством. Для компьютерных технологий данные — это информация в дискретном, фиксированном виде, удобная для хранения, обработки на ЭВМ, а также для передачи по каналам связи.
- База данных (БД) — именованная совокупность данных, отражающая состояние объектов и их отношений в рассматриваемой предметной области, или иначе БД — это совокупность взаимосвязанных данных при такой минимальной избыточности, которая допускает их использование оптимальным образом для одного или нескольких приложений в определенной предметной области. БД состоит из множества связанных файлов.
- Система управления базами данных (СУБД) — DBMS - Database management system - совокупность языковых и программных средств, предназначенных для создания, ведения и совместного использования БД многими пользователями. Автоматизированная информационная система (АИС) — это система, реализующая автоматизированный сбор, обработку, манипулирование данными, функционирующая на основе ЭВМ и других технических средств и включающая соответствующее программное обеспечение (ПО) и персонал.
- Банк данных (БнД) является разновидностью ИС. БНД — это система специальным образом организованных данных: баз данных, программных, технических, языковых, организационно-методических средств, предназначенных для обеспечения централизованного накопления и коллективного многоцелевого использования данных.
- Модель – способ структурирования данных, описания взаимосвязей между данными:
- Иерархическая модель. Модель представляет данные в виде иерархии. Модель ориентирована на описание объектов, находящихся между собой в неких отношениях. Например, структура кадров некоторой организации.
- Сетевая модель. Сетевая модель представляет собой развитие иерархической. Модель позволяет описывать более сложные виды взаимоотношений между данными. Однако расширение возможностей достигается за счет большей сложности реализации самой модели и трудности манипулирования данными.
- Реляционная модель. В реляционной модели данные представляются в виде таблиц, состоящих из строк и столбцов. Каждая строка таблицы – информация об одном конкретном объекте, столбцы содержат свойства этого объекта. Взаимоотношения между объектами задаются с помощью связей между столбцами таблиц. Реляционная модель на сегодняшний день наиболее распространена. Она достаточно универсальна и проста в проектировании. Строки таблиц называют записями или кортежами, столбцы – полями или атрибутами. Для того чтобы можно было сослаться на отдельную запись (строку) в некоторой таблице, каждая запись этой таблицы должна содержать уникальный идентификатор. Поле таблицы, значения которого гарантированно уникальны для каждой записи этой таблицы, называют ключевым полем или ключом. Ключ не обязательно должен быть числовым. Иногда

уникальным идентификатором может служить не одно поле, а комбинация полей. При этом сочетание значений этих полей должно быть уникальным. Такие поля образуют составной ключ таблицы

- Объектная модель. В этой модели данные представляются в форме объектов. Объект имеет набор свойств, называемых атрибутами, и может включать в себя также процедуры для обработки данных, которые называют методами. Объекты, имеющие одинаковые наборы атрибутов и различающиеся только их значениями, образуют некоторый класс объектов. Например, класс «клиент» может иметь следующие атрибуты: «фамилия», «имя», «отчество», «номер кредитной карты». Для каждого объекта из этого класса определены конкретные значения перечисленных атрибутов. Говорят, что объект является экземпляром класса. На основе существующего класса могут создаваться новые, наследующие свойства исходного. При этом исходный класс именуется родителем нового класса. Производный класс называют потомком исходного. При этом объекты – экземпляры класса-потомка принадлежат также и родительскому классу, поскольку обладают всеми его атрибутами. Пример: на основе класса «клиент» может быть определен класс «постоянный клиент»
- Гибридные модели. В некоторых приложениях предпринимаются попытки смешения различных моделей представления данных. Пример такого смешения – объектно-реляционная модель. В ней использовано некоторое сходство между реляционной и объектной идеологией. Строки таблиц реляционной модели соответствуют объектам объектной модели, столбцы таблиц – атрибутам объектов. Таблицы в целом являются аналогом классов. Отсюда вытекает возможность введения наследования при определении таблиц – таблица-потомок содержит те же столбцы, что и родительская, и, кроме того – дополнительные, определенные при наследовании. По идее создателей, объектно-реляционная модель должна унаследовать от реляционной легкость описания и манипулирования данными, а от объектной – возможность определения более сложных взаимоотношений между объектами.

Доп. материал:
Базы данных: большой обзор типов и подходов. Доклад Яндекса
Может ли у ПО быть сразу несколько баз данных?
Может и даже разного типа. Но в простых случаях делать это стоит только когда все упрется в предел производительности. Начиная с миллиардов записей у одной даже хорошо оптимизированной БД на одной hardware дисковой подсистеме уже могут начаться проблемы с performance, поэтому компания может принять решение разнести одну базу на несколько баз на разных серверах, но вместе с этим появляются вопросы к сетевому аспекту этого решения (задержки и т.п.). Помимо производительности, разделение на несколько БД может быть в угоду безопасности.
Что такое SQL?
structured query language — «язык структурированных запросов» — декларативный язык программирования, применяемый для создания, модификации и управления данными в реляционной базе данных, управляемой соответствующей системой управления базами данных.
Что вы знаете о NoSQL?
“NoSQL” имеет абсолютно стихийное происхождение и не имеет общепризнанного определения или научного учреждения за спиной. Это название скорее характеризует вектор развития ИТ в сторону от реляционных баз данных. Расшифровывается как Not Only SQL. Базы данных NoSQL специально созданы для определенных моделей

данных и обладают гибкими схемами, что позволяет разрабатывать современные приложения. Базы данных NoSQL получили широкое распространение в связи с простотой разработки, функциональностью и производительностью при любых масштабах.

Доп. материал:
Что такое NoSQL? | Нереляционные базы данных, модели данных с гибкой схемой | AWS
Что такое транзакция?
Транзакция — это набор операций по работе с базой данных (БД), объединенных в одну атомарную пачку. Или, если говорить по-научному, то транзакция — упорядоченное множество операций, переводящих базу данных из одного согласованного состояния в другое. Согласованное состояние — это состояние, которое подходит под бизнес-логику системы.
Источник: Что такое транзакция
Что такое нормальные формы?
Терминология:

- Атрибут — свойство некоторой сущности. Часто называется полем таблицы.
- Домен атрибута — множество допустимых значений, которые может принимать атрибут.
- Кортеж — конечное множество взаимосвязанных допустимых значений атрибутов, которые вместе описывают некоторую сущность (строка таблицы).
- Отношение — конечное множество кортежей (таблица).
- Схема отношения — конечное множество атрибутов, определяющих некоторую сущность. Иными словами, это структура таблицы, состоящей из конкретного набора полей.
- Проекция — отношение, полученное из заданного путем удаления и (или) перестановки некоторых атрибутов.
- Функциональная зависимость между атрибутами (множествами атрибутов) X и Y означает, что для любого допустимого набора кортежей в данном отношении: если два кортежа совпадают по значению X, то они совпадают по значению Y. Например, если значение атрибута «Название компании» — Canonical Ltd, то значением атрибута «Штаб-квартира» в таком кортеже всегда будет Millbank Tower, London, United Kingdom. Обозначение: {X} -> {Y}.
- Нормальная форма — требование, предъявляемое к структуре таблиц в теории реляционных баз данных для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц).
- Метод нормальных форм (НФ) состоит в сборе информации о объектах решения задачи в рамках одного отношения и последующей декомпозиции этого отношения на несколько взаимосвязанных отношений на основе процедур нормализации отношений.

Цель нормализации: исключить избыточное дублирование данных, которое является причиной аномалий, возникших при добавлении, редактировании и удалении кортежей(строк таблицы).

- Аномалией называется такая ситуация в таблице БД, которая приводит к противоречию в БД либо существенно усложняет обработку БД. Причиной является излишнее дублирование данных в таблице, которое вызывается наличием функциональных зависимостей от не ключевых атрибутов.

- Аномалии-модификации проявляются в том, что изменение одних данных может повлечь просмотр всей таблицы и соответствующее изменение некоторых записей таблицы.
- Аномалии-удаления — при удалении какого-либо кортежа из таблицы может пропасть информация, которая не связана на прямую с удаляемой записью.
- Аномалии-добавления возникают, когда информацию в таблицу нельзя поместить, пока она не полная, либо вставка записи требует дополнительного просмотра таблицы.

Нормальные формы:

- Первая нормальная форма: Отношение находится в 1НФ, если все его атрибуты являются простыми, все используемые домены должны содержать только скалярные значения. Не должно быть повторений строк в таблице.
- Вторая нормальная форма: Отношение находится во 2НФ, если оно находится в 1НФ и каждый не ключевой атрибут неприводимо зависит от Первичного Ключа(ПК). Неприводимость означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость.
- Третья нормальная форма: Отношение находится в 3НФ, когда находится во 2НФ и каждый не ключевой атрибут нетранзитивно зависит от первичного ключа. Проще говоря, второе правило требует выносить все не ключевые поля, содержимое которых может относиться к нескольким записям таблицы в отдельные таблицы.
- Четвертая нормальная форма: Отношение находится в 4НФ, если оно находится в НФБК и все нетривиальные многозначные зависимости фактически являются функциональными зависимостями от ее потенциальных ключей. В отношении R (A, B, C) существует многозначная зависимость R.A -> -> R.B в том и только в том случае, если множество значений B, соответствующее паре значений A и C, зависит только от A и не зависит от C.
- Пятая нормальная форма: Отношения находятся в 5НФ, если оно находится в 4НФ и отсутствуют сложные зависимые соединения между атрибутами. Если «Атрибут_1» зависит от «Атрибута_2», а «Атрибут_2» в свою очередь зависит от «Атрибута_3», а «Атрибут_3» зависит от «Атрибута_1», то все три атрибута обязательно входят в один кортеж. Это очень жесткое требование, которое можно выполнить лишь при дополнительных условиях. На практике трудно найти пример реализации этого требования в чистом виде.
- Шестая нормальная форма: Переменная отношения находится в шестой нормальной форме тогда и только тогда, когда она удовлетворяет всем нетривиальным зависимостям соединения. Из определения следует, что переменная находится в 6НФ тогда и только тогда, когда она неприводима, то есть не может быть подвергнута дальнейшей декомпозиции без потерь. Каждая переменная отношения, которая находится в 6НФ, также находится и в 5НФ. Идея «декомпозиции до конца» выдвигалась до начала исследований в области хронологических данных, но не нашла поддержки. Однако для хронологических баз данных максимально возможная декомпозиция позволяет бороться с избыточностью и упрощает поддержание целостности базы данных.

Понятие хранимой процедуры?

Хранимые процедуры представляют собой группы связанных между собой операторов SQL, применение которых делает работу программиста более легкой и гибкой,

поскольку выполнить хранимую процедуру часто оказывается гораздо проще, чем последовательность отдельных операторов SQL. Хранимые процедуры представляют собой набор команд, состоящий из одного или нескольких операторов SQL или функций и сохраняемый в базе данных в откомпилированном виде.

Понятие триггера?

Три́ггер (англ. trigger) — хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных: добавлением INSERT, удалением DELETE строки в заданной таблице, или изменением UPDATE данных в определенном столбце заданной таблицы реляционной базы данных. Триггеры применяются для обеспечения целостности данных и реализации сложной бизнес-логики. Триггер запускается автоматически при попытке изменения данных в таблице, с которой он связан. Все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера. Соответственно, в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

Что такое индексы? (Indexes)

Индекс - объект базы данных, создаваемый с целью повышения производительности поиска данных. Таблицы в базе данных могут иметь большое количество строк, которые хранятся в произвольном порядке, и их поиск по заданному критерию путем последовательного просмотра таблицы строка за строкой может занимать много времени. Индекс формируется из значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы и, таким образом, позволяет искать строки, удовлетворяющие критерию поиска. Ускорение работы с использованием индексов достигается в первую очередь за счет того, что индекс имеет структуру, оптимизированную под поиск — например, сбалансированного дерева. Различные типы индексов:

- B-Tree index
- Bitmap index
- Clustered index
- Covering index
- Non-unique index
- Unique index

Какие шаги выполняет тестировщик при тестировании хранимых процедур?

Тестировщик проверяет стандартный формат хранимых процедур, а также проверяет правильность полей, таких как updates, joins, indexes, deletions как указано в хранимой процедуре.

Как бы вы узнали для тестирования базы данных, сработал триггер или нет?

В журнале аудита (audit log) вы можете увидеть срабатывание триггеров.

Как тестировать загрузку данных при тестировании базы данных?

- Исходные данные должны быть известны
- Целевые данные должны быть известны
- Совместимость источника и цели должна быть проверена
- В диспетчере SQL Enterprise запустите пакет DTS после открытия соответствующего пакета DTS.
- Вы должны сравнить столбцы цели и источника данных
- Количество строк цели и источника должны быть проверены

- После обновления данных в источнике проверьте, появляются ли изменения в цели или нет.
- Проверьте на NULL и ненужные символы

Основные команды SQL?

- Просмотр доступных баз данных

SHOW DATABASES;

- Создание новой базы данных

CREATE DATABASE;

- Выбор базы данных для использования

USE <database_name>;

- Импорт SQL-команд из файла .sql

SOURCE <path_of_.sql_file>;

- Удаление базы данных

DROP DATABASE <database_name>;

- Просмотр таблиц, доступных в базе данных

SHOW TABLES;

- Создание новой таблицы

```
CREATE TABLE <table_name1> (
<col_name1> <col_type1>,
<col_name2> <col_type2>,
<col_name3> <col_type3>
PRIMARY KEY (<col_name1>),
FOREIGN KEY (<col_name2>) REFERENCES <table_name2>(<col_name2>)
);
```

- Добавление данных в таблицу

```
INSERT INTO <table_name> (<col_name1>, <col_name2>, <col_name3>, …)
VALUES (<value1>, <value2>, <value3>, …);
```

При добавлении данных в каждый столбец таблицы не требуется указывать названия столбцов.

```
INSERT INTO <table_name>
VALUES (<value1>, <value2>, <value3>, …);
```

- Обновление данных таблицы

```
UPDATE <table_name>
SET <col_name1> = <value1>, <col_name2> = <value2>, ...
WHERE <condition>;
```

- Удаление всех данных из таблицы

```
DELETE FROM <table_name>;
```

- Удаление таблицы

```
DROP TABLE <table_name>;
```

SELECT используется для получения данных из определенной таблицы:

```
SELECT <col_name1>, <col_name2>, …
FROM <table_name>;
```

- Следующей командой можно вывести все данные из таблицы:

```
SELECT * FROM <table_name>;
```

SELECT DISTINCT

- В столбцах таблицы могут содержаться повторяющиеся данные. Используйте SELECT DISTINCT для получения только неповторяющихся данных.

```
SELECT DISTINCT <col_name1>, <col_name2>, …
```

FROM <table_name>;
WHERE

- Можно использовать ключевое слово WHERE в SELECT для указания условий в запросе:

SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <condition>;

- В запросе можно задавать следующие условия:

сравнение текста;
сравнение численных значений;
логические операции AND (и), OR (или) и NOT (отрицание).
Пример
Попробуйте выполнить следующие команды. Обратите внимание на условия, заданные в WHERE:

```
SELECT * FROM course WHERE dept_name=’Comp. Sci.’;
SELECT * FROM course WHERE credits>3;
SELECT * FROM course WHERE dept_name='Comp. Sci.' AND credits>3;
```

- Оператор GROUP BY часто используется с агрегатными функциями, такими как COUNT, MAX, MIN, SUM и AVG, для группировки выходных значений.

```
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
GROUP BY <col_namex>;
```

Пример
Выведем количество курсов для каждого факультета:

```
SELECT COUNT(course_id), dept_name
FROM course
GROUP BY dept_name;
```

- Ключевое слово HAVING было добавлено в SQL потому, что WHERE не может быть использовано для работы с агрегатными функциями.

```
SELECT <col_name1>, <col_name2>, ...
FROM <table_name>
GROUP BY <column_namex>
HAVING <condition>
```

Пример
Выведем список факультетов, у которых более одного курса:

```
SELECT COUNT(course_id), dept_name
FROM course
GROUP BY dept_name
HAVING COUNT(course_id)>1;
```

- ORDER BY используется для сортировки результатов запроса по убыванию или возрастанию. ORDERBY отсортирует по возрастанию, если не будет указан способ сортировки ASC или DESC.

```
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
ORDER BY <col_name1>, <col_name2>, … ASC|DESC;
```

Пример
Выведем список курсов по возрастанию и убыванию количества кредитов:

```
SELECT * FROM course ORDER BY credits;
```

SELECT * FROM course ORDER BY credits DESC;

- BETWEEN используется для выбора значений данных из определенного промежутка. Могут быть использованы числовые и текстовые значения, а также даты.

SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <col_namex> BETWEEN <value1> AND <value2>;

Пример

Выведем список инструкторов, чья зарплата больше 50 000, но меньше 100 000:

SELECT * FROM instructor
WHERE salary BETWEEN 50000 AND 100000;

- Оператор LIKE используется в WHERE, чтобы задать шаблон поиска похожего значения.

Есть два свободных оператора, которые используются в LIKE:
% (ни одного, один или несколько символов);
_ (один символ).

SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <col_namex> LIKE <pattern>;

Пример

Выведем список курсов, в имени которых содержится «to», и список курсов, название которых начинается с «CS-»:

SELECT * FROM course WHERE title LIKE ‘%to%’;
SELECT * FROM course WHERE course_id LIKE 'CS-___';

- С помощью IN можно указать несколько значений для оператора WHERE:

SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <col_namen> IN (<value1>, <value2>, …);

Пример

Выведем список студентов с направлений Comp. Sci., Physics и Elec. Eng.:

SELECT * FROM student
WHERE dept_name IN (‘Comp. Sci.’, ‘Physics’, ‘Elec. Eng.’);

- JOIN используется для связи двух или более таблиц с помощью общих атрибутов внутри них.
- View — это виртуальная таблица SQL, созданная в результате выполнения выражения. Она содержит строки и столбцы и очень похожа на обычную SQL-таблицу. View всегда показывает самую свежую информацию из базы данных.

Создание

CREATE VIEW <view_name> AS
SELECT <col_name1>, <col_name2>, …
FROM <table_name>
WHERE <condition>;

Удаление

DROP VIEW <view_name>;

Пример

Создадим view, состоящую из курсов с 3 кредитами:

- 24. Агрегатные функции - эти функции используются для получения совокупного результата, относящегося к рассматриваемым данным. Ниже приведены общеупотребительные агрегированные функции:

COUNT (col_name) — возвращает количество строк;
SUM (col_name) — возвращает сумму значений в данном столбце;
AVG (col_name) — возвращает среднее значение данного столбца;
MIN (col_name) — возвращает наименьшее значение данного столбца;
MAX (col_name) — возвращает наибольшее значение данного столбца.

- Вложенные подзапросы — это SQL-запросы, которые включают выражения SELECT, FROM и WHERE, вложенные в другой запрос.

Пример
Найдем курсы, которые преподавались осенью 2009 и весной 2010 годов:

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year= 2009 AND course_id IN (
SELECT course_id
FROM section
WHERE semester = 'Spring' AND year= 2010
);
```

Доп. материал:
SQL запросы быстро. Часть 1

Подробнее о джойнах? (Join)
Как и было сказано выше, различные виды JOIN помогают объединить некие данные из нескольких таблиц каким-либо образом.

Так чем отличается INNER JOIN от LEFT JOIN? Чаще всего ответ примерно такой: "inner join — это как бы пересечение множеств, т.е. остается только то, что есть в обеих таблицах, a left join — это когда левая таблица остается без изменений, а от правой добавляется пересечение множеств. Для всех остальных строк добавляется null". Еще, бывает, рисуют пересекающиеся круги.
Это понимание и подобные ответы – по сути не верны, т.к. все джойны – декартово произведение (cross join) с фильтрами (предикатом и, возможно, UNION). Также стоит обратить внимание на порядок таблиц при различных джойнах.

Доп. материал:
Понимание джойнов сломано. Это точно не пересечение кругов, честно

Типы данных в SQL?

- Exact Numeric SQL Data Types:
- bigint = Range from -2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807)

int = Range from -2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)
smallint = Range from -2^15 (-32,768) to 2^15-1 (32,767)
tinyint = Range from 0 to 255
bit = 0 and 1
decimal = Range from –10^38 +1 to 10^38 -1
numeric = Range from -10^38 +1 to 10^38 -1
money = Range from -922,337,203,685,477.5808 to +922,337,203,685,477.5807
small money = Range from -214,748.3648 to +214,748.3647

- Approximate Numeric SQL Data Types:
- float = Range from -1.79E + 308 to 1.79E + 308

real = Range from -3.40E + 38 to 3.40E + 38

- Date and Time SQL Data Types:
- datetime = From Jan 1, 1753 to Dec 31, 9999

smalldatetime = From Jan 1, 1900 to Jun 6, 2079
date = To store a date like March 27, 1986
time = To store a time of day like 12:00 A.M.

- Character Strings SQL Data Types:
- char = Maximum length of 8,000 characters

varchar = Maximum of 8,000 characters
varchar(max) = Maximum length of 231 characters
text = Maximum length of 2,147,483,647 characters.

- Unicode Character Strings SQL Data Types:
- nchar = Maximum length of 4,000 characters

nvarchar = Maximum length of 4,000 characters
nvarchar(max) = Maximum length of 231 characters
ntext = Maximum length of 1,073,741,823 characters

- Binary SQL Data Types:
- binary = Maximum length of 8,000 bytes

varbinary = Maximum length of 8,000 bytes
varbinary(max) = Maximum length of 231 bytes
image = Maximum length of 2,147,483,647 bytes

Шпаргалки SQL

----- Практическая часть -----

Дана форма для регистрации. Протестируйте.

Вопрос номер один практически на всех собеседованиях на младшую позицию. Он хорош еще и тем, что в зависимости от уровня кандидата будет раскрыт в разной степени. Всегда в первую очередь уточняйте хотя бы какие-то минимальные требования, даже если вначале озвучивают, что требования не формализованы.

- Начальный уровень представляет из себя простые позитивные и негативные кейсы (в основном на валидацию):
- Обязательные поля отмечены *
- Обязательные поля заполнены/нет
- Галочки на соглашениях проставлены/нет
- Поле password и подтверждение имеет соответствующий тип (в полях формы прописан корректный атрибут TYPE, сообщающий браузеру тип элементов формы.)
- Проверяется, что пароли одинаковы
- Имя пользователя валидируется как минимум на длину и спец. символы, остальное по ТЗ

- Адрес почты валидируется в соответствии со стандартом (наличие символа @, несколько символов @, длины частей до и после @, допустимые символы до и после, наличие пробелов перед адресом и после, корректная доменная часть и т.п.)
- Поля с ожидаемым числовым вводом и текстовым соответственно проверить позитивными и негативными кейсами по типам данных
- Следующий уровень:
- Все из предыдущего
- Кроссбраузерность
- Понятность формы. Присутствует описание полей или плейсхолдеры
- Сенситив данные не должны передаваться в URL
- Проверяем, как форма отображается до сабмита и после
- Поведение, если нажать сабмит несколько раз подряд
- Если формы очищаются после сабмита, проверить регистрацию существующего пользователя
- Проверка глобализации – номер телефона, дата, почтовый индекс, валюта, вертикальное или RTL письмо и т.п. (опционально)
- Проверка простых инъекций
- Правильная работа многошаговых форм (Навигация рядом с формой показывает текущий этап и количество оставшихся шагов.)
- Для полей, предполагающих загрузку файлов, прописан атрибут accept, определяющий тип загружаемых документов
- Текстовое многострочное поле при вводе объемного сообщения изменяет высоту либо в правой части появляется скроллбар для просмотра всего содержимого
- Для авторизованного пользователя в поля формы автоматически подставляются все известные о посетителе данные.
- Форма сохраняется в веб-формах (админ-панели) или SQL-таблицах.
- Прописан реальный e-mail лица, отвечающего за обработку заявок (если предполагается ОС)
- Опционально. Пользователь получает уведомление на свой e-mail об успешно полученной заявке и последующих действиях, которые от него требуются.
- Прописан атрибут autocomplete для полей, поддерживающих это значение
- Extra:
- Проверяем, отправились ли данные после сабмита
- Проверяем, добавились ли соответствующие записи в бд
- Проверка загрузки формы и сабмита при медленном/нестабильном интернет-соединении
- Корректность cookies/токена и т.п. после сабмита

Есть еще форма посложнее (с просторов коммьюнити, автор @azshoo):

Или вот еще с просторов, реальное тестовое задание. Можно их много найти, если поискать.

Доп. материал:
Пароли, их тестирование и использование
Принципы и тестовые сценарии для тестирования паролей
Как Тестировать? Форма Входа

Определение серьезности и приоритета
Очень частый вопрос на собеседованиях. Либо вам дают конкретные примеры дефектов, для которых вы должны определить серьезность и приоритет, либо вас самих просят придумать варианты дефектов в каких-либо ситуациях. Например, дверь в магазине. Какой дефект может быть с низкой серьезностью, но высоким приоритетом? У каждой компании найдутся свои варианты вопросов, так что потренируйтесь заранее.
Определение граничных значений и классов эквивалентности
Следующий по популярности вопрос. Зная азы этих техник тест-дизайна, ответить довольно просто, но все равно будьте внимательнее. Потренироваться можно на просторах интернета.
Логические задачи
Могут быть буквально на логику (тесты Войнаровского):
«Саша смотрит на Ольгу, а Ольга смотрит на Андрея. У Саши есть дети, у Андрея нет. Смотрит ли человек, у которого есть дети, на человека, у которого детей нет? Варианты ответа: «Да», «Нет», «Нельзя определить». Объясните свою точку зрения.»

На рассуждение и перебор вариантов, цель - увидеть, как думает кандидат и насколько он эрудирован:

- Мессенджер. Один пользователь отправляет другому сообщение – не доходит. В чем может быть причина?
- Два абсолютно идентичных компьютера (аппаратная и программная конфигурация), файлы скачиваются с разной скоростью. Почему?
- Два абсолютно идентичных компьютера (аппаратная и программная конфигурация), на одном баг воспроизводится, на другом нет. Почему?
- Два разных мобильных устройства с одинаковой версией приложения. Бэк и связь стабильны. На одну приходят нотификации, на другую нет. В чем может быть причина?
- Есть форма с 5 полями, после отправки в БД записываются только 4. В чем может быть причина?
- Приложение при старте запрашивает по API профиль пользователя и на основе полученных данных расставляет в правильном порядке свои блоки интерфейса на главном экране. То есть ему нужны только цифры, остальное рендерится из готовых элементов приложения. На основе только этих данных, можно ли сказать что приложение является нативным или гибридным?

Или на «логику»:
«Есть две изолированные друг от друга комнаты. В одной находятся 3 лампочки, в другой - три выключателя. Вы стоите в комнате с выключателями и можете перейти в комнату с лампочками лишь один раз. Необходимо определить, какая лампочка включается каким выключателем.»

К первому типу можно подготовиться, изучив самые азы мат. логики и порешав несколько примеров. Многие относятся к этому несерьезно и проваливают этот тип заданий, между тем такие задачки щелкают на олимпиадах 5-клашки.
Второй тип задач показывает эрудированность в области computer science, здесь помогут только базовые общие курсы.

Про подготовку ко третьему типу задач, если опустить дискуссии об их бесполезности, можно сказать только то, что проще их просто прочитать и запомнить решение.
Подробнее изучить тему можно в популярной книге «Как сдвинуть гору Фудзи».
Еще примеры

- Дан веб-сайт, на котором есть каталог и реализована регистрация. На каких уровнях и что будете тестировать, конкретно по пунктам?
- Дана багтрекинговая система. Протестируйте воркфлоу (жизненный цикл бага).
- Аутлук - протестировать форму отправки письма (только этот функционал).
- Дано мобильное приложение: случайное подбрасывание игрального кубика. Как будете тестировать (кейсы)?
- Могут попросить накидать кейсов и в другом направлении. Например, придумайте кейсы для метода замены строки.
- Возможно не актуально в СНГ, но в англоязычных ресурсах встречаются задачи на decision/statement/branch coverage
- Спроектировать спецификацию API для калькулятора
- На просторах есть: лексический анализатор Яндекса, листбоксер в Veeam, крестики-нолики в Fora-soft
- GIT: ты на новом рабочем месте. Перечисли действия и команды как ты склонируешь себе репозиторий и создашь свою ветку;

Тут можно запросить платное тестовое с проверкой (просто наткнулся, так что хз что там)
Сайт для тренировки тестирования
Несколько примеров задач с решениями

Доп. материал:
Решаем тестовое задание на позицию тестировщика (Junior QA) / Ответы на вопросы тестировщику
ТЕСТОВОЕ ЗАДАНИЕ ТЕСТИРОВЩИКА / Какие бывают тестовые задания для QA, как делать тестовое
Набор небольших задач по SQL
Дана база:

```
CREATE TABLE Departments
    ([Id] int, [Name] varchar(100))
;

CREATE TABLE Employees
    ([Id] int, [Name] varchar(100), [Salary] int, [ManagerId] int, [DepartamentId] int)
;

INSERT INTO Departments
    ([Id], [Name])
VALUES
    (1, 'Sales'),
    (2, 'Development')
;
```

```
INSERT INTO Employees
   ([Id], [Name], [Salary], [ManagerId], [DepartamentId])
VALUES
   (1, 'Ivanov', 100000, null, 1),
   (2, 'Petrov', 120000, 1, 1),
   (3, 'Sidorov', 130000, 2, 1),
   (4, 'Korotkov', 120000, 2, 1),
   (5, 'Filev', 90000, 1, 1),
   (6, 'Smirnov', 125000, null, 2),
   (7, 'Godov', 125000, null, null)

/*1. Фамилии и зарплаты всех сотрудников*/
select Name, Salary from Employees
/*2. Фамилии и зарплаты всех сотрудников, отсортированные по зарплате по возрастанию*/
select Name, Salary from Employees order by Salary asc
/*3. Фамилии и зарплаты всех сотрудников, отсортированные по зарплате по убыванию*/
select Name, Salary from Employees order by Salary desc
/*4. Фамилии и зарплаты всех сотрудников, у которых зарплата больше 100000*/
select Name, Salary from Employees where Salary > 100000
/*5. Фамилии и зарплаты всех сотрудников, у которых зарплата равна 100000*/
select Name, Salary from Employees where Salary = 100000
/*6. Фамилии и зарплаты всех сотрудников, у которых зарплата больше либо равна 100000*/
select Name, Salary from Employees where Salary >= 100000
/*7. Фамилии и зарплаты всех сотрудников, у которых фамилия Ivanov*/
select Name, Salary from Employees where Name = ‘Ivanov’
/*8. Ид департамента и максимальная зарплата в этом департаменте*/
select DepartamentId, max(Salary) from Employees group by DepartamentId
/*9. Имя сотрудника и название его отдела*/
select emp.Name, dep.Name from Employees as emp
join Departments as dep on dep.Id = emp.DepartamentId
/*10. Имя сотрудника и имя его начальника*/
select emp2.Name, emp1.Name from Employees as emp1
join Employees as emp2 on emp1.Id = emp2.ManagerId
/*11. Добавить сотрудника с именем Semenov, зарплатой 70000, менеджером Ivanov и отделом Sales*/
insert into Employees
([Id], [Name], [Salary], [ManagerId], [DepartamentId])
values
(8, ‘Semenov’, 70000, 1, 1)
/*12. Изменить зарплату сотрудника Godov на 80000*/
update Employees
set Salary = 80000
where Id = 7
/*13. Удалить сотрудника Godov*/
delete from Employees
```

where Id = 7

Доп. материал:
SQL • для аналитики • —• • рейтинг прикладных задач с решениями
• Много ресурсов для практики есть в разделе инструментов тестировщика
Тестирование чашки для кофе
Не уверен, что это еще популярно спрашивать, но на всякий случай пара примеров с просторов.

Сначала – позитивное тестирование.

Функции чашки:
- вмещать напитки,
- переносить напитки,
- возможность пить из нее,
- возможность греть напитки в микроволновке.

Проверим сначала «вмещать»:
Поставили на поверхность – стоит, не падает - все ок.
Холодной воды налили – вода внутри - все ок.
Кипятку налили - не треснула, не течет – все ок.
*сперва хотела лить только горячую (раз горячую выдержит, то холодную – само собой), но потом решила, что это будет заодно и стрессовое тестирование (испытание на перепад температур).

Теперь – «переносить»:
Есть ручка, за нее можно взяться и поднять/перенести даже полную и горячую чашку (пустую и холодную носить не станем, если предусмотрен более тяжелый тест).

Теперь – «пить из нее»:
Наклонять удается, отхлебывать – тоже. Все ок.

«Возможность греть в микроволновке»:
Если в инструкции к чашке не указана максимальная допустимая температура при подогреве в печи, наливаем воду, ставим чашку в печь и включаем на максимум

По идее, нужно ставить таймер на время, достаточное для нагрева напитка до 100 градусов. Потому что если он выкипит, а чашка перегреется, это уже не будет позитивным тестированием.

Если позитивное тестирование удачно, проведем негативное (ошибочные или нестандартные, но возможные действия с предметом):
Подвергнем ее
- механическому (об пол, ап стену),
- химическому (кротом, адриланом, уксусной кислотой),
- физическому воздействию (перегреем в микроволновке, поставим на раскаленную горелку).

Еще - «тестирование удобства пользователя»:
удобно ли ставить, не горячо ли носить, приятно ли браться за ручку и т. д.

Еще – «тестирование безопасности»:
Вот на работе у меня чашка – маленькая, и край отбитый. Поэтому ее никто не трогает, когда меня нет. А эту наверняка все таскали бы – большая, удобная, красивая… Мне не жалко, но тест безопасности она бы не прошла.

Еще – «тестирование взаимодействия»:
встает ли чашка на блюдца от других сервизов.

Тестирование карандаша

HR: Как вы будете решать конфликты между членами вашей команды?
А также еще уйма подобных каверзных вопросов, в ответах на которые вам нужно проявить чудеса маневрирования, менеджерских качеств, софт-скилов и т.п. В общем-то не всегда можно угадать, что от вас ожидают услышать, но в целом это вопросы на оценку вашей адекватности, умения работать в коллективе и прочих софт-скилов.
HR: Что делать, если разработчик утверждает, что найденный дефект таковым не является?
Указывать на требования, апеллировать к здравому смыслу, подключать аналитика, чтобы объяснил. Если это поведение не описано в доке, то это баг, либо недоработка. Но недоработка в терминах джиры все равно баг
Проверить ТЗ. Если есть расхождение с ожидаемым результатом – привязываем ссылку на ТЗ.
Если формально это не зафиксировано, но вы чувствуете, что на это стоит обратить внимание – идете к писателю/аналитику/менеджеру, объясняете и в случае согласия это попадает в ТЗ.
Если формально не зафиксировано и менеджер с вами не согласен – дефект закрывается.
Вот тебе комп и работающий сайт. Сделай мне 401-ю ошибку
Задача на умение пользоваться инструментами, позволяющими подменять трафик. По обстоятельствам.
Пришел баг из продакшена, что делаем?
Воспроизводим, запускаем по пути жизненного цикла дефекта и анализируем причины, как данный дефект прошел в прод. После исправления дефекта разработчиком проводим повторное тестирование. Добавляем данный дефект в регрессию. В зависимости от охватываемого функционала и Root Cause этих багов принимается решение о проведении санитарного/регрессионного тестирования после подтверждающего.

Оценить время на тестирование лендинга
“Нужны ТЗ, макет, идеально - прототип. “А дальше считаете по самой простой эстимации по тест-кейсам. Функциональное: позитивные и негативные сценарии на поля ввода; чек-боксы, подписки, имэйлы, тултипы, хинты, кнопки, линки и футэр; Кроссбраузерное и кроссплатформенное: здесь надо уточнить, для каких браузеров и

девайсов тестируете; UI/UX: сверка с элементами макета в дев тулз на разных браузерах и девайсах; Грей бокс метод: смотрим, как сторятся отправленные данные в админке/бд. Плюс полный флоу прохождения регистрации в качестве смоука. Ну а потом прикидываете количество кейсов, на каждый кейс закладываете, сколько вы на него потратите времени (по-разному, здесь вы учитываете свою скорость)+20% на риски. Плюс закладываете время для регрессии.
В общем, в этом задании вам важно задать правильные вопросы) иначе будет из оперы "не зная тз - получишь хз" (с) @DorityTM

Для оценок в общем виде существует Метод оценки по 3 точкам (Three Point Estimation)
Один из самых распространенных и простых методов. В рамках него сначала определяются оптимистичная (O = Optimistic), пессимистичная (P = Pessimistic) и реалистичная/средняя (M = Middle) оценки.
Значения P, M и O определяются экспертно (в часах, днях, $), например, в ходе обсуждения внутри проектной команды. Для этого задаются вопросы такого типа: «сколько времени займет проект, если все пойдет хорошо, не будет никаких рисков и проблем?», «каким может быть самый негативный сценарий и сколько на него потребуется времени/усилий?» и т.п.
Далее полученные значения P, M и O подставляются в формулу: (O + 4 M + P) / 6
Результат расчета дает усредненную оценку. Такая формула позволяет с одной стороны учесть возможные позитивные и негативные сценарии, а с другой – «сгладить» их влияние и получить более реальное значение оценки.

Доп. материал:
Truthful Estimations by James Bach at ThinkTest 2015 (вкратце: "начну, потестирую немного, прикину, скажу")

----- Источники -----
www.software-testing.ru
www.techbeamers.com
www.guru99.com/software-testing.html
wikipedia.org
www.softwaretestinghelp.com/mobile-testing-interview-questions-answers/
medium.com/@sheidaievkostiantyn/capacity-testing-273c87ff03b4
tproger.ru/translations/sql-recap/
www.youtube.com/watch?v=SJwXK-2rw4M
lsreg.ru/shpargalka-po-sql/
lib.ssga.ru/
vc.ru/design/93884-32-otlichiya-dizayna-mobilnogo-prilozheniya-pod-ios-i-android
yamobi.ru/posts/kak_rabotaet_mobilnaya_svyaz_likbez.html
mobile-review.com/articles/2016/likbez-1.shtml
wifigid.ru/besprovodnye-tehnologii/kak-rabotaet-wi-fi
thecode.media/wifi/
html5book.ru/otzyvchivyj-dizayn-saita/#part5
lpgenerator.ru/blog/2015/10/21/responsivnyj-vs-adaptivnyj-dizajn-chto-luchshe-dlya-polzovatelya/

xakep.ru/2011/05/24/55557/
habr.com/ru/company/livetyping/blog/307860/
zen.yandex.ru/media/habr/kak-ty-realizuesh-autentifikaciiu-priiatel-5ec4cc1e033b1f6bec4ce836
interface31.ru/tech_it/2019/07/kak-ustroen-i-rabotaet-protokol-dhcp.html
www.youtube.com/watch?v=n_kl9sPQhXA
www.bigdataschool.ru/wiki/agile
www.youtube.com/watch?v=lKXGhh5un58
habr.com/ru/company/otus/blog/502720/
doitsmartly.ru/all-articles/blog-with-left-sidebar/88-requirements-for-qa-test-environment.html
withsecurity.ru/chto-takoe-pentest-i-dlya-chego-on-nuzhen
espressocode.top/software-testing-portability-testing/
https://testmatick.com/ru/testirovanie-globalizatsii/
artoftesting.com/
www.antula.ru/cookies.htm
yandex.ru/turbo?text=https%3A%2F%2Fnuancesprog.ru%2Fp%2F7833%2F
yandex.ru/blog/company/77455
www.rea.ru/ru/org/cathedries/infkaf/Documents/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5%20%D0%B2%D0%B5%D0%B1-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B%20%D0%B2%20%D1%8D%D0%BA%D0%BE%D0%BD%D0%BE%D0%BC%D0%B8%D0%BA%D0%B5.pdf
habr.com/ru/company/sibirix/blog/223777/
habr.com/ru/post/46374/
www.intervolga.ru/blog/projects/relsy-veb-integratsii-rest-i-soap/
flylib.com/books/en/2.156.1/control_flow_testing.html
www.protesting.ru/testing/testcoverage.html
kaner.com/pdfs/ScenarioIntroVer4.pdf
www.simbirsoft.com/blog/tekhniki-test-dizayna-i-ikh-prednaznachenie/
www.bullseye.com/coverage.html
sysgears.com/articles/test-design-techniques-overview/
www.youtube.com/watch?v=NrIN0qVBpZ4
www.intuit.ru
https://martinfowler.com/
33testers.blogspot.com/2015/06/blog-post_17.html
Microsoft Corporation - Performance testing Guidance for Web Applications
С. Куликов - Тестирование программного обеспечения. Базовый курс.

----- Поблагодарить автора -----

Приятно читать ваши письма с благодарностями и понимать, что куча моих сил и огромное количество времени, проведенные за проектом не пропадают зря. Знать, что мой проект помогает экономить время, устраиваться на работу, нанимать, используется как обучающее пособие для новых сотрудников и для студентов в университетах. Все это очень мотивирует продолжать :)

И все же некоторые люди считают более правильным говорить спасибо донатами, так что я наконец-то добавил возможность отправить мне скромную чашечку кофе :) а там кто знает, может так и до издания полноценной книги дойдет когда-нибудь))

По России можно просто пополнить связь Теле2 +79044121375
Яндекс.Деньги https://sobe.ru/na/QA_Bible
QIWI qiwi.com/n/VLADISLAV610
PAYPAL paypal.me/QAbible
PAYEER P1044386908

Спасибо и удачи в карьере!