

БИБЛИЯ QA v. 2.5

[Читать](#) • [Скачать](#) • [Контакты](#)

Что это за проект?

“Библия QA” - это копилка полезностей объемом 370+ страниц:

- ответы на самые популярные вопросы новичков о профессии и старте карьеры;
- крупнейшая подборка ссылок и полезных ресурсов;
- конспект всевозможной теории и ответов на вопросы с реальных собеседований;

Дисклеймер:

- проект не задуман чтобы по нему обучались, за этим на хорошие курсы или в фундаментальные книги;
- конспект авторский и составлен одним простым человеком, который не сеньор;
- проект бесплатный, без рекламы и пиратства.

----- Оглавление -----

| | |
|---|----|
| ----- Оглавление ----- | 1 |
| ----- F.A.Q. для новичков ----- | 9 |
| Ответы на самые популярные вопросы новичков в чатах | 9 |
| Качества и навыки, которыми нужно обладать тестировщику? | 10 |
| Что должен знать junior? А что спросят на собеседовании? | 11 |
| С чего начать обучение? | 12 |
| Как составить резюме? | 13 |
| Как вообще происходит процесс найма? | 15 |
| Как проходить собеседование? | 15 |
| Я получил оффер, что будет в первые рабочие дни? | 17 |
| Ошибки в работе у начинающих тестировщиков? | 18 |
| Как правильно задавать вопросы? | 18 |
| О процессах и что делать единственному тестировщику в команде | 19 |
| ----- Полезные ссылки ----- | 21 |
| Список полезных ресурсов на разных платформах | 21 |
| Список ресурсов по инструментам тестировщика | 24 |
| ----- HR-вопросы на собеседовании ----- | 30 |

| | |
|--|----|
| Вопросы с реальных собеседований с этапа HR | 30 |
| HR: Что делать, если разработчик утверждает, что найденный дефект таковым не является? | 30 |
| HR: Пришел баг из продакшена, что делаем? | 31 |
| HR: Кто виноват в багах, найденных в процессе регресса? | 31 |
| HR: Как решать конфликты в удаленной команде? | 31 |
| HR: Как понять, что тестировщик хорошо сделал свою работу? | 31 |
| ----- Теоретическая часть ----- | 33 |
| Общее | 33 |
| Обеспечение качества (Quality Assurance - QA) | 33 |
| Контроль качества (Quality Control - QC) | 34 |
| Тестирование (Testing) | 34 |
| Почему требуется тестирование ПО? | 35 |
| Качество ПО (Software Quality) | 35 |
| Принципы тестирования | 35 |
| Верификация и валидация (Verification and Validation) | 37 |
| Дефекты и ошибки | 39 |
| Серьезность и приоритет Дефекта (Severity & Priority) | 45 |
| Подход к тестированию (Test Approach) | 47 |
| Тестовое покрытие (Test Coverage) | 48 |
| Импакт анализ (анализ влияния, Impact Analysis)? | 49 |
| Анализ первопричин (RCA - Root Cause Analysis) | 50 |
| Модель зрелости тестирования (TMM - Test Maturity Model) | 53 |
| Тестирование со сдвигом влево (Shift left testing) | 54 |
| Независимое тестирование (Independent testing) | 55 |
| Тестирование как сервис (TaaS – testing as a Service) | 56 |
| Альфа- и бета- тестирование (Alpha Testing and Beta Testing) | 56 |
| Как протестировать продукт без требований? | 60 |
| Роли/должности в команде | 60 |
| Тестовая среда и тестовый стенд (Test Environment/Test Bed) | 62 |
| Тестовые данные (Test Data) | 63 |
| Бизнес-логика | 63 |
| Политика отсутствия багов (ZBP - Zero Bug Policy) | 63 |
| Эвристики и мнемоники | 64 |
| Виды/типы/уровни тестирования | 68 |
| Типы/методы тестирования (White/Black/Grey Box) | 68 |
| Тестирование черного ящика (Black Box Testing) | 68 |

| | |
|--|-----|
| Тестирование белого ящика (White Box Testing) | 72 |
| Тестирование серого ящика (Grey Box Testing) | 74 |
| Статическое и динамическое тестирование (Static Testing, Dynamic Testing) | 75 |
| Пирамида / уровни тестирования (Test Pyramid / Testing Levels) | 75 |
| Модульное/юнит/компонентное тестирование (Module/Unit/Component testing) | 76 |
| Интеграционное тестирование (Integration testing) | 78 |
| Системное тестирование (System Testing) | 81 |
| Приемочное тестирование (AT – Acceptance testing) | 83 |
| Основные виды тестирования ПО | 84 |
| Функциональное тестирование (Functional/Behavioral testing) | 86 |
| Нефункциональное тестирование (Non-Functional testing) | 87 |
| Тестирование производительности (Performance testing) | 89 |
| Тестирование емкости (Capacity testing) | 94 |
| Нагрузочное тестирование (Load testing) | 96 |
| Стрессовое тестирование (Stress testing) | 98 |
| Тестирование масштабируемости (Scalability testing) | 100 |
| Объемное тестирование (Volume testing) | 101 |
| Тестирование выносливости/стабильности (Endurance/Soak/Stability testing) | 102 |
| Тестирование устойчивости (Resilience testing) | 102 |
| Тестирование надежности (Reliability Testing) | 103 |
| Тестирование на отказ и восстановление (Failover and Recovery testing) | 104 |
| Эталонное и базовое тестирование (Benchmark and Baseline Testing) | 104 |
| Тестирование хранилища (Storage testing) | 106 |
| Одновременное / многопользовательское тестирование (Concurrency/Multi-user testing) | 106 |
| Тестирование сервиса (Service Testing) | 107 |
| Тестирование безопасности (Security and Access Control testing) | 108 |
| Оценка уязвимости/защищенности (Vulnerability Assessment) | 113 |
| Фаззинг-тестирование (Fuzz testing) | 118 |
| Можно ли отнести тестирование безопасности или нагрузочное тестирование к функциональным видам тестирования? | 119 |
| Тестирование совместимости/взаимодействия (Compatibility/Interoperability testing) | 119 |
| Конфигурационное тестирование (Configuration testing) | 121 |
| Тестирование на соответствие (Conformance/Compliance testing) | 123 |
| Тестирование удобства пользования (Usability testing) | 123 |
| Тестирование доступности (Accessibility testing) | 125 |
| Инсталляционное тестирование (Installation Testing) | 126 |

| | |
|--|-----|
| Тестирование локализации, глобализации и интернационализации (Localization/globalization/internationalization testing) | 127 |
| Исследовательское тестирование (Exploratory testing) | 130 |
| Свободное / Интуитивное тестирование (Adhoc, Ad-hoc Testing) | 131 |
| Тестирование поддержки (Maintenance testing) | 133 |
| Регрессионные виды тестирования (Regression testing) | 135 |
| Тестирование клиентской части и серверной (Frontend testing Vs. Backend testing) | 140 |
| Тестирование графического интерфейса/визуальное тестирование (GUI - Graphical User Interface testing) | 140 |
| Тестирование API (API - Application Programming Interface) | 142 |
| A/B тестирование (A/B Testing) | 146 |
| Деструктивное и недеструктивное тестирование (DT - Destructive testing and NDT – Non Destructive testing) | 149 |
| Выборочное/хаотическое тестирование (Random/monkey testing) | 149 |
| Тестирование рабочего процесса/воркфлоу (Workflow testing) | 151 |
| Тестирование документации (Documentation testing) | 152 |
| Мутационное тестирование (Mutation testing) | 154 |
| Разница тестирования ПО и железа (Software Vs. Hardware testing) | 155 |
| Параллельное тестирование (Parallel testing) | 157 |
| Тестирование качества данных (Data Quality Testing) | 158 |
| Подкожный тест (Subcutaneous test) | 159 |
| Тест дизайн | 161 |
| Тест-дизайн и техники тест-дизайна (Test Design and Software Testing Techniques) | 161 |
| Static -> Reviews | 164 |
| Static -> Static Analysis | 166 |
| Dynamic -> White box | 169 |
| Dynamic -> Black box | 174 |
| Dynamic -> Experience based | 189 |
| Тестовая документация/артефакты (Test Deliverables/test artifacts) | 191 |
| Виды тестовой документации | 191 |
| Политика качества и политика тестирования (Quality policy and Test policy) | 193 |
| Стратегия тестирования (Test strategy) | 194 |
| План тестирования (Test plan) | 195 |
| Тестовый сценарий (Test scenario) | 197 |
| Тест-кейс (Test case) | 198 |
| Чек-лист (Check List) | 200 |
| Баг-репорт (Defect/bug report) | 200 |

| | |
|---|-----|
| Требования (Requirements) | 201 |
| Пользовательские истории (User stories) | 210 |
| Критерии приемки (Acceptance Criteria) | 212 |
| Виды отчетов (Reports) | 214 |
| Базис тестирования (Test basis) | 215 |
| Матрица трассируемости (RTM - Requirement Traceability Matrix) | 216 |
| Метрики тестирования (Software Test Metrics) | 218 |
| Тестовый оракул (Test oracle) | 219 |
| SDLC и STLC | 222 |
| Жизненный цикл разработки ПО (SDLC - Software Development Lifecycle) | 222 |
| Жизненный цикл тестирования ПО (STLC – Software Testing Lifecycle) | 223 |
| Модели разработки ПО | 226 |
| Agile | 235 |
| Scrum | 239 |
| Подходы к разработке/тестированию (... - driven development/testing) | 242 |
| (Не обновлялось) Тестирование в разных сферах/областях (testing different domains) | 244 |
| Веб-тестирование | 244 |
| Тестирование банковского ПО | 250 |
| Тестирование электронной коммерции (eCommerce) | 250 |
| Тестирование платежного шлюза (Payment Gateway) | 253 |
| Тестирование систем розничной торговли (POS - Point Of Sale) | 254 |
| Тестирование в сфере страхования (Insurance) | 256 |
| Тестирование в сфере телекоммуникаций (Telecom) | 259 |
| Тестирование протокола: L2 и L3 OSI | 260 |
| Тестирование интернета вещей (IoT - Internet of Things) | 262 |
| Облачное тестирование (Cloud testing) | 264 |
| Тестирование сервис-ориентированной архитектуры (SOA - Service Oriented Architecture) | 266 |
| Тестирование планирования ресурсов предприятия (ERP - Enterprise Resource Planning) | 269 |
| Тестирование качества видеосвязи WebRTC-based сервиса видеоконференций | 270 |
| Тестирование ETL | 271 |
| Тестирование VR программного обеспечения | 272 |
| Тестирование мессенджера | 272 |
| Тестирование чат-бота | 272 |
| Тестирование микросервисной архитектуры | 272 |
| Тестирование e-mail | 272 |
| Тестирование платформы электронного обучения | 272 |

| | |
|--|-----|
| Тестирование игр (Gamedev) | 272 |
| Тестирование блокчейна (Blockchain) | 274 |
| Тестирование банкомата (ATM) | 274 |
| Тестирование Salesforce | 274 |
| Тестирование SaaS | 274 |
| Мобильное тестирование | 275 |
| Особенности в тестировании мобильных приложений | 275 |
| Типы мобильных приложений | 275 |
| Архитектура Android OS | 280 |
| Архитектура Android Application | 284 |
| Архитектура iOS | 294 |
| Архитектура iOS Application | 299 |
| iOS/Android Developer Settings | 305 |
| Основные различия iOS и Android | 308 |
| Последнее обновление Android/iOS, что нового? | 309 |
| Основные проверки при тестировании мобильного приложения | 310 |
| Каким образом тестировщик получает приложение на тест? | 314 |
| Симуляторы и эмуляторы | 314 |
| Push-уведомления: принципы работы и способы тестирования | 315 |
| Покрытие девайсов | 316 |
| Middleware | 317 |
| Как проверить использование процессора на мобильных устройствах? | 318 |
| Как успешно зарелизить продукт в App Store и Google Play | 318 |
| Android Debug Bridge (ADB) | 320 |
| Тестирование требований к мобильным приложениям | 320 |
| Разработка и тестирование мобильных дип линков (mobile deep links) | 320 |
| Тестирование сохраненных поисков | 320 |
| Тестирование покупок в приложениях | 320 |
| (Не обновлялось) Сети и около них | 322 |
| Клиент - серверная архитектура? | 322 |
| Уровни/модель OSI? | 323 |
| HTTP | 324 |
| TCP/IP | 329 |
| Endpoint, ресурс, URI, URL, URN | 330 |
| Веб-сервис (WS - Web service) | 332 |
| Сокет/веб-сокет (socket/web-socket) | 333 |

| | |
|---|-----|
| REST и SOAP | 334 |
| gRPC | 334 |
| JSON и XML | 334 |
| Какие еще бывают протоколы? | 336 |
| Куки (cookies) и их тестирование | 336 |
| Web Storage | 338 |
| Статические и динамические веб-сайты | 339 |
| Отличие stateless и stateful | 339 |
| Основные команды Linux | 339 |
| Почему важно тестировать в разных браузерах? | 342 |
| Адаптивный и отзывчивый веб-дизайн (Adaptive vs. Responsive) | 343 |
| Как сервер узнает, с какого типа устройства/браузера/ОС/языка вы открываете веб-сайт? (Например, для Adaptive design) | 344 |
| Какие заголовки (headers, хедеры) важны тестировщику? | 344 |
| Авторизация и аутентификация | 344 |
| Кэш и зачем его очищать при тестировании | 349 |
| Брокер сообщений (Message broker) | 349 |
| AJAX | 350 |
| Как работает браузер (коротко) | 350 |
| Как работает сотовая связь | 351 |
| Как работает подключение к Wi-Fi | 352 |
| (Не обновлялось) Базы данных | 352 |
| Базовые понятия | 353 |
| Может ли у ПО быть сразу несколько баз данных? | 354 |
| Что такое SQL? | 354 |
| Что вы знаете о NoSQL? | 354 |
| Что такое транзакция? | 355 |
| Что такое нормальные формы? | 355 |
| Понятие хранимой процедуры? | 356 |
| Понятие триггера? | 356 |
| Что такое индексы? (Indexes) | 357 |
| Что вы знаете о требованиях ACID? | 357 |
| Что такое “федеративные таблицы” в Mysql? | 357 |
| Тип памяти BLACKHOLE | 357 |
| Так как тестировать базы данных? | 357 |
| Какие шаги выполняет тестировщик при тестировании хранимых процедур? | 357 |

| | |
|--|------------|
| Как бы вы узнали для тестирования базы данных, сработал триггер или нет? | 357 |
| Как тестировать загрузку данных при тестировании базы данных? | 357 |
| Основные команды SQL? | 358 |
| Подробнее о джойнах? (Join) | 362 |
| Типы данных в SQL? | 364 |
| Шпаргалки SQL | 365 |
| ----- (Не обновлялось) Практическая часть ----- | 367 |
| Дана форма для регистрации. Протестируйте. | 367 |
| Определение серьезности и приоритета | 370 |
| Определение граничных значений и классов эквивалентности | 371 |
| Логические задачи | 371 |
| Еще примеры | 372 |
| Тестирование чашки для кофе | 376 |
| Тестирование карандаша | 377 |
| Вот тебе комп и работающий сайт. Сделай мне 401-ю ошибку | 378 |
| Оценить время на тестирование лендинга | 378 |
| ----- Контакты ----- | 379 |

----- F.A.Q. для новичков -----

Ответы на самые популярные вопросы новичков в чатах

Хочу войти в айти (в разработку) через тестирование, хороший план?

Это работало несколько лет назад, потому что тогда к новичкам-тестировщикам могло вообще не быть никаких требований кроме здравомыслия, но общий технический уровень в индустрии сильно вырос, а 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 и не нужно вестись на рекламу этих инфоцыган. 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 переобучение непосредственно той специальности, куда хотели изначально, т.к. опыт по сути не релевантен и спрашивать будут с нуля. Кейс перехода из тестирования в разработку (впрочем как и наоборот) встречается только у опытных специалистов и по совсем другим причинам.

Если же говорить просто о желании работать где-нибудь, лишь бы в сфере информационных технологий, то тяп-ляп быстренько прочитать пару книжек и сразу устроиться чтобы получать много денег тут не прокатит. Стоит изучить огромный перечень специальностей и понять для себя, что из этого ближе и развиваться сразу в нужном направлении, как бы невозможно это ни казалось.

Доп. материал:

- [Мифы о тестировании #2 / О чем не говорят на курсах по тестированию / Правда о работе в IT](#)
- [Почему профессия QA сложная и интересная, а не только простой «вход в IT»](#)
- [Why I Love Software Testing](#)
- [Программирование — это сложно](#)

Хочу зарабатывать много денег, мне сюда?

Первые зарплаты будут небольшими (особенно учитывая конкурс на места), а подняться выше по карьерной лестнице без искреннего интереса 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 ёкнуло и хочется взахлеб осваивать именно тестирование. Но даже в этих случаях нужно понимать, что тестирование - не рекордсмен по зарплатам. Далеко не всем компаниям требуется эксперты тестировщики, как это может быть в случае с разработчиками. 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 или спрос на свои навыки в текущем рынке при желаемых условиях. Соответственно и зарплаты здесь в среднем ниже, чем на многих специальностях в IT.

Доп. материал:

- Статистика зарплат: [Россия, Украина, Беларусь](#)
- [Why Experience Is More Important Than Money Early in Your Career](#)

Мне 30/40/50, кому-то нужен такой старик в команде?

Хорошая новость в том, что на возраст в IT чаще не смотрят. Безусловно, есть и молодые команды, куда возрастной коллега не впишется. Но в это же время существуют множество других команд, где возраст нового коллеги не имеет значения. Другой вопрос в том, готовы ли вы подчиняться 25-летнему сеньору и поспевать в

обучении за вчерашними студентами. Можно поискать по истории сообщений в QA-сообществах телеграмма, там этот вопрос неоднократно поднимался и там же есть истории смены сферы деятельности и успешного трудоустройства и в 40+ и в 50+.

Доп. материал:

- [Джуном? в 40 лет? Ещё и на удаленку? Да ну, не выдумывайте...](#)
- [Как стать ТЕСТИРОВЩИКОМ в 45 лет ? Плюсы и Минусы работы в QA](#)
- [В тестировщики после 40 лет - плюсы в смене профессии, обучении. Мотивация.](#)

Хочу работать удаленно джуном, это возможно?

Плохая новость в том, что шансы устроиться на удаленку специалисту без опыта сами по себе 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.

У меня нет высшего образования или я гуманитарий, всё пропало?

Зависит от требований нанимателя, но в большинстве случаев само наличие диплома никому не интересно, т.к. это ещё ни о чём не говорит (разве что есть какие-то достижения за время учёбы). Обычно такое требование можно увидеть только в вакансиях крупных компаний, да и то на деле оно чаще оказывается не обязательным. Не-техническая же специализация может оказаться вполне полезной, потому как бухгалтеров-экономистов охотно возьмут тестировать финтех, медиков соответственно медицинское ПО и т.д., а диплом переводчика вообще открывает некоторые двери автоматически.

Я всю жизнь работал %название должности%, это может как-то пригодиться в тестировании?

По аналогии со сказанным выше, потенциально почти любой опыт можно как-то использовать в тестировании. Если говорить о максимально близкой работе, с которой чаще всего и растут сюда, то это техническая поддержка, реже энгинееры и начинающие "сисадмины".

Доп. материал:

- [Как я перешла в тестирование](#)
- [From Support to QA: A Journey in Testing](#)

У меня старый компьютер, придется купить макбук про для работы?

В офисе или на удаленке в приличной компании вас всем обеспечит работодатель. Если же вы сами по себе или просто хотите узнать приемлемую конфигурацию, то для мануального тестировщика ресурсы рабочей машины и операционная система глобального значения не имеют, для комфортной работы ориентируйтесь на последнюю версию ОС, CPU 4 ядра/4 потока с поддержкой виртуализации + SSD + 8Gb RAM - этого хватит для любых задач. Если смотреть на перспективу, то для контейнеров, эмуляторов и всего такого потребуется больше потоков и от 16Gb оперативки. При использовании эмулятора вам пригодится поддержка аппаратного ускорения видео у APU или дискретной видеокарты.

Качества и навыки, которыми нужно обладать тестировщику?

- Самые главные:
 - Развитые софт-скиллы;
 - Прокачанное умение гуглить и постоянно самостоятельно обучаться на огромных массивах новой информации;
- Второстепенные, но всё еще важные:
 - Крайне полезно иметь склонность к развитию T-shaped skills (когда специалист углубляется в свою профессиональную область, но также развивается поверхностно в смежных областях);
 - QA Engineer это всё же engineer, желательно иметь склонности к problem solving;
 - Пытливый ум и желание докопаться до первопричины проблемы;

- Общая грамотность;
- Устойчивость к рутине;

Доп. материал:

- [Миф об образе мышления в тестировании](#)
- [Важные навыки тестировщика](#)

Про soft-skills:

- [Podlodka Soft Skills Crew – Интервью "Как я научился софт-скиллам и захватил мир"](#)
- [3 Tips To Improve Your Verbal Communication Skills](#)
- [Как разговаривать с му*аками \(краткое содержание\) - Марк Гоулстон](#)
- [Конспект по книге Марка Гаулстона "Я слышу вас насквозь"](#)
- [Патрик Ленсиони - Пять пороков команды \(краткое содержание\)](#)
- [Моя роль в конфликте. Елена Татти. СОМАQA Piter 2017](#)
- [Checklist молодого QA менеджера глазами тестировщика. Антон Семенченко](#)

Что должен знать junior? А что спросят на собеседовании?

Конкретного ответа на этот вопрос нет, всё зависит от конкретной компании и вакансии. Но вот пара ссылок на карты знаний для тестировщиков, можете на просторах найти и другие.

- [Awesome Quality Assurance Roadmap](#)
- <https://www.mindmeister.com/ru/1324282825/junior-qa?fullscreen=1>
- <https://www.mindmeister.com/ru/1558647509?t=973hdS2AKb>
- https://miro.com/app/board/o9J_IxUgVuQ=/
- <https://medium.com/slalom-build/quality-engineer-learning-roadmap-fddfcbb77409e>

Некоторые компании подробно расписывают на своих порталах ожидания от каждой стадии развития сотрудника, по этой же теме много видео на Youtube ([раз](#), [два](#), [три](#), [четыре](#), ...). Еще один ориентир – просто открыть и почитать вакансии, выписывая повторяющиеся пункты.

Если же попытаться выделить самые частые вопросы на собеседованиях, то получится примерно следующее:

- Что такое тестирование и зачем оно нужно
- Разница QA/QC/Тестирование
- Качество ПО
- Принципы тестирования
- Верификация и валидация
- Виды, типы, уровни тестирования
- Тестовые артефакты
- Баг и его жизненный цикл
- Severity/priority
- Техники тест-дизайна
- SDLC, STLC; Методологии разработки ПО
- Особенности мобильного тестирования*
- API
- Базовое знание сетей: Клиент-серверная архитектура, HTTP(s), его методы, коды ответов, TCP/IP, REST/SOAP, JSON/XML
- Базы данных: основы БД, что такое SQL, СУБД, основные команды (селекты, джойны).
- Инструменты: Chrome DevTools, Postman, Charles/Fiddler, GIT
- 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.

*Просто для цельной картины, если говорить про уровень middle, то работодателей теория уже не так интересует, если только это не крупная галера с высоким конкурсом (вопросы в таком случае будут +- те же, что и для junior, просто копнут глубже). Мидл - самостоятельная в решении рядовых задач боевая единица. Такой специалист уже имеет опыт в задачах, инструментах, видел какие-то процессы и уже хотя бы примерно может давать оценку времени на выполнение задачи. Соответственно и спрашивать будут больше по таким кейсам и по предыдущему опыту работы. Senior это уже про серьезный опыт, а также: автоматизация, CI/CD и место автоматизации в нём, менеджмент процессов и их построение, планирование, метрики, ROI, знание стандартов, опыт в тест планах и стратегиях, декомпозиция и распределение задач, оценка времени и т.п..

Помимо вышеперечисленного нужно помнить об английском языке. Он нужен для чтения документации и актуальных статей, просмотра вебинаров, поиска ответов на вопросы, т.к. в русскоязычном сегменте информации в разы меньше и пока ее переведут она уже устаревает. В РБ и Украине гораздо чаще чем в РФ язык нужен для ведения проектной документации и общения с иностранными коллегами (не надейтесь особо на переводчики и авто субтитры, всё это еще далеко от совершенства). Конечно, компании работающие на внутренние рынки могут не требовать знание языка, но тут, опять же, остается открытым вопрос личного развития. Если же в вакансии указан необходимый уровень владения языком, то будьте готовы к тому, что как минимум попросят ответить на какой-нибудь простенький житейский или HR-вопрос 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](#)
- [Святослав Куликов про QA, Курсы тестировщиков / Как развиваться тестировщику](#)
- [Что должен знать тестировщик бэкенда](#)
- [Тестировщик ПО / что делает QA Engineer / интервью с Artsiom Rusau QA](#)
- [Чек-лист подготовки к собеседованию на позицию ручного web-тестировщика](#)
- [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](#)
- [Пути развития тестировщика. Карьера QA Engineer](#)
- [QAGuild#54: Что должен знать тестировщик? Топ 3 навыка для QA Automation engineer](#)
- [Challenge accepted: карьера тестировщика](#)
- [Как тестировщику учить английский язык](#)
- [Таблица уровней английского языка](#)
- Марафон “Как IT-специалисту заговорить по-английски за 6 недель”: [часть 1](#) + [часть 2](#) + [часть 3](#)

С чего начать обучение?

Начать нужно с простого ознакомления с тестированием и самый частый совет для этого - книга Романа Савина “Тестирование дот ком”, которая не учебник, а скорее худ. лит. на 1-2 вечера, и местами спорная, но простыми словами расскажет о тестировании. После ознакомления я бы посоветовал выбрать по отзывам в коммьюнити хороший базовый онлайн-курс и пройти его, либо по возможности пойти на онлайн-курсы местной компании с возможностью последующего трудоустройства - это вообще лучший вариант. Если нет возможности, то хорошим выбором будет бесплатная [книга Святослава Куликова “Тестирование программного обеспечения. Базовый курс”](#) + бесплатный [курс в дополнение к ней](#) и далее уже имея общее представление и понимая азы равномерно восполнять пробелы, подготавливаясь к собеседованиям.

Тестирование – очень широкая область и, хотя базовая теория и не сильно сложная, ее довольно много. В отрыве от практике она плохо усваивается и быстро забывается, вы начинаете путаться. Нужно пытаться как можно быстрее найти применение своим навыкам. Начать стоит с тестирования приложений и сайтов, которыми нравится пользоваться или любых других, а также классики типа тестирования форм,

тренировочных сайтов с дефектами специально для тестировщиков и т.п. Отдельно советую действительно вдумываться во все практические примеры до полного понимания и способности решать аналогичные кейсы самостоятельно, просто за факт прочтения деньги платить не будут. Не стоит забывать и о софт скилах (учитесь адекватно общаться с людьми в профильных чатах) и базовой грамотности (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 - снова взяться тестировать какое-либо приложение или сайт, но теперь 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.). Багов хватает в любом популярном приложении/сайте, стоит только поискать, хотя баг-репорты и не главное. Главное показать понимание что и как тестировать.

Когда вы устроитесь на свою первую работу, спустя некоторое 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., либо 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) если работодатель оплачивает банкет, но вообще istqb если где и смотрят, то на западе и обычно не более чем как небольшой бонус.

Доп. материал:

- [Как стать тестировщиком с нуля](#)
- [Тестирование в эпоху ИИ](#)
- [12 Important Software Testing Trends for 2021 You Need to Know](#)
- [Как учиться, чтобы научиться](#)
- [James Whittaker — The role of testing in the age of AI \(ENG + RUS SUB\)](#)
- [Software Testing Trends to Watch 2021](#)
- [Тренды тестирования 2020-2021: правда и мифы](#)
- [Где начинающему тестировщику взять опыт для первой QA работы?](#)
- [Где начинающему тестировщику получить первый опыт: проект «Хомячки»](#)
- [Курс тестировщика пройден. А дальше что?](#)
- [Где начинающему тестировщику взять опыт для первой QA работы?](#)
- [Как получить первый опыт работы тестировщиком / Практика для тестировщика](#)
- [Заработок для QA - Практика - Фриланс - DIGIVANTE](#)
- [Устаревшие концепции тестирования: сертификация](#)
- [end-to-end discussion of ISTQB Foundation syllabus tutorials](#)
- [Geekhub Podcast: про образование в QA с Артемом Ерошенко и Всеволодом Брекеловым](#)

Как составить резюме?

Кратко о базовых рекомендациях.

Размер резюме стажера или джуниора – ровно 1 страница (имеется в виду вариант в файле, а не на площадках). Вариант минимум: PDF + расшаренная копия на google-диске. Вариант получше: резюме в веб-версии на github-pages + из варианта минимум.

Язык резюме – русский, если нацелены на компании из РФ с клиентами из РФ (или рекрутерами без знания английского :). В остальных случаях – английский.

Шапку можно оформить без наворотов: Нейтральное фото, по которому вас можно узнать, ФИО, на какую позицию претендуете, актуальные контакты, опционально локация.

Опыт работы (любые практические навыки) идет сразу после шапки. Это самая важная часть резюме. Без лишней воды кратко и емко описывается чем конкретно вы занимались. 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».

Навыки и технологии. С чем работали, что умеете. 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. Не забудьте упомянуть знание иностранных языков. Сориентироваться поможет, например, бесплатный тест [EFSET](#) с сертификатом или андроид-приложение [EnglishScore: Free British Council English Test](#).

Образование и самообразование. Университет, курсы, книги и т.п. Кратко и по существу.

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. Чаще всего их всё-таки читают, но не надо их писать просто для галочки, это сразу бросается в глаза и идет в минус. Тут лучше поступать аналогично с разделом “О себе” - пишите только если это действительно нужно, т.е. вы ознакомились с информацией о компании, вам есть чем выделиться среди других кандидатов и вы точно можете описать какие ваши навыки и как пригодятся бизнесу нанимателя. Для компании найм джуна это очень дорого и хорошо если кандидат начнет окупаться после полугода в штате, поэтому разговор с работодателем на понятном ему языке доход/расход может вас выделить из серой массы.

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 следовать базовым рекомендациям и периодически корректировать в зависимости от результатов, а если всё плохо, то можно скинуть итоговый вариант на оценку в коммюни티.

Насчет самих откликов, тут на мой взгляд, уместна аналогия с холодными звонками, т.е. не нужно на начальном этапе выбирать место работы так, будто у вас лишь один шанс и вы собираетесь в ней состариться. Вообще слать резюме стоит не только откликаясь на вакансии. Есть мнение, что когда компания выкинула вакансию на работные сайты, это уже тупик (т.к. не нашли кандидата по своим каналам). Шлите на почты IT-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?

Дополнительно я бы посоветовал ознакомиться с разными типами компаний. Работа в инхаус и аутсорс заметно отличается и помимо просто понимания, куда бы вам больше хотелось, преимуществом будет знать, как подогнать резюме и отклик под конкретный тип.

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](#)
- [Аутсорсинг или продуктовая компания для тестировщика \(QA\)](#)
- [Как составить резюме на английском для иностранной компании](#)

- [Резюме для тестировщика. Структура, оформление, рекомендации](#)
- [Разбор резюме тестировщика \(QA\) / Ответы на вопросы](#)

Как вообще происходит процесс найма?

Все зависит от компании и в меньшей степени от уровня позиции. В среднем это выглядит так:

1. Отклик на вакансию;
2. *Опционально: выполнение тестового задания, п.2 и п.3 могут меняться местами;
3. Скрининг по телефону (небольшая беседа с HR);
4. Полнотенное собеседование с HR;
5. Техническое собеседование, п.4 и п.5 иногда делают за раз;
6. *Опционально: знакомство с боссом/лидом/командой);

В очень больших компаниях с потоковым наймом бывает и по 6-7 этапов интервью, в мелких местных же всё может обойтись одной встречей с беседой за кофе.

Как проходить собеседование?

Прежде всего, конечно, на него не нужно опаздывать. В случае оффлайна стоит прийти немного заранее, оглядеться, привыкнуть к атмосфере и настроиться на нужный лад. Требований по дресс-коду обычно нет, стоит просто быть опрятным и ориентироваться на “среднюю температуру по больнице” (району), т.е. не идти в Москва-сити в пляжных шортах и сланцах, так же как и не идти в офис на берегу моря в костюме-тройке, всё остальное приемлемо.

В случае онлайна так же необходимо быть на связи уже за 5-10 минут до назначенного времени, поздороваться в чате и сообщить что вы онлайн, устройства должны быть заряжены, интернет стабилен, микрофон и звук настроены и проверены. Приятное впечатление оставят хороший свет и отсутствие постороннего шума. Внешний вид особого значения не имеет, но сидеть с голым торсом и хлебать борщ не стоит (помните про опрятность и здравый смысл). На фон за вами большинству всё равно, хотя сейчас почти везде можно включить размытие.

Главное, что нужно сказать о собеседованиях – их нужно проходить. 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 и 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 выбрать себе возможность пройти интервью. Цифры примерно такие: 10 откликов на подходящие вакансии или 100 рассылок = 1 собес. 10 не заваленных собесов = 1 оффер. Принятие решения компанией может занимать пару недель. В некоторых компаниях этапов может быть штук 7. 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 (и даже год не редкость).

Основное, что хочется сказать про само собеседование:

Узнай всё что можешь 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 взглядов 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, подходит 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 если вопрос максимально глупый. Или вас попробуют заставить сомневаться в заведомо правильном ответе (в этот момент вы вспомните передачу “кто хочет стать миллионером”). Умение адекватно и грамотно отвечать на вопросы и 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 закапаетесь в новых вопросах.

В конце (хотя бывает и в начале) спрашивает собеседуемый! Помните, что это общий 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 типичных вопросов на любом собеседовании (не бойтесь, что если такое 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 заработка 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? Офис 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?

Хотя я бы лучше уточнил про вентиляцию и свет в помещении, где будет рабочее место, ситуация с перерывами/обедами и всё такое более насущное. “Профильные” вопросы работодателю ищите в доп. материалах и выбирайте что нужно знать именно вам.

Можно еще обратить внимание на такие моменты, как:

- пришлось ли вам ждать начала 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 деловой беседе

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.

Как отвечать на вопросы об ожидаемой компенсации труда каждый для себя решает сам, я лишь призываю не демпинговать и помнить, что любой труд должен оплачиваться. Обычно спрашивают 3 цифры: на испытательном сроке, после, через год. Для столиц цифры в \$ очень приблизительно такие: 300-500/500-800/800-1300; для регионов: 200-400/300-600/500-800. Разумеется это при соответствующем росте ваших навыков и в этой перспективе вы уже должны были убедить собеседующего за время интервью.

Доп. материал:

- [Вопросы для собеседования — от кандидата к работодателю](#)
- [Как собеседовать работодателя?](#)
- [Level up 2021: как собрать лучшие офферы в ИТ](#)
- [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)
- [Круглый стол «Все грани собеседования»](#)
- [Идеальное собеседование на позицию QA-инженера: как провести / как пройти](#)
- [Лайфхаки: как получить больше обратной связи после собеседования](#)
- [Оцениваем работодателя на собеседовании. Как понять, что за компания перед тобой?](#)
- [25 актуальных вопросов работодателю + комментарии разработчика](#)
- [Почему вам не дают подробный фидбек после собеседования](#)
- [Страх перед собеседованием: как перестать бояться и начать ходить на интервью](#)
- [50 вопросов потенциальному IT-работодателю](#)
- [Собеседование QA: лайфхаки и скрипты успеха](#)
- [Как оценить процессы в компании + комментарии разработчика](#)
- [Вы не просите дать вам работу, вы продаёте услугу](#)
- [Как новичку в ИТ-сфере показать HR свои лучшие стороны \(поиск работы в ИТ\)](#)
- [Типичные ошибки начинающего QA на собеседовании](#)
- [Как найти удаленную работу в зарубежной компании. 10 шагов](#)

Я получил оффер, что будет в первые рабочие дни?

Если вас угораздило быть на первой работе единственным тестировщиком, то скорее всего вас познакомят с коллегами и расскажут о продукте (а вы уже сами должны знать о нем всё что возможно узнать заранее).

Вероятно вас просто с ходу бросят на амбразуру искать баги, возможно писать кейсы и попутно будут отвечать на вопросы, а потом уже всё остальное.

В крупных же компаниях обычно есть налаженный процесс онбординга, который начинается с выдачи прав доступа, настройки рабочих учеток и рабочей станции, знакомства с внутренней структурой проекта и

компании, ознакомления со стеком технологий и инструментами. К вам прикрепляют ментора и рассказывают к кому и с какими вопросами можно обращаться.

В любом случае каких-то значимых результатов от новичков первое время никто ждать не будет, обычно есть минимум один “золотой” месяц, когда сотрудника сюсюкают как младенчика на второстепенных задачах, пока тот не вольется в работу, так что не накручивайте себя заранее.

Доп. материал:

- [Welcome on board или по ту сторону оффера](#)
- [Введение тестировщика в проект и процесс тестирования](#)
- [Как выжить на новой работе или онбординг снизу](#)

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, а коллеге придется тратить время чтобы заново сфокусироваться на задаче).

Доп. материал:

- [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 совершила как junior QA engineer](#)
- [7 QA-шных грехов, которые помогут или помешают тестировщику \(стать тем, кем ты хочешь\)](#)
- [Сегодня — трейни, а завтра — сеньор. Что может быть не так с самооценкой у новичков](#)

Как правильно задавать вопросы?

В чатах, коллегам, на форумах, где угодно:

- прочитайте внимательно <https://nometa.xyz/>;
- убедитесь, что вы потратили уже достаточно времени в гугле и у вас ничего не вышло;
- сформулируйте вопрос со всем контекстом, чтобы любой человек мог не напрягаясь его понять;

- опишите все ваши действия и результаты, в т.ч. не увенчавшиеся успехом;
- сформулируйте предельно ясно и четко с чем вы просите помочь;

Доп. материал:

[Как решать сложные \(технические\) проблемы?](#)

О процессах и что делать единственному тестировщику в команде

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 СММ и ТММ, зрелость проекта (новый/старый-зрелый/старые-зрелые где будут глобальные изменения) и команды, определите методологию разработки. Соберите метрики, подбейте статистику, подумайте как на эту статистику можно повлиять. Проведите вводную лекцию команде: что такое QA, как оно может помочь, с какими проблемами обращаться и зачем всё это надо. Далее в зависимости от всего этого выбирайте в доп. материалах или на просторах интернета вебинар/статью о процессах (можно поинтересоваться опытом коллег в коммюниити или поискать по истории) и сверяясь с целями компании аккуратно приступайте к внедрению оных.

Вообще создавать отдел тестирования нанимают QA Lead, а если вы джун без опыта работы, то либо работодатель не понимает что делает, либо у него есть вполне конкретная “боль”, которую он с помощью тестировщика хочет решить, в таком случае проводить целое расследование не придется - наверняка все объяснят еще на собеседовании.

В нормальной ситуации вы проведете исследовательское тестирование, составите несколько наборов кейсов, задокументируете все текущие баги и в дальнейшем будете заниматься тестированием новых сборок, проверкой исправления найденных дефектов и проведением регрессии. Т.е. если вернуться в плоскость процессов, то нужно будет их обдумать и внедрить хотя бы на ключевые моменты: этап составления требований, этап проверки нового функционала перед вливанием в основной, этап регрессии, этап предоставления отчетности (и в целом все заинтересованные лица всегда должны иметь актуальную информацию о текущем состоянии качества релиза/продукта)

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](#)
- [Управляемое тестирование: с чего мы начинаем, чтобы не было мучительно больно](#)
- [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](#)
- [QAGuild #49: Самая частая проблема в сфере тестирования - Проблемы QA](#)
- [QA-митап Redmadrobot 19/11, Современные паттерны тестирования, Марина Куликова](#)
- [Оптимизируем процесс тестирования: на какие подходы стоит обратить внимание](#)
- [Heisenbug Show / Методологии и процессы в тестировании // 29 сентября 2020](#)
- [Blog: Testers: Focus on Problems](#)
- [Один тестировщик на проекте - Советы по организации работы](#)
- [Leadership in test: executing a test project](#)
- [ISTQB Foundation Level Syllabus, Chapter 5 of 6: Test Management](#)

----- Полезные ссылки -----

Список полезных ресурсов на разных платформах

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, дую, медиум и т.п., вам отберут все самое полезное. В конце концов, в этих каналах публикуются анонсы грядущих онлайн-мероприятий, чтобы ничего не упустить.

- [Список интересных групп, каналов и ботов телеграма](#)
- Всем новичкам сюда! Тренировочные собеседования, интересные обсуждения [@qa_interviews](#)
- Огромный чат, ориентированный на джуниоров [@gajuniors](#)
- Огромный чат, ориентированный на уже работающих в сфере тестирования [@qa_ru](#)
- Чат по геймдеву [@qa_gamedev](#)
- Обсуждение курсов, отзывы о них [@qa_courses](#)
- Тут можно размещать свое резюме [@qa_resumes](#)
- Тут отзывы о компаниях [@qa_bad_company](#)
- Обсуждение финансов [@qa_fin](#)
- Публикация книг [@booksqa](#)
- Авторский бложик [@shooandendlessagony](#)
- Флудилки [@gachanellflood](#) [@qaflood](#)
- Чат про производительность и инструменты [@qa_load](#)
- Jobs abroad - Работа за рубежом [@jobs_abroad](#)
- Чаты по penetration testing (pen test, pentest):
 - [@qa_security](#)
 - [@true_secator](#)
 - [@pentesting_channel](#)
 - [@pentesting_chat](#)
 - [@AAAAAAE_Cu9rTSOgF7uolrg](#)
 - [@hackerlib](#)
- Репосты новых статей и полезных ссылок с разных сайтов:
 - [@qa_wiki](#)
 - [@serious_tester](#)
 - [@qa_pro](#)
 - [@qa_chillout](#)
 - [@yetanotherqa](#)

Youtube-каналы:

- [Вадим Ксендзов \(тренировочные собеседования!\)](#)
- [Artsiom Rusau QA Life \(бесплатный курс!\)](#)
- [Podlodka Podcast](#)
- [Heisenbug](#)
- [QA START UP](#)
- [Компания DINS](#)
- [Hillel IT School](#)
- [QAGuild](#)
- [Mikhail Portnov](#)
- [Alexei Barantsev](#)

- [LearnQA](#)
- [Леша Маршал](#)
- [Тестирование](#)
- [Fest Group](#)
- [Andrey Sozykin](#)
- [All about QA](#)
- [Look Live UI](#)
- [Test Club RU](#)

Web:

- <https://software-testing.ru/>
- <https://protesting.ru/>
- <http://www.mobileappstesting.com/>
- <https://www.guru99.com>
- <https://www.softwaretestingmaterial.com/>
- <https://www.ministryoftesting.com/>
- [Black Box Software Testing course site](#)
- <https://www.learnqa.ru/>
- <https://mobilenativefoundation.org/>
- [Хроники тестировщика](#)

Книги и материалы:

- Очень хвалят вот эту [подборку](#);
- Если выделить самые советуемые книги в коммьюнити, то получится следующее:
 - Самая спорная книга для новичков Р. Савин - “Тестирование дот ком”. Является скорее вариантом худ. лита “для чайников” на один вечер и может местами быть устаревшей;
 - [Святослав Куликов - “Тестирование программного обеспечения. Базовый курс.”](#);
 - Lee Copeland - “A Practitioner's Guide to Software Test Design” (есть в переводе)
 - Rex Black - “Critical Testing Processes” (есть в переводе)
 - Гленфорд Майерс, Том Баджетт, Кори Сандлер «Искусство тестирования программ.»
- [IEEE Guide to the Software Engineering Body of Knowledge](#)
- [RSTQB » Downloads](#) и [ISTQB » Downloads](#)

Курсы:

Списка и конкретных рекомендаций здесь не будет по понятным причинам. Я лишь посоветую обходить стороной разного рода инфоцыган, реклама которых вылезает из каждого утюга и что обещают трудоустройство по записанным лекциям за конский ценник, думаю тут и без конкретики все поймут о каких компаниях речь.

В телеграмме есть чат @[qa_courses](#) с обсуждением разнообразных нормальных курсов и отзывами о них, ищите по истории сообщений по хештегам, читайте, выбирайте сами (только держите в уме, что админы и сами авторы некоторых курсов). Также есть таблица с [отзывами](#) от Артёма Русова.

По поводу всяких sharewood и т.п. Использование слитых курсов - это личное дело каждого, ситуации в жизни бывают разные. Но имейте в виду, что в IT пиратство крайне не приветствуется, упоминать такие поступки не стоит, а обсуждение в чатах может легко закончиться баном. Кроме того, вся ценность курсов именно в практике и обратной связи от преподавателей.

Мобильные приложения:

Подборка для Android, с iOS дела не имел. Возможно, указанные в списке приложения есть на обеих платформах.

- [StrimQa — ваш карьерный навигатор!](#)
- [Interview Questions and Answers 2021](#)
- [Software Engineering](#)
- [QA Wizard подготовка к собеседованию для тестеров](#)
- [Learn Software Testing-Interview questions & quiz](#)
- [Software Testing Interview FAQ](#)
- [Software Testing - QA Learning](#)
- [Mobile Testing](#)

Другие сборники материала и ответов на вопросы:

- [Подборка от Артёма Русова](#)
- [Подборка от сообщества QA juniors](#)
- [Всё о QA: 80 бесплатных материалов по грамотному тестированию](#)
- [Текстовая версия курса Артема Русова с youtube](#)
- [Что должен уметь начинающий тестировщик](#)
- [Полезные ссылки для тестировщика](#)
- [Библиотека QA](#)
- [Обзор частых вопросов по тестированию ПО на собеседованиях и ответы на них](#)
- [Ресурсы для развития тестировщика 2021](#)
- [14 самых вдохновляющих статей о тестировании ПО, которые я когда-либо читал](#)
- [Что почитать продолжающему тестировщику \(часть 2\)](#)
- [Фундаментальная теория тестирования](#)
- [Wikipedia - Тестирование программного обеспечения](#)

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/1LgytNrl7ep9wlr3A_3u0NitQsrZzKhEQwC-OTQfbLAM/edit?usp=sharing
- https://docs.google.com/spreadsheets/d/1r5Ek83V4IHkOsW52DyVT8iJepR20oZu_Jy5vAkq7Srl/edit?usp=sharing
- [Элементы интерфейса сайта](#)
- [UI-элементы и жесты в мобильных приложениях](#)
- [Словарь тестировщика](#)
- [User Interface Elements Every Designer Should Know](#)
- [Glossary WEB - project](#)

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 для веб-форм](#)
- [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](#)
- [Чеклист: 217 пунктов для отличного интернет-магазина](#)

Список ресурсов по инструментам тестировщика

- Мультитул: DevTools;
- Снiffeры: 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 mind map - like продукт;
- Фермы устройств для тестирования мобильных приложений: BrowserStack, Xamarin, AWS;
- Инструменты тестирования мобильных 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](#)
- [Урок 10: Введение в Тестирование ПО - QA с Нуля - DevTools, Web Console, Device Toolbar](#)
- [Основные Use case использования Dev Tools для QA](#)
- [Изучаем инструменты разработчика Google Chrome \(ЧАСТЬ 1\)](#)
- [DevTools для «чайников»](#)
- [Devtools для тестировщика - Devtools chrome - Что такое Devtools](#)
- [Chrome DevTools Official Documentation](#)
- [Safari DevTools Official Documentation](#)
- [Полезные функции DevTools для тестировщиков](#)
- [Chrome DevTools. Обзор основных возможностей веб-инспектора](#)

- [Documentation - Chrome DevTools - Remote debugging](#)

Тестирование API:

Postman 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. Постман хорошо подойдет в простых случаях автоматизации или как инструмент поддержки 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](#)
- [Сергей Махетов — Воркшоп: Исследуем возможности Postman \(часть 1\)](#)
- [Сергей Махетов — Воркшоп: Исследуем возможности Postman \(часть 2\)](#)
- [API testing using Postman](#)
- [Postman Beginner's Course - API Testing](#)
- [Погружение qa junior в пучину API с использованием SoapUI\(Open Source\)](#)
- [Swagger Petstore - тренировочный API](#)
- [xml response](#)

Proxy (снiffeры 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](#)
- [Начинающему QA: полезные функции снiffeров на примере Charles Proxy](#)
- [Перехват SSL трафика с Android-приложения](#)
- [mitmproxy is a free and open source interactive HTTPS proxy](#)
- [Charles Proxy meetup](#)

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](#)
- [Santoku Linux](#)
- [Kali Linux](#)
- <https://github.com/FSecureLABS/drozer>

GIT:

Git - это система контроля версий, которая упрощает работу нескольких человек над одним проектом, помогая разрешать конфликты слияния изменений, следить за историей, откатывать эти изменения и т.п.

Ваш репозиторий может быть локальным и/или находиться в: [GitHub](#), [Bitbucket](#), [GitLab](#)

Даже ручному тестировщику пригодятся навыки работы с Git: хранить там портфолио для резюме с подтверждением навыков использования инструментов и написания документации, можно само резюме разместить на github pages, уже на работе иногда будет требоваться самостоятельно сбилдить себе сборку на тест или разобраться, в какой момент (в каком коммите) появился баг или наоборот был пофиксен и т.п. Про автоматизацию, очевидно, даже и говорить не стоит - гит там используется ежедневно.

Все что нужно для работы с GIT

- [Git для тестировщиков](#)
- [Git для новичков \(часть 1\)](#)
- [Git изнутри и на практике](#)
- [Git, я хочу все отменить! Команды исправления допущенных ошибок](#)
- [Getting solid at Git rebase vs. merge](#)
- [Git How To — это интерактивный тур, который познакомит вас с основами Git](#)

SQL:

Это язык программирования, применяемый для создания, модификации и управления данными в базе данных.

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 "Изучаем SQL"](#)
 - [Линн Бейли "Изучаем SQL"](#)
 - [W3C Introduction to SQL](#)
 - [guru99 - SQL Tutorial for Beginners: Learn SQL in 7 Days](#)
- Продвинутый уровень
 - [Энтони Молинаро "SQL. Сборник рецептов"](#)
 - [Алекс Кригель "SQL. Библия пользователя"](#)
 - [Джеймс Грофф, Пол Вайнберг, Эндрю Оппель "SQL Полное руководство. Третье издание."](#)
- Практика
 - [SQLAcademy - Онлайн тренажер с упражнениями по SQL](#)
 - [SQLBolt - Introduction to SQL](#)
 - [W3C - The Try-SQL Editor](#)
 - [HackerRank 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](#)
- [Ресурсы и инструменты для обучения и практической работы с базами данных - SQL](#)
- [The 10 best sql analytics services for qa teams in 2021](#)

Инструменты тестирования мобильных приложений:

- [Android Debug Bridge \(adb\)](#), [Minimal ADB](#), [Инструменты тестирования Android приложений. Часть 2, Отладка по ADB](#)
- [Logcat](#), [Инструменты тестирования Android приложений. Часть 3](#)
- [ANR-WatchDog](#), [Инструменты тестирования Android приложений. Часть 5](#)
- [Performance tracing](#)
- [Xcode profiler](#)
- [On-device developer options](#)
- [apkanalyzer](#)
- [Top 10 Mobile Performance Testing Tools in 2020](#)
- [UI/Application Monkey Tester](#), [Monkey Testing - Как тестировать мобильные приложения](#)
- [Mobile App Beta Testing Services \(IOS And Android Beta Testing Tools\)](#)
- Инструменты скорее разработчика, чем тестировщика, но наверняка когда-то придется столкнуться:
 - Google Firebase: некоторые из самых популярных функций платформы включают в себя базы данных, аутентификацию, push-уведомления, аналитику (в т.ч. по крешам), хостинг и многое другое: [документация](#), [youtube](#), [обзор](#)
 - OneSignal: Лидер на рынке взаимодействия с клиентами, мобильных и веб пушей, электронной почты, SMS и in-app сообщений.

Эмуляторы, симуляторы, фермы устройств:

- [Android studio emulator](#)
- [Genymotion - Android Virtual Devices for all your development & testing needs](#)
- [BrowserStack - Test instantly on a wide range of real iOS and Android devices on the cloud](#)
- [10 лучших альтернатив BrowserStack \(бесплатные и платные\) 2021](#)
- [Xcode simulator](#)
- [Центр приложений Visual Studio](#)
- [Samsung Remote Test Lab](#)
- [AWS Device Farm](#)
- [Huawei cloud debugging](#)
- [Device Farmer is a web application for debugging smartphones, smartwatches and other gadgets remotely](#)
- [Appetize.io - Run native mobile apps in your browser](#)
- [Genymobile/scrcpy - обеспечивает отображение и управление устройствами Android через USB или TCP/IP](#)

Работа с логами:

- [Логи для тестировщика / Работа с логами в тестировании](#)
- [Tools for Log Analysis](#)

Тестирование производительности:

- [Apache JMeter](#)
- [Яндекс.Танк](#)
- [LoadRunner](#)
- [Google Lighthouse](#)
- [artillery.io](#)
- [Top 10 лучших инструментов для нагрузочного тестирования](#)
- [10 инструментов тестирования производительности мобильных приложений](#)

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](#)

- [Mind Map для QA - Интеллектуальные карты](#)

TMS:

- [Allure TestOps](#)
- [Топ-12 лучших систем управления тестированием 2020](#)
- [Инструмент на века - гугл таблицы](#)
- [*Пришла пора отправить в отставку инструменты управления кейсами](#)

Полезные расширения для браузера:

- [Vimbox Переводчик от Skyeng](#)
- [Violentmonkey, Tampermonkey + script](#)
- [Talend API Tester - Free Edition](#)
- [Bug Magnet](#)
- [Dimensions](#)
- [Tape](#)
- [PerfectPixel](#)
- [GoFullPage - Full Page Screen Capture](#)
- [8 Browser Plugins for Testing](#)

Программы для снятия скриншотов и записи видео:

- Windows: скриншот всего экрана Prtsc+Fn, выделяемой части Win+Shift+S, запись видео Win+G
- [Screenpic - больше чем программа для скриншотов](#)
- [Делайте снимки экрана в один клик со Скриншотером Mail.ru](#)
- [ShareX - Screen capture, file sharing and productivity tool](#)
- [Greenshot is a light-weight screenshot software tool for Windows](#)
- [Flameshot - powerful yet simple to use screenshot software](#)
- [Bandicam — это лучшая программа для записи экрана, игр и видеоустройств](#)
- [Recordit - fast screencasts](#)
- [ScreenToGif - screen, webcam and sketchboard recorder with an integrated editor](#)
- [OBS Studio - бесплатная программа с открытым исходным кодом для записи видео и потокового вещания](#)
- [Snagit lets you quickly capture your screen](#)
- [Joxi - бесплатная программа для снятия скриншотов](#)
- [Movavi Screen Recorder - захват экрана в один клик](#)
- [PicPick - захват экрана, редактор изображений, выбор цвета, цветовая палитра, пиксельная линейка, угломер, перекрестье, грифельная доска и многое другое](#)
- [Apowersoft Screen Capture Pro - multi-functional Screenshot Program](#)
- [Screencast-o-matic - Free Screen Recorder](#)

RegExp:

- [Регулярные выражения \(regexp\) — основы](#)
- [Мягкое введение в Regex](#)
- <https://regex101.com/>
- <http://myregexp.com/>
- <https://regexr.com/>

Разное:

- [Полезные ресурсы для тестировщика. Генераторы данных, изображений, текста. Сравнение текста, файлов.](#)
- [Screen Dimensions for Devices + my device](#)
- [Супер простой сервис для генерации разных HTTP-кодов](#)

- [Бесплатные одноразовые e-mail](#)
- [Tools for Software Testing](#)
- [Mock API](#)
- Еще один [mock API](#)
- [Get Credit Card Numbers](#)
- [Тестовые банковские карты](#)
- [Stripe test card numbers](#)
- <https://caniuse.com/ciu/index>
- [Chrome Remote Desktop — теперь подключаемся к ПК и со смартфона на Android](#)
- [Пингуем из Excel](#)
- [Тулзы ручного тестировщика приложений на базе Windows](#)
- [One click website testing tool](#)
- [Инструменты для тестирования - Что должен знать тестировщик без опыта.](#)
- [Генератор номеров банковских счетов](#)
- [Программа для генерации банковских счетов и генерации ключа проверки](#)
- [mChat is a real-time messaging app written in Swift for iOS devices](#)
- [Telegram iOS Source Code Compilation Guide](#)
- [Как установить, настроить и использовать подсистему Linux в Windows 10. Обновленный Windows Terminal](#)
- [Багред - ставите задачу в баг-трекер? Проверьте название на стоп-слова!](#)
- [Top Cross-Browser Testing Tools to Test from Different Geo-Locations](#)
- [10 best data engineering tools and technologies in 2021](#)
- [Кракозябры](#)
- [Прорисовка и визуализация сервисов, систем, архитектуры и всего остального](#)

----- HR-вопросы на собеседовании -----

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?
- Парням: наличие военного билета, девушкам: планы на ближайшие годы по поводу декрета
- Представь, что ты работаешь уже полгода. Опиши свой рабочий день.
- Что если при выполнении задачи понимаешь, что не укладываешься в сроки?
- Что делать, если нет времени на регрессионное тестирование?
- Что делать, если разработчик утверждает, что найденный дефект таковым не является?
- Пришел баг из продакшена, что делаем?
- Какое самое важное влияние оказывает тестировщик на команду разработки? (не продукт!)
- Кто виноват в багах, найденных в процессе регресса?
- Как решать конфликты в удаленной команде?
- Как понять, что тестировщик хорошо сделал свою работу?

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.

Доп. материал:

Доверие между тестировщиками и разработчиками

HR: Пришел баг из продакшена, что делаем?

Воспроизводим, запускаем по пути жизненного цикла дефекта и анализируем причины, как данный дефект прошел в прод: устаревшие кейсы, специфика конкретных устройств или конфигураций которых у тестировщика нет, или это вообще была ситуация срочного релиза и кейсы был но решили выпускать без регрессии и тогда это вопрос к процессам. После исправления дефекта разработчиком проводим повторное тестирование. Добавляем данный дефект в регрессию если его там еще нет. В зависимости от охватываемого функционала и Root Cause этих багов принимается решение о проведении sanity/регрессионного тестирования после подтверждающего.

HR: Кто виноват в багах, найденных в процессе регресса?

Начнем с того, что хорошо, что эти баги нашлись сейчас тестированием, а не после релиза пользователями. Дальше надо уже разбираться отдельно с каждым конкретным багом. Какие могут быть причины:

- Низко-компетентная разработка. В таком случае, баги в процессе регресса будут регулярными. Это может быть связаны с кривым мерджем задач версии, некорректно решенными конфликтами при мерже, невнимательности.
- Плохое качество тестирования. Когда тестировщик во время тестирования своей задачи пропустил какой-то кейс, а он потом всплыл при регрессе у другого тестера.
- Другие недостатки в процессах на проекте. Например, в одном релизе едут 2 фичи, которые как-то пересекаются друг с другом по функционалу, но которые тестировали 2 команды (тут может быть как проблема с коммуникациями в проекте, так и проблема в уровне квалификации тестировщиков и разработчиков, которые не взяли во внимание такое пересечение и менеджеров, которые впихнули 2 фичи в один релиз - такое должно быть допустимо только в крайних случаях и в качестве единичных акций, в остальных случаях стоит разбивать такое на 2 релиза и не рисковать качеством и конечной датой релиза).

Также не стоит забывать про обычный человеческий фактор и если такие ситуации проявляются не часто - то берите во внимание, анализируйте, проговаривайте с командой, но не делайте поспешных выводов и не принимайте радикальных решений.

Источник: https://t.me/qa_chillout/92

HR: Как решать конфликты в удаленной команде?

- Для начала запланируйте встречу с коллегой на определенное время;
- Во время разговора сформулируйте свою позицию в такой последовательности:
 - Озвучьте свое наблюдение: что коллега сделал, что вам не понравилось;
 - Выразите словами свои чувства и свою реакцию;
 - Выразите словами значимую для вас потребность, которая была проигнорирована;
 - Предложите решение или сформулируйте ясный запрос;
- После разговора следует зафиксировать ваши договоренности.

Источник:

Как решать конфликты в удаленной команде

HR: Как понять, что тестировщик хорошо сделал свою работу?

Бывает мнение, что основная задача тестировщиков - сломать продукт, на самом деле тот уже приходит на тестирование с дефектами, одна из задач тестировщика как раз их выявить.

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

- продукт проверен на соответствие требованиям;
- сведено к минимуму количество дефектов, которые обнаружит конечный пользователь;

- представлена отчетность по актуальному качеству продукта заинтересованным лицам.

----- Теоретическая часть -----

Общее

Обеспечение качества (Quality Assurance - QA)

Это часть Quality Management - совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и поддержки ПО, предпринимаемых на разных стадиях жизненного цикла ПО, для обеспечения требуемого уровня качества выпускаемого продукта. Т.е. QA обеспечивает создание правильных процессов для получения в результате качественного продукта. Это также означает создание процессов контроля качества (QC), которые в свою очередь гарантируют, что процессы, установленные QA, соблюдаются. Проще говоря, если тестировщик вступает уже после этапа разработки (не считая, возможно, создания артефактов заранее), то QA участвует в каждом этапе SDLC, начиная еще с составления требований, и занимается не проверкой постфактум уже готового ПО на соответствие требованиям и наличие дефектов, а пытается предотвратить само появление этих дефектов, являясь эдаким "инфлюенсером", специалистом, влияющим на процессы разработки и улучшающим их качество для обеспечения качества итогового продукта.

Если попытаться перечислить некоторые активности, то получится что-то вроде:

- Проверка/тестирование/верификация требований и спецификаций;
- Оценка рисков (Risk assessment);
- Планирование задач по повышению качества продукции;
- Подготовка тестовой документации (сроки, подход (approach), плана тестирования, тест кейсов и чек листов), тестовых сред и данных. По сравнению с QC и тестированием на этом этапе разрабатывается эффективная модель и последовательность проведения различных тестов продукта, а также описываются инструменты и сценарии, которые обеспечат необходимый уровень покрытия функциональности;
- Процесс тестирования продукта;
- Анализ результатов испытаний, отчетов и других приемочных документов.

Перечислите типичные возможные обязанности QA?

- Мониторинг всего процесса разработки;
- Несет ответственность за отслеживание результатов каждого этапа SDLC и корректировку в соответствии с ожиданиями;
- Несет 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.

Источник: [Real Time Software QA Interview Questions And Answers](#)

Доп. материал:

- [QA — специалист по пожарной безопасности вашего проекта](#)
- [Being an Influencer of Quality](#)

- [What is quality engineering?](#)
- [QA in Production](#)
- [Кто такой QA Engineer, QC Engineer и Software Engineer in Test](#)
- [В чем отличие QA-инженеров от тестеров](#)
- [Обеспечение качества - чья это работа?](#)
- [Quality Assurance vs Quality Control vs Testing](#)

Контроль качества (Quality Control - QC)

Это часть QA - процесс установления стандартов и проверки, что ПО сделано правильно. Цель контроля качества - проверить, соблюдалась ли предписанная модель или нет. Это может быть достигнуто путем проведения аудитов и определения того, следовала ли команда определенной модели для достижения качества. Аудит качества - это оценка процесса на месте для обеспечения его соответствия требованиям. Они выполняются под наблюдением аудитора, который проверяет, соблюдались ли установленные руководящие принципы во время создания продукта. Аудиты предназначены не для проверки качества продукта, а для проверки типа работы, выполняемой при создании продукта. Он оценивает, насколько точно соблюдалась предписанная модель. Есть ли вариации? Если да, то в чем причина вариаций? Целью аудитов является постоянное улучшение качества работы, отныне повышая качество продукции. Инспекция может быть одним из аспектов аудита. Инспекция исследует свойства продукта. Он проверяет, насколько хорошо продукт соответствует требованиям, есть ли какие-либо различия между разработанным и желаемым продуктом. Если да, то будет соответствовать требованиям или нет. Сколько продуктов нагрузки / стресса могут выдержать? Какая неблагоприятная ситуация может привести к сбою? Короче говоря, аудит - это проверка качества процесса, используемого при создании продукта. Инспекция - это проверка того, насколько хорошо продукт соответствует требованиям, предъявляемым заинтересованными сторонами. Качество и контроль практикуются в самых разных отраслях, таких как программное обеспечение, производство, автомобилестроение, розничная торговля и т.д., чтобы гарантировать, что все соблюдают стандартную процедуру и практику. Как и при массовом производстве, любое отклонение от стандартной процедуры может привести к ошибкам, которые могут привести к потере огромного количества денег и времени.

Источник: [Testing vs Quality Assurance vs. Quality Control What's the Difference?](#)

Тестирование (Testing)

Пара официальных определений:

- Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и ее ожидаемым поведением на конечном наборе тестов, выбранных определенным образом (ISO/IEC TR 19759:2005).
- Процесс, содержащий в себе все активности жизненного цикла, как динамические, так и статические, касающиеся планирования, подготовки и оценки программного продукта и связанных с этим результатов работ с целью определить, что они соответствуют описанным требованиям, показать, что они подходят для заявленных целей и для определения дефектов (ISTQB Ru);

Это часть QC - процесс проверки того, ведет ли себя спроектированный продукт должным образом в различных условиях. Требования документируются в виде test case. Тестировщик верифицирует и валидирует продукт. Если тестируемое требование к продукту ведет себя так, как ожидалось, test case помечается как пройденный (pass/passed), иначе - как неудачный (fail/failed).

Основная цель тестирования - как можно эффективнее найти дефекты / ошибки. Дефект выявляется тестировщиком для failed scenarios и передается разработчику. После того, как дефект будет исправлен разработчиком, требование проверяется для проверки исправления, и соответствующий test case считается passed. Эти петли обратной связи важны на каждом этапе product delivery lifecycle.

Источник: [Testing vs Quality Assurance vs. Quality Control What's the Difference?](#)

Доп. материал:

- [Что же такое тестирование?](#)
- [Уроки ретроспективного анализа: наука о тестировании](#)

Почему требуется тестирование ПО?

- Процесс тестирования гарантирует, что ПО будет работать в соответствии с ожиданиями клиентов и на имеющемся у них оборудовании;
- Рентабельность - одно из важных преимуществ тестирования программного обеспечения. Своевременное тестирование любого ИТ-проекта поможет вам сэкономить деньги в долгосрочной перспективе. В случае, если ошибки были обнаружены на более раннем этапе тестирования программного обеспечения, их исправление обходится дешевле.
- 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.
- Безопасность - наиболее уязвимое и важное преимущество тестирования ПО. Люди ищут проверенные продукты. Тестирование помогает избавиться от рисков и проблем.

Доп. материал:

- [Мир без QA](#)
- [Продукт без тестирования](#)
- [«Ответственность должна быть на инженерах, которые пишут код». Почему в People.ai отказались от QA-команды и что это дало](#)
- [Why is software testing necessary?](#)

Качество ПО (Software Quality)

Формально стандарт ISO 8402-1986 определяет качество как совокупность функций и характеристик продукта или сервиса, которые обладают способностью удовлетворять явные или неявные требования. Иными словами, качество заключается в соответствии требованиям (conformance to requirements) и пригодности к использованию (fitness for use), т.е. характеризуется набором свойств, определяющих, насколько продукт "хорош" с точки зрения заинтересованных сторон, например, заказчик продукта или пользователь. 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:

- [Качество программного обеспечения \(Software Quality\)](#)
- [Лекция 9: Особенности индустриального тестирования](#)
- [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](#)
- [Качество вместо контроля качества](#)

Принципы тестирования

1. Тестирование демонстрирует наличие дефектов (Testing shows presence of defects)
2. Исчерпывающее тестирование недостижимо (Exhaustive testing is not possible)

3. Раннее тестирование (Early testing)
4. Скопление/кластеризация дефектов (Defect clustering)
5. Парадокс пестицида (Pesticide paradox)
6. Тестирование зависит от контекста (Testing is context dependent)
7. Заблуждение об отсутствии ошибок (Absence of errors fallacy)

Принцип 1. Тестирование показывает наличие дефектов

Тестирование может показать, что дефекты присутствуют, но не может доказать, что дефектов больше нет.

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

Принцип 2. Исчерпывающее тестирование невозможно

Для проведения исчерпывающего тестирования придется протестировать все возможные входные значения и все пути выполнения программы, в большинстве случаев число таких вариаций стремится к бесконечности или просто на порядки превосходит отведенное время и бюджет. 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 SDLC, а именно когда сформированы требования.

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 «куклются». 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

Boris Beizer в своей книге Software Testing Techniques объяснил парадокс пестицида как феномен, согласно которому чем больше вы тестируете ПО, тем более невосприимчивым оно становится к имеющимся тестам, т.е.

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

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

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 не равна валидации.

Доп. материал:

- [Парadox пестицида и поддержка эффективности тест-кейсов](#)
- [The Cold Hard Truth About Zero-Defect Software](#)

Верификация и валидация (Verification and Validation)

Верификация – это проверки, выполняемые в процессе разработки ПО для ответа на вопрос: “правильно ли мы разрабатываем продукт?”. Это в т.ч. включает проверку документации: requirements specification, design documents, database table design, ER diagrams, test cases, traceability matrix и т.д. Верификация гарантирует, что ПО разрабатывается в соответствии со стандартами и процессами организации, полагаясь на [reviews](#) и статические методы тестирования (т.е. без запуска ПО, но, например, с unit/integration tests). Верификация является превентивным подходом (Preventative approach).

| Этап верификации | Действующие лица | Описание | На выходе |
|---|---|---|---|
| Review бизнес / функциональных требований | Команда разработки / клиент для бизнес-требований | Это необходимый шаг не только для того, чтобы убедиться, что требования собраны и / или корректны, но и для того, чтобы убедиться, выполнимы ли они | Завершенные требования, которые готовы к использованию на следующем этапе - дизайне |

| | | | |
|---|--|---|---|
| Review дизайна | Команда разработки | После создания дизайна команда разработчиков тщательно его просматривает, чтобы убедиться, что функциональные требования могут быть выполнены с помощью предложенного дизайна | Дизайн готов к имплементации |
| Прохождение кода (Code Walkthrough) | Отдельный разработчик | Написанный код проверяется на наличие синтаксических ошибок. Это более обыденно и выполняется индивидуальным разработчиком на основе кода, разработанного им самим | Код готов к unit testing |
| Проверка кода (Code Inspection) | Команда разработки | Это уже более формально. Специалисты в данной области и разработчики проверяют код, чтобы убедиться, что он соответствует бизнес-целям и функциональным целям | Код готов к тестированию |
| Test Plan Review (внутренней командой QA) | QA команда | План тестирования проходит внутреннюю проверку командой QA, чтобы убедиться в его точности и полноте | test plan готов к передаче внешним командам (Project Management, Business Analysis, development, Environment, client, etc.) |
| Test Plan Review (внешнее) | Project Manager, Business Analyst, and Developer | Формальный анализ test plan, чтобы убедиться, что график и другие соображения команды QA соответствуют другим командам и всему проекту | Подписанный или утвержденный test plan, на котором будет основываться деятельность по тестированию |
| Test documentation review (Peer review) | Членый команды QA | Экспертная проверка - это когда члены команды проверяют работу друг друга, чтобы убедиться, что в самой документации нет ошибок. | Документация по тестированию готова к передаче внешним командам |

| | | | |
|---------------------------------|--|---|---|
| Test documentation final review | Business Analyst and development team. | A test documentation review чтобы убедиться, что test cases охватывают все бизнес-условия и функциональные элементы системы | Тестовая документация готова к выполнению |
|---------------------------------|--|---|---|

Валидация - это процесс оценки конечного продукта, чтобы проверить, соответствует ли он потребностям бизнеса и ожиданиям клиентов, т.е. отвечает на вопрос: “правильный ли мы разработали продукт?”.

Валидация является динамическим тестированием, т.е. происходит с помощью выполнения кода и прогона тестов на нём (UAT/CAT, usability, всё что угодно). Валидация является реактивным подходом (Reactive approach).

Если попробовать привести очень упрощенный пример, представим блюдо в ресторане. Верификация будет включать проверку технологической карты, оценку процесса приготовления (температуры, времени и т.п.). На протяжении этого процесса можно будет примерно быть уверенным, что блюдо получится именно тем, какое задумывалось и в итоге формально мы его приготовим. Валидация же - это, по сути, попробовать приготовленное блюдо, чтобы удостовериться, действительно ли получилось то, что ожидал бизнес и клиент.

Источник: [Exact Difference Between Verification And Validation With Examples](#)

Доп. материал:

- [В. В. Куламин - Работа на тему “Методы верификации программного обеспечения”](#)
- [Форум тестировщиков: Verification & Validation - что это такое?](#)

Дефекты и ошибки

Прежде всего, стоит разобраться с терминологией. В определениях Error/Mistake/Defect/Bug/Failure/Fault три из них переводятся на русский язык как ошибка. Определения из ISTQB:

- *Просчет (mistake): См. ошибка;*
- *Помеха (bug): См. дефект;*
- *Недочет (fault): См. дефект;*
- *Ошибка (error): Действие человека, которое приводит к неправильному результату;*
- *Дефект (defect): Изъян в компоненте или системе, который может привести компонент или систему к невозможности выполнить требуемую функцию, например неверный оператор или определение данных. Дефект, обнаруженный во время выполнения, может привести к отказам компонента или системы;*
- *Отказ (failure): Отклонение компонента или системы от ожидаемого выполнения, эксплуатации или результата.*

Неофициальные же источники показывают более широкую картину:

- Ошибка (Error) возникает из-за просчета (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).

Так что же такое баг на практике? Когда мы имеем ситуацию “1 требование = 1 тест-кейс”, то вопрос отпадает сам собой - тест-кейс не прошёл, значит требование реализовано не правильно, значит баг. Но обычно вариантов куда больше:

- работало, но вдруг перестало;
- работает, но неправильно;

- реализация не соответствует описанию и в задаче в явном виде не зафиксированы корректировки;
- нужно изменить название кнопки/страницы/раздела, потому что в них есть опечатка или “Отменить отмену” (классика!);
- опечатки в принципе (легко может иметь разный приоритет в зависимости от целей и задач проекта);
- после сохранения информация не появляется на странице, даже если в консоли 200 OK;
- не все указанные при сохранении поля отображаются на странице, но поля неизменно показываются при редактировании;
- при нажатии на кнопку “УДАЛИТЬ ВООБЩЕ ВСЕ ДАННЫЕ КЛИЕНТА” нет модального окна с подтверждением Да/Нет, да и сделать это может любой пользователь без авторизации, который нашел ссылку;
- по переходу по прямой ссылке на услугу не проверяется какой пользователь сейчас авторизован и таким образом можно посмотреть чужие профили или детали услуг, если подобран валидный id;
- можно cURL’ом заказать услугу другому клиенту или в Elements через DevTools изменить стоимость в корзине (не проворачивайте такие сценарии не на своих рабочих проектах);
- информация торчит за границами своего блока или “наслаивается” на другой (ж-ж-ж-жуть, но на некоторых проектах этим можно легко пренебречь);
- страница очень долго открывается, ну о-о-очень долго — секунд 30 на стабильном интернете (взбешенный клиент гарантирован);
- система делает что-то, что она не должна делать согласно изначальной задумке. Например, закрытие аккаунта не только переводит его в статус “Закрыто”, но и возвращает клиенту все деньги, которые он принес проекту за всё время сотрудничества за уже оказанные услуги (о-о-ой!);
- неудобно пользоваться. Например, чтобы посмотреть детали услуги клиента, нужно зайти на три вкладки вглубь аккаунта, а смотреть нужно 2–3 раза в день. Или неудобно копировать информацию со страницы, а по рабочим вопросам это нужно делать несколько раз в день — это баг интерфейса и он должен быть исправлен.

При этом часто может возникнуть извечный вопрос “баг или фича?”, когда баг-репорт заводить не нужно. Это фича-реквест, если:

- нужно изменить название кнопки/страницы/раздела, потому что есть ощущение, что оно не отражает действительности;
- фичу сделали, но после использования видно, что есть простор для существенных улучшений. Например, по услуге не хватает мониторинга или статистических данных по использованию, а за перерасход может взиматься дополнительная плата — клиент точно будет несчастлив в неведении;
- знаете как улучшить ту или иную часть системы, чтобы было удобней. Например, меню необоснованно занимает 30% ширины экрана, а полезная информация ютится на оставшихся 70%;
- пользователь регулярно делает рутинные монотонные действия, которые можно автоматизировать. Например, копировать однотипную информацию с 12 страниц пагинации, когда простая выгрузка бы решила проблему;
- изобретаете велосипед из действующих фич продукта, чтобы добиться желаемого результата;
- на странице не хватает какой-то информации или возможности её добавить;
- на странице не хватает фильтров и пагинации, когда информации много и трудно найти нужное или отображение 1000+ элементов существенно сказывается на скорости загрузки страницы;
- пользователь ведет дополнительную отчетность в блокноте/экселе, когда проблему можно решить выводом ID на странице и несколькими фильтрами.

Хорошо если в команде есть UX/UI дизайнер, а если нет? Тестирующему стоит различать что в дизайне баг, который может привести к печальным последствиям, а что запрос на улучшение, который сделает взаимодействие пользователей с системой более гладким и удобным, но может быть реализован позднее.

Классификация дефектов

Дефекты можно классифицировать по-разному. Для организации важно следовать единой схеме классификации и применять ее ко всем проектам. Некоторые дефекты можно отнести к нескольким классам

или категориям. Из-за этой проблемы разработчики, тестировщики и сотрудники SQA должны стараться быть максимально последовательными при записи данных о дефектах.

Классы дефектов:

- **Дефекты требований и спецификаций** (Requirements and Specifications Defects): Начало жизненного цикла программного обеспечения важно для обеспечения высокого качества разрабатываемого программного обеспечения. Дефекты, введенные на ранних этапах, очень трудно устранить на более поздних этапах. Поскольку многие документы с требованиями написаны с использованием представления на естественном языке, они могут стать
 - Двусмысленными;
 - Противоречивыми;
 - Непонятными;
 - Избыточными;
 - Неточными.

Некоторые специфические дефекты требований / спецификаций:

- Дефекты функционального описания: Общее описание того, что делает продукт и как он должен себя вести (входы / выходы), неверно, двусмысленно и / или неполно;
- Дефекты функций: описываются как отличительные характеристики программного компонента или системы. Дефекты функций связаны с отсутствием, неправильным, неполным или ненужным описанием функций;
- Дефекты взаимодействия функций: это происходит из-за неправильного описания того, как функции должны взаимодействовать друг с другом;
- Дефекты описания интерфейсов: это дефекты, которые возникают в описании взаимодействия целевого программного обеспечения с внешним программным обеспечением, оборудованием и пользователями.
- **Дефекты дизайна:** Дефекты дизайна возникают когда неправильно спроектированы: Системные компоненты, Взаимодействие между компонентами системы, Взаимодействие между компонентами и внешним программным / аппаратным обеспечением или пользователями. Они включают дефекты в конструкции алгоритмов, управления, логики, элементов данных, описаний интерфейсов модулей и описаний внешнего программного обеспечения / оборудования / пользовательского интерфейса. К дефектам дизайна относятся:
 - Алгоритмические дефекты и дефекты обработки: это происходит, когда этапы обработки в алгоритме, описанном псевдокодом, неверны;
 - Дефекты управления, логики и последовательности: Дефекты управления возникают, когда логический поток в псевдокоде неверен;
 - Дефекты данных: Они связаны с неправильным дизайном структур данных;
 - Дефекты описания интерфейсов модулей: эти дефекты возникают из-за неправильного или непоследовательного использования типов параметров, неправильного количества параметров или неправильного порядка параметров;
 - Дефекты функционального описания: к дефектам этой категории относятся неправильные, отсутствующие или неясные элементы дизайна;
 - Дефекты описания внешних интерфейсов: они возникают из-за неправильных описаний дизайна интерфейсов с компонентами COTS, внешними программными системами, базами данных и аппаратными устройствами.
- **Дефекты кода:** Дефекты кодирования возникают из-за ошибок при реализации кода. Классы дефектов кодирования аналогичны классам дефектов дизайна. Некоторые дефекты кодирования возникают из-за непонимания конструкций языка программирования и недопонимания с разработчиками.
 - Алгоритмические дефекты и дефекты обработки:
 - Непроверенные условия overflow and underflow;
 - Сравнение несоответствующих типов данных;

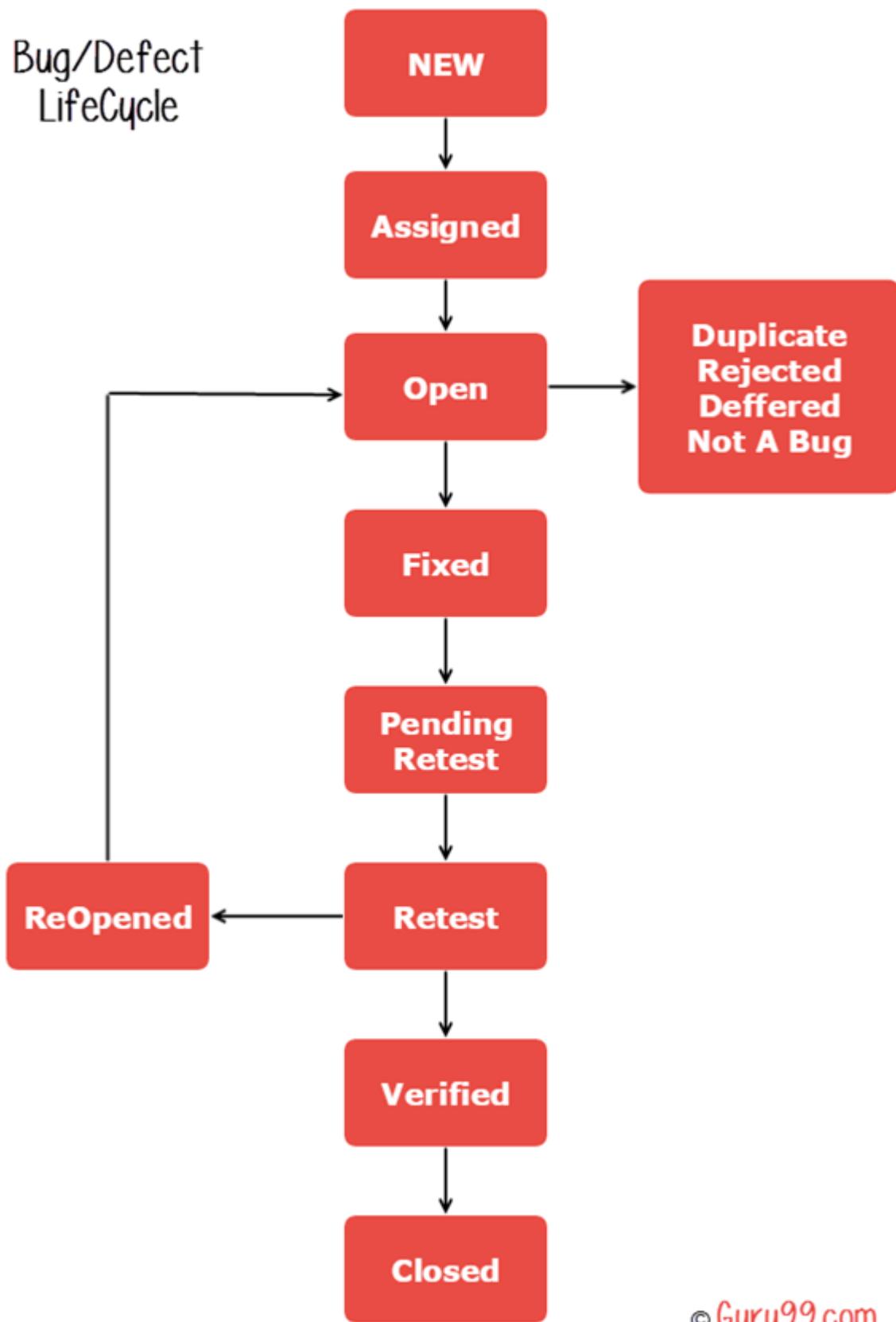
- Преобразование одного типа данных в другой;
- Неправильный порядок арифметических операторов;
- Неправильное использование или пропуск круглых скобок;
- Потеря точности (Precision loss);
- Неправильное использование знаков.
- Дефекты управления, логики и последовательности: этот тип дефектов включает неправильное выражение операторов case, неправильное повторение циклов и пропущенные пути;
- Типографические дефекты: в основном это синтаксические ошибки, например неправильное написание имени переменной, которые обычно обнаруживаются компилятором, self-reviews, or peer reviews;
- Дефекты инициализации: этот тип дефектов возникает, когда операторы инициализации пропущены или неверны. Это может произойти из-за недопонимания или отсутствия связи между программистами или программиста и дизайнера, небрежности или непонимания среды программирования;
- Дефекты потока данных: дефекты потока данных возникают, когда код не следует необходимым условиям потока данных;
- Дефекты данных: на это указывает неправильная реализация структур данных;
- Дефекты интерфейса модуля: возникают из-за использования неправильных или несовместимых типов параметров, неправильного количества параметров или неправильного порядка параметров;
- Дефекты документации кода: когда документация по коду не описывает, что программа на самом деле делает, либо является неполной или двусмысленной;
- Внешнее оборудование, дефекты программных интерфейсов: эти дефекты возникают из-за проблем, связанных с Системными вызовами, Ссылками на базы данных, Последовательностью ввода / вывода, Использованием памяти, Использованием ресурсов, Обработкой прерываний и исключений, Обменом данными с оборудованием, Протоколами, Форматами, Интерфейсами с файлами сборки, Временными последовательностями.
- **Дефекты тестирования:** Планы тестирования, тестовые наборы, средства тестирования и процедуры тестирования также могут содержать дефекты. Эти дефекты называются дефектами тестирования. Дефекты в планах тестирования лучше всего обнаруживать с помощью методов review.
 - Дефекты тестовой обвязки: Для тестирования программного обеспечения на уровне модулей и интеграции необходимо разработать вспомогательный код. Это называется Test Harness или scaffolding code. Test Harness должен быть тщательно спроектирован, реализован и протестирован, поскольку это рабочий продукт, и этот код можно повторно использовать при разработке новых версий программного обеспечения;
 - Дизайн тестового случая и дефекты процедуры тестирования: сюда входят неправильные, неполные, отсутствующие, несоответствующие тестовые примеры и процедуры тестирования.

[В англоязычной Wikipedia описано плюс-минус то же самое.](#)

Жизненный цикл дефекта (Defect/Bug Life Cycle)

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 и настраиваться под процессы конкретного проекта.

Bug/Defect LifeCycle

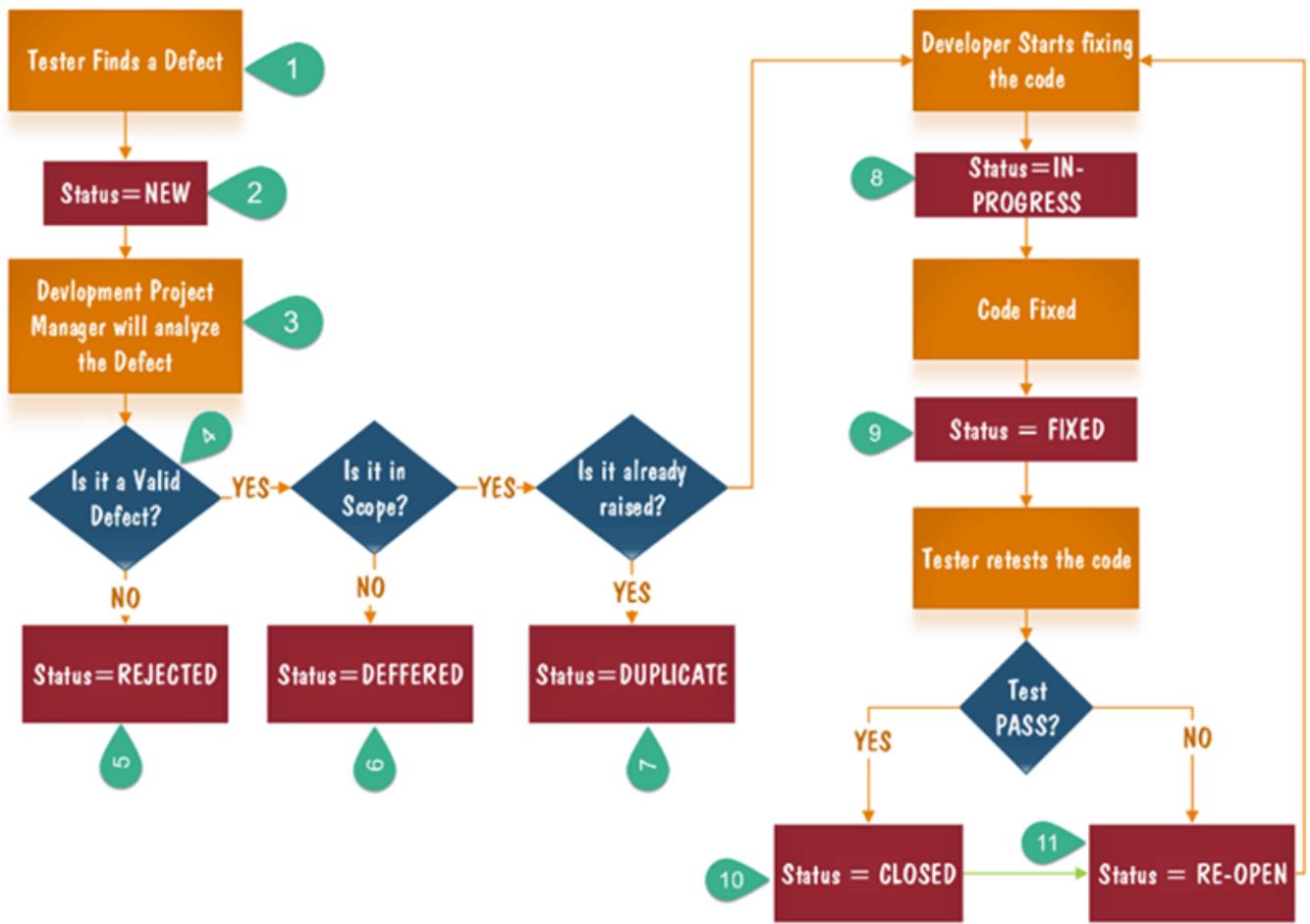


© Guru99.com

Статусы дефекта:

- **Новый (New)**: когда новый дефект регистрируется и публикуется впервые;
- **Назначен (Assigned)**: после публикации бага тестировщиком руководитель тестировщика утверждает ошибку и передает ее команде разработчиков;
- **Открыт (Open)**: разработчик начинает анализ и работает над исправлением бага;
- **Исправлен (Fixed)**: разработчик внес необходимое изменение в код и проверил его;

- **Ожидает повторного тестирования** (Pending retest): как только дефект будет исправлен, разработчик предоставляет тестировщику конкретный код для повторного тестирования кода. Поскольку тестирование программного обеспечения остается незавершенным со стороны тестировщиков, ему присваивается статус «ожидает повторного тестирования»;
- **Повторное тестирование** (Retest): на этом этапе тестировщик выполняет повторное тестирование кода, чтобы проверить, исправлен ли дефект разработчиком;
- **Проверен** (Verified): тестировщик повторно тестирует баг после его исправления разработчиком. Если баг исправлен, то присваивается статус «проверено»;
- **Переоткрыт** (Reopen): если баг сохраняется даже после того, как разработчик исправил баг, тестировщик меняет статус на «повторно открыт». И снова баг проходит жизненный цикл.
- **Закрыт** (Closed): если баг больше не существует, тестировщик присваивает статус «Закрыто».
- **Дубль** (Duplicate): если дефект повторяется дважды или дефект соответствует той же концепции ошибки, статус изменяется на «дублировать».
- **Отклонен** (Rejected): если разработчик считает, что дефект не является таковым, он меняет статус на «отклонен»;
- **Отложен** (Deferred): если текущий баг не является приоритетным и ожидается, что он будет исправлен в следующем выпуске, таким багам присваивается статус «Отложено»;
- **Не является багом** (Not a bug): если это не влияет на функциональность приложения, то багу присваивается статус «Не является багом».



Утечка дефектов и релиз бага (Bug Leakage & Bug Release)

Утечка бага (Bug Leakage): возникает когда пропускается баг в билде, который вышел в Production. Если баг был обнаружен конечным пользователем или заказчиком, мы называем это утечкой ошибок.

Выпуск бага (Bug release): выпуск программного обеспечения в Production с некоторыми известными багами. Эти известные баги следует включить в примечания к выпуску (release notes). Другой вариант - передача программного обеспечения группе тестирования с некоторыми известными багами, серьезность и приоритет которых невысоки. Эти ошибки можно исправить перед выпуском в Production.

Основное отличие отладки от тестирования (Debugging Vs. Testing)

После того, как разработчик получил баг-репорт, он приступает к исправлению бага. Но, прежде чем ошибку исправить, нужно ее воспроизвести, понять, как она происходит и где ее найти в коде. Дебаг, буквально “de”+“bug” - это и есть процесс поиска и устранения ошибок в коде. Специальная debug-версия билда приложения может иметь расширенный вывод для более информативных логов или любые другие модификации для упрощения понимания проблемы. Тактика отладки может включать интерактивную отладку, анализ потока управления, модульное тестирование, интеграционное тестирование, анализ логов, мониторинг на уровне приложения или системы, дампы памяти и профилирование. Многие языки программирования и инструменты разработки программного обеспечения также предлагают программы для помощи в отладке, известные как отладчики/дебаггеры.

Маскировка дефектов (Defect masking)

Ситуация, когда наличие одного дефекта скрывает присутствие другого дефекта в системе.

Скрытый дефект (Latent defect)

Дефект, который является существующим дефектом в системе, но еще не вызывал сбоев, поскольку подходящий набор входных данных для его проявления не был введен или его проявлению мешает другой дефект (Defect masking).

Сортировка дефектов (Bug triage)

Это формальный процесс определения серьезности и приоритета дефектов в зависимости от их severity, риска, повторяемости и т. д. во время Defect Triage Meeting. Такая встреча полезна в условиях ограниченных ресурсов, когда нужно разобраться с множеством ошибок и тем, какие из них приоритетные.

Понятие сортировки пришло из медицины, где это процесс быстрого обследования пациентов, доставленных в больницу, чтобы решить, какие из них наиболее серьезно больны и нуждаются в лечении в первую очередь. В тестировании мы используем ту же концепцию к ошибкам, обнаруженным на этапе тестирования.

Источники:

- [Баг или фича? Вот в чём вопрос!](#)
- [Defect Classes, the Defect Repository, and Test Design](#)
- [Defect/Bug Life Cycle in Software Testing](#)
- [Real Time Software QA Interview Questions And Answers](#)
- [Top 150 Software Testing Interview Questions and Answers for Freshers and Experienced](#)
- [Defect Triage Process in Software Testing](#)

Серьезность и приоритет Дефекта (Severity & Priority)

Bug management включает в себя процесс документирования, категоризации, назначения, воспроизведения, исправления и выпуска исправленного кода. Предлагаемые изменения в программном обеспечении - баги, запросы на улучшения и даже целые [релизы](#) - обычно отслеживаются и управляются с помощью баг-трекинговых систем. Добавленные элементы могут называться дефектами, заявками, проблемами или, в соответствии с парадигмой гибкой разработки, эпиками и историями (stories and epics). Категории могут быть объективными, субъективными или комбинированными, такими как номер версии, область программного обеспечения, серьезность и приоритет, а также тип проблемы, такой как фича-реквест или баг.

Критичность (severity): Важность воздействия конкретного дефекта на разработку или функционирование компонента или системы. (IEEE 610)

Приоритет (priority): Степень важности, присваиваемая объекту. Например, дефекту. (ISTQB)

Иными словами, серьезность представляет техническое влияние ошибки в контексте работоспособности самого ПО, а приоритет указывает на очередность выполнения задачи или устранения дефекта, т.е. точку зрения бизнеса. Приоритет выставляется любыми business stakeholders, включая project managers, business

analysts, product owner, а серьезность сам тестировщик (или в сложных случаях тот, кто лучше разбирается). Разработчик берет таски исходя из приоритета.

Градация Серьезности (Severity):

- Критическая (critical) — существование дефекта приводит к масштабным последствиям катастрофического характера, например: потеря данных, раскрытие конфиденциальной информации, нарушение ключевой функциональности приложения и т.д.;
- Высокая (major) — существование дефекта приносит ощутимые неудобства многим пользователям в рамках их типичной деятельности, например: недоступность вставки из буфера обмена, неработоспособность общепринятых клавиатурных комбинаций, необходимость перезапуска приложения при выполнении типичных сценариев работы;
- Средняя (medium) — существование дефекта слабо влияет на типичные сценарии работы пользователей, и/или существует обходной путь достижения цели, например: диалоговое окно не закрывается автоматически после нажатия кнопок «OK»/«Cancel», при распечатке нескольких документов подряд не сохраняется значение поля «Двусторонняя печать», перепутаны направления сортировок по некоему полю таблицы;
- Низкая (minor) — существование дефекта редко обнаруживается незначительным процентом пользователей и (почти) не влияет на их работу, например: опечатка в глубоко вложенном пункте меню настроек, некое окно сразу при отображении расположено неудобно (нужно перетянуть его в удобное место), неточно отображается время до завершения операции копирования файлов.

Градация Срочности/приоритета (Priority):

- Наивысшая (ASAP, as soon as possible) срочность указывает на необходимость устранить дефект настолько быстро, насколько это возможно. В зависимости от контекста «настолько быстро, насколько возможно» может варьироваться от «в ближайшем билде» до единиц минут;
- Высокая (high) срочность означает, что дефект следует исправить вне очереди, т.к. его существование или уже объективно мешает работе, или начнёт создавать такие помехи в самом ближайшем будущем;
- Обычная (normal) срочность означает, что дефект следует исправить в порядке общей очередности. Такое значение срочности получает большинство дефектов;
- Низкая (low) срочность означает, что в обозримом будущем исправление данного дефекта не окажет существенного влияния на повышение качества продукта.

Сочетания Severity и Priority

- **High Priority and High Severity:** Любой Critical/major сбой бизнес-модели, критическая проблема, при которой полностью не работает большая часть функциональности или основной компонент системы:
 - нажатие на определенную кнопку не запускает саму функцию, например, не работает кнопка отправки на странице входа, и клиенты не могут войти в приложение;
 - выполнение определенной функции постоянно приводит к 500 ошибке сервера и потере данных;
 - система дает сбой после того, как вы совершили платеж или когда вы не можете добавить товары в корзину;
 - функция банкомата, при которой после ввода правильного имени пользователя и пароля автомат не выдает деньги, но списывает их с вашего счета;
 - на веб-сайте банка появляется сообщение об ошибке, когда клиент нажимает кнопку перевода денег.
- **High Priority and Low Severity:** Любые minor severity дефекты, которые влияют на взаимодействие с пользователями / репутацию:
 - ожидается, что функция покажет пользователю конкретную ошибку по коду ответа. В этом случае функционально код выдает ошибку, но сообщение должно быть более релевантным коду;

- ошибка в логотипе или названии компании на главной странице, или опечатки, бросающиеся в глаза и способные повлиять на репутацию компании;
 - опечатки в контактных данных;
 - важные ошибки в соглашениях и юридических документах.
- **Low Priority and High Severity:** Проблема, которая пока не влияет на бизнес, но имеет большое влияние с точки зрения функциональности:
 - присутствует серьезный баг, но есть workaround и исправление уже может быть запланировано в следующем релизе или функция будет удалена;
 - функция генерации годового отчета, которая будет использована только через полгода;
 - редкость проявления дефекта/сложность воспроизведения для юзеров.
- **Low Priority and Low Severity:** Любые орфографические ошибки / начертание / несовпадение шрифта в абзаце 3-й или 4-й страницы заявки, а не на главной или титульной странице / заголовке. Эти дефекты возникают, когда это не влияет на функциональность, но все же в небольшой степени не соответствует стандартам. Обычно сюда классифицируются косметические ошибки или, скажем, размеры ячеек в таблице пользовательского интерфейса:
 - в политике конфиденциальности веб-сайта есть орфографическая ошибка;
 - страница часто задаваемых вопросов загружается очень долго;
 - семейство шрифтов, размер шрифта, цвет или орфографическая ошибка в приложении или отчетах.

Источники:

- [Святослав Куликов “Тестирование программного обеспечения. Базовый курс”](#). Раздел 2.5
- [Defect Severity And Priority In Testing With Examples And Difference](#)
- [Difference Between Defect Severity And Priority In Software Testing](#)

Подход к тестированию (Test Approach)

Подход к тестированию - это реализация стратегии тестирования для конкретного проекта.

Подход к тестированию определяется и уточняется в test plans and test designs. Подход к тестированию обычно включает решения, принимаемые на основе цели (тестового) проекта и оценки рисков (risk assessment). Подход к тестированию является отправной точкой для планирования процесса тестирования, для выбора применяемых методов проектирования тестов и типов тестов, а также для определения критериев начала и окончания тестирования. Выбранный подход зависит от контекста и может учитывать риски, опасности и безопасность, доступные ресурсы и навыки, технологии, характер системы (например, [custom built](#) vs. [commercially available off-the-shelf \(COTS\)](#)), цели тестирования (test objectives) и правила.

Подход к тестированию включает две техники:

- Упреждающий (Proactive) - подход, при котором test design process запускается как можно раньше, чтобы найти и исправить дефекты до создания сборки (build);
- Реактивный (Reactive) - подход, при котором тестирование не начинается до завершения проектирования и разработки.

Различные подходы к тестированию:

- **Аналитические подходы (Analytical approaches)**, такие как risk-based testing, когда тестирование направлено на области наибольшего риска;
- **Подходы на основе моделей (Model-based approaches)**, такие как стохастическое тестирование с использованием статистической информации о частоте отказов (например, модели роста надежности) или использовании (например, рабочие профили);
- **Методические подходы (Methodical approaches)**, такие как основанные на отказах (failure-based) (включая error guessing and fault attacks), основанные на опыте, на основе чек-листов и на основе характеристик качества (experience-based, checklist-based, and quality characteristic-based);
- **Подходы, соответствующие процессам или стандартам (Process- or standard-compliant approaches)**, например, указанные в отраслевых стандартах или различных гибких методологиях;

- **Динамические и эвристические подходы (Dynamic and heuristic approaches)**, такие как exploratory testing, при котором тестирование более реагирующее (reactive) на события, чем при запланированном заранее (pre-planned), и где выполнение и оценка (execution and evaluation) являются параллельными задачами;
- **Консультативные подходы (Consultative approaches)** - подходы, при которых test coverage определяется в первую очередь советами и руководством экспертов в области технологий и / или бизнеса, не входящих в группу тестирования;
- **Подходы против регрессии (Regression-averse approaches)** - подходы, которые включают повторное использование существующего тестового материала, обширную автоматизацию функциональных регрессионных тестов и стандартные наборы тестов.

Можно комбинировать разные подходы, например, динамический подход, основанный на оценке риска.

Источник:

[ISTQB Foundation - 5.2.6 Test Strategy, Test Approach](#)

Тестовое покрытие (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 с близким к 100% покрытием кода, в котором по-прежнему остаются баги и проблемы. В реальности нужно следить за другими метриками — хорошо известными показателями СТМ (Codepipes testing Metrics).

| Название метрики | Описание | Идеальное значение | Обычное значение | Проблемное значение |
|------------------|---|--------------------|------------------|---------------------|
| PDWT | Процент разработчиков, пишущих тесты | 100% | 20%-70% | Любое меньше 100% |
| PBCNT | Процент багов, приводящих к созданию новых тестов | 100% | 0%-5% | Любое меньше 100% |
| PTVB | Процент тестов, проверяющих поведение | 100% | 10% | Любое меньше 100% |
| PTD | Процент детерминированных тестов | 100% | 50%-80% | Любое меньше 100% |

- **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 выигрывает 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

Источники:

[Антипаттерны тестирования ПО](#)

Доп. материал:

- [Лекция 4: Оценка оттестированности проекта: метрики и методика интегральной оценки](#)
- [Самый полный список метрик тестирования на русском языке](#)
- [Тестовое покрытие по Бейзеру // Бесплатный урок OTUS](#)

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.

Есть 3 типа импакт анализа:

- Анализ влияния зависимостей (Dependency impact analysis) фокусируется на обнаружении зависимостей: потенциальных последствий изменений или частей продукта, которые необходимо переработать при реализации этих изменений
- Эмпирический анализ влияния направлен на оценку рисков, связанных с изменениями продукта, с точки зрения всего процесса разработки, включая потребность в дополнительном времени и ресурсах для разработки
- Анализ влияния прослеживаемости (Traceability impact analysis), согласно определению в глоссарии ISTQB, оценивает, что необходимо изменить на разных уровнях документации, чтобы внести конкретное изменение в продукт.

...

Источники:

- [Dependency Impact Analysis in Software Testing and Development: What It Is and How to Do It](#)
- [Impact Analysis: 6 шагов, которые облегчат тестирование изменений](#)

Доп. материал:

[The Rise of Test Impact Analysis](#)

[Анализ первопричин \(RCA - Root Cause Analysis\)](#)

Любой процесс, неважно, разработка это или тестирование, сопровождение, управление качеством и т.д., всегда должен быть цикличен. Существуют 4 основных подхода к работе с процессами, и самый популярный из них, это уже общепризнанный цикл Деминга. Именно на его основе строится работа процесса и все другие методологии, такие как DMAIC (6 сигм), IDEAL, EFQM, которые всегда говорят нам о том, что нужно не только требовать выполнение процесса, но и постоянно его анализировать и непрерывно совершенствовать. Эти модели позволяют нам понять, как мы должны работать с процессом, и самое главное, мы должны всегда видеть проблемы нашего процесса и стараться их решить.

Говоря о тестировании, существуют 2 основополагающих подхода к совершенствованию процесса тестирования, это MBI и ABI.

MBI или Model Based Improvement – подход к совершенствованию процесса тестирования, который основан на референтных моделях совершенствования процесса тестирования. Модели могут быть процессные, такие как TMMi, TPI и контекстные, такие как STEP или СТР. Эти модели позволяют нам на основе практик строить наш процесс тестирования по конкретным шагам, тем самым развивая процесс тестирования равномерно и последовательно.

Но подходят ли нам такие модели для аудитов уже существующих процессов?

Основная проблема в том, что каждый процесс тестирования различается в зависимости от организации, что ставит под сомнение применение тех практик, которые дает нам модельный подход. Ну и многие специалисты, которые проводили именно аудит процесса тестирования возможно слышали фразу, ставящую в тупик все результаты вашей работы: "Я могу сейчас взять ваши результаты, принести их в другую компанию и они тоже там будут применимы. Нет конкретики!" И все это потому, что многие руководители, тест-менеджеры,

особенно в России, не понимают различия между аудитом и оценкой уровня зрелости процесса тестирования.

Аудит – это анализ текущего состояния процесса с целью решения конкретных проблем, не позволяющих процессу выполнять поставленные задачи.

Оценка зрелости – это анализ текущего состояния процесса с целью понимания его зрелости относительно общепринятых моделей процесса тестирования. Оценка уровня зрелости зачастую может не решать вообще никаких проблем, т.к. ее задача только оценить процесс.

Поэтому, говоря о модельном подходе MBI разумно его применять только для выполнения задач по оценке уровня зрелости процесса тестирования и написанию стратегии развития процесса тестирования на длительных срок. Во всех остальных случаях, а особенно, когда вам нужно решить какую-то проблему, MBI не поможет вам. Для этого существует подход ABI.

ABI или Analytical Based Improvement – это подход к совершенствованию процесса тестирования, который основывается на аналитических подходах анализа процесса.

Главное отличие модельного подхода от аналитического в том, что когда мы анализируем процесс по MBI, то мы анализируем процесс сверху вниз, то есть, сначала мы смотрим на процесс в целом, потому делим его на области, этапы и т.д., тем самым постепенно погружаясь в детали процесса. Аналитический подход работает наоборот, мы сначала погружаемся в самые детали нашего процесса и уже потом идя, как по цепочке, вверх, доходим до решения нашей главной проблемы.

Существуют несколько аналитических подходов к совершенствованию процессов, но я ни разу не видел их применение к процессу тестирования, т.к. они содержат именно модель анализа и могут быть применимы, в том числе, и в производстве фабрики, расследовании аварий, построении мостов и многом другом.

Задайте себе вопрос, как часто к вам приходил ваш руководитель со словами, у нас есть такая проблема в процессе и ее нужно решить? Что вы делаете? Вы ищете причину этой проблемы и пытаетесь ее устраниить. Но очень часто бывает так, что вроде вы решаете причину, но процесс все равно не работает как надо. И все это потому, что вы выбрали для решения не то узкое место!

Поэтому для проведения аудита процесса тестирования, с целью решения конкретных проблем стоит обратить свое внимание на Root Cause Analysis.

Анализ первопричин (RCA) - это систематический процесс выявления скрытых первопричин проблем или событий и подхода к их устранению. В основе RCA лежит основная идея о том, что эффективное управление требует большего, чем просто «тушение пожаров» возникающих проблем, но и поиск способов их предотвращения. Таким образом RCA представляет собой древовидную иерархическую структуру зависимости причин, как с проблемой, так и между собой.

Стандартно, к проблемам процесса тестирования, я отношу только те проблемы, которые связаны с классическим треугольником – цена/качество/сроки. Почему это так? Любой процесс ИТ, в том числе и тестирование, должен обеспечивать бизнес организации. ИТ – это помощник бизнеса, поэтому, когда вам бизнес говорит, что они не успевают внедрять все запланированные финчи, продукты, то это не проблема бизнеса (что они генерят много задач), а проблема тестирования. Все остальное, что не связано с этим “треугольником проблем” является причинами возникновения проблем, которые зачастую бывают непонятны и скрываются от нас.

Процесс аудита по RCA состоит из 5 этапов:

- Определить проблему и ее влияние на общие цели;
- Собрать всю информацию и данные;
- Определить любые инциденты (issues), которые способствовали возникновению проблемы;
- Определить первопричины;
- Определить рекомендации на случай повторения проблем в будущем;
- Реализовать необходимые решения;

Итак, что такое проблема? Проблема – это вопрос или задача, требующая разрешения. Проблемы в нашей работе мы находим постоянно, критичность проблемы мы определяем симптомами, т.е. признаками существования проблемы. Допустим, мы идентифицировали проблему, как постоянный сдвиг сроков внедрения релиза. Признаком существования в проблеме в данном случае может быть систематичность ее возникновения. Я думаю, вы понимаете разницу, один раз у вас произошел сдвиг за 6 релизов, или уже 6-й раз подряд. Во втором случае проблема уже начинает носить критический характер. Решать все проблемы невозможно, поэтому если у Вас есть большое количество проблем, то вы можете выбрать наиболее важные из них по степени влияния и частоте возникновения.

Имея проблему, мы можем приступать к поиску вероятностных причин. Самый простой способ определения причин – метод брейншторма всех вовлеченных в процесс участников. Вы можете собрать со всех специалистов их мнение относительно того, какие они видят причины возникновения проблемы и после этого выбрать из них наиболее часто называемые сотрудниками.

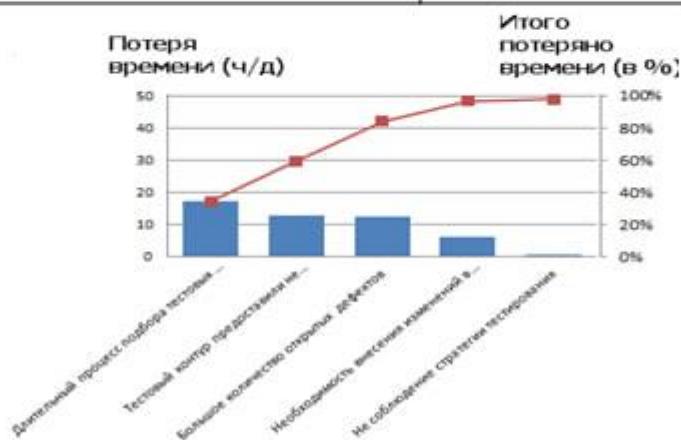
Следующий и очень важный шаг – это анализ вероятностных причин.

Существуют несколько подходов к проведению анализа, таких как диаграммы зависимостей или рассеивания, аффинная диаграмма, но самым распространенным подходом к проведению анализа является диаграмма Парето.

Путем брейншторма мы определили 5 основных причин возникновения проблемы, связанной со сдвигом сроков внедрения. После этого наша задача понять, какие из этих причин наиболее серьезно влияют на нашу проблему. Используя принцип Парето мы определяем количество человеко-дней, которые мы теряем из-за возникновения той или иной проблемы.

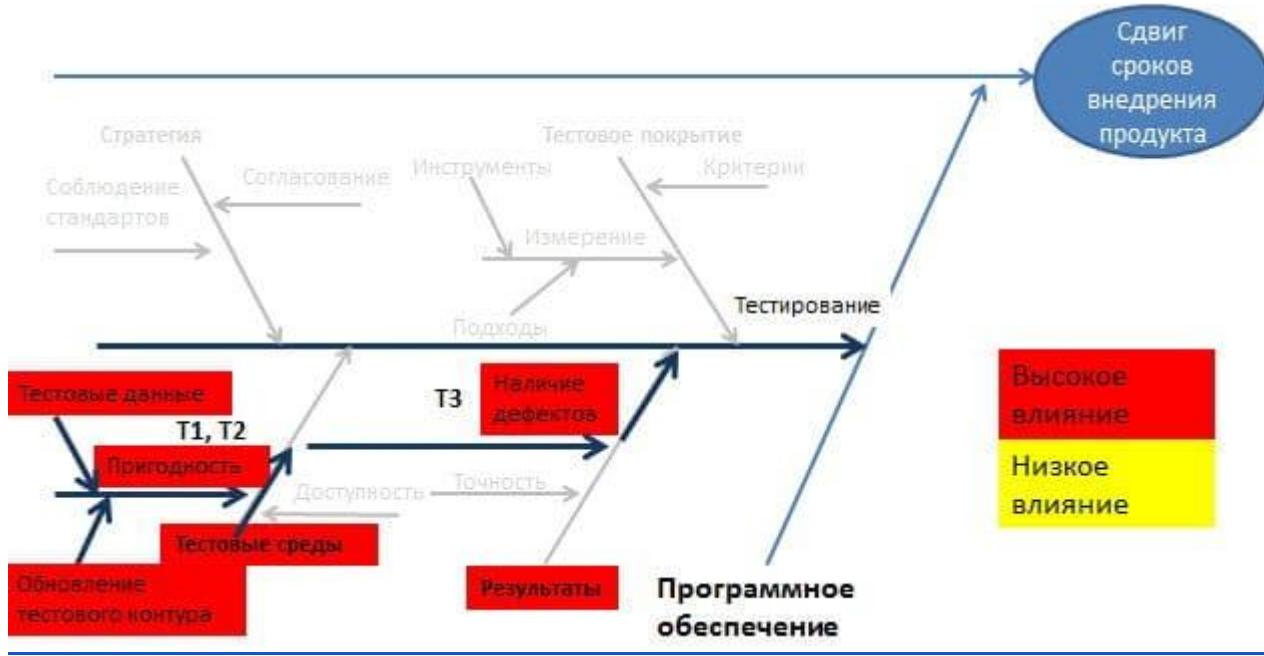
В результате анализа мы видим, что только первые 2 причины на 60% влияют на сдвиг сроков внедрения, что существенно больше, чем остальные 3 причины суммарно.

| Причины сдвига сроков внедрения | Потеря времени в ч/д на команду | Всего потеряно времени |
|--|---------------------------------|------------------------|
| Тестовый контур предоставили не своевременно | 12; 0,5, 0,2 | 12,7 |
| Длительный процесс подбора тестовых данных | 10; 4;3 | 17 |
| Большое количество открытых дефектов | 1;1;0,5;3;5;3 | 12,5 |
| Не соблюдение стратегии тестирования | 0,2;0,4 | 0,6 |
| Необходимость внесения изменений в тестовые сценарии | 1,2;3;2 | 6,2 |



Следующим этапом является проведение причинно-следственного анализа (ПСА) с целью выявления коренных причин и их зависимостей друг с другом. Для этого используется диаграмма причинно-следственного анализа, основная задача которой заключается в том, что идя от проблемы по цепочке мы погружаемся на каждом уровне в причины возникновения причин, тем самым доходя до истинной или коренной причины возникновения проблем. В результате, решив коренную причину, мы автоматически решаем все остальные наши причины, что приводит к минимизации влияния или полного устранения нашей проблемы.

Для выполнения ПСА мы наносим на наше дерево все наши причины, которые были определены ранее принципом Парето. После этого мы их приоритезируем и определяем их взаимозависимость. Например, говоря о тестовых средах, а именно проблемах, связанных с подбором тестовых данных, это приводит к возникновению дефектов "тестирования", что увеличивает сроки выполнения работ по тестированию ПО. Соответственно, решив эту причину, мы автоматически сможем снизить влияние причины, связанной с дефектами. Поэтому, мы можем решать не 3 причины, а всего 2, тем самым сокращая затраты на оптимизацию процесса тестирования.



Ну и заключительный этап – это выработка решений. Проведя анализ RCA решения будут уже вполне понятны, но очень важно, чтобы эти решения действительно были нацелены на конкретную причину и были выполнимы всей командой, на которую ложится их реализация.

Наиболее понятной книгой, рассматривающей модель RCA, я считаю Андерсон Бьерн — «Анализ основной причины. Упрощенные инструменты и методы», в которой на обычных жизненных примерах рассматриваются различные возможности применения модели RCA.

Поэтому, если Вам поставили задачу оптимизировать ваш процесс тестирования, но для этого вам нужно решить какие-то текущие проблемы, то вам нужен аудит с применением ABI, т.к. именно ABI позволяют точечно решать проблемы и любые ваши рекомендации будут носить не рекомендательный характер, а детальный, направленный на оптимизацию именно для вашего проекта или процесса.

Источник:

[Root Cause Analysis. Как минимизировать затраты на оптимизацию процесса тестирования](#)

Доп. материал:

- [Guide To Root Cause Analysis – Steps, Techniques & Examples](#)
- [A Brief History of Root Cause Analysis](#)
- [How to Conduct a Root Cause Analysis](#)

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 ММ - модель зрелости тестирования, которая тоже состоит из 5 уровней. ТММ 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. ТММ это подробная модель для улучшения процесса тестирования. Она может быть дополнена любой моделью улучшения процесса или может использоваться как одиночная модель.

Модель ТММ имеет два основных компонента:

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

Пять уровней ТММ помогают организации определить зрелость своего процесса и определить следующие шаги по улучшению, которые необходимы для достижения более высокого уровня зрелости тестирования.

- 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 управляемое 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.

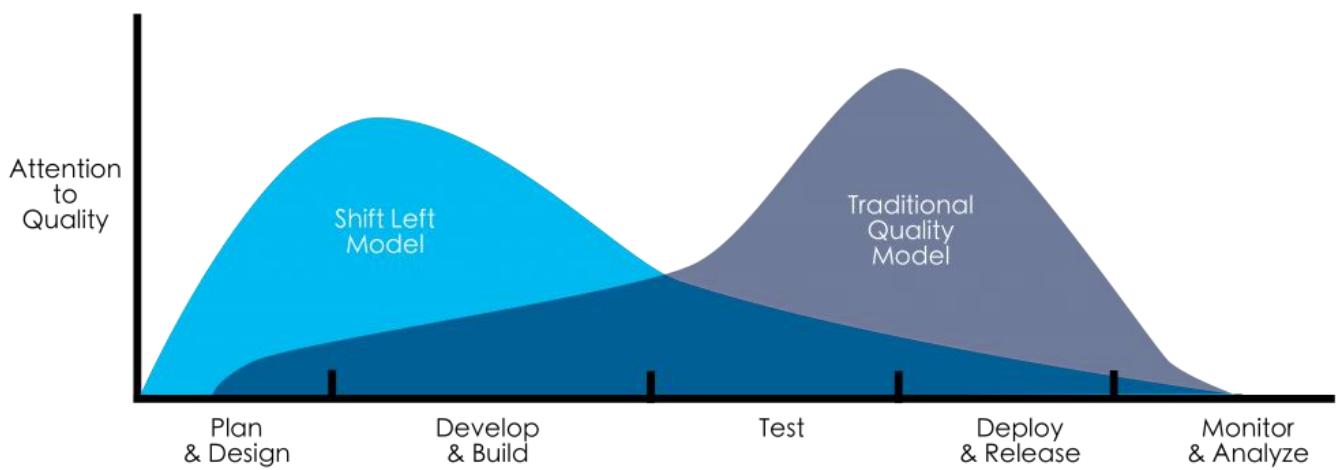
Доп. материал:

- [Модель зрелости тестирования TPI Next: преимущества, недостатки и варианты внедрения](#)
- [Test Maturity Model: как тестировщику оценить проект и спланировать процессы](#)
- [Как достичь Уровня 5 по модели СММ в области QA и тестирования](#)
- [7 подходов к тестированию](#)

Тестирование со сдвигом влево (Shift left testing)

Тестирование со сдвигом влево - это практика включения тестирования на более ранних этапах SDLC. Конечная цель состоит в том, чтобы тестировщики работали вместе с командой разработчиков на ранних этапах разработки, чтобы те могли работать над предотвращением проблем до их возникновения, а не выявлять их в конце проекта. Тестируя на раннем этапе, тестировщики могут сократить время, необходимое

для проведения тестовых спринтов, могут помочь повысить качество программного обеспечения и помочь устранить фатальные проблемы, которые обычно обнаруживаются в конце.



Доп. материал:

- [Меньше «сложного» тестирования, больше — «умного» тестирования](#)
- [Экономим ресурсы и успеваем в срок: зачем подключать QA-инженера в начале работы над фичей](#)
- [Эволюция идеи shift left - shift up and spread: a new testing concept](#)
- [Shift Left Testing Benefits and Approach](#)

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. 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.

Тестирование по уровням независимости:

- 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 упусанию 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.

Источник:

[Independent Testing Guide – How It Delivers Quality Driven Product](#)

Тестирование как сервис (TaaS – testing as a Service)

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.

Источник: [Software Testing as a Service \(TaaS\)](#)

Альфа- и бета- тестирование (Alpha Testing and Beta Testing)

Альфа- и бета-тестирование - это Customer Validation methodologies (Acceptance Testing types) которые помогают укрепить веру в запуске продукта и, таким образом, привести к успеху продукта на рынке. Несмотря на то, что они оба полагаются на реальных пользователей и обратную связь разных команд, ими движут разные процессы, стратегии и цели. Эти два типа тестирования вместе увеличивают успех и продолжительность жизни продукта на рынке. Эти этапы можно адаптировать к продуктам Consumer, Business или Enterprise. Этапы альфа- и бета-тестирования в основном сосредоточены на обнаружении ошибок в уже протестированном продукте и дают четкое представление о том, как продукт на самом деле используется пользователями в реальном времени. Они также помогают получить опыт работы с продуктом перед его запуском, а ценные отзывы эффективно используются для повышения удобства использования продукта. Цели и методы альфа- и бета-тестирования переключаются между собой в зависимости от процесса, которому следуют в проекте, и могут быть изменены в соответствии с процессами.

[Альфа-тестирование](#) - это форма внутреннего приемочного тестирования (internal acceptance testing), выполняемого, в основном, собственными командами по обеспечению качества и тестированию ПО. Альфа-тестирование - это последнее тестирование, проводимое группами тестирования на месте разработки после приемочного тестирования и перед выпуском программного обеспечения для бета-тестирования. Альфа-тестирование также может быть выполнено потенциальными пользователями или клиентами приложения. Но все же это форма внутреннего приемочного тестирования.

[Бета-тестирование](#) - это следующий этап после альфа-тестирования. Это заключительный этап тестирования, на котором компании выпускают ПО для нескольких внешних групп пользователей, не входящих в группы

тестирования компании или сотрудников. Эта начальная версия программного обеспечения известна как бета-версия. Большинство компаний собирают отзывы пользователей в этом выпуске. Короче говоря, бета-тестирование можно определить как тестирование, проводимое реальными пользователями в боевой среде. Несмотря на то, что компании проводят строгую внутреннюю проверку качества с помощью специальных групп тестирования, практически невозможно протестировать приложение для каждой комбинации тестовой среды. Бета-версии упрощают тестирование приложения на тысячах тестовых машин и исправление проблем перед выпуском приложения для широкой публики. Выбор групп для бета-тестирования может производиться в зависимости от потребностей компании. Компания может либо пригласить нескольких пользователей для тестирования предварительной версии приложения, либо выпустить ее открыто, чтобы это мог сделать любой. Устранение проблем в бета-версии может значительно снизить затраты на разработку, поскольку большинство незначительных сбоев будут исправлены до окончательной версии. До сих пор многие крупные компании успешно использовали бета-версии своих самых ожидаемых приложений.

| Alpha Testing | Beta Testing |
|---|---|
| Testing environment | Real environment |
| Functional, usability | Functional, Usability, Reliability, Security |
| White box and / or Black box testing | Black box testing |
| На найденные дефекты создаются баг-репорты с high priority, после чего они немедленно исправляются | Дефекты собираются из обратной связи от пользователей и записываются как улучшения для будущей версии |
| Цели: <ul style="list-style-type: none"> • Оценить качество продукта; • Убедиться в готовности к бета-тестированию; • Фокус на поиске ошибок; • Работает ли ПО? | Цели: <ul style="list-style-type: none"> • Оценить удовлетворенность клиентов; • Убедиться в готовности к релизу (в прод); • Фокус на сборе отзывов и предложений; • Нравится ли заказчикам (customers) продукт? |
| Когда? <ul style="list-style-type: none"> • Обычно после System testing phase или когда продукт готов на 70-90%; • Фичи почти заморожены, и нет возможности для серьезных улучшений; • Сборка должна быть стабильной для технического пользователя; | Когда? <ul style="list-style-type: none"> • Обычно после альфа-тестирования и продукт готов на 90-95%; • Фичи заблокированы и улучшения уже не принимаются; • Сборка должна быть стабильной для реальных пользователей; |
| Продолжительность теста: <ul style="list-style-type: none"> • Проведение множества циклов испытаний; • Каждый цикл тестирования длится 1-2 недели; • Продолжительность также зависит от количества обнаруженных проблем и количества добавленных новых функций; | Продолжительность теста: <ul style="list-style-type: none"> • Проведение всего 1 или 2 цикла испытаний; • Каждый цикл тестирования длится 4-6 недель; • Циклы тестирования могут увеличиваться в зависимости от отзывов / предложений реальных пользователей; |
| Stakeholders: Engineers (in-house developers), Quality Assurance Team, and Product Management Team | Stakeholders: Product Management, Quality Management, and User Experience teams |

| | |
|--|--|
| <p>Участники:</p> <ul style="list-style-type: none"> • Технические эксперты, специализированные тестировщики с хорошими знаниями предметной области (новые или уже участвовавшие в фазе тестирования системы), предметная экспертиза (Subject Matter Expertise); • В некоторых случаях клиенты и / или конечные пользователи могут участвовать в альфа-тестировании; | <p>Участники:</p> <ul style="list-style-type: none"> • Конечные пользователи, для которых предназначен продукт; • Customers также обычно участвуют в бета-тестировании; |
| <p>Ожидания:</p> <ul style="list-style-type: none"> • Приемлемое количество ошибок, которые были пропущены при предыдущих тестовых мероприятиях; • Неполные функции и документация; | <p>Ожидания:</p> <ul style="list-style-type: none"> • Почти готовый продукт с гораздо меньшим количеством ошибок и сбоев; • Почти готовые функции и документация; |
| <p>Критерии начала (Entry Criteria):</p> <ul style="list-style-type: none"> • Альфа-тесты, разработанные и проверенные с учетом требований бизнеса (Business requirements); • Требования покрыты тестами в Traceability matrix; • Команда тестирования со знанием предметной области (domain) и продукта; • Настройка среды и сборка для выполнения (Environment setup and build for execution); • Набор инструментов должен быть готов для регистрации ошибок и управления тестированием; • Системное тестирование, в идеале, должно быть закончено; | <p>Критерии начала (Entry Criteria):</p> <ul style="list-style-type: none"> • Бета-тесты, например, что тестировать, и процедуры, задокументированные для использования на проде; • Нет необходимости в матрице прослеживаемости; • Конечные пользователи и заказчик объединяются; • Настройка среды конечного пользователя; • Набор инструментов должен быть готов для сбора отзывов / предложений; • Alpha Testing должно быть закончено; |
| <p>Критерии окончания (Exit Criteria):</p> <ul style="list-style-type: none"> • Все альфа-тесты должны быть выполнены, и все циклы должны быть завершены; • Critical / Major дефекты должны быть исправлены и повторно протестированы; • Должен быть завершен эффективный анализ отзывов, предоставленных участниками; • Alpha Test Summary report; • Alpha Testing должно быть закончено; | <p>Критерии окончания (Exit Criteria):</p> <ul style="list-style-type: none"> • Все циклы должны быть завершены; • Critical / Major дефекты должны быть исправлены и повторно протестированы; • Должен быть завершен эффективный анализ отзывов, предоставленных участниками; • Beta Test Summary report; • Beta Testing должно быть закончено; |

| | |
|---|--|
| <p>Плюсы (Pros):</p> <ul style="list-style-type: none"> • Помогает обнаружить ошибки, которые не были обнаружены во время предыдущих тестовых мероприятий; • Лучшее представление об использовании и надежности продукта; • Анализ возможных рисков во время и после запуска продукта; • Помогает подготовиться к будущей поддержке клиентов; • Помогает укрепить доверие клиентов к продукту; • Снижение затрат на обслуживание за счет выявления и исправления ошибок перед запуском бета-версии / production версии; • Простое управление тестированием (Test Management); | <p>Плюсы (Pros):</p> <ul style="list-style-type: none"> • Тестирование продукта не поддается контролю, и пользователь может протестировать любую доступную функцию любым способом - в этом случае угловые области (corner areas) хорошо протестированы; • Помогает обнаружить ошибки, которые не были обнаружены во время предыдущих тестовых мероприятий (включая альфа-версию); • Лучшее представление об использовании продукта, надежности и безопасности; • Анализ точки зрения и мнение реального пользователя о продукте; • Отзывы / предложения реальных пользователей помогают в дальнейшем импровизировать продукт; • Помогает повысить удовлетворенность клиентов продуктом; |
| <p>Минусы (Cons):</p> <ul style="list-style-type: none"> • Ожидается, что не вся функциональность продукта будет проверена; • Ограничено только бизнес-требованиями; | <p>Минусы (Cons):</p> <ul style="list-style-type: none"> • Определенный объем (Scope) может соблюдаться или не соблюдаться участниками; • Документация больше и требует больше времени - требуется для использования инструмента регистрации ошибок (при необходимости), использования инструмента для сбора отзывов / предложений, процедуры тестирования (установка / удаление, руководства пользователя); • Не все участники гарантируют, что проводят качественное тестирование; • Не все отзывы эффективны - на рассмотрение отзывов уходит много времени; • Управление тестированием слишком сложно; |

Помимо альфа- и бета-тестирования, существуют еще гамма-тестирования и пилотное.

Gamma Testing - это заключительный этап тестирования, который выполняется, когда продукт готов к выпуску с особыми требованиями. Не все действия по внутреннему тестированию, которые решено пройти через этот этап тестирования, выполняются на продукте. Этот этап не позволяет вносить в продукт какие-либо изменения, кроме исправления критических ошибок, которые необходимо выполнить. Это тестирование проводится, чтобы убедиться, что продукт является более безопасным с точки зрения качества продукта, удобства использования, безопасности и производительности перед выпуском в прод.

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.

Источник:

[Alpha Testing And Beta Testing \(A Complete Guide\)](#)

Как протестировать продукт без требований?

Продукта без требований не существует, просто они могут быть не формализованы и не записаны. Если вам дали протестировать какое-то готовое решение, к которому нет документации, то нужно попытаться восстановить все явные и неявные требования: изучив функционал (какую проблему решает продукт? для чего его создали?), целевую аудиторию (для кого его создали?), попытаться найти тех, кто мог знать что-то и уже не в команде; список источников получения неявных требований вообще огромен. На основе всего этого уже как минимум можно составить user stories для покрытия основными тестами.

Доп. материал:

[Тестирование без требований. Где искать требования к продукту, если отсутствует ТЗ?](#)

[Роли/должности в команде](#)

| Name | Description | Soft skills | Hard Skills |
|-----------------------|---|---|---|
| Product Manager | Создает и управляет стратегией развития продукта. Знает хорошо портрет своих потребителей и их желания, относительно продукта. Следит за тенденциями, рынком, конкурентами и превносит улучшения в продукт. | Communication Leadership Stakeholders management Networking Creative solutions Persuasion Competitive Intelligence Innovative Vision Creative Passion | Product Management Product Strategy Product Roadmap Technical skills Design Marketing Research Analytical Skills Time management English (from Upper+) |
| Business Analyst | Исследует проблему заказчика, ищет решение и оформляет его концепцию в форме требований, на которые в дальнейшем будут ориентироваться разработчики при создании продукта. | Communication Listening Influencing Decision making Presentation | Meeting management Time management Technical skills Analytical Skills Research English (from Upper+) |
| Project Manager | Ответственность за управление проектом, достижение целей проекта в рамках бюджета, в срок и с заданным уровнем качества. | Leadership Motivation Communication Influencing Decision making Negotiation Trust building Coaching Conflict management Team building | Risk Management Budgeting and Scheduling Planning Project Lifecycle Management Scrum Management Performance Tracking Kanban management English (from Upper+) |
| QA engineer | Обеспечивает качество, деятельность направлена на улучшение процесса разработки ПО, предотвращение дефектов и выявление ошибок в работе продукта. | Communication Listening Adaptability Teamwork Work Ethic Critical Thinking Fast Learning Confrontation skill Proactive Organized Sixth Sense Knowledge Sharing Empathy | Time management Prioritizing skills Technical skills Analytical Skills Identifying problems Programming skills Problem solving English (from Intermediate) |
| Support Engineer | Обрабатывает заявки от пользователей продукта/сервиса. В зависимости от типа заявки, этот специалист решает возникшую проблему самостоятельно или передает другому специалисту для ее решения. | Communication Flexibility Friendly End User Empathy & Understanding Enthusiasm Organized | Customer service Research Identifying problems Problem solving Troubleshoot English (from Intermediate) |
| DevOps engineer | Работает на стыке областей разработки и системного администрирования, обеспечивая эффективность процесса поставки ПО. | Collaboration Empathy Passion Proactive | Testing Skills DevOps Tools Security Skills Coding and Scripting Skills Cloud Skills Automation skills English (from Intermediate) |
| Frontend Developer | Разработчик клиентской стороны пользовательского интерфейса к программно-аппаратной части. | Teamwork Thinking visually Flexibility Communication Organized Patience Open-mindedness Accountability Approachability Helpfulness Problem solving | HTML/CSS JavaScript JavaScript Frameworks Front End Frameworks CSS Preprocessors RESTful Services/APIs Responsive/Mobile Design Cross-Browser Development Testing/Debugging Git/Version Control Problem solving English (from Intermediate) |
| Backend Developer | Разработчик программно-аппаратной части комплексного ПО. | Teamwork Decision making Organized Patience Open-mindedness Accountability Approachability Helpfulness Problem solving | Server Database API Web Development Languages Cache Git/Version Control Problem solving English (from Intermediate) |
| Android/iOS Developer | Разработчик, специализирующийся на создании программ для планшетов и смартфонов на iOS и Android. | Teamwork Thinking visually Flexibility Organized Patience Accountability Approachability Problem solving Helpfulness | Analytical Skills Mobile Programming Languages Git/Version Control SDK CSS Testing/Debugging English (from Intermediate) |
| UX/UI Designer | Занимается проектированием пользовательских интерфейсов, в которых удобство использования так же важно, как и внешний вид. | Communication Teamwork Creative Patience Work Ethic Organized Proactive Thinking visually Flexibility Open-mindedness | Wireframing UI prototyping Interaction design Visualization Skills Analytical Skills Design Design Tools English (from Intermediate) |

Роли в тестировании ПО:

- Software Test Engineer: тестирует всю систему, используя соответствующие методы и инструменты тестирования;
- Test Analyst: определяет test conditions and features для тестирования, разрабатывает тестовые сценарии и документацию;
- Test Automation Engineer: разрабатывает сценарии для запуска автоматизированных тестов;
- SDET (Software Development Engineer in Test) - это специалист, который может одинаково эффективно работать в сфере разработки и тестирования и принимает участие в полном процессе разработки ПО, в т.ч. в его обязанности может входить разработка внутренних инструментов для поддержки тестирования или других функций, написание тестового фреймворка, фикс найденных дефектов за разработчика, unit/integration testing и т.п.;
- Test Architect: проектирует комплексную инфраструктуру тестирования, выбирает инструменты для реализации;
- Test Manager: подготавливает стратегию тестирования, контролирует процесс тестирования и членов команды;

Источники:

- [Куда идти в IT. Подробная инструкция от Project Manager](#)
- [QA Engineering Roles: Skills, Tools, and Responsibilities in a Testing Team](#)

Доп. материал:

- [Product Owner vs Product Manager или Product Owner/Product Manager](#)
- [Business Analyst, Requirement Specialist, Product Owner и другие. Чем отличаются схожие на первый взгляд роли?](#)
- [Роль QA Lead в продуктовой компании: особенности и зоны ответственности](#)
- [Продакт-менеджмент как профессия: востребованность, зарплата и другие нюансы](#)
- [Кто такой продакт-менеджер? Или не все РМ'ы — продакт-менеджеры](#)
- [Project Management in QA and Testing](#)
- [Knowledge management: как перестать изобретать велосипеды](#)
- [Заметки knowledge manager'a. Как работает управление знаниями в Exness](#)
- [Профессия СТО](#)
- [Кто такой DevOps-инженер, что он делает, сколько зарабатывает и как им стать](#)
- [Гайд по DevOps для начинающих](#)
- [Распространенные поисковые запросы, часть 3: когда должно начинаться тестирование?](#)
- [«Вам звонок». Как выстроить отношения между QA и техподдержкой](#)

Тестовая среда и тестовый стенд (Test Environment/Test Bed)

В общем случае среда тестирования - это конфигурация ПО/виртуального контейнера и/или сервера, т.е. эдакая песочница для тестирования. Позволяет не испытывать судьбу с production-версией, а также дает множество возможностей, которые недоступны на боевом окружении. 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 тестируют новые билды: 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, отвечаая за работоспособность всех систем. Инциденты и проблемы требующие исправления ПО передаются в работу команде на L3

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](#)

Тестовые данные (Test Data)

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.

Доп. материал:

[What Is Test Data? Test Data Preparation Techniques With Example](#)

Бизнес-логика

Бизнес-логика - это реализация работы бизнес-процессов внутри ПО, т.е. это реализация предметной области (domain) в информационной системе. К ней относятся, например, формулы расчёта ежемесячных выплат по ссудам (в финансовой индустрии), автоматизированная отправка сообщений электронной почты руководителю проекта по окончании выполнения частей задания всеми подчиненными (в системах управления проектами), отказ от отеля при отмене рейса авиакомпанией (в туристическом бизнесе) и т. д.

Источники:

- [Что такое бизнес-логика](#)
- [Бизнес-логика](#)

Политика отсутствия багов (ZBP - Zero Bug Policy)

Она означает, что все баги имеют приоритет над разработкой новых фич или улучшениями. Важным следствием этого подхода является отсутствие таких вещей, как приоритет багов, critical bugs или minor bugs. Либо issue является багом, либо нет. И если это баг, вам нужно исправить его, прежде чем выполнять другую работу.

Преимущества:

- снижение затрат на разработку;
- лучшие оценки (estimates);
- повышение гибкости;
- повышение удовлетворенности клиентов/заказчиков;

Источник:

[The Zero Bug Policy](#)

Доп. материал:

- [QA Crew #4: Круглый стол: Zero Bug Policy: о политике, ее преимуществах/недостатках, нюансах внедрения](#)
- [How We Got to Zero Bugs and Implemented a Zero Bug Policy](#)

Эвристики и мнемоники

Эвристики – это быстрые, недорогие способы решения проблемы или принятия решения. В мире тестирования ПО мы можем использовать эвристики как выведенные опытным путём подсказки для принятия решений и решения проблем в ходе тестирования. Они могут быть особенно полезными для генерации предположений, если мы не уверены, как начать тестирование, или исчерпали идеи, что делать дальше.

Мнемоника – это тип эвристики, набор правил и приемов, которые помогают эффективно запоминать необходимые сведения (информацию), обычно это слово-аббревиатура или фраза. Например, все помнят детскую мнемонику “каждый охотник желает знать где сидит фазан”, в которой по первым буквам каждого слова можно вспомнить порядок цветов радуги.

Множество известных тест-эвристик использует мнемоники и широко распространено заблуждение, что у эвристик должна быть мнемоника, или что это одно и то же. Это не так. Эвристики не требуют мнемоник, они просто создаются таким образом, чтобы их было легче запомнить.

Эвристики и мнемоники могут быть придуманы, модифицированы и скрещены как удобно автору для своих нужд. Вот наиболее известные:

- **I SLICED UP FUN:** Для тестирования мобильных приложений;
- **COP FLUNG GUN:** Еще одна;
- **MOBILE APP TESTING:** И еще одна;
- **SPIES:** Для тестирования локализации;
- **PAOLO:** Тестирование мобильных приложений и смены ориентации экрана;
- **GO DaRE=M:** Для составления тест-плана;
- **PAPAS BE @ SFO:** Мнемоника для API-тестов функционала;
- **DEED HELP GC:** Еще одна мнемоника по API-тестам;
- **DVLA PC:** Для поддержки API-тестов;
- **ICE OVER MAD!**: Мнемоника по тестированию API;
- **INVEST:** Атрибуты хорошей юзер-стори;
- **CIRCUS MATTA:** Для ревью пользовательских историй;
- **CAN I USE THIS:** Для тестирования Usability;
- **SAQSII meeting:** Для улучшения эффективности любого собрания;
- **SFPDO & SFDIPOT:** Для знакомства с продуктом, новых тест-идей и т.п.;
- **RCRCRC:** Для регрессионного тестирования;
- **CRUSSPIC STMPL:** Эвристика качественных характеристик системы;
- **FEW HICCUPS:** Тестовые оракулы;
- **RIMGEA:** Для описания багов;
- **MOCHA:** Описывает стиль собеседований для найма тестировщиков
- **HEENA:** Для тестирования сложных продуктов;
- **SCAMPER:** Для того, чтобы задать вопросы по продукту, которые принесут креативные идеи и кейсы;
- **DUFFSSCRA:** Для техник тестирования;
- **MCOASTER:** Для составления баг-репортов;
- **FAILURE:** Для составления грамотных сообщений об ошибке;
- **W5HE (WWWWWH/KE):** Для анализа требований;
- **PROOF:** Для написания тестового отчета после сессионного тестирования;
- **GRATEDD SCRIPTS & B GRADED SCRIPTTS:** Для тестовой стратегии;
- **CIDTESTD:** Для высокоуровневого планирования процесса тестирования;
- **MAC RUSS:** Для приемочного тестирования;

- **SACKED SCOWS:** Для обучения;
- **MR.Q COMP GRABC R&R:** Для проведения исследовательского тестирования;
- **FCC CUTS VIDS:** Эвристика тестовых туров;
- **SLIME:** Эвристика приоритетов для тестирования;
- **CCD IS EARL:** Основные принципы нагрузочного тестирования;
- **IVECTRAS:** Классификация нагрузочных тестов;
- **FIBLOTS:** Модель нагрузки для нагрузочного тестирования;
- **RSTLLL:** Эвристика тестирования сообщений, отправляемых приложением;
- **MUTII:** Эвристика для тестирования;

Расшифровка, больше вариантов и дополнительные ссылки в первом источнике.

Эвристики окончания тестирования (когда пора прекратить тестировать продукт):

1. **Эвристика «Время вышло!».** Для многих специалистов по тестированию это наиболее распространенная эвристика: мы останавливаем тестирование, когда заканчивается выделенное на него время. Получили ли мы информацию, которую нам требуется знать о продукте? Не слишком ли высок риск прекращения тестирования? Не был ли срок искусственным, произвольным? Будет ли выполняться дополнительная разработка, которая потребует дополнительного тестирования?
2. **Эвристика пиньят (The Piñata Heuristic).** Мы прекращаем ломать программу, когда начинают выпадать конфеты – мы останавливаем тестирование, когда видим первую достаточно серьезную проблему. Не застряло ли в ноге пиньята еще несколько конфет? Является ли первая серьезная проблема самой важной? Единственной, о которой стоит беспокоиться? Не найдем ли мы другие интересные проблемы, если продолжим тестирование? Что если наше ощущение «серьезности» ошибочно и проблема не столь грандиозна?
3. **Эвристика «мертвой лошади» (The Dead Horse Heuristic).** 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. Здесь мы предполагаем, что уже найдено много интересного и важного. Если мы сейчас остановимся, не пропустим ли мы что-то еще более важное или более интересное?
4. **Эвристика «Задание выполнено» (The Mission Accomplished Heuristic).** Мы останавливаем тестирование, когда найдены ответы на все поставленные вопросы. В процессе нашего тестирования могут возникнуть новые вопросы. Это приводит нас к эвристике Рамсфелда (Rumsfeld Heuristic): «Есть то, про что мы знаем, что мы это не знаем, и есть то, про что мы не знаем, что мы этого не знаем». Достаточно ли неизвестных переместило наше тестирование в область известного? Обнаружило ли наше тестирование новые неизвестные? И сложный для разбора, но важный вопрос: удовлетворены ли мы тем, что мы переместили достаточно неизвестных неизвестных в область известного или по крайней мере сделали их известными неизвестными.
5. **Эвристика «Отмена задания» (The Mission Revoked Heuristic).** 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.) В достаточной ли степени наш клиент осознает ценность продолжения тестирования или риски прекращения? Если мы не согласны с клиентом, то в достаточной ли мере мы осознаем бизнес-причины приостановки тестирования?
6. **Эвристика «Я зашел в тупик!» (The I Feel Stuck! Heuristic).** 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. Существует масса способов выйти из тупика. Может быть, нам нужна помощь, а может быть нам просто надо сделать перерыв (смотрите ниже). Может быть, продолжение тестирования позволит

нам получить требуемые знания. Может быть, вся цель тестирования и заключается в исследовании продукта и получении недостающей информации. Возможно, имеется путь, позволяющий обойти блокирующую ошибку; возможно инструменты и оборудование имеются, но мы просто не знаем о них или никогда не задавали правильных вопросов тем, кому надо; возможно имеются доступные для нас эксперты – в команде тестирования, среди программистов или на стороне бизнеса – и мы этого просто не знаем. Есть разница между ощущением тупика и нахождением в тупике.

7. **Эвристика «освежающей паузы»** (The Pause That Refreshes Heuristic). 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, поразмысльить 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. Конечно, мы можем чувствовать себя уставшими, нам может быть скучно, но не нужно ли проявить упорство и продолжить двигаться вперед? Не получится ли изучить требуемое в процессе работы с программой, вместо того, чтобы делать это отдельно? Не найдется ли тот критичный бит информации, которого нам не хватает, благодаря лишь еще одному тесту? Является ли функция с «более высоким приоритетом» действительно более приоритетной? Готова ли она к тестированию? Не протестировали ли мы ее и так уже достаточно?
8. **Эвристика «Отсутствие продвижения»** (The Flatline Heuristic). 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!" Действительно ли приложение упало или, возможно, оно восстанавливается? Не является ли отсутствие отклика само по себе важным результатом тестирования? Включает ли в себя понятие «что бы мы ни делали» достаточное разнообразие вариантов или нагрузок, чтобы покрыть потенциальные риски?
9. **Эвристика Привычного завершения** (The Customary Conclusion Heuristic). 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». Эвальд Руденриджс (Ewald Roodenrijns) приводит в своем блоге пример этой эвристики в статье «Когда прекращать тестирование». Он говорит, что он останавливается, «когда выполнено определенное количество тестовых циклов, включая регрессионное тестирование». Отличие от эвристики «Время вышло!» в том, что временные ограничения могут изменяться более гибко, чем некоторые другие. Поскольку в большинстве проектов главенствует именно график проекта, и у меня и у Джеймса заняло некоторое время осознание того, что эта эвристика также очень распространена. Иногда мы можем слышать фразы типа «один тест на требование» или «один положительный и один отрицательный тест на требование», в качестве соглашения для определения «достаточно хорошего» тестирования. (Конечно же, мы не согласны с этим, но мы слышим это). Достаточно ли мы задумываемся о том, почему мы всегда останавливаемся на этом? Не должны ли мы на самом деле провести дополнительное тестирование? Или наоборот наше тестирование избыточно? Нет ли у нас информации – например, от службы технической поддержки, от службы продаж, от внешних рецензентов – которая подсказала бы, как нам изменить наши шаблоны? Рассмотрели ли мы все прочие эвристики?
10. **Больше нет интересных вопросов** (No more interesting questions). 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. Что мы думаем о наших моделях рисков? Нет ли опасности недооценки или наоборот переоценки риска, не случилось ли так, что мы не заметили Чёрного лебедя (а может быть даже Белого лебедя)? Достигли ли мы достаточного покрытия? Достаточно ли тщательно мы проверили свои оракулы?

11. **Эвристика уклонения/безразличия** (The Avoidance/Indifference Heuristic). 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. Если это безразлично нам сейчас, то почему мы вообще тестировали? У нас сменились приоритеты? Если кто-то закончил работу, то почему? Иногда компанию меньше беспокоит незнание о существовании проблемы, чем знание и отсутствие действий по ее устраниению – не может ли это быть нашим случаем?

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

Источники:

- [Мнемоники в тестировании](#)
- [Эвристики тестирования: будьте внимательны!](#)
- [Когда нужно прекращать тестирование?](#)

Виды/типы/уровни тестирования

Примечание: понятие типов, методов и видов в англоязычной литературе часто не разделяется и может быть перечислено вообще вперемешку, а также зачастую там выделяют только функциональное и нефункциональное тестирование. Не зацикливайтесь на категоризации и терминологии, пытайтесь понять саму суть.

Типы/методы тестирования (White/Black/Grey Box)

Самым высоким уровнем в иерархии подходов к тестированию будет понятие типа/метода, 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.

Доп. материал:

[What is red box, yellow box and green box testing?](#)

Тестирование черного ящика (Black Box Testing)

Другие названия: Behavioral Testing, Specification-Based Testing, Input-Output Testing, непрозрачный ящик (opaque-box), закрытый ящик (closed-box), тестирование на основе спецификации (specification-based testing) или тестирование с глазу на глаз (eye-to-eye testing).

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

Functional Testing: этот тип касается функциональных требований или спецификаций приложения (functional requirements or specifications). Здесь различные действия или функции системы тестируются путем предоставления входных данных и сравнения фактического выхода с ожидаемым выходом. Например, когда мы тестируем раскрывающийся список, мы нажимаем на него и проверяем, что он раскрывается и все ожидаемые значения отображаются. Вот несколько основных типов функционального тестирования:

- Smoke Testing;
- Sanity Testing;
- Integration Testing;
- System Testing;
- Regression Testing;
- User Acceptance Testing;

Non-Functional Testing: Помимо функциональности требований, есть несколько нефункциональных аспектов, которые необходимо протестировать, чтобы улучшить качество и производительность приложения.

Несколько основных типов нефункционального тестирования включают:

- Usability Testing;
- Load Testing;
- Performance Testing;
- Compatibility Testing;
- Stress Testing;
- Scalability Testing;

Приемущества Black box testing:

- Тестировщику не обязательно иметь технический опыт. Важно проводить тестирование, оказываясь на месте пользователя и думая с его точки зрения;
- Тестирование можно начинать после завершения разработки проекта / приложения. И тестировщики, и разработчики работают независимо, не мешая друг другу;
- Это более эффективно для больших и сложных приложений;

- Дефекты и несоответствия можно выявить на ранней стадии тестирования;

Недостатки Black box testing:

- Без каких-либо технических или программных знаний есть вероятность пропустить возможные условия тестируемого сценария;
- В оговоренное время есть вероятность протестировать не все входные и выходные значения;
- Полный Test Coverage невозможен для больших и сложных проектов;

Парадигмы тестирования методом черного ящика (Paradigms of Black Box Software Testing)

Парадигма создает основное направление мышления - она дает понимание и направление для дальнейших исследований или работы. Парадигма тестирования будет определять типы тестов, которые (для человека, работающего в рамках этой парадигмы) актуальны и интересны. Список парадигм:

- **Domain driven**
 - Ключевые идеи:
 - «Попробуйте диапазоны и варианты»;
 - «Разделите мир на классы»;
 - Основной вопрос или цель:
 - Стратегия стратифицированной выборки. Разделите большое пространство возможных тестов на подмножества. Выберите лучших представителей из каждого набора;
 - Примеры кейсов:
 - Анализ эквивалентности простого числового поля;
 - Тестирование совместимости принтеров;
 - Сильные стороны:
 - Нахождение ошибок с наибольшей вероятностью с помощью относительно небольшого набора тестов;
 - Интуитивно понятный подход, хорошо обобщает;
 - Слепые зоны:
 - Ошибки, выходящие за рамки границ, или в очевидных особых случаях;
 - Кроме того, фактические домены часто остаются неизвестными;
- **Stress driven**
 - Ключевые идеи:
 - «Сокруши продукт»;
 - «Проведи его через отказы»;
 - Основной вопрос или цель:
 - Узнать о возможностях и слабых сторонах продукта, проведя его через отказ и за его пределами. Что сбои в экстремальных случаях говорят нам об изменениях, необходимых в работе программы в нормальных случаях?
 - Примеры кейсов:
 - Большие объемы данных, подключения устройств, длинные цепочки транзакций;
 - Условия нехватки памяти, сбои устройств, вирусы и другие проблемы;
 - Сильные стороны:
 - Выявление слабых мест, в т.ч. дыр в безопасности;
 - Слепые зоны:
 - Слабости, которые не становятся более заметными из-за стресса;
- **Specification driven**
 - Ключевые идеи:
 - «Проверяйте каждое требование»;
 - Основной вопрос или цель:
 - Проверяйте соответствие (conformance) продукта каждому заявлению в каждой спецификации, документе с требованиями и т. д.;
 - Примеры кейсов:

- Матрица прослеживаемости, отслеживает тестовые случаи, связанные с каждым элементом спецификации;
 - Сильные стороны:
 - Критическая защита от гарантийных претензий, обвинений в мошенничестве, потери доверия со стороны клиентов;
 - Слепые зоны:
 - Любые проблемы, не указанные в спецификациях или плохо решенные в спецификациях;
- **Risk driven**
 - Ключевые идеи:
 - «Сначала найди наибольшие ошибки»;
 - Основной вопрос или цель:
 - Расставьте приоритеты при тестировании с точки зрения относительного риска различных областей или проблем, которые мы могли бы проверить;
 - Примеры кейсов:
 - Переформулированный анализ классов эквивалентности;
 - Тестируйте в порядке частоты использования;
 - Стресстесты, тесты на обработку ошибок, тесты безопасности, тесты для поиска прогнозируемых или предполагаемых ошибок;
 - Образец из списка предсказанных ошибок;
 - Сильные стороны:
 - Оптимальная приоритизация (при условии, что мы правильно идентифицируем и расставляем по приоритетам риски);
 - Тесты высокой мощности;
 - Слепые зоны:
 - Риски, которые не были идентифицированы или которые на удивление более вероятны;
- **Random / statistical testing**
 - Ключевые идеи:
 - «Объемное тестирование с новыми кейсами»;
 - Основной вопрос или цель:
 - Пусть компьютер создает, выполняет и оценивает огромное количество тестов;
 - Примеры кейсов:
 - Валидация функции или подсистемы (например, тестирование эквивалентности функций) на основе оракулов (Oracle-driven);
 - Стохастическое (переход между состояниями) тестирование для выявления конкретных сбоев (ассерты, утечки и т. д.);
 - Оценка статистической надежности;
 - Частичный или эвристический оракул, чтобы найти некоторые типы ошибок без общей проверки;
 - Сильные стороны:
 - Регрессия не зависит каждый раз от одного и того же старого теста;
 - Частичные оракулы могут быстро и дешево находить ошибки в молодом коде;
 - Меньше вероятность пропустить невидимые извне внутренние оптимизации;
 - Может обнаруживать сбои, возникающие из-за длинных сложных цепочек, которые было бы трудно создать в соответствии с запланированными испытаниями;
 - Слепые зоны:
 - Нужно уметь отличать pass от failure. Слишком много людей думают: «Not crash = not fail»;
 - Кроме того, эти методы часто охватывают многие типы рисков, но затемняют необходимость в других тестах, которые не поддаются автоматизации;
- **Function Testing**
 - Ключевые идеи:

- «Модульное тестирование черного ящика»;
 - Основной вопрос или цель:
 - Тщательно проверяйте каждую функцию по очереди;
 - Примеры кейсов:
 - Таблица, тестируйте каждый элемент по отдельности;
 - База данных, тестируйте каждый отчет по отдельности;
 - Сильные стороны:
 - Тщательный анализ каждого протестированного элемента;
 - Слепые зоны:
 - Упускает взаимодействия, пропускает исследование преимуществ предлагаемые программой;
- **Regression Testing**
 - Ключевые идеи:
 - «Повторить тестирование после изменений»;
 - Основной вопрос или цель:
 - Управляйте рисками, связанными с тем, что (а) исправление ошибки не устраняет ошибку или (б) исправление (или другое изменение) имело побочный эффект (side effect);
 - Примеры кейсов:
 - Регрессия ошибок, регрессия старых исправлений, общая функциональная регрессия;
 - Наборы автоматизированной регрессии графического интерфейса;
 - Сильные стороны:
 - Обнадеживает, укрепляет доверие, удобен для регуляторов;
 - Слепые зоны:
 - Все, что не вошло в регрессионную серию. Кроме того, поддержание этого стандартного списка может быть очень дорогостоящим;
- **Scenario / use case / transaction flow**
 - Ключевые идеи:
 - «Делай что-нибудь полезное и интересное»;
 - «Делайте одно за другим»;
 - Основной вопрос или цель:
 - Сложные случаи, отражающие реальное использование;
 - Примеры кейсов:
 - Оценивайте продукт на предмет соответствия бизнес-правилам, данным о клиентах и продукции конкурентов;
 - Тестирование жизненного цикла / Life history testing ([Hans Buwalda's "soap opera testing"](#));
 - Варианты использования (Use cases) - это более простая форма, часто основанная на возможностях продукта и пользовательской модели, а не на естественном наблюдении за системами такого типа;
 - Сильные стороны:
 - Сложные, реалистичные события. Может помогать справляться в ситуациях, которые слишком сложны для моделирования;
 - Выявляет сбои, которые происходят (развиваются) с течением времени;
 - Слепые зоны:
 - Отказ одной функции может сделать этот тест неэффективным;
 - Необходимо хорошо подумать, чтобы добиться хорошего покрытия;
- **User testing**
 - Ключевые идеи:
 - Стремитесь к реализму;
 - Давайте попробуем это с настоящими людьми (для разнообразия);
 - Основной вопрос или цель:

- Выявить сбои, которые могут возникнуть по вине человека, то есть сбои в общей системе человек / машина / программное обеспечение;
- Примеры кейсов:
 - Бета-тестирование;
 - Собственные эксперименты с использованием стратифицированной выборки целевого рынка;
- Сильные стороны:
 - Проблемы дизайна более достоверны;
 - Может продемонстрировать, что некоторые аспекты продукта непонятны или приводят к высокому проценту ошибок при использовании;
 - Внутренние тесты можно контролировать с помощью логов, видео, отладчиков и других инструментов;
 - Внутренние тесты могут быть сосредоточены на областях / задачах, которые, по вашему мнению, являются (или должны быть) спорными;
- Слепые зоны:
 - Покрытие не гарантировано (серьезные пропуски бета-тестирования, других пользовательских тестов);
 - Тестовые примеры могут быть плохо спроектированы, тривиальны, вряд ли выявляют малозаметные ошибки;
 - Бета-тестирование стоит денег;
- **Exploratory testing**
 - Ключевые идеи:
 - «Интерактивное, одновременное исследование, разработка тестов и тестирование»;
 - Основной вопрос или цель:
 - ПО поступает тестировщику без документации. Тестировщик должен одновременно узнавать о продукте и о тестовых примерах / стратегиях, которые позволят выявить продукт и его дефекты;
 - Примеры кейсов:
 - Полное тестирование test-it-today;
 - Сторонние компоненты;
 - Горилла-тестинг;
 - Сильные стороны:
 - Продуманная стратегия получения результата в неизвестности;
 - Стратегия выявления несоответствия ожиданиям клиентов;
 - Слепые зоны:
 - Чем меньше мы знаем, тем больше рискуем упустить.

Источники:

- [Black Box Testing: An In-Depth Tutorial With Examples And Techniques](#)
- [Cem Kaner, James Bach - “Paradigms of Black Box Software Testing”](#)

Тестирование белого ящика (White Box Testing)

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 тому, кто её тестирует. 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 - модульном, интеграционном и системном, но чаще применяется для юнит-тестирования этого участка кода самим разработчиком или SDET. Тестирование белого ящика - это больше, чем тестирование кода: это **тестирование путей**. Обычно

тестируемые пути находятся внутри модуля (модульное тестирование). Но мы можем применить эту же методику для тестирования путей между модулями внутри подсистем, между подсистемами внутри систем, и даже между целыми системами.

Тестирование белого ящика - это покрытие требований в коде:

- Code coverage;
- Segment coverage: каждый оператор кода выполняется один раз;
- Branch Coverage or Node Testing: покрытие каждой ветки кода из всех возможных было выполнено;
- Compound Condition Coverage: Для нескольких условий проверяется каждое условие с несколькими путями и комбинацией разных путей для достижения этого условия;
- Basis Path Testing: каждый независимый путь в коде взят на тестирование;
- Data Flow Testing (DFT): в этом подходе вы отслеживаете конкретные переменные посредством каждого возможного вычисления, тем самым определяя набор промежуточных путей через код. DFT имеет тенденцию отражать зависимости, но в основном это происходит через последовательности манипуляций с данными. Короче говоря, каждая переменная данных отслеживается, и ее использование проверяется. Этот подход имеет тенденцию обнаруживать ошибки, такие как переменные, которые используются, но не инициализируются, или объявлены, но не используются, и т.д. (компиляторы/линтеры/IDE уже вполне способны на это сами);
- Path Testing: тестирование пути - это определение и покрытие всех возможных путей прохождения через код;
- Loop Testing: эти стратегии относятся к тестированию одиночных циклов, составных (concatenated) циклов и вложенных циклов;

Используя покрытие Statement и Branch, вы обычно достигаете 80-90% покрытия кода, что является достаточным.

Процесс White box testing:

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

Преимущества White box testing:

- 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 White box testing:

- Количество выполняемых путей может быть настолько большим, что не удастся проверить их все. Как правило, попытка протестировать все пути выполнения с помощью тестирования белого ящика так же невозможна, как и тестирование всех комбинаций всех входных данных при тестировании черного ящика;
- Выбранные тест-кейсы могут не содержать данные, которые будут чувствительны к ошибкам. Например: $p=q/r$; может выполняться корректно, за исключением случая, когда $r=0$. $y=x^2$; тест не выявит ошибок в случаях, когда $x=0$, $y=0$ и $x=2$, $y=4$;
- Тестирование белого ящика предполагает, что поток управления правильный (или близок к правильному). Поскольку эти тесты основаны на существующих путях, с помощью нельзя обнаружить несуществующие пути;
- Тестировщик должен обладать навыками программирования для того, чтобы понять и оценить тестируемое программное обеспечение;

White box testing нужно:

Чтобы убедиться, что:

- Все независимые пути в модуле были проверены хотя бы один раз;
- Все логические решения проверены на их истинное и ложное значения;
- Все циклы выполняются на своих границах и в пределах своих рабочих границ валидности внутренних структур данных;

Чтобы обнаружить следующие типы ошибок:

- Логическая ошибка имеет тенденцию закрасться в нашу работу, когда мы разрабатываем и реализуем функции, условия или элементы управления, которые не входят в программу;
- Ошибки проектирования из-за разницы между логическим потоком программы и фактической реализацией;
- Типографические ошибки и проверка синтаксиса;

Источники:

- [White Box Testing: A Complete Guide With Techniques, Examples, & Tools](#)
- Ли Копланд - “A Practitioner's Guide to Software Test Design”, Секция II. Методы тестирования белого ящика

Тестирование серого ящика (Grey Box Testing)

Тестирования методом серого ящика вообще нет в ISTQB, тем не менее много где можно встретить упоминания этого типа тестирования. В целом оно определяется как метод тестирования ПО, который предполагает комбинацию White Box и Black Box подходов или как дополненный черный ящик. Т.е., внутреннее устройство/код известны/используется лишь частично, и, например, имея доступ к внутренней структуре и алгоритмам работы ПО, можно написать более эффективные тест-кейсы, но само тестирование проводится с помощью техники черного ящика, то есть, с позиции пользователя.

Примеры: тестирование с проверкой корректности записей в БД; работа с логами и метриками для поиска root cause проблем.

Техники:

- **Матричное тестирование (Matrix Testing):** разработчики предоставляют все переменные в программе, а также связанные с ними технические и бизнес-риски. Методика матричного тестирования проверяет риски, определенные разработчиками. Матричный метод устанавливает все используемые переменные в программе. Этот метод помогает идентифицировать и удалять переменные, которые не используются в программе, и, в свою очередь, помогает увеличить скорость работы программного обеспечения;
- **Регрессионное тестирование (Regression Testing):** регрессионное тестирование выполняется, когда в программное обеспечение вносятся какие-либо изменения или исправляется какой-либо дефект. Это делается для того, чтобы новое изменение или исправление не повлияло на существующие функциональные возможности программного обеспечения;
- **Тестирование ортогональных массивов или OAT (Orthogonal Array Testing or OAT):** этот метод тестирования больше используется для сложных функций или приложений, когда требуется максимальное покрытие кода с минимальным количеством test cases и имеет большие тестовые данные с п числом комбинаций;
- **Pattern testing:** тестирование по образцу выполняется на основе предыдущих дефектов, обнаруженных в ПО. Записи о дефектах анализируются на предмет причин дефектов, и создаются test cases на основе этих дефектов и их причин;

Источник: [Grey Box Testing Tutorial With Examples, Tools And Techniques](#)

Статическое и динамическое тестирование (Static Testing, Dynamic Testing)

Статическое тестирование (Static Testing, Non-execution technique или verification) подразумевает проверку вручную или с помощью инструментов кода без его запуска и всей документации.

Почему требуется статическое тестирование:

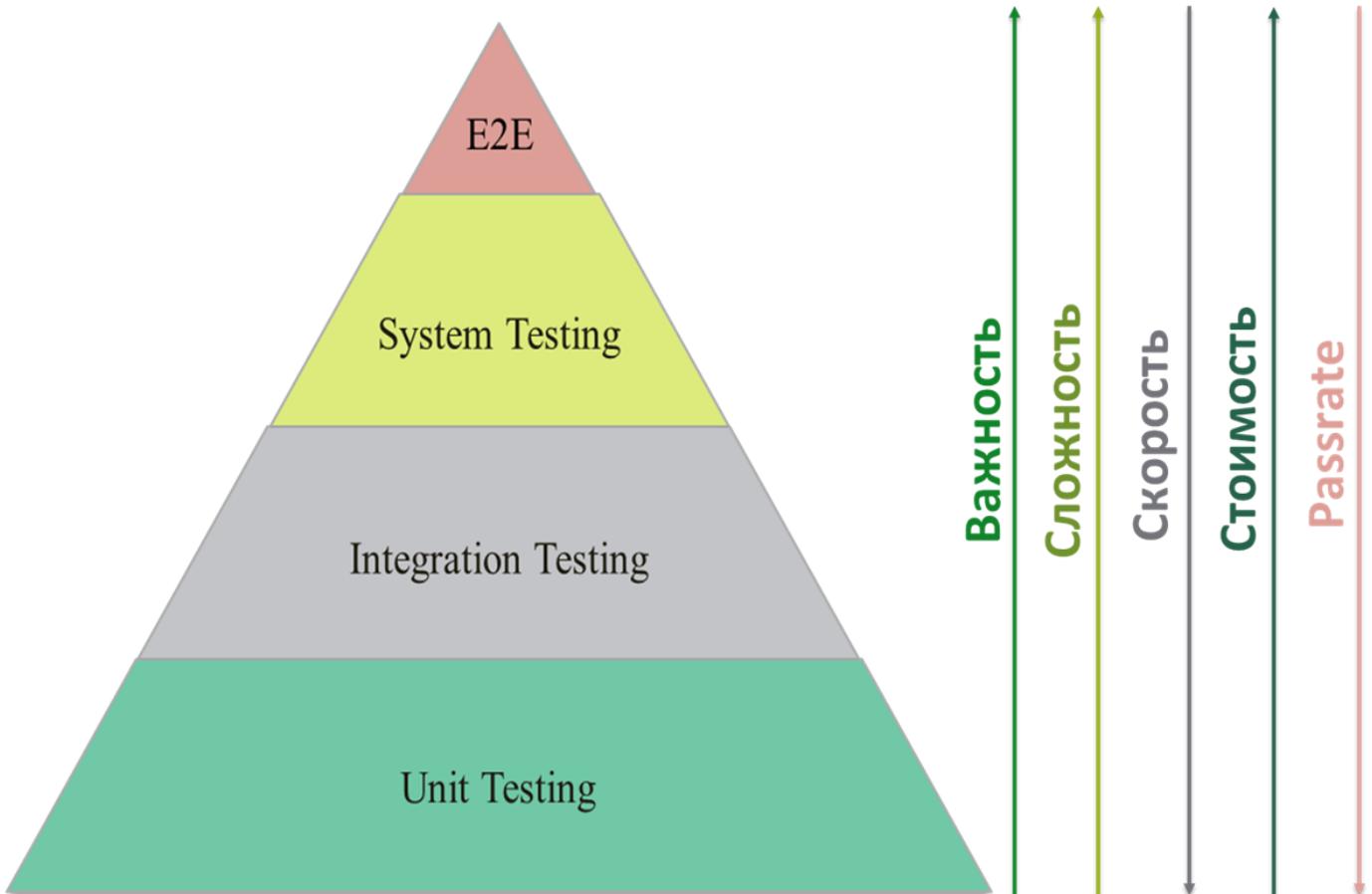
- Обнаружение ошибок / недостатков на ранних этапах: при создании ПО нельзя полагаться исключительно на динамическое тестирование, поскольку оно выявляет ошибки или недостатки программного продукта на более позднем этапе, что может стоить разработчикам много времени и усилий для отладки;
- Увеличение размера ПО: по мере увеличения размера программного продукта становится трудно справиться с ним, поскольку эффективность покрытия кода снижается;
- Динамическое тестирование занимает много времени: несмотря на то, что динамическое тестирование выявляет ошибку и предоставляет некоторые подробности относительно ошибки, исправление ошибки по-прежнему требует времени и усилий, поскольку оно включает в себя обнаружение сбоя от тестового примера до основной причины, что в целом усложняет процесс;
- Динамическое тестирование дорогое: как упоминалось ранее, для динамического тестирования требуются тестовые примеры, и выполнение этого само по себе является дорогостоящим, потому что тестовые примеры должны быть сначала созданы, затем выполнены и проверены, а также должны поддерживаться, что требует большой работы со стороны тестировщиков;

Динамическое тестирование (Dynamic Testing, Execution technique или validation) подразумевает запуск кода для проведения функциональных и нефункциональных проверок ПО. Основная цель этого тестирования - подтвердить, что программный продукт работает в соответствии с требованиями бизнеса. Преимуществами динамического тестирования являются выявление сложных дефектов, которые не могут быть обнаружены статическим тестированием, обнаружение угроз безопасности, проблем с производительностью и т.п.

Источник:

Пирамида / уровни тестирования (Test Pyramid / Testing Levels)

«Пирамида тестов» — метафора, которая означает группировку динамических тестов программного обеспечения по разным уровням. Она также дает представление, какое количество тестов должно быть в каждой из этих групп. Основной принцип разделения уровней - тест должен быть на том же уровне, что и тестируемый объект. В teste более высокого уровня вы не тестируете всю условную логику и пограничные случаи, которые уже покрыты тестами более низкого уровня.



Уровни тестирования:

- Unit/component/program/module testing - тестируется минимально-атомарный модуль программы, чаще всего это одна функция или метод. Таких тестов должно быть больше всего;
- Integration testing - несколько модулей программы тестируются вместе;
- System testing - вся программа тестируется полностью;
- Acceptance testing - программа принимается заказчиком на соответствие заявленным требованиям либо тестировщики проходят end-to-end сценарии с точки зрения пользователя;

Доп. материал:

- [Test Pyramid](#)
- [The Practical Test Pyramid](#) + перевод на русский [Пирамида тестов на практике](#)
- [Software Testing Anti-patterns](#)
- [Пирамида Автоматизации Тестирования: Версия 2021 года](#)
- [Антипаттерны тестирования ПО](#)
- [Unit, API и GUI тесты — чем отличаются](#)
- [Почему тестировать должны не только QA. Распределяем тест-кейсы между Dev, Analyst и QA](#)
- [Пирамида тестирования на практике. Как работает QA в Jiji](#)

Модульное/юнит/компонентное тестирование (Module/Unit/Component testing)

С этими терминами часто происходит путаница. Если ссылаться на глоссарий ISTQB, то все они - синонимы:

- **Модуль, юнит** (module, unit): См. компонент.
- **Модульное, юнит тестирование** (module testing, unit testing): См. компонентное тестирование.
- **Компонент** (component): Наименьший элемент программного обеспечения, который может быть протестирован отдельно.
- **Компонентное тестирование** (component testing): Тестирование отдельных компонентов программного обеспечения (IEEE 610).

Тем не менее, некоторые источники описывают ситуацию несколько иначе и я решил выписать другую точку зрения.

Модульное тестирование (оно же юнит-тестирование) используется 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).

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;
- Тестирование компонентов в целом (CTIL — Component testing In Large) - тестирование компонентов, выполненное без изоляции других компонентов в тестируемом программном обеспечении или приложении;

| Module/Unit testing | Component testing |
|--|--|
| Тестирование отдельных классов, функций для демонстрации того, что программа выполняется согласно спецификации | Тестирование 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 разработчиками | Тестировщиками |
| Выполняется первым | Выполняется после Unit |

Другой источник:

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/unit тестировании - конкретные значения.

*В контексте юнит-тестирования еще можно встретить понятие [golden testing](#). Оно означает те же юнит тесты, но с ожидаемыми результатами хранящимися в отдельном файле. Таким образом после прогона выходные значения тестов сравниваются с golden (эталонным) файлом.

*Иногда юнит-тесты называют одинокими (solitary) в случае тотального применения имитаций и заглушек или общительными (sociable) в случае реальных коммуникаций с другими участниками.

*Правило трех А(AAA) (arrange, act, assert) или триада «дано, когда, тогда» — хорошая мнемоника, чтобы поддерживать хорошую структуру тестов.

Источники:

- [What is Component Testing? Techniques, Example Test Cases](#)
- [Лекция 5: Модульное и интеграционное тестирование](#)

Доп. материал:

- [Я сомневался в юнит-тестах, но...](#)
- [Юнит-тесты переоценены](#)
- [Elliotte Rusty Harold — Effective Unit Testing](#)
- [Kevlin Henney — What we talk about when we talk about unit testing](#)
- [Андрей Сербин — Компонентное тестирование инфраструктуры](#)
- [Анатомия юнит тестирования](#)
- [Unit Test](#)
- [Component Test](#)
- [Анатомия юнит-теста](#)
- [Почему большинство юнит тестов — пустая трата времени? \(перевод статьи\)](#)
- [Unit Testing Guide](#)
- [Лекция 2: Тестирование программного кода \(методы+окружение\)](#)

Интеграционное тестирование (Integration testing)

Интеграционное тестирование предназначено для проверки насколько хорошо два или более компонента 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 компонентного, поскольку также оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки (Stub) на месте отсутствующих модулей. Основная разница между компонентным 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.

Уровни интеграционного тестирования:

- **Компонентный** интеграционный уровень (CIT - [Component Integration testing](#)): Проверяется взаимодействие между компонентами одной системы после проведения компонентного тестирования. Программные компоненты или модули могут быть определены в разное время совершенно разными группами спецификаций, component integration testing выполняется чтобы убедиться, что даже после различий в разработке модулей интеграция всего работает вместе. В этом случае также важно учесть отрицательные случаи, так как компоненты могут делать предположения относительно данных;
- **Системный** интеграционный уровень (SIT - [System Integration testing](#)): - это полное тестирование всей системы, состоящей из множества подсистем. Основная цель SIT - обеспечить правильное функционирование всех зависимостей программных модулей и сохранение целостности данных между отдельными модулями всей системы. SUT ([System Under Test](#)) может состоять из аппаратного обеспечения, базы данных, программного обеспечения, комбинации аппаратного и программного

обеспечения или системы, требующей взаимодействия с человеком (HITL - [Human in the Loop](#) Testing). SIT имеет предварительное условие, при котором несколько базовых интегрированных систем уже прошли системное тестирование. Затем SIT проверяет необходимые взаимодействия между этими системами в целом. Результаты SIT передаются в UAT (пользовательское приемочное тестирование);

Интеграция может быть как программной, так и софт-железо:

- **HSIT** - Hardware Software Integration Testing: представляет собой процесс тестирования компонентов компьютерного программного обеспечения (CSC - Computer Software Components) на предмет функциональности высокого уровня в целевой аппаратной среде. Тестирование черного ящика - это основной тип тестирования, используемый на этом уровне тестирования. Целью тестирования интеграции аппаратного / программного обеспечения является проверка поведения разработанного программного обеспечения, интегрированного в аппаратный компонент. Цель тестирования интеграции аппаратного и программного обеспечения на основе требований (Requirement based Hardware-Software Integration Testing) - убедиться, что программное обеспечение на целевом компьютере удовлетворяет высокоуровневым требованиям (high-level requirements);
- **SSIT** - Software Software Integration Testing: это Computer Software Component Testing, работающего в среде целевого компьютера при моделировании всей системы [других CSC], и на функциональности высокого уровня. Оно фокусируется на поведении CSC в смоделированной среде хоста / цели. Для проверки интеграции программного обеспечения используются разные подходы;

Подходы к интеграционному тестированию:

- **Подход 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 неверно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования;
- **Инкрементальный подход (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 разными методами:
 - **Нисходящий подход (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;
 - **Подход 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 помочь 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. Пример низкоуровневого модуля - модуль, который заведует хранением токенов авторизации. Высокоуровневый - модуль авторизации, в состав которого помимо прочего входит модуль токенов. 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;
 - **Гибридный/сэндвич-подход (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;

Критерии начала и окончания Integration Testing:

Обычно при выполнении интеграционного тестирования используется стратегия [ETVX](#) (Entry Criteria, Task, Validation, Exit Criteria).

- Критерии начала:
 - завершено модульное тестирование;
- На входе:
 - Software Requirements Data;
 - Software Design Document;
 - Software Verification Plan;
 - Software Integration Documents;
- Действия:
 - На основе требований высокого и низкого уровня (High and Low-level requirements) создайте test cases and procedures;
 - Комбинируйте сборки низкоуровневых модулей, которые реализуют общую функциональность;
 - Разработайте тестовую обвязку (test harness);
 - Протестируйте сборку;
 - После прохождения теста сборка объединяется с другими сборками и тестируется до тех пор, пока система не будет интегрирована как единое целое;
 - Повторите все тесты на целевой processor-based platform и получите результаты;
- Критерии выхода:
 - Успешное завершение интеграции Программного модуля на целевое Hardware;
 - Правильная работа программного обеспечения в соответствии с указанными требованиями;
- На выходе:
 - Integration test reports;
 - SVCP - Software Test Cases and Procedures;

Test Harness- это набор заглушек, драйверов и других вспомогательных инструментов, необходимых для объединения двух компонентов кода или модуля, которые взаимодействуют друг с другом, чтобы проверить, соответствует ли комбинированное поведение ожидаемому или нет.

Test Driver и Test Stub являются искусственными заменами компонентов программы на время тестов по аналогии с моками в тестировании API. Тестовый драйвер - то, что вызывает тестируемый компонент. Тестовая заглушка - то, что возвращает тестируемому компоненту фиктивный ответ. Т.е. заглушки и драйверы не реализуют всю логику программного модуля, а только моделируют обмен данными с тестируемым модулем.

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 интерфейса включает в себя тестирование двух основных сегментов:

- Интерфейс веб-сервера и сервера приложений
- Интерфейс сервера приложений и базы данных

Тестирование потоков (Thread testing) - это вид 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 System Integration Testing. Поэтому его, вероятно, следует более правильно назвать «тестом взаимодействия потоков» (thread interaction test).

Thread Testing подразделяется на две категории:

- Однопоточное тестирование (Single thread testing) включает одну транзакцию приложения за раз;
- Многопоточное тестирование (Multi-thread testing) включает одновременно несколько активных транзакций;

Как проводить Thread Testing:

- 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 тестируть;
- Вместо непосредственного тестирования программных компонентов требуется, чтобы интеграторы сосредоточились на тестировании логических путей выполнения в контексте всей системы;

Советы:

- Протестируйте свою многопоточную программу, многократно выполняя ее с другим набором запущенных приложений;
- Протестируйте свою многопоточную программу, активировав одновременно несколько экземпляров программы;
- Выполняйте многопоточную программу на разных моделях оборудования с различными уровнями нагрузки и рабочими нагрузками;
- Инспекция кода;
- Собирайте только ошибки и сбои, которые произошли в потоках, отличных от основного;

Источники:

- [Integration Testing](#)
- [Лекция 5: Модульное и интеграционное тестирование](#)
- [What is Thread Testing in Software Testing?](#)

Доп. материал:

- [Интеграционные тесты в микросервисах](#)
- [Лекция 6: Интеграционное тестирование и его особенности для объектно-ориентированного программирования](#)
- [Для чего нужно интеграционное тестирование?](#)
- [Component / System integration testing examples](#)
- [#11 Артем и Сева. Моки\(Mocks\) и стабы\(Stubs\)](#)
- [Mocks Aren't Stubs](#)
- [Почему мы решили создать отдел кросс-системного тестирования](#)
- [Кто такой кросс-системный тестировщик и почему он не должен быть «agile»?](#)
- [What Is Thread Testing In Software Testing](#)

Системное тестирование (System Testing)

Системное тестирование означает тестирование всей системы в целом, оно выполняется после интеграционного тестирования, чтобы проверить, работает ли вся система целиком должным образом. В основном это тестирование типа «черный ящик», которое оценивает работу системы с точки зрения пользователя с помощью документа спецификации и оно не требует каких-либо внутренних знаний о системе, таких как дизайн или структура кода.

Основное внимание уделяется следующему:

- Внешние интерфейсы;

- Многопрограммность и сложный функционал;
- Безопасность;
- Восстановление;
- Производительность;
- Гладкое (smooth) взаимодействие оператора и пользователя с системой;
- Возможность установки;
- Документация;
- Удобство использование;
- Нагрузка / стресс;

Зачем нужно системное тестирование?

- Очень важно завершить полный цикл тестирования, и ST - это этап, на котором это делается;
- ST выполняется в среде, аналогичной production environment, и, следовательно, заинтересованные стороны могут получить хорошее представление о реакции пользователя;
- Это помогает свести к минимуму устранение неполадок после развертывания и количество обращений в службу поддержки;
- На этом этапе STLC тестируются архитектура приложения и бизнес-требования. Это тестирование очень важно, и оно играет важную роль в предоставлении клиенту качественного продукта;

Критерии начала системного тестирования:

- Система должна соответствовать критериям окончания интеграционного тестирования, то есть все test cases должны быть выполнены, и не должно быть открытых критических ошибок или ошибок с приоритетом P1, P2;
- System Test Plan должен быть одобрен и подписан;
- Test cases/scenarios/scripts должны быть готовы к выполнению;
- Все нефункциональные требования должны быть доступны, и для них должны быть созданы test cases;
- Среда тестирования должна быть готова;

Критерии окончания системного тестирования:

- Все test cases должны быть выполнены;
- В открытом состоянии не должно быть критических, приоритетных или связанных с безопасностью ошибок;
- Если какие-либо ошибки со средним или низким приоритетом находятся в открытом состоянии, они должны быть исправлены с согласия клиента;
- Отчет о выходе (Exit Report) должен быть отправлен;

Чем отличается системное тестирование от сквозного (E2E - end-to-end testing)?

Сквозное тестирование - это методология тестирования программного обеспечения для тестирования flow приложения от начала до конца. Целью сквозного тестирования является моделирование реального пользовательского сценария и проверка тестируемой системы и ее компонентов на предмет интеграции и целостности данных.

Системное тестирование - этап предпоследний этап STLC и уровень тестирования, а E2E - подход к тестам. Обычно сквозные тесты выполняют после системного тестирования и перед приемочным, а также после внесения изменений (smoke и regression). E2E выполняется от начала до конца в реальных сценариях, таких как взаимодействие приложения с оборудованием, сетью, базой данных и другими приложениями. Основная причина проведения этого тестирования - определение различных зависимостей приложения, а также обеспечение передачи точной информации между различными компонентами системы.

Источники:

- [What Is System Testing – A Ultimate Beginner’s Guide](#)
- [What Is End To End Testing: E2E Testing Framework With Examples](#)

Доп. материал:

[Лекция 7: Разновидности тестирования: системное и регрессионное тестирование](#)

Приемочное тестирование (AT – Acceptance testing)

После того, как процесс тестирования системы завершен командой тестирования, весь продукт передается клиенту и/или нескольким его пользователям для проверки приемлемости (acceptability). E2E бизнес-потоки проверяются аналогично в сценариях в реальном времени. Подобная производственной среда будет тестовой средой для приемочного тестирования (Staging, Pre-Prod, Fail-Over, UAT environment). Это метод тестирования черного ящика, при котором проверяется только функциональность, чтобы убедиться, что продукт соответствует указанным критериям приемки.

Виды приемочного тестирования:

- **Пользовательское** приемочное тестирование (UAT - User Acceptance Testing, validation, end-user testing) выполняется пользователем или клиентом чтобы определить, может ли ПО быть принято (accepted) или нет и проверить ПО на соответствие бизнес-требованиям. Могут существовать такие бизнес-требования и процессы, которые известны только конечным пользователям, и они либо пропускаются, либо неправильно интерпретируются, поэтому приемочное тестирование выполняется конечными пользователями, знакомыми с бизнес-требованиями;
- **Бизнес** - приемочное тестирование (BAT - Business Acceptance Testing) необходимо для оценки того, соответствует ли Продукт бизнес-целям и задачам. BAT в основном фокусируется на бизнес-преимуществах (финансах), которые являются довольно сложными из-за меняющихся рыночных условий / прогрессирующих технологий, так что текущая реализация может претерпеть изменения, которые приведут к дополнительным затратам. Даже Продукт, отвечающий техническим требованиям, может не пройти BAT по этим причинам;
- **Контрактное** приемочное тестирование (CAT - Contract Acceptance Testing) - это контракт, который определяет, что после того, как Продукт будет запущен в течение заранее определенного периода, должен быть проведен приемочный тест, и он должен пройти все приемочные тест-кейсы. Подписанный здесь контракт называется Соглашением об уровне обслуживания (SLA), которое включает условия, по которым платеж будет производиться только в том случае, если услуги Продукта соответствуют всем требованиям, что означает, что контракт выполнен. Иногда этот контракт может заключаться до того, как Продукт будет запущен. В любом случае, контракт должен быть четко определен с точки зрения периода тестирования, областей тестирования, условий по проблемам, возникающим на более поздних этапах, платежей и т. д.;
- **Правовое** приемочное тестирование (RAT - Regulations/Compliance Acceptance Testing) необходимо для оценки того, нарушает ли Продукт правила и нормы, установленные правительством страны, в которой он выпускается. Это может быть непреднамеренным, но отрицательно скажется на бизнесе. Обычно разрабатываемый Продукт / приложение, предназначенный для выпуска во всем мире, должен пройти RAT, поскольку в разных странах / регионах действуют разные правила и положения, определенные его руководящими органами. Если какие-либо правила и нормы нарушаются для какой-либо страны, то этой стране или конкретному региону в этой стране не будет разрешено использовать Продукт и это будет считаться отказом (Failure). Вендоры Продукта несут прямую ответственность, если Продукт будет выпущен даже при наличии нарушения;
- **Эксплуатационное** приемочное тестирование ([OAT - Operational Acceptance testing](#)) - это тип тестирования программного обеспечения, который оценивает эксплуатационную готовность программного приложения до его выпуска в производство. Целью эксплуатационного тестирования является обеспечение бесперебойной работы системы в ее стандартной эксплуатационной среде (SOE - standard operating environment). В основном это тестирование восстановления, совместимости, ремонтопригодности, доступности технической поддержки, надежности, восстановления после сбоя, локализации и т. д (recovery, compatibility, maintainability, technical support availability, reliability, fail-over, localization);
- **Альфа-тестирование** ([Alpha Testing](#)) проводят для оценки продукта в среде разработки / тестирования специализированной командой тестировщиков, обычно называемой альфа-тестерами. Здесь отзывы и

предложения тестировщиков помогают улучшить использование Продукта, а также исправить определенные ошибки;

- **Бета-тестирование, полевые испытания** ([Beta Testing](#), Field Testing) проводят для оценки Продукта, предоставляя его реальным конечным пользователям, обычно называемым бета-тестерами / бета-пользователями, в их среде. Собирается постоянная обратная связь от пользователей, и проблемы устраняются. Кроме того, это помогает в улучшении Продукта, чтобы обеспечить удобство работы пользователей. Тестирование происходит неконтролируемым образом, что означает, что у пользователя нет ограничений на использование Продукта;

Источники:

- [What Is Acceptance Testing \(A Complete Guide\)](#)
- [What Is User Acceptance Testing \(UAT\): A Complete Guide](#)

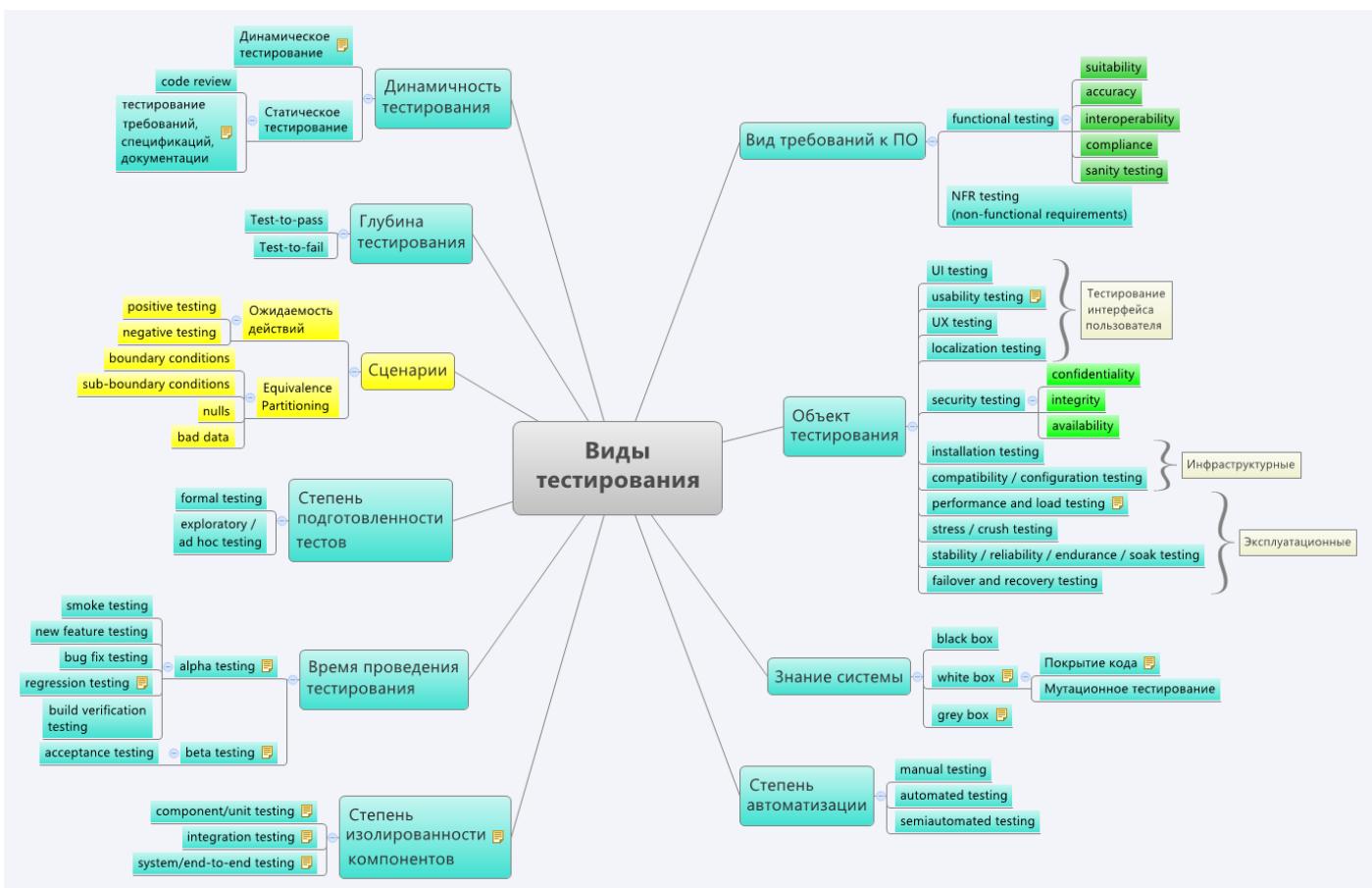
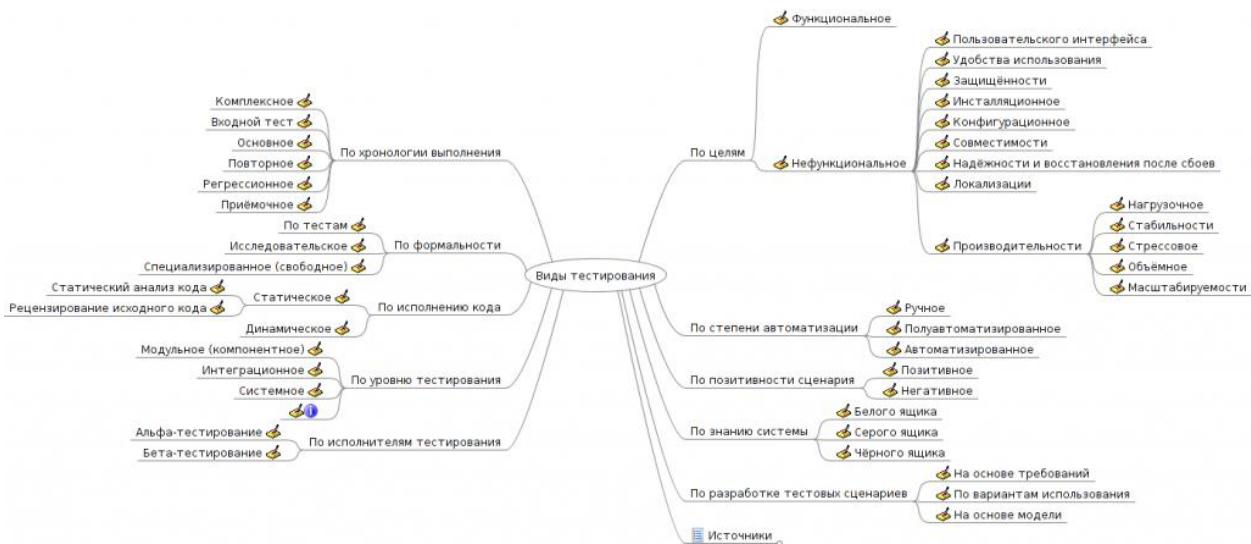
Доп. материал:

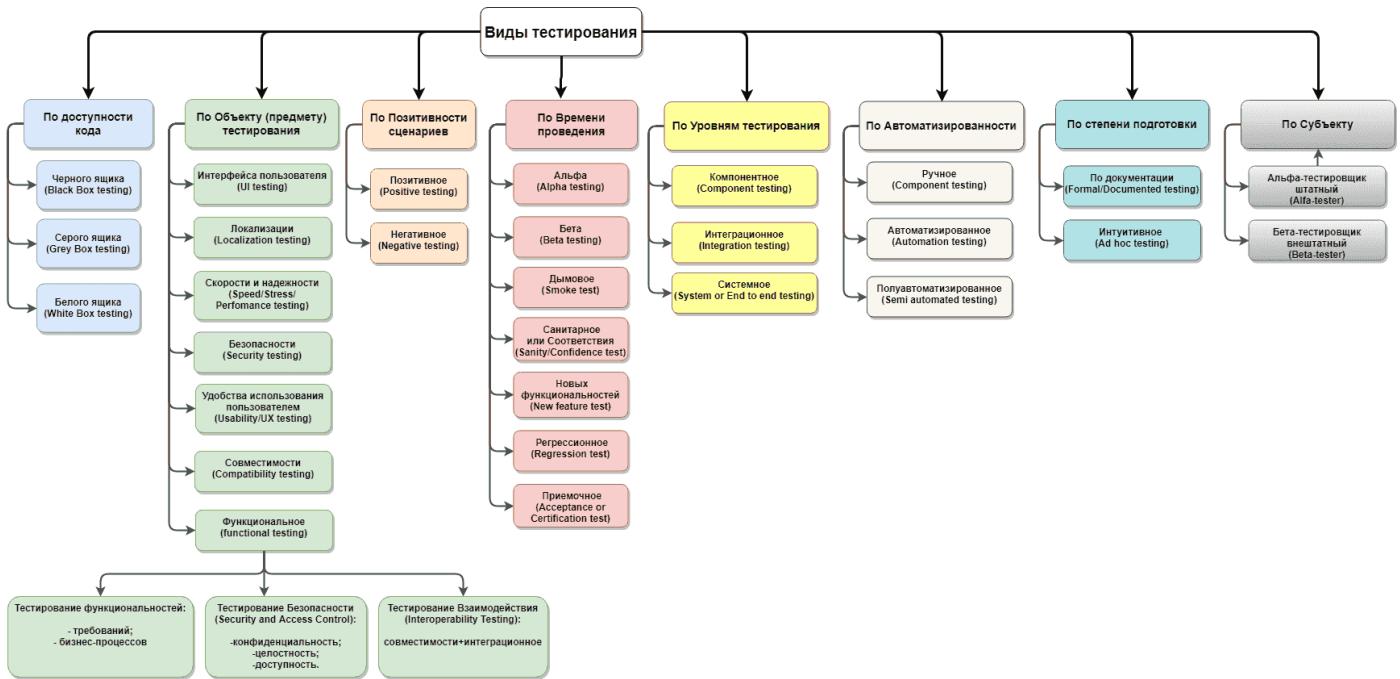
- [Business-facing tests](#)
- [Как проводить приемочное тестирование](#)

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 (Capacity testing)
 - Нагрузочное (Load testing)
 - Стressовое (Stress testing)
 - Масштабируемости (Scalability test)
 - Объемное тестирование (Volume testing)
 - Выносливости (Soak/Endurance testing)
 - Устойчивости (Resilience testing)
 - Стабильности/надежности (Stability / Reliability testing)
 - Отказ и восстановление (Failover and Recovery testing)
 - Эталонное и базовое тестирование (Benchmark and Baseline Testing)
 - Тестирование безопасности (Security and Access Control testing)
 - Удобство пользования (Usability testing)
 - Тестирование доступности (Accessibility testing)
 - Тестирование установки (Installation testing)
 - Тестирование на соответствие (Conformance/Compliance testing)
 - Конфигурационное (Configuration testing)
 - Тестирование локализации, глобализации и интернационализации
- Связанное с изменениями:
 - Регрессионное (Regression testing)
 - Тест работоспособности (Sanity testing)
 - Дымовое (Smoke testing)

Вообще виды тестирования можно классифицировать по самым разным критериям, поэтому можно встретить и такие схемы:





Еще классификаций на десерт: exploratory vs scripted; traditional vs agile; testing vs checking; standards-driven vs context-driven; phased vs. threaded.

Доп. материал:

- [Types of Testing - Software Testing Types Every QA Should Know](#)
- [Phased vs Threaded Testing](#)

Функциональное тестирование (Functional/Behavioral testing)

Функциональное тестирование выполняется чтобы убедиться, что каждая функция программного приложения ведет себя так, как указано в документе с требованиями. В большинстве случаев это выполняется методом black box testing.

Для функционального тестирования принято использовать две техники:

- Тестирование на основе требований: содержит все функциональные спецификации, которые составляют основу для всех тестов, которые будут проводиться;
- Тестирование на основе бизнес-сценариев: содержит информацию о том, как система будет восприниматься с точки зрения бизнес-процесса;

Основные виды функционального тестирования:

- Unit Testing: модульное тестирование обычно выполняется разработчиком и влечет за собой написание тестов, которые будут вызывать методы в каждом модуле и проверять их, передавая требуемые параметры и проверяя соответствие возвращаемого значения ожидаемому. Покрытие кода - важная часть модульного тестирования, где должны существовать test cases, охватывающие:
 - Line coverage;
 - Code path coverage;
 - Method coverage;
- Smoke Testing: тестирование, которое проводится после выпуска каждой сборки. Это также называется build verification testing;
- Sanity Testing: тестирование, которое проводится для того, чтобы убедиться, что все основные и жизненно важные функции приложения / системы работают правильно. Обычно это делается после Smoke Testing;
- Regression Tests: тестирование проводится для того, чтобы убедиться, что добавление нового кода, улучшений, исправление ошибок не нарушает существующую функциональность или не вызывает нестабильности и ПО все еще работает в соответствии со спецификациями. Регрессионные тесты не

должны быть такими обширными, как фактические функциональные тесты, но должны гарантировать объем покрытия, подтверждающий стабильность функциональности;

- Integration Tests: когда система полагается на несколько функциональных модулей, которые работают по отдельности, но должны работать согласованно когда объединены вместе, чтобы достичь сквозного сценария, проверка таких сценариев называется интеграционным тестированием;
- Beta/Usability Testing: продукт демонстрируется реальному пользователю в среде, приближенной к проду, и они тестируют продукт. Это похоже на User Acceptance testing;
- System testing: тестирование, которое выполняется для всей системы, чтобы проверить, работает ли она должным образом после интеграции всех модулей или компонентов;
- End to end testing: проводится для проверки функциональности продукта. Это тестирование выполняется только после завершения тестирования системной интеграции, включая функциональные и нефункциональные требования;

Критерии начала функционального тестирования:

- Requirement Specification document определен и утвержден;
- Подготовлены тест-кейсы;
- Созданы тестовые данные;
- Среда для тестирования готова, все необходимые инструменты доступны и готовы;
- Всё или часть приложения разработано, модульно протестировано и готово к тестированию;

Критерии окончания функционального тестирования:

- Выполнение всех функциональных тестов завершено;
- Нет критических или открытых ошибок Р1, Р2;
- Сообщенные ошибки были подтверждены;

Этапы функционального тестирования:

- Самый первый шаг заключается в определении функциональности продукта, который необходимо протестировать, и он включает в себя тестирование основных функций, условий ошибок и сообщений, тестирование удобства использования, то есть, является ли продукт удобным для пользователя или нет, и т. д.
- Следующим шагом является создание входных данных для проверяемой функциональности в соответствии со спецификацией требований.
- Позже, из спецификации требований, определяется результат для тестируемой функциональности.
- Подготовленные тест-кейсы исполняются.
- Фактический результат, то есть результат после выполнения тест-кейса, и ожидаемый результат (определенный из спецификации требований) сравниваются, чтобы определить, работает ли функциональность должным образом или нет.

Источник: [Complete Functional Testing Guide With Its Types And Example](#)

Нефункциональное тестирование (Non-Functional testing)

Нефункциональное тестирование проводится для проверки нефункциональных требований приложения, таких как производительность, безопасность, совместимость, надежность, удобство использования и т. д. В большинстве случаев это выполняется методом black box testing. Оно проверяет, соответствует ли поведение системы требованиям по всем аспектам, не охваченные функциональным тестированием. В нашем повседневном тестировании много внимания уделяется функциональному тестированию и функциональным требованиям и клиенты также заинтересованы в выполнении функциональных требований, которые напрямую связаны с функциональностью приложения, но когда ПО выходит на рынок и используется реальными конечными пользователями, у них есть шансы столкнуться с проблемами. Эти проблемы не связаны с функциональностью системы, но могут негативно повлиять на пользовательский опыт.

Нефункциональные требования могут быть отражены как:

- Пользовательские / Технические истории (User /Technical Stories): запись нефункциональных требований в виде пользовательской истории такая же, как и запись любых других требований. Единственная разница между пользователем и технической историей заключается в том, что пользовательская история требует обсуждения и имеет видимость (? visibility);
- В критериях приемки (Acceptance criteria): это точка, которая определяется для принятия продукта заказчиком. Нефункциональное требование должно быть включено в критерии приемки, но иногда невозможно проверить нефункциональные требования с каждой историей, то есть с каждой итерацией. Следовательно, требования следует добавлять или тестировать только с соответствующей итерацией;
- В артефактах (Artifact): для нефункциональных требований следует подготовить отдельный артефакт, это, в свою очередь, поможет лучше понять, что нужно тестировать и как это можно делать в итерациях;

Документ подхода к тестированию (Approach Document):

Разработайте конкретный подход к этапу тестирования, уточнив общую стратегию тестирования. Этот подход к тестированию помогает при планировании и выполнении всех задач тестирования:

- Объем испытаний (Test Scope);
- Метрики тестирования;
- Инструменты тестирования;
- Основные даты и результаты;

Виды нефункционального тестирования (список не полный):

- Тестирование производительности (Performance Testing)
- Нагрузочное тестирование (Load Testing)
- Стрессовое тестирование (Stress Testing)
- Объемное тестирование (Volume Testing)
- Тестирование восстановления (Recovery Testing)
- Тестирование отказоустойчивости (Failover Testing)
- Тестирование эффективности (Efficiency Testing)
- Тестирование аварийного восстановления (Disaster Recovery Testing)
- Тестирование установки (Installation Testing)
- Тестирование документации (Documentation Testing)
- Тестирование на удобство использования (Usability Testing)
- Тестирование графического интерфейса пользователя (User Interface Testing)
- Тестирование совместимости (Compatibility Testing)
- Тестирование обслуживаемости (Maintainability Testing)
- Тестирование безопасности (Security Testing)
- Тестирование масштабируемости (Scalability Testing)
- Тестирование выносливости (Endurance Testing)
- Тестирование надежности (Reliability Testing)
- Тестирование соответствия (Compliance Testing)
- Тестирование локализации (Localization Testing)
- Тестирование интернационализации (Internationalization Testing)
- Тестирование переносимости (Portability Testing)
- Тестирование на основе базового уровня (Baseline Testing)

Примеры чек-листов:

Тестирование производительности:

- Время отклика (The response time) приложения, то есть сколько времени требуется для загрузки приложения, за какое время любой ввод, предоставленный приложению, обеспечивает вывод, время обновления браузера и т. д.;

- Пропускную способность (Throughput) следует проверять по количеству транзакций, завершенных во время нагружочного теста;
- Настройка среды (Environment) должна быть такой же, как и в реальной среде, иначе результаты не будут такими же;
- Время процесса (Process time) - такие действия, как импорт и экспорт Excel, любые вычисления в приложении должны быть протестированы;
- Совместимость (Interoperability) должна быть проверена, т.е. программное обеспечение должно иметь возможность взаимодействовать с другим программным обеспечением или системами;
- Необходимо проверить время ETL, то есть время, затраченное на извлечение, преобразование и загрузку данных из одной базы данных в другую;
- Необходимо проверить возрастающую нагрузку (Load) на приложение;

Тестирование безопасности:

- Аутентификация (Authentication): только достоверный пользователь может войти в систему;
- Авторизация (Authorized): пользователь должен иметь возможность входить в те модули, для которых он авторизован или к которым пользователю был предоставлен доступ;
- Пароль: Требование пароля должно быть подтверждено, т.е. пароль должен соответствовать тому, как это требование определяется, то есть длине, специальным символам, числам и т. д.;
- Тайм-аут: если приложение неактивно, оно должно истечь по таймауту в указанное время;
- Резервное копирование данных: резервное копирование данных должно быть выполнено в указанное время и данные должны быть скопированы в безопасное место;
- Внутренние ссылки на веб-приложение не должны быть доступны, если размещены непосредственно в браузере;
- Вся коммуникация должна быть зашифрована;

Тестирование документации:

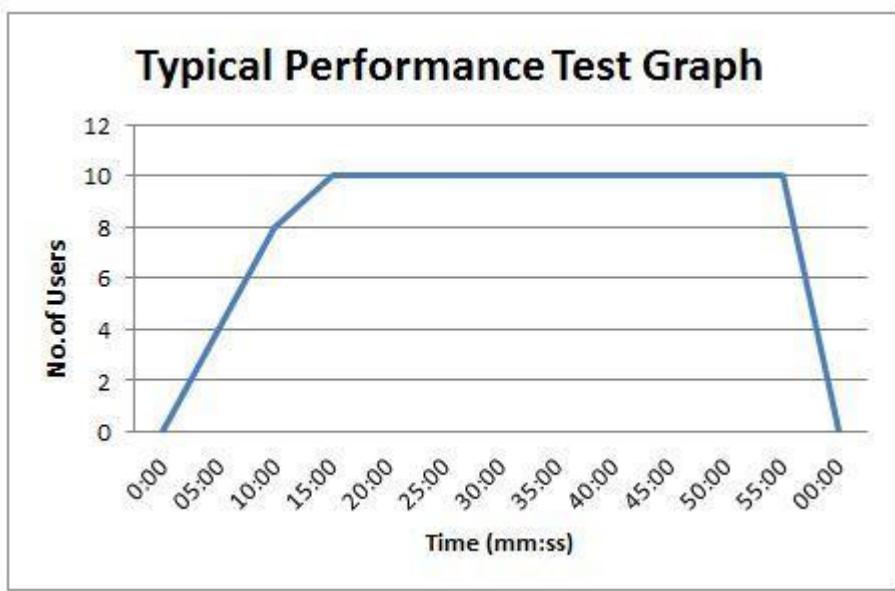
- Пользовательская и системная документация;
- Документы для учебных целей;

Источник: [A Complete Non-Functional Testing Guide For Beginners](#)

Тестирование производительности (Performance testing)

Тестирование производительности - это нефункциональный вид тестирования программного обеспечения, используемый для проверки скорости, времени отклика, стабильности, надежности, масштабируемости и использования ресурсов программного приложения при определенной рабочей нагрузке, обычно регрессионным образом, когда в приложение ежедневно или еженедельно вносятся небольшие изменения. Основная цель тестирования производительности - выявить и устранить узкие места производительности в программном приложении. Это подмножество performance engineering, также известное как «Perf Testing». Само по себе оно не призвано находить дефекты, но оно помогает в обнаружении узких мест в системе.

Обычно продолжительность теста производительности составляет 1 час (устойчивое состояние) на средней / ожидаемой нагрузке; это может варьироваться в зависимости от вашего SLA / требований.



Примечание 1: все подвиды тестирования производительности отличаются, грубо говоря, только параметрами (тип возрастания нагрузки, ее количество, длительность и т.п.) и собираемыми метриками (без которых это тестирование бессмысленно). Точкой отсчета для всех подвидов принято брать результаты Capacity testing.

| Risks | Performance test types | | | | | | | | | |
|---|------------------------|-----------|-----------|---------------|------|-------|-------|--------|------|------------|
| | Capacity | Component | Endurance | Investigation | Load | Smoke | Spike | Stress | Unit | Validation |
| Speed-related risks | | | | | | | | | | |
| User satisfaction | | | X | X | X | | | X | | X |
| Synchronicity | | X | X | X | X | | X | X | X | |
| Service Level Agreement (SLA) violation | | | X | X | X | | | | | X |
| Response time trend | | X | X | X | X | X | | | X | |
| Configuration | | | X | X | X | X | | X | | X |
| Consistency | X | X | X | X | | | | X | X | X |
| Scalability-related risks | | | | | | | | | | |
| Capacity | X | X | X | X | X | | | | | X |
| Volume | X | X | X | X | X | | | | | X |
| SLA violation | | | X | X | | | | | | X |
| Optimization | X | X | | X | X | | | | | X |
| Efficiency | X | X | | X | | | | | X | |
| Future growth | X | X | | X | X | | | | | X |
| Resource consumption | X | X | X | X | X | X | | X | X | X |
| Hardware / environment | X | X | X | X | X | X | | X | | X |
| Service Level Agreement (SLA) violation | X | X | X | X | X | | | | | X |
| Stability-related risks | | | | | | | | | | |
| Reliability | | X | X | X | X | X | X | X | X | |
| Robustness | | X | X | X | X | X | X | X | X | |
| Hardware / environment | | | X | X | X | X | X | X | | X |
| Failure mode | | X | X | X | X | X | X | X | X | X |
| Slow leak | X | X | X | X | | | | | | X |
| Service Level Agreement (SLA) violation | X | X | X | X | | | | | | |
| Recovery | X | | | X | | X | X | X | | X |
| Data accuracy and security | X | X | X | X | | X | X | X | | X |
| Interfaces | X | X | X | X | X | | X | X | | X |

Примечание 2: 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 (те же перцентили).

Общие проблемы с производительностью.

Большинство проблем с производительностью связаны со скоростью, временем отклика, временем загрузки и плохой масштабируемостью. Скорость часто является одним из самых важных атрибутов приложения. Медленно работающее приложение потеряет потенциальных пользователей. Тестирование производительности проводится для того, чтобы убедиться, что приложение работает достаточно быстро, чтобы удерживать внимание и интерес пользователя. Взгляните на следующий список распространенных проблем с производительностью и обратите внимание на то, что скорость является общим фактором во многих из них:

- Длительное время загрузки - обычно время загрузки - это начальное время, необходимое приложению для запуска. Обычно оно должно быть сведено к минимуму. Хотя некоторые приложения невозможно загрузить менее чем за минуту, время загрузки должно быть меньше нескольких секунд, если это возможно;
- Плохое время отклика. Время отклика - это время, которое проходит с момента ввода пользователем данных в приложение до того, как приложение выдаст ответ на этот ввод. Как правило, это должно происходить очень быстро. Опять же, если пользователю приходится ждать слишком долго, он теряет интерес;
- Плохая масштабируемость - программный продукт страдает от плохой масштабируемости, когда он не может обрабатывать ожидаемое количество пользователей.
- Узкие места (Bottlenecking)- это препятствия в системе, которые ухудшают общую производительность системы. Узкое место - это либо ошибки в коде, либо проблемы с оборудованием, которые вызывают снижение пропускной способности при определенных нагрузках. Узкие места обычно устраняются путем исправления плохо работающих процессов или добавления дополнительного оборудования.
Некоторые общие узкие места производительности:
 - Загрузка ЦП;
 - Использование памяти;
 - Использование сети;
 - Ограничения операционной системы;
 - Использование диска;

Как проводить тестирование производительности?

- Определите необходимую среду тестирования (testing environment): перед тем, как начать процесс тестирования, изучите детали аппаратного, программного обеспечения и сетевых конфигураций, используемых во время тестирования. Это поможет тестировщикам создавать более эффективные тесты. Это также поможет определить возможные проблемы, с которыми тестировщики могут столкнуться во время процедур тестирования производительности;
- Определите критерии приемки: сюда входят цели и ограничения по пропускной способности, времени отклика и распределению ресурсов. Также необходимо определить критерии успеха проекта, выходящие за рамки этих целей и ограничений. Тестировщики должны иметь право устанавливать критерии и цели производительности, потому что часто спецификации проекта не включают достаточно широкий спектр тестов производительности. Иногда их может и вовсе не быть. Если возможно, найти похожее приложение для сравнения - это хороший способ установить цели производительности;
- Планирование и проектирование тестов производительности: Определите, как использование может варьироваться среди конечных пользователей, и определите ключевые сценарии для тестирования всех возможных вариантов использования. Необходимо смоделировать различных конечных пользователей, спланировать данные тестирования производительности и наметить, какие метрики будут собраны;
- Настройка тестовой среды: Подготовьте тестовую среду перед выполнением. Также подготовьте инструменты и другие ресурсы;
- Имплементирование тест-дизайна: Создайте тесты производительности в соответствии с вашим тест-дизайном;
- Запуск тестов: Выполнить и промониторить тесты;

- Анализ, настройка и повторное тестирование: объединяйте, анализируйте и делитесь результатами тестирования. Затем выполните точную настройку и снова проверьте, есть ли улучшение или снижение производительности. Поскольку улучшения обычно становятся меньше с каждым повторным тестом, остановитесь, когда узкое место вызвано ЦП. Тогда вы можете рассмотреть вариант увеличения мощности процессора.

Метрики тестирования производительности (Performance Testing Metrics):

- **Использование процессора (Processor Usage):** время, затрачиваемое процессором на выполнение non-idle потоков;
- **Использование памяти (Memory use):** объем физической памяти, доступной процессам на компьютере;
- **Время диска (Disk time):** время, в течение которого диск занят выполнением запроса на чтение или запись;
- **Время отклика (Response time):** время с момента ввода пользователем запроса до получения первого символа ответа. Подробнее: [раз](#), [два](#);
- **Задержка (Latency):** временной интервал между запросом и ответом;
- **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.
- **Частные байты (Private bytes):** количество байтов, выделенных процессом, которые не могут использоваться другими процессами. Они используются для измерения утечек и использования памяти;
- **Выделенная память (Committed memory)** - объем используемой виртуальной памяти;
- **Страниц памяти в секунду (Memory pages/second)** - количество страниц, записываемых на диск или считываемых с диска для устранения аппаратных ошибок страниц. Сбои аппаратной страницы - это когда код не из текущего рабочего набора вызывается из другого места и извлекается с диска;
- **Ошибок страниц в секунду (Page faults/second)** - общая скорость, с которой страницы ошибок обрабатываются процессором. Это происходит, когда процессу требуется код из-за пределов своего рабочего набора;
- **Число прерываний процессора в секунду (CPU interrupts per second):** это среднее количество аппаратных прерываний, которые процессор принимает и обрабатывает каждую секунду;
- **Длина дисковой очереди (Disk queue length):** среднее количество запросов на чтение и запись, поставленных в очередь для выбранного диска в течение интервала выборки;
- **Длина сетевой выходной очереди (Network output queue length):** длина очереди выходных пакетов в пакетах. Значение больше двух означает, что необходимо убрать задержку и возникновение узких мест;
- **Всего сетевых байтов в секунду (Network bytes total per second):** скорость, с которой байты отправляются и принимаются интерфейсом, включая символы кадрирования;
- **Количество пулов соединений (Amount of connection pooling):** количество запросов пользователей, которые удовлетворяются объединенными соединениями. Чем больше запросов будет выполнено подключениями в пуле, тем выше будет производительность;
- **Максимальное количество активных сессий (Maximum active sessions):** максимальное количество сессий, которые могут быть активны одновременно;
- **Коэффициент хитов (Hit ratios):** это связано с количеством операторов SQL, которые обрабатываются кэшированными данными вместо дорогостоящих операций ввода-вывода. Это хорошее начало для решения проблем, связанных с узкими местами;

- **Хитов в секунду** (Hits per second): количество хитов веб-сервера в течение каждой секунды нагрузочного теста;
- **Сегмент отката** (Rollback segment): объем данных, который можно откатить в любой момент времени;
- **Блокировки баз данных** (Database locks) - блокировки таблиц и баз данных необходимо отслеживать и тщательно настраивать;
- **Максимальное время ожидания** (Top waits) - отслеживаются, чтобы определить, какое время ожидания можно сократить при работе с тем, насколько быстро данные извлекаются из памяти;
- **Количество потоков** (Thread counts): состояние приложения можно измерить по количеству потоков, которые работают и в настоящее время активны;
- **Сборка мусора** (Garbage collection): это связано с возвратом неиспользуемой памяти обратно в систему. Сборку мусора необходимо отслеживать на предмет эффективности;
- *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, медленный 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.

Примеры проверок:

- Время отклика не превышает 4 секунды при одновременном доступе к сайту 1000 пользователей;
- Время отклика приложения под нагрузкой находится в допустимом диапазоне при медленном сетевом подключении;
- Проверьте максимальное количество пользователей, с которыми приложение может справиться, прежде чем оно выйдет из строя;
- Проверить время выполнения базы данных при одновременном чтении / записи 500 записей;

- Проверьте использование ЦП и памяти приложением и сервером базы данных в условиях пиковой нагрузки;
- Проверьте время отклика приложения в условиях низкой, нормальной, средней и высокой нагрузки;

Во время фактического выполнения теста производительности расплывчатые термины, такие как допустимый диапазон, большая нагрузка и т. д., заменяются конкретными цифрами. Инженеры по производительности устанавливают эти числа в соответствии с бизнес-требованиями и техническим ландшафтом приложения.

Источники:

- [Performance Testing Tutorial: What is, Types, Metrics & Example](#)
- [Do you really know all types of Performance Tests \(Non-Functional Tests\)?](#)

Доп. материал:

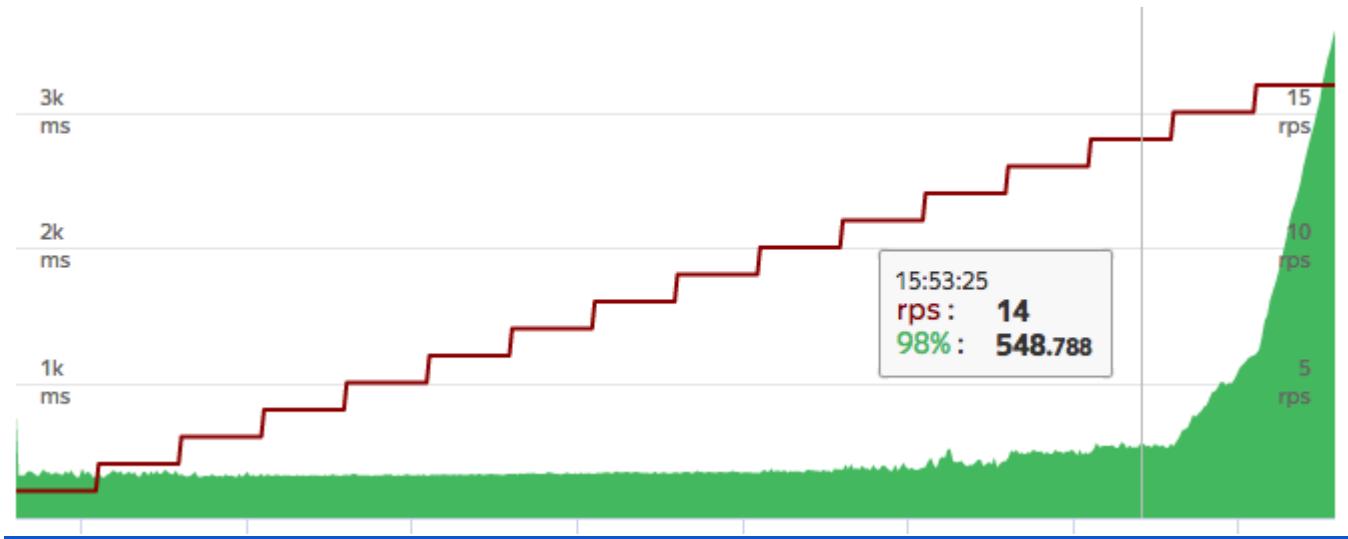
- [Анализ результатов нагрузочного тестирования](#)
- [Нагрузочное тестирование выполнять сложно, а инструменты далеки от совершенства. Почему?](#)
- [Нагрузочное тестирование: особенности профессии](#)
- [Андрей Акиньшин — Анализируем перформанс с пользой для себя и окружающих](#)
- [Антон Поцюс — Нагрузочное тестирование игрового сервера](#)
- [Сергей Махетов — Воркшоп \(часть 1\): Стартуем в тестировании производительности](#)
- [Сергей Махетов — Воркшоп \(часть 2\): Стартуем в тестировании производительности](#)
- [Инструментарий для нагрузочного тестирования и не только](#)
- [Представляем RAIL: модель оценки производительности сайта](#)
- [30+ Performance Testing Interview Questions And Answers](#)

Тестирование емкости (Capacity testing)

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.

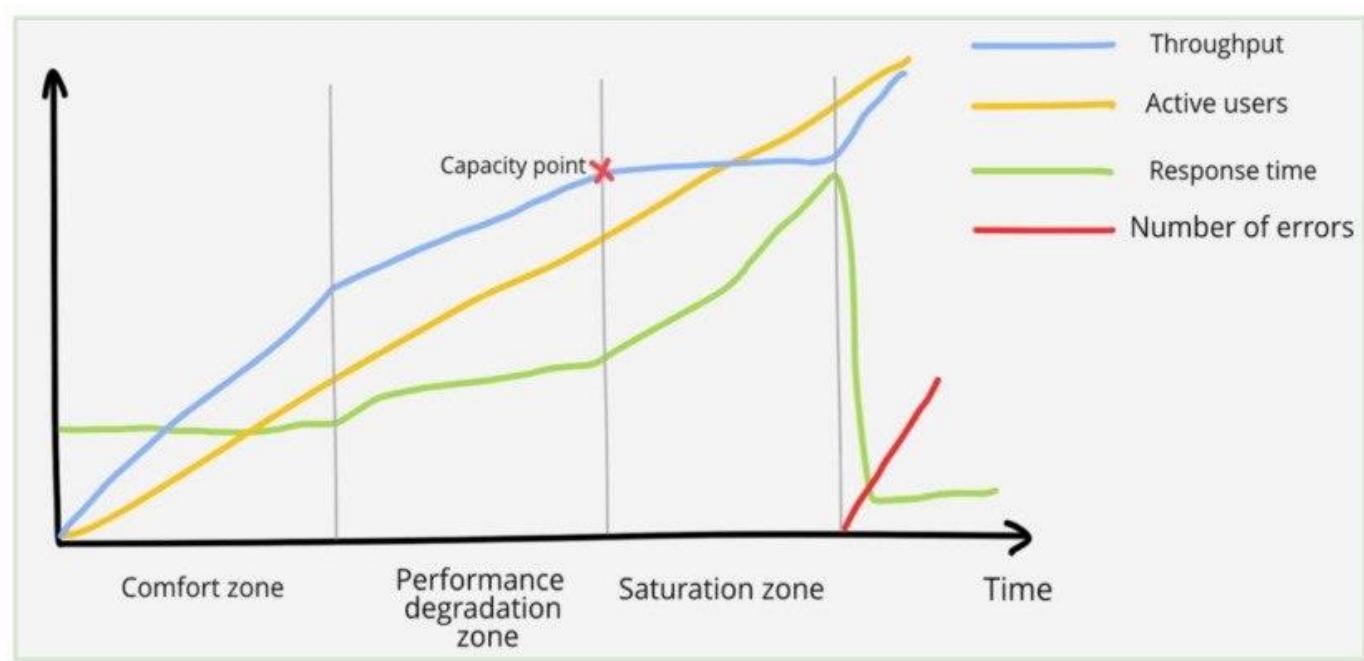
Тестирование емкости гарантирует, что приложение и среда могут беспрепятственно обрабатывать максимальное количество пользователей или транзакций в соответствии с требованиями к производительности, определенными в вашем соглашении об уровне обслуживания (SLA). Тестирование емкости нацелено на тестирование максимальной емкости вашей системы с точки зрения трафика, при этом обеспечивая оптимальное взаимодействие с пользователем. Общие примеры SLA по производительности включают время загрузки домашней страницы, время ответа веб-страницы, продолжительность транзакции (например, время входа в учетную запись, время поиска и платежа). Общая цель состоит в том, чтобы идентифицировать «зону безопасности» системы и поддерживать ее в максимально возможной степени. Тестирование емкости помогает определить, в какой степени она может быть расширена без ущерба для опыта конечного пользователя.

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.



Обратите внимание: от 1 до 12 rps процентиль времени ответа практически не изменяется. Только на 13 и 14 rps мы видим незначительный рост, который не изменяется с течением времени. Это значит, что мы практически исчерпали ресурс, в который упремся, но очередь еще не образуется. На 15 rps время ответа начало расти экспоненциально, значит это и есть наш предел. Таким образом, можно сделать вывод, что емкость =14 rps. Следующий шаг — поиск ресурса который исчерпался и не дает системе обрабатывать больше 14 rps.

Capacity test flow:



Capacity point – точка, где перестает расти пропускная способность и увеличивается время ответа.

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. Начать стоит с минимального набора:

- Основные физические, функциональные, компоненты сервера(железо): процессор, память, сеть, жесткий диск.

- Метрики операционной системы: прерывания, переключение контекстов, среднее значение загрузки системы за 1, 5 и 15 минут.
- Метрики программного обеспечения, используемого на сервере.
- По возможности постарайтесь определять в каких пропорциях ресурсы используются вашим программным обеспечением.

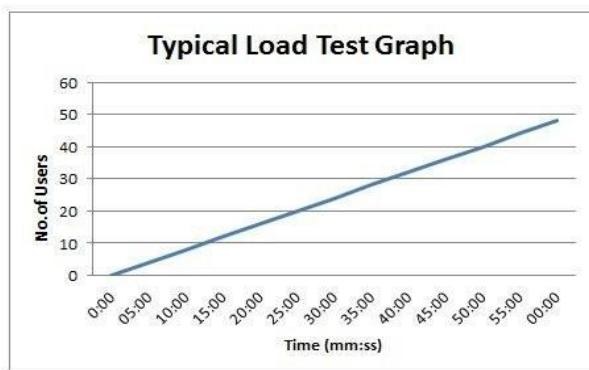
Источники:

- [All You Need to Know About Capacity Testing](#)
- [Тестируем производительность: как определить емкость системы](#)

Нагрузочное тестирование (Load testing)

Нагрузочное тестирование - это тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе. Этот подвид тестирования производительности выполняется для диагностики поведения системы при увеличении рабочей нагрузки.

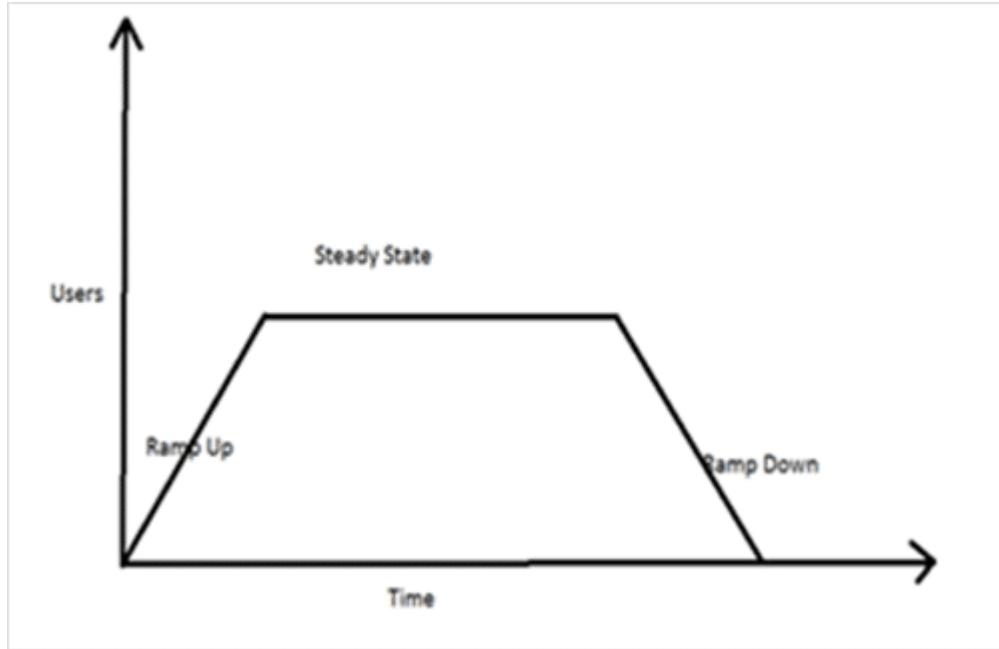
Пример. Предположим, что требование нашего клиента для страницы входа составляет 2–5 секунд, и эти 2–5 секунд должны быть всегда, пока нагрузка не достигнет 5000 пользователей, а также если количество пользователей постепенно увеличивается, то сколько ЦП, памяти будет потребляться, каково состояние сети, время отклика, пропускная способность и т.д.



Подход к нагрузочному тестированию:

1. Определите критерии приемки нагрузочного теста. Например:
 - Время отклика страницы входа не должно превышать 5 секунд даже в условиях максимальной нагрузки;
 - Загрузка ЦП не должна превышать 80%;
 - Пропускная способность системы должна составлять 100 транзакций в секунду;
2. Определите бизнес-сценарии, которые необходимо протестировать. Не тестируйте все потоки, постарайтесь понять основные business flows, которые, как ожидается, будут происходить в производственной среде. Если это уже существующее приложение, мы можем получить информацию из логов. Если это новое приложение, нам нужно работать с бизнес-группами, чтобы понять закономерности потока, использование приложения и т. д. Иногда команда проекта проводит семинары, чтобы дать обзор или подробности о каждом компоненте приложения;
3. Моделирование рабочей нагрузки. Получив подробную информацию о бизнес-потоках, шаблонах доступа пользователей и количестве пользователей, нам нужно спроектировать рабочую нагрузку таким образом, чтобы она имитировала фактическую навигацию пользователя в производственной среде или как она ожидается в будущем, когда приложение будет в производстве. Ключевые моменты, которые следует помнить при разработке модели рабочей нагрузки, - это увидеть, сколько времени потребуется для завершения конкретного бизнес-потока. Здесь нам нужно назначить время обдумывания (think time) таким образом, чтобы пользователь мог более реалистично перемещаться по приложению. Схема рабочей нагрузки обычно будет с нарастанием, спадом и устойчивым

состоянием (Ramp up, Ramp down and a steady state).



Устойчивое состояние обычно представляет собой одн часовое испытание под нагрузкой с нарастанием 15 минут и замедлением 15 минут.

Пример модели рабочей нагрузки: Предположим, что это интернет-магазин, где пользователи войдут в приложение и получат широкий выбор платьев для покупок и могут перемещаться по каждому продукту. Чтобы просмотреть подробную информацию о каждом продукте, им нужно щелкнуть по продукту. Если им нравится цена и качество продукта, они могут добавить его в корзину и купить продукт, выполнив проверку и произведя оплату. Список сценариев:

- 1. Обзор - здесь пользователь запускает приложение, входит в приложение, просматривает различные категории и выходит из приложения;
- 2. Обзор, Просмотр продукта, Добавить в корзину - здесь пользователь входит в приложение, просматривает различные категории, просматривает сведения о продукте, добавляет продукт в корзину и выходит из системы;
- 3. Обзор, просмотр продукта, добавление в корзину и оформление заказа - в этом сценарии пользователь входит в приложение, просматривает различные категории, просматривает сведения о продукте, добавляет продукт в корзину, оформляет заказ и выходит из системы;
- 4. Обзор, Просмотр продукта, Добавить в корзину, Оформить заказ и произвести оплату - здесь пользователь входит в приложение, просматривает различные категории, просматривает сведения о продукте, добавляет продукт в корзину, оформляет заказ, производит оплату и выходит из системы.

| Business Flow | Количество транзакций | Виртуальная пользовательская нагрузка | Время отклика (сек) | % Допустимая частота отказов | Транзакций в час |
|---------------|-----------------------|---------------------------------------|---------------------|------------------------------|------------------|
| 1 | 17 | 1600 | 3 | Менее 2% | 96000 |
| 2 | 17 | 200 | 3 | Менее 2% | 12000 |
| 3 | 18 | 120 | 3 | Менее 2% | 7200 |
| 4 | 20 | 80 | 3 | Менее 2% | 4800 |

Приведенные выше значения были получены на основе следующих расчетов: Транзакций в час = Количество пользователей * Транзакции, совершенные одним пользователем за один час. Количество пользователей = 1600. Общее количество транзакций в сценарии просмотра = 17. Время отклика для каждой транзакции = 3. Общее время, за которое один пользователь совершил 17 транзакций = $17 \times 3 = 51$, округленное до 60 секунд (1 мин). Транзакций в час = $1600 \times 60 = 96000$ Транзакций.

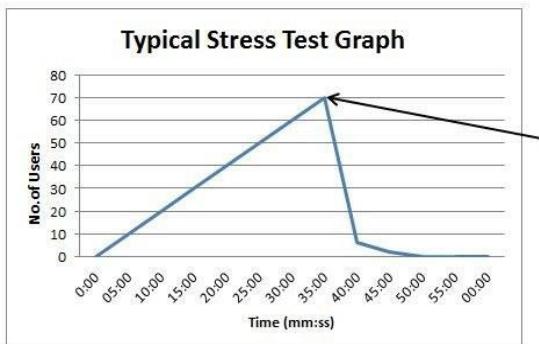
4. Дизайн / разработка нагрузочных тестов - Нагрузочный тест должен быть разработан с использованием данных, которые мы уже собрали, то есть бизнес-потоков, количества пользователей, пользовательских шаблонов, показателей, которые необходимо собрать и проанализировать. Более того, тесты должны быть максимально реалистичными.
5. Выполнение нагрузочного теста - перед тем, как мы выполним нагрузочный тест, убедитесь, что приложение запущено и работает. Среда нагрузочного тестирования готова. Приложение функционально протестировано и работает стабильно. Проверьте параметры конфигурации среды нагрузочного тестирования. Она должна быть такой же, как и в производственной среде. Убедитесь, что доступны все тестовые данные. Не забудьте добавить необходимые метрики для отслеживания производительности системы во время выполнения теста. Всегда начинайте с небольшой нагрузки и постепенно увеличивайте ее. Никогда не начинайте работу с полной нагрузкой.
6. Анализ результатов нагрузочного теста - имейте базовый тест (baseline test), чтобы всегда сравнивать его с другими тестами. Соберите метрики и журналы сервера после запуска теста, чтобы найти узкие места. Некоторые проекты используют инструменты мониторинга производительности приложений для мониторинга системы во время тестового запуска, эти инструменты APM (Application Performance Monitoring) помогают легче определить основную причину и сэкономить много времени.
7. Отчетность - после завершения тестового прогона соберите все показатели и отправьте сводный отчет теста соответствующей группе со своими наблюдениями и рекомендациями.

Источники:

- [Load Testing Complete Guide For Beginners](#)
- [Do you really know all types of Performance Tests \(Non-Functional Tests\)?](#)

Стрессовое тестирование (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». Стress-тестирование - это отрицательное / негативное тестирование, которые проводят при больших нагрузках или нагрузках, выходящих за допустимые пределы, чтобы определить поведение системы при таких обстоятельствах, точку отказа системы (числовые показатели метрик), показываются ли корректные ошибки при этом и не теряются ли данные. Это тестирование также называют Fatigue testing.



Виды стресс-тестирования:

- Распределенное стресс-тестирование: тесты выполняются для всех клиентов с сервера для отслеживания их статуса, а также для выявления сбоев из-за чрезмерного стресса;
- Стress-тестирование приложения: основное внимание в этом тестировании уделяется обнаружению дефектов в программном обеспечении, связанных с блокировкой данных (locking and blocking), проблемами сети и узкими местами производительности;

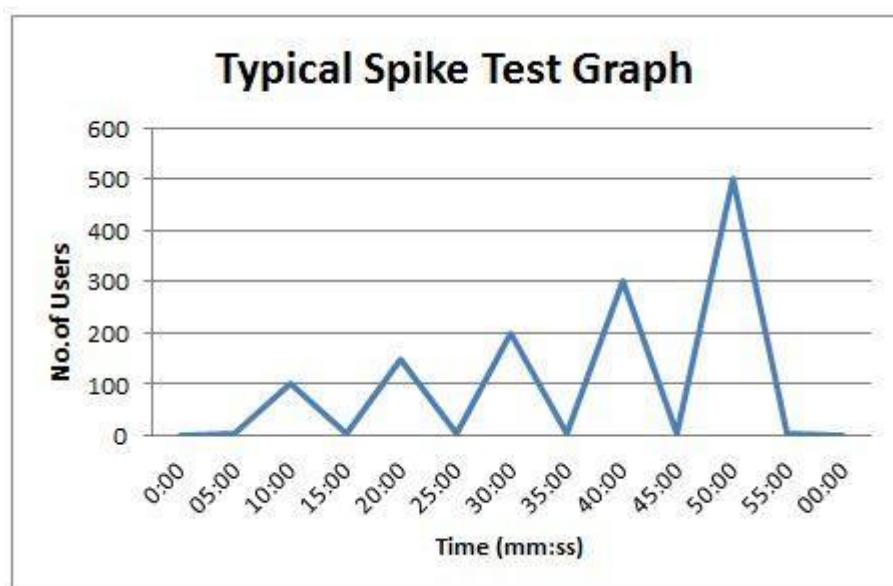
- Транзакционное стресс-тестирование: транзакционное тестирование выполняет стресс-тестирование одной или нескольких транзакций между различными программными продуктами или приложениями. Его основная цель - точная настройка и оптимизация системы для повышения ее производительности;
- Систематическое стресс-тестирование: интегрированное стресс-тестирование, систематическое стресс-тестирование, используется для тестирования нескольких систем, работающих на одном сервере. Это позволяет группе тестирования обнаруживать дефекты, когда данные одного программного обеспечения блокируют другое программное обеспечение;
- Исследовательское стресс-тестирование: используется для тестирования системы в необычных условиях, которые маловероятны в реальном сценарии. Эти сценарии стресс-тестирования позволяют группе обнаруживать различные необнаруженные проблемы и ошибки в системе;

Подход / процесс содержит те же этапы, что описаны в нагрузочном, и то же самое в остальных подвидах, поэтому дублировать нет смысла.

Примеры:

- 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);
- MS Word может выдать сообщение об ошибке «Не отвечает» при попытке скопировать файл размером 7–8 ГБ. Вы засыпали Word файлом огромного размера, и он не смог обработать такой большой файл, и в результате он завис;
- Когда большое количество пользователей одновременно получают доступ к логину / регистрации;
- Когда несколько пользователей пытаются загрузить один файл одновременно;
- База данных отключается, когда к ней обращается большое количество пользователей;
- Много пользователей заходят на веб-сайт, в то время как на сервере запускается сканирование на вирусы;
- Огромный объем данных копируется из буфера обмена и сохраняется в веб-форме большим количеством пользователей одновременно;
- Огромное количество пользователей одновременно нажимают кнопку «Добавить в корзину» на сайте покупок;

Пиковое тестирование (Spike Testing) - это разновидность стресс-тестирования, проводится для проверки характеристик производительности, когда тестируемая система подвергается моделям рабочей нагрузки и объемам нагрузки, которые многократно увеличиваются за пределы ожидаемых производственных операций в течение коротких периодов времени. Обычно продолжительность теста составляет 1 час; он может отличаться в зависимости от вашего SLA / требований.

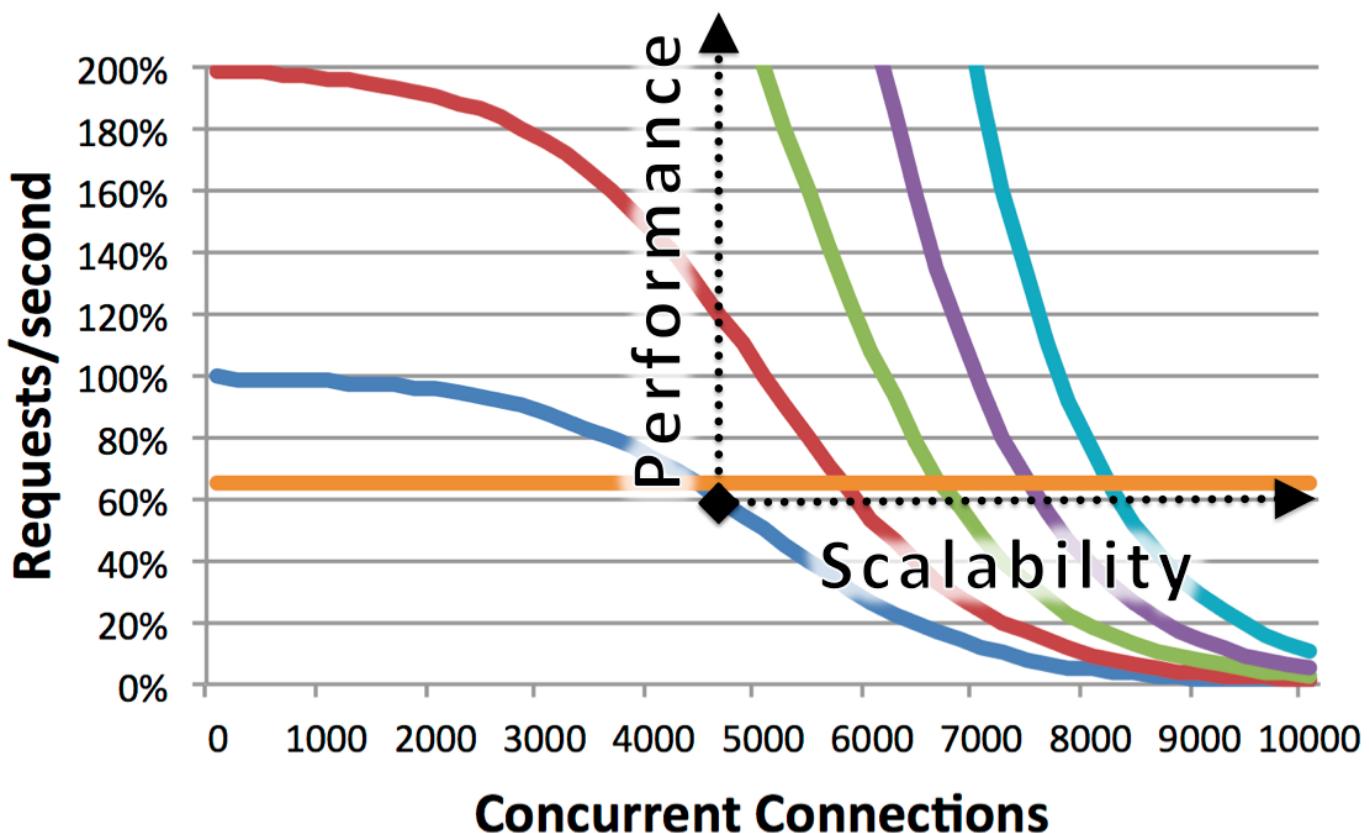


Источники:

- [What is Stress Testing?](#)
- [Stress Testing](#)
- [Stress Testing Guide For Beginners](#)
- [Do you really know all types of Performance Tests \(Non-Functional Tests\)?](#)

Тестирование масштабируемости (Scalability testing)

Тестирование масштабируемости проводится для определения способности приложения масштабироваться с точки зрения пользовательской нагрузки, количества транзакций, объема данных и т. д. Цель теста масштабируемости отличается от стрессового или нагрузочного тестирования. Например, компания ожидает шестикратного увеличения нагрузки на серверы в течение следующих двух месяцев. Им может потребоваться увеличить производительность сервера и сократить время обработки запроса, чтобы лучше обслуживать посетителей. Если приложение масштабируемое, вы можете сократить это время, обновив оборудование сервера, например, вы можете увеличить частоту ЦП и добавить больше ОЗУ. Также вы можете улучшить производительность запросов, изменив программное обеспечение сервера, например, заменив хранилища данных в текстовых файлах базами данных SQL Server. Чтобы найти лучшее решение, вы можете сначала протестировать изменения оборудования, затем изменения программного обеспечения, а затем сравнить результаты тестов.



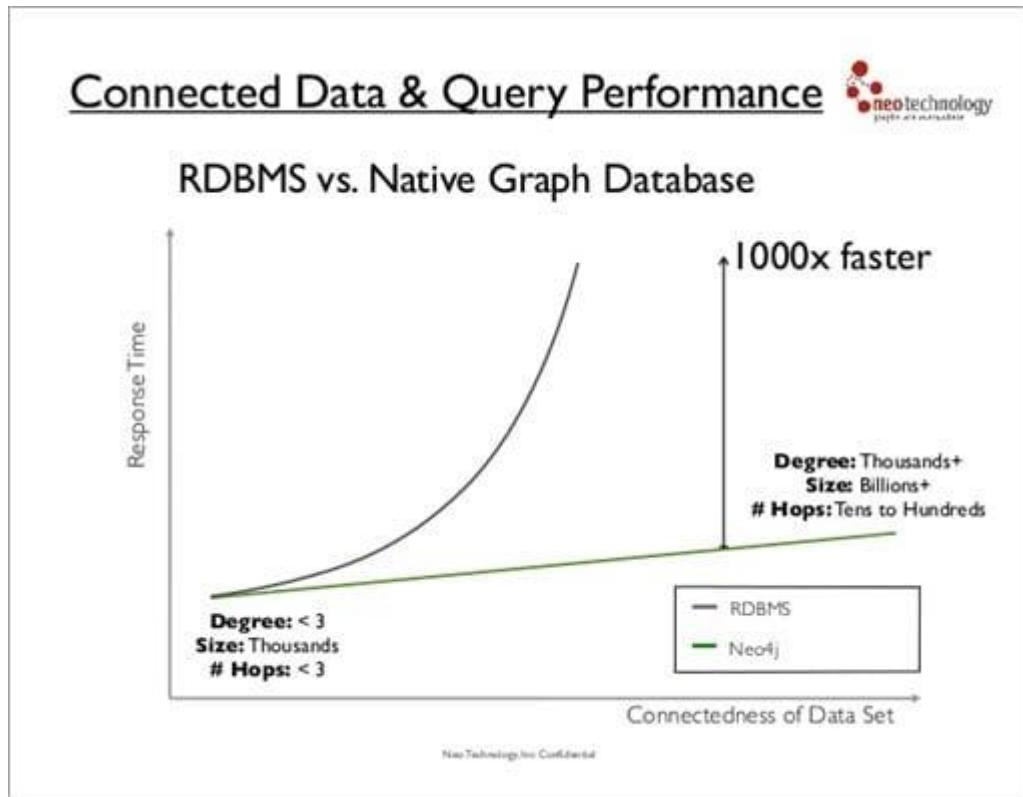
Пример: если тестирование масштабируемости определяет максимальную нагрузку в 10 000 пользователей, то для обеспечения масштабируемости системы разработчикам необходимо принять меры по таким факторам, как уменьшение времени отклика после достижения лимита в 10 000 пользователей или увеличение размера ОЗУ для соответствия растущему количеству пользовательских данных.

Источники:

- [What Is Scalability Testing? How To Test The Scalability Of An Application](#)
- [Do you really know all types of Performance Tests \(Non-Functional Tests\)?](#)

Объемное тестирование (Volume testing)

Объемное тестирование (также flood testing) предназначено для прогнозирования того, может ли система / приложение обрабатывать большой объем данных в плане проверки объема данных, обрабатываемых базой данных. Это тестирование сосредоточено на наполнении БД продукта в реальных сценариях использования, отслеживании производительности приложения при различных объемах БД. Обычно продолжительность проверки объема составляет 1 час или время, необходимое для обработки записей; оно может варьироваться в зависимости от вашего SLA / требований.



Причины для проведения этого тестирования:

- Самая основная потребность - проанализировать производительность вашей системы при увеличивающемся объеме данных. Создание огромного объема данных поможет вам понять производительность вашей системы с точки зрения времени отклика, потери данных и т. д.;
- Выявление проблем, которые могут возникнуть с огромными данными, а также пороговой точки (threshold point);
- За пределами устойчивой или пороговой точки (то есть при сбое БД) система перестает отвечать на запросы или появляются таймауты;
- Реализация решений по перегрузке БД и даже их проверка;
- Выявление крайней точки вашей БД (которая не может быть исправлена), за которой система выйдет из строя, и, следовательно, необходимо принять меры предосторожности;
- В случае наличия более одного сервера БД, выявление проблем с коммуникациями между БД;

Примеры тест-кейсов:

- Добавление данных может быть выполнено успешно и отражено ли оно в приложении или на веб-сайте;
- Удаление данных может быть выполнено успешно и отражается ли оно в приложении или на веб-сайте;
- Обновление данных может быть выполнено успешно, и отражается ли оно в приложении или на веб-сайте;
- Отсутствуют потери данных и вся информация отображается в приложении или на веб-сайте должным образом;
- Время ожидания приложения или веб-страниц не истекло из-за большого объема данных;

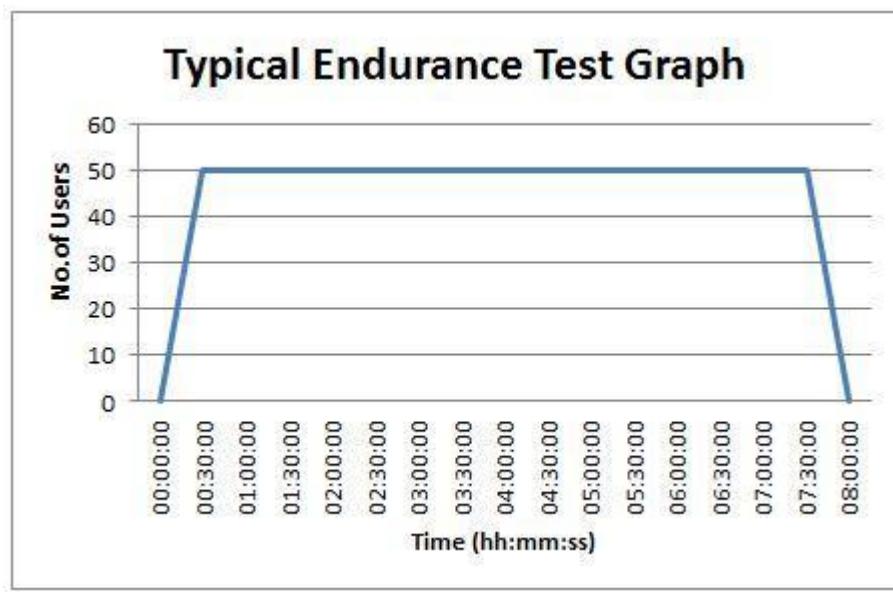
- При большом объеме данных нет сообщений о крашах;
- Данные не перезаписаны и отображаются соответствующие предупреждения;
- Другие модули вашего веб-сайта или приложения не завершают работу аварийно или не работают по таймауту из-за большого объема данных;
- Время отклика базы данных находится в допустимом диапазоне;

Источники:

- [Volume Testing Tutorial: Examples And Volume Testing Tools](#)
- [Do you really know all types of Performance Tests \(Non-Functional Tests\)?](#)

Тестирование выносливости/стабильности (Endurance/Soak/Stability testing)

Тестирование на выносливость ([Endurance Testing](#), оно же [Stability Testing](#), [Soak Testing](#), [Longevity Testing](#)) включает в себя тестирование системы со значительной нагрузкой в течение длительного периода времени, чтобы выяснить, как система ведет себя при длительном использовании. То есть для обеспечения того, чтобы производительность и / или время отклика после некоторого длительного периода устойчивой активности были не хуже, чем в начале теста. В основном используется для проверки утечек памяти, времени отклика, правильности подключения и закрытия подключения к модулям (например, БД) и т.п. Обычно продолжительность испытания на выносливость составляет 6-8 часов; может отличаться в зависимости от вашего SLA / требований.



Источники:

- [Do you really know all types of Performance Tests \(Non-Functional Tests\)?](#)

Тестирование устойчивости (Resilience testing)

Тестирование устойчивости направлено на обеспечение хорошей работы приложений в реальных или хаотичных условиях. Другими словами, оно проверяет отказоустойчивость приложения или способность противостоять стрессовым или сложным факторам, а также включает в себя тестирование на соответствие (compliance), выносливость (endurance), нагрузочное тестирование и тестирование восстановления (recovery testing). Эту форму тестирования иногда называют chaos engineering. Поскольку отказов невозможно избежать, тестирование устойчивости гарантирует, что программное обеспечение может продолжать выполнять основные функции и избежать потери данных даже в критических условиях, например, при сбое сети, отказе базы данных, при необходимости выдавая соответствующие сообщения об ошибках. При тестировании на устойчивость запланированным результатом является надежность (Reliability). Устойчивость также известна как восстанавливаемость (recoverability). Тестирование устойчивости можно рассматривать как часть плана обеспечения непрерывности бизнеса (BCP - business continuity plan) организации.

Шаги:

- Определение метрик: Разработчики должны выбрать, какие показатели следует измерять, чтобы отразить производительность программного обеспечения. Это может включать скорость ввода и вывода, пропускную способность, время восстановления, задержку и взаимосвязь между метриками;
- Определение базового уровня производительности;
- Ввести и измерить сбои: шаг, на котором вводятся проблемы, чтобы попытаться сломать систему. Взлом системы может быть осуществлен различными способами, например, нарушением связи с внешними зависимостями, введением злонамеренного ввода, манипулированием управлением трафиком, ограничением полосы пропускания, отключением систем сопряжения, удалением источников данных и потреблением системных ресурсов. После завершения этих сценариев следует измерить показатели и построить график в соответствии с тем, как каждый из них влияет на производительность;
- Делайте выводы и реагируйте на результаты: результаты следует использовать для начала обсуждения, исправления программного обеспечения и оценки практики команды разработчиков. Команды также должны использовать эти результаты для улучшения последующих сценариев тестирования;

Источник:

[software resilience testing](#)

Доп.материал:

- [Хаос на практике: зачем ломать production?](#)
- [Resilience testing: Which tool to choose?](#)

Тестирование надежности (Reliability Testing)

Надежность (Reliability) - это «вероятность безотказной работы программного обеспечения в течение определенного периода времени в определенной среде», т.е. это результат, к которому стремятся разработчики, способом достижения которого является устойчивость. Тестирование надежности связано с качеством программного обеспечения и стандартизацией продуктов. Если мы можем повторять тест-кейсы и постоянно получать один и тот же результат, то продукт считается «надежным». Тестирование надежности выполняется, чтобы убедиться, что программное обеспечение надежно, соответствует цели, для которой оно создано, и в течение определенного периода времени в данной среде способно обеспечить безотказную работу. Тестирование надежности может включать в себя Feature Testing, Security testing, Load Testing, Regression Testing и др.

Основные варианты для оценки надежности:

- Надежность повторного тестирования (Test-retest Reliability): при тестировании функционала одинаковыми тест-кейсами в разное время каждый раз мы получаем высокую корреляцию результатов. Тогда мы можем сказать, что тест «надежен». Обычно надежность 0,8 или более означает, что систему можно рассматривать как высоконадежный продукт;
- Параллельная или альтернативная форма надежности (Parallel or Alternate form of Reliability): разные версии одного теста должны давать одинаковый результат;
- Надежность между оценщиками (Inter-Rater Reliability): Надежность между оценщиками иначе известна как надежность между наблюдателями (Inter-Observer) или кодировщиками (Inter-Coder). Это особый тип надежности, состоящий из нескольких оценщиков или судей. Он касается согласованности рейтинга, выставляемого разными оценщиками / наблюдателями;

План тестирования надежности:

Имея правильную модель, мы можем предсказать качество продукта. К двум типам моделей относятся:

- Модель прогноза (Prediction Model): В прогнозном тестировании (Predictive testing) мы прогнозируем результат на основе исторических данных, статистики, машинного обучения. Все, что нам нужно, это написать отчет. В прогнозной модели мы получаем только некоторую историческую информацию. Используя эту информацию, мы можем экстраполировать имеющиеся данные на будущее;

- Модель оценки (Estimation Model): Этот тип модели выполняется перед самой стадией разработки или тестирования. В оценочном тестировании (Estimation Testing), помимо использования исторических данных, мы будем использовать текущие данные. Здесь мы можем спрогнозировать надежность продукта в настоящее время или в будущем. Этот тип тестирования выполняется на последних этапах жизненного цикла разработки программного обеспечения.

Источники:

- [What Is Reliability Testing: Definition, Method And Tools](#)
- [Reliability Series #1: Reliability vs. resilience](#)

Тестирование на отказ и восстановление (Failover and Recovery testing)

Тестирование на отказ и восстановление (Failover and Recovery testing, Disaster Recovery Testing) - подвид тестирования производительности, 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. Failover - проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта. Методика подобного тестирования заключается в симулировании различных условий сбоя, последующем изучении и оценке реакции защитных систем. В процессе подобных проверок выясняется, была ли достигнута требуемая степень восстановления системы после возникновения сбоя. В отличие от тестирования надежности (Reliability Testing), которое проводится, чтобы найти отказ в конкретной точке, где он происходит, Recovery Testing проводится для проверки того, насколько хорошо система восстанавливается после сбоя или аварии.

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

- Проблемы с сетью;
- Сбой питания;
- Внешний сервер недоступен (External server not reachable);
- Сервер не отвечает (Server not responding);
- Отсутствует файл dll;
- Перегрузка базы данных;
- Остановленные сервисы/службы;
- Физические условия;
- Внешнее устройство не отвечает;
- Потеря сигнала беспроводной сети;

Источник:

[What Is Recovery Testing In Software Testing](#)

Эталонное и базовое тестирование (Benchmark and Baseline Testing)

Эталонное тестирование (Benchmark Testing) - это набор стандартов, метрик или контрольных точек (reference point), по которым оценивается (assessed or evaluated) качество работы продукта или услуги, через нагрузочное тестирование модуля или всей комплексной программной системы для определения ее производительности. Оно определяет повторяемый набор экспериментальных результатов, которые помогают определить функциональные возможности как для текущих, так и для будущих выпусков программного обеспечения.

Базовое тестирование (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 Baseline Testing тесты прогоняют, сохраняют все результаты и сравнивают с 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 базовой. Определение из ISTQB/IEEE 610: базовая версия (baseline): Спецификация или программный продукт, который был формально отрецензирован или согласован, впоследствии используется как базовая версия для дальнейшей разработки, и который может быть изменен только в процессе формального контроля процесса изменений.

Самым первым этапом жизненного цикла разработки программного обеспечения является сбор и анализ требований. Этот тест играет важную роль, так как в случае, если начальная фаза не протестирована должным образом, в дальнейшем могут возникнуть серьезные проблемы. Это также может повлиять на стоимость, сроки, бюджет и репутацию клиента. После того, как требование собрано бизнес-аналитиком, он обсуждает то же самое с менеджером проекта для тестирования требования и его осуществимости. В случае неясности прототип требования создается разработчиком и представляется клиенту. Если это соответствует ожиданиям клиента и одобрено, командам предоставляется документ с требованиями для начала дальнейшего процесса. На основе этого документа с требованиями создаются другие документы для разработки и тестирования программного обеспечения, такие как план проекта, проектный документ, план тестирования, тестовые примеры и т. д. Поэтому очень важно правильно выполнить базовый тест. Если документ с требованиями не подтвержден должным образом, дальнейшие документы и процессы не пройдут. После завершения тестирования начинается процесс разработки и тестирования.

Разница между Baseline и Benchmark testing:

| Benchmark testing | Baseline Testing |
|---|--|
| Метрики уже созданы для оценки производительности приложения. Оно сравнивает характеристики продукта с отраслевыми стандартами; | Метрики создаются после завершения тестирования производительности. Набор тест-кейсов запускается для сбора информации о производительности; |
| Проводится с точки зрения бизнеса. SLA создаются на основе того же; | Выполняется на самом начальном этапе, с которого начинаются разработка, внедрение, тестирование и сравнение; |
| Можно пользоваться для всех продуктов / программного обеспечения в организации; | Проводится для конкретных продуктов / программного обеспечения; |
| Это состояние, в котором вы хотите достичь или превзойти то, чего вы уже достигли; | Выполняется на самом начальном этапе, с которого начинаются разработка, внедрение, тестирование и сравнение; |
| Предназначено для измерения производительности приложения вместе с другим приложением, имеющим аналогичные функции; | Определяет производительность приложения для сравнения в будущем; |

Пороговый тест (Threshold Test) - это тест, вставленный в Deployment Pipeline, который отслеживает некоторое измеримое явление, сравнивая значение в текущей сборке с пороговым значением. Если значение текущей сборки превышает пороговое значение, тест завершается неудачно, и сборка не выполняется. Типичный пример использования пороговых тестов - производительность. Команда берет репрезентативный набор операций и засекает их. Затем они устанавливают пороговый тест и если эти операции занимают значительное количество времени, превышающее текущее значение, тест завершается неудачей.

Источники:

- [What Is Benchmark Testing In Performance Testing](#)
- [What Is Baseline Testing And Its Benefits For Software Quality](#)
- [Threshold Test](#)

Тестирование хранилища (Storage testing)

Тестирование хранилища (Storage testing, Storage Performance testing) - это вид тестирования ПО, используемого для проверки того, как тестируемое ПО хранит данные в соответствующих каталогах и достаточно ли в них места для предотвращения неожиданного завершения работы из-за недостатка места на диске. ПО должно обрабатывать такие исключения и отображать предупреждающее сообщение для пользователя.

Зачем оно нужно?

- Медленное хранилище означает медленное время отклика, длительные запросы и более низкую доступность (availability) приложений;
- Медленное хранилище - это накладные расходы на обслуживание серверной инфраструктуры;
- Помогает найти практические ограничения хранилища перед развертыванием;
- Это помогает понять, как система отреагирует на замену или обновление оборудования;

Типы:

- Application testing: Тестирование приложений с примерами запросов в среде, похожей на боевую:
 - Сравните время ответа [OLTP](#);
 - Сравните время выполнения batch;
 - Сравните стабильные скорости потоковой передачи;
- Application Simulation: тестирование с использованием стандартного программного обеспечения, аналогичного целевому приложению:
 - Протестировать на пиковые значения IOPS для баз данных
 - Протестируйте на пиковые значения для среды потоковой передачи данных;
 - Проверка задержек хранилища для обмена сообщениями или других однопоточных приложений;
- Benchmarking: Провести тестирование с использованием стандартного программного обеспечения;
 - Проверка на повреждение данных;

Источник:

[Storage Testing Tutorial: What is, Type, Concepts](#)

Доп. материал:

- [Производительность распределенного хранилища: препродакшн тесты](#)
- [Тестирование хранилищ данных \(Data Warehouse\)](#)

Одновременное / многопользовательское тестирование (Concurrency/Multi-user testing)

Concurrency testing - это подвид нагрузочного тестирования, при котором проверяется поведение системы (веб-приложение, веб-страница или API) в момент, когда одновременно происходят два или более событий, или выполняется одновременный вход нескольких пользователей в систему с выполнением одного и того же действия. Во время теста наблюдаются и записываются определенные метрики, а также измеряется время отклика системы в периоды устойчивой большой нагрузки. 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;

Concurrency Testing Techniques: Reviewing code, Static Analysis, Con testing, Reachability Testing, Fuzz Testing, Random testing, Extending [concolic testing](#).

Источники:

- [Concurrency Testing: Challenges, Techniques & Process](#)
- [LoadView – Concurrent User Testing](#)

Тестирование сервиса (Service Testing)

Качество обслуживания, предоставляемого веб-приложением, может быть определено включением всех его атрибутов, таких как функциональность, производительность, надежность, удобство использования, безопасность и так далее. Однако для наших целей здесь мы выделяем три конкретные задачи обслуживания (*?service objectives*), которые подвергаются тщательной проверке в рамках того, что мы называем «?тестированием услуг»:

- Performance;
- Reliability;
- Manageability;

Во всех трех случаях нам необходимо моделировать пользовательскую нагрузку для эффективного проведения тестов. Цели производительности, надежности и управляемости существуют в контексте реальных клиентов, использующих сайт для бизнеса. Скорость отклика (в данном случае время, необходимое одному системному узлу для ответа на запрос другого) сайта напрямую зависит от ресурсов, доступных в технической архитектуре. Чем больше клиентов используют эту услугу, тем меньше технических ресурсов доступно для обслуживания запросов каждого пользователя, и время отклика сокращается. Очевидно, что слабо загруженная служба с меньшей вероятностью выйдет из строя. Большая часть сложности программного и аппаратного обеспечения существует для удовлетворения требований к ресурсам в рамках технической архитектуры, когда сайт сильно загружен. Когда сайт загружен (или перегружен), конфликтующие запросы ресурсов должны управляться различными компонентами инфраструктуры, такими как серверные и сетевые операционные системы, системы управления базами данных, продукты веб-серверов, брокеры объектных запросов, промежуточное ПО и т. д. Эти компоненты инфраструктуры обычно более надежны, чем код специально созданного приложения, которому требуются ресурсы, но сбои могут произойти в одном из следующих случаев:

- Компоненты инфраструктуры выходят из строя, потому что код приложения (из-за плохой разработки или реализации) предъявляет чрезмерные требования к ресурсам.
- Компоненты приложения могут выйти из строя, потому что требуемые им ресурсы не всегда могут быть доступны (вовремя).

Моделируя типичные и необычные производственные нагрузки в течение длительного периода, тестировщики могут выявить недостатки в конструкции или реализации системы. Когда эти недостатки будут устранены, те же тесты продемонстрируют устойчивость системы. Для всех служб обычно существует ряд критических процессов управления, которые необходимо выполнить для обеспечения бесперебойной работы службы. Можно было бы закрыть службу для выполнения планового обслуживания в нерабочее время, но большинство онлайн-служб работают 24 часа в сутки. Рабочий день сервиса никогда не заканчивается. Неизбежно, что процедуры управления должны выполняться, пока служба работает и пользователи находятся в системе. Эти процедуры необходимо тестировать при наличии нагрузки на систему, чтобы убедиться, что они не оказывают отрицательного воздействия на живую службу, известную как тестирование производительности.

Тестирование управления услугами (Service Management Testing). Когда служба развертывается в производственной среде, ею необходимо управлять. Для поддержания работоспособности службы необходимо, чтобы ее отслеживали, обновляли, создавали резервные копии и быстро исправляли, когда что-то пойдет не так. Процедуры, которые менеджеры служб используют для выполнения обновлений, резервного копирования, выпусков и восстановлений после сбоев, критически важны для обеспечения надежной службы, поэтому им необходимо тестирование, особенно если служба претерпит быстрые изменения после развертывания. Необходимо решить следующие конкретные проблемы:

- Процедуры не дают желаемого эффекта;
- Процедуры неработоспособны или непригодны;
- Процедуры нарушают прямую трансляцию;

По возможности тесты должны проводиться реалистично.

Источник:

Leadership in test: service testing

Тестирование 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 - скрытие 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;
- перегрузка 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 вручную, так и автоматизированно;
- Тестирование на проникновение (Penetration testing) имитирует атаку злоумышленника. Это тестирование включает анализ конкретной системы для проверки потенциальных уязвимостей при попытке внешнего взлома;
- Оценка рисков (Risk Assessment) 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 используют свои навыки законным способом для помощи в выявлении уязвимостей системы, в отличии от Black Hat (преступников);
- 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 |

Тестирование безопасности может выполняться методом любого ящика, а также Tiger Box: этот взлом обычно выполняется на ноутбуке с набором операционных систем и инструментов для взлома. Это тестирование помогает тестерам на проникновение и тестерам безопасности проводить оценку уязвимостей и атаки.

Методы тестирования безопасности:

Доступ к приложению. Будь то настольное приложение или веб-сайт, безопасность доступа обеспечивается функцией «Управление ролями и правами». Часто это делается неявно при описании функциональности, Например, в системе управления больницей администратор меньше всего беспокоится о лабораторных исследованиях, поскольку его работа состоит в том, чтобы просто зарегистрировать пациентов и назначить их встречи с врачами. Таким образом, все меню, формы и экраны, относящиеся к лабораторным тестам, не будут доступны для роли «регистратора». Следовательно, правильная реализация ролей и прав гарантирует безопасность доступа.

Как протестировать доступ к приложению: Чтобы это проверить, необходимо выполнить тщательное тестирование всех ролей и прав. Тестировщик должен создать несколько учетных записей пользователей с разными, а также с несколькими ролями. Затем он должен использовать приложение с помощью этих учетных записей и убедиться, что каждая роль имеет доступ только к своим собственным модулям, экранам, формам и меню. Если тестировщик обнаруживает конфликт, он должен с полной уверенностью зарегистрировать проблему безопасности. Некоторые из тестов аутентификации включают в себя проверку правил качества пароля, проверку входа в систему по умолчанию, проверку восстановления пароля, проверку проверки подлинности, проверку функциональности выхода, проверку смены пароля, проверку контрольного вопроса / ответа и т. д. Аналогичным образом, некоторые из тестов авторизации включают в себя тест на обход пути, тест на отсутствие авторизации, тест на проблемы горизонтального контроля доступа и т. д.

Защита данных. Есть три аспекта безопасности данных. Во-первых, пользователь может просматривать или использовать только те данные, которые он должен использовать. Это также обеспечивается ролями и правами. Например, TSR (представитель по телефону) компании может просматривать данные об имеющихся запасах, но не может видеть, сколько сырья было закуплено для производства. Итак, этот аспект тестирования безопасности уже объяснен выше. Второй аспект защиты данных связан с тем, [как эти данные хранятся в БД](#). Все конфиденциальные данные должны быть зашифрованы, чтобы сделать их безопасными. Шифрование должно быть надежным, особенно для конфиденциальных данных, таких как пароли учетных записей пользователей, номера кредитных карт или другой критически важной для бизнеса информации. Третий и последний аспект является продолжением этого второго аспекта. При передаче конфиденциальных или важных для бизнеса данных необходимо принять надлежащие меры безопасности. Независимо от того, перемещаются ли эти данные между разными модулями одного и того же приложения или передаются в разные приложения, они должны быть зашифрованы для обеспечения безопасности.

Как протестировать защиту данных: Тестировщик должен запросить в базе данных «пароли» учетной записи пользователя, информацию о выставлении счетов клиентов, другие важные для бизнеса и конфиденциальные данные и убедиться, что все такие данные хранятся в зашифрованной форме. Точно так

же он должен убедиться, что данные передаются между различными формами или экранами только после надлежащего шифрования. Более того, тестировщик должен убедиться, что зашифрованные данные должным образом расшифрованы в месте назначения. Особое внимание следует уделить различным действиям «отправить». Тестировщик должен убедиться, что информация, передаваемая между клиентом и сервером, не отображается в адресной строке веб-браузера в понятном формате. Если какая-либо из этих проверок завершится неудачно, значит, в приложении определенно есть баги безопасности. Тестировщик также должен проверить правильность использования соления (salting - добавление дополнительного секретного значения к конечному вводу, например пароля, что делает его более надежным и трудным для взлома). Небезопасная случайность также должна быть проверена, поскольку это своего рода уязвимость. Другой способ проверить защиту данных - проверить использование слабого алгоритма. Например, поскольку HTTP - это протокол открытого текста, если конфиденциальные данные, такие как учетные данные пользователя, передаются через HTTP, то это угроза безопасности приложения. Вместо HTTP конфиденциальные данные следует передавать через HTTPS (защищенный через SSL, туннель TLS). Однако HTTPS увеличивает поверхность атаки, поэтому необходимо проверить правильность конфигурации сервера и гарантировать действительность сертификата;

Атака грубой силы (Brute-Force Attack, атака полным перебором) в основном выполняется некоторыми программными инструментами. Идея состоит в том, что, используя действительный идентификатор пользователя, программное обеспечение пытается подобрать связанный пароль, пытаясь войти в систему снова и снова. Простым примером защиты от такой атаки является приостановка учетной записи на короткий период времени, как это делают все почтовые приложения, такие как Yahoo, Gmail и Hotmail. Если определенное количество последовательных попыток (в основном 3) не позволяют войти в систему, эта учетная запись блокируется на некоторое время (от 30 минут до 24 часов).

Как протестировать атаку грубой силы: Тестировщик должен убедиться, что какой-то механизм блокировки учетной записи доступен и работает правильно. Он должен попытаться войти в систему с недопустимыми идентификаторами пользователя и паролями, в качестве альтернативы, чтобы убедиться, что программное обеспечение блокирует учетные записи, если предпринимаются постоянные попытки входа с недопустимыми учетными данными. Если приложение делает это, оно защищено от атаки методом перебора. В противном случае тестировщик должен сообщить об этой уязвимости системы безопасности. Тестирование перебором также можно разделить на две части - тестирование черного ящика и тестирование серого ящика. При тестировании «черного ящика» метод аутентификации, используемый приложением, обнаруживается и тестируется. Тестирование серого ящика основано на частичном знании пароля и данных учетной записи, а также на атаках компромисса памяти ([подробнее](#)).

Вышеупомянутые три аспекта безопасности следует учитывать как для веб-приложений, так и для настольных приложений, в то время как **следующие моменты относятся только к веб-приложениям**.

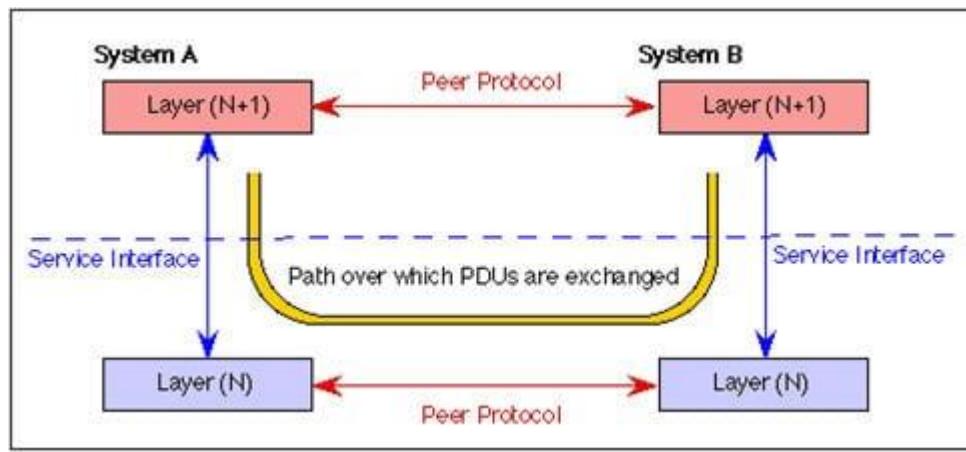
SQL-инъекции и **XSS** (межсайтовый скрипting). С концептуальной точки зрения, темы обеих попыток взлома схожи, поэтому они обсуждаются вместе. В этом подходе вредоносный сценарий используется хакерами для манипулирования веб-сайтом. Есть несколько способов застраховаться от таких попыток. Для всех полей ввода веб-сайта длины полей должны быть достаточно малыми, чтобы ограничить ввод любого скрипта. Например, поле «Фамилия» должно иметь длину поля 30 вместо 255. Могут быть некоторые поля ввода, в которых требуется ввод больших данных, для таких полей необходимо выполнить правильную проверку ввода до сохранения этих данных в приложении. Более того, в таких полях должны быть запрещены любые HTML-теги или ввод тегов скрипта. Чтобы спровоцировать XSS-атаки, приложение должно отклонять перенаправления скриптов от неизвестных или ненадежных приложений.

Как протестировать SQL-инъекцию и XSS: Тестер должен убедиться, что максимальная длина всех полей ввода определена и реализована. Он также должен гарантировать, что заданная длина полей ввода не соответствует вводу сценария, а также вводу тега. Оба они могут быть легко протестированы. Например, если 20 - максимальная длина, указанная для поля «Имя», а входная строка «`<p>thequickbrownfoxjumpsoverthelazydog`» можно проверить оба этих ограничения. Тестировщик также должен убедиться, что приложение не поддерживает методы анонимного доступа. Если какая-либо из этих

уязвимостей существует, приложение находится в опасности. В основном, тестирование SQL-инъекции может быть выполнено следующими пятью способами:

- Методы обнаружения;
- Стандартные техники SQL-инъекций;
- [Отпечаток базы данных](#);
- Эксплойты;
- [Методы внедрения в сигнатуру SQL-инъекции](#) (SQL Injection Signature Invasion Techniques);

Точки доступа к сервису (закрытые и безопасные открытые).



Сегодня компании зависят друг от друга и сотрудничают друг с другом, то же самое относится и к приложениям, особенно к веб-сайтам. В таком случае оба участника должны определить и опубликовать несколько точек доступа друг для друга. Пока сценарий кажется довольно простым и понятным, но для некоторых веб-продуктов, таких как торговля акциями, все не так просто и легко. Когда существует большое количество целевой аудитории, точки доступа должны быть достаточно открытыми, чтобы облегчить работу всех пользователей, достаточно приспособленными для выполнения всех запросов пользователей и достаточно безопасными, чтобы справиться с любой опасностью.

Как протестировать точки доступа к сервису (Service Access Points): позвольте мне объяснить это на примере веб-приложения для торговли акциями; инвестор (желающий приобрести акции) должен иметь доступ к текущим и историческим данным о ценах на акции. Это требует, чтобы приложение было достаточно открытым. Под удобством и безопасностью я подразумеваю, что приложение должно облегчить инвесторам возможность свободно торговать (в соответствии с законом). Они могут покупать или продавать 24/7, и данные транзакций должны быть защищены от любых хакерских атак. Более того, большое количество пользователей будет взаимодействовать с приложением одновременно, поэтому приложение должно предоставлять достаточно точек доступа, чтобы удовлетворить всех пользователей. В некоторых случаях эти точки доступа могут быть закрыты для нежелательных приложений или людей. Это зависит от бизнес-домена приложения и его пользователей, например, настраиваемая веб-система управления офисом может распознавать своих пользователей на основе IP-адресов и отказывать в установлении соединения со всеми другими системами (приложениями), которые не попадают в диапазон допустимых IP-адресов для этого приложения. Тестировщик должен гарантировать, что весь межсетевой и внутрисетевой доступ к приложению осуществляется доверенными приложениями, машинами (IP) и пользователями. Чтобы убедиться, что открытая точка доступа достаточно безопасна, тестировщик должен попытаться получить к ней доступ с разных машин, имеющих как доверенные, так и ненадежные IP-адреса. Следует опробовать различные типы транзакций в реальном времени сразу, чтобы быть уверенным в производительности приложения. Таким образом, пропускная способность точек доступа приложения также будет четко отслеживаться. Тестировщик должен убедиться, что приложение обрабатывает все запросы связи от доверенных IP-адресов и приложений только тогда, когда все остальные запросы отклоняются. Точно так же, если в приложении есть открытая точка доступа, тестировщик должен убедиться, что она разрешает (при необходимости) загрузку данных пользователями безопасным способом. Под этим безопасным способом я

имею в виду ограничение размера файла, ограничение типа файла и сканирование загруженного файла на вирусы или другие угрозы безопасности.

Управление сессией. Веб-сеанс - это последовательность транзакций HTTP-запроса и ответа, связанных с одним и тем же пользователем. Тесты управления сеансом проверяют, как управление сеансом обрабатывается в веб-приложении. Вы можете проверить истечение срока действия сеанса после определенного времени простоя, завершение сеанса после максимального времени жизни, завершение сеанса после выхода из системы, проверить объем и продолжительность сеанса cookie, проверить, может ли один пользователь иметь несколько одновременных сеансов и т. д.

Обработка ошибок. Тестирование на обработку ошибок включает:

- проверку кодов ошибок, которые возвращаются с подробным сообщением. Эти сообщения не должны содержать критической информации, которая может быть использована для взлома;
- проверку трассировки стека: в основном это включает в себя передачу в приложение некоторых исключительных данных, так что возвращаемое сообщение об ошибке содержит трассировку стека, которые содержат интересную информацию для хакеров;

Конкретные опасные функции. В основном, две рискованные функции - это платежи и загрузка файлов. Эти функции следует очень хорошо протестировать. Для загрузки файлов вам необходимо в первую очередь проверить, ограничена ли загрузка нежелательных или вредоносных файлов. Для платежей вам необходимо в первую очередь протестировать на наличие уязвимостей инъекций, небезопасного криптографического хранилища, переполнения буфера, подбора пароля и т. д.

Источники:

- [What is Security Testing? Types with Example](#)
- [Security Testing \(A Complete Guide\)](#)

Доп. материал:

- [Список книг по наступательной информационной безопасности](#)
- [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](#)
- [сHack](#)
- [OWASP Top Ten](#)
- [SQL-инъекции' union select null,null,null --](#)
- [Чек-лист устранения SQL-инъекций](#)
- [Что такое XSS-уязвимость и как тестировщику не пропустить ее](#)
- [What Is Database Security Testing – Complete Guide](#)
- [QA-митап Redmadrobot 19/11, QA vs Hackers: безопасность в web, Вика Бегенчева](#)
- [Как я получил награду Facebook по баунти-программе дважды](#)
- [Безопасность REST API от А до ПИ](#)
- [Тестирование безопасности API - Катерина Овченко. QA Fest 2019](#)
- [Как провести тестирование на безопасность: руководство для Manual QA](#)
- [Типы атак и уязвимостей](#)
- [Open Web Application Security Project \(OWASP\) TOP 10 2017](#)
- [Что такое OWASP Top-10 и как использовать указанные риски и уязвимости](#)
- [SQL Injection Prevention Cheat Sheet](#)
- [Web Authentication, Session Management, and Access Control Cheat Sheet](#)
- [Forgot Password Cheat Sheet](#)
- [Authentication Cheat Sheet](#)

- [Password Storage Cheat Sheet](#)
- [Cross Site Scripting \(XSS\) Prevention Cheat Sheet](#)
- [Unvalidated Redirects and Forwards Cheat Sheet](#)
- [Безопасность iOS-приложений: гайд для новичков](#)
- [Уязвимости Android 2020](#)
- [Обман обманщиков: форк-бомба нового уровня](#)
- [Software security](#)

Оценка уязвимости/защищенности (Vulnerability Assessment)

Уязвимость - это любые ошибки или недостатки в процедурах безопасности системы, разработке, реализации или любом внутреннем контроле, которые могут привести к нарушению политики безопасности системы.

Оценка уязвимости - это процесс оценки рисков безопасности в программной системе с целью уменьшения вероятности угрозы. 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).

Классификация уязвимостей:

- Уязвимость оборудования - это недостатки, возникающие из-за проблем с оборудованием, таких как чрезмерная влажность, пыль и незащищенное хранение оборудования;
- Уязвимость программного обеспечения. Недостаток в методике разработки проекта, несоответствующее тестирование и отсутствие своевременного аудита активов приводят к уязвимости программного обеспечения;
- Уязвимость сети: из-за использования открытых сетевых подключений, незащищенной сетевой архитектуры и слабого канала связи возникают проблемы этого типа;
- Физическая уязвимость: если система расположена в зоне, подверженной сильному дождю, наводнению, нестабильному электроснабжению и т. д., тогда она подвержена физической уязвимости;
- Уязвимость организации: эта уязвимость возникает из-за использования несоответствующих инструментов безопасности, правил аудита и ошибок в административных действиях;

Наиболее распространенные уязвимости сетевой безопасности:

- **USB-накопители.** Использование USB-накопителей - самый обычный способ воздействия на любую сетьевую систему и даже брандмауэр не сможет остановить вирусную атаку, поскольку они используются между многими компьютерами для обмена большим объемом информации и могут переносить большие объемы данных внутри себя. USB-накопители, зараженные вирусами, такими как червь, автоматически устанавливаются в ОС и подключаются к USB-порту, поскольку большая часть ОС по умолчанию позволяет запускать эти программы.

Способ устранения: мы можем остановить автоматическую установку их в ОС, изменив настройки по умолчанию в операционных системах, и можем сделать их более безопасными от вирусных атак с USB-накопителями.

- **Ноутбуки.** Такие устройства, как лэптопы и ноутбуки очень удобны и портативны, оснащены всеми новейшими драйверами, ОС и имеют порт Ethernet, через который их можно легко подключить к любой сетевой системе. Ноутбуки очень небезопасны с точки зрения организации, ноутбук сотрудника содержит конфиденциальные данные, такие как зарплата сотрудника, адрес, контактная информация, личные данные, важная база данных компании, личные банковские пароли и т. д. Любая организация не может допустить утечки всей этой информации, так как это повлияет на бизнес, и организация может пострадать от бизнес-потерь.

Способ устранения: все конфиденциальные и важные данные должны храниться в зашифрованном виде, чтобы трети лица не могли легко получить к ним доступ. Права доступа к базе данных должны

быть ограничены. В дополнение к этому, должен быть включен только порт LAN, а все остальные порты должны быть отключены администратором.

- **Разные USB-устройства.** Помимо флэш-накопителей USB, в сети присутствуют некоторые другие устройства, которые могут считывать и хранить данные в них и могут подвергнуть вашу систему уязвимости. Зараженные этим вирусом устройства, такие как цифровая камера, принтер, сканер, MP3-плеер и т. д., вступают в контакт с вашей системой через порт USB и могут нанести вред вашей сетевой системе.

Способ устранения: установите такие политики, которые могут контролировать автоматический запуск программ порта USB в вашей системе.

- **Оптические носители.** Оптические носители являются носителями важных пакетов данных, которыми обмениваются в сетевой системе WAN для связи на большие расстояния. Следовательно, данные по этим ссылкам также могут быть пропущены или использованы не по назначению третьей стороной в интересах кого-то другого, как в случае с USB-устройствами.

Способ устранения: руководство должно ввести такие политики и правила контроля активов, которые могут отслеживать и контролировать неправомерное использование данных.

- **Электронная почта.** Электронная почта является наиболее распространенным источником связи внутри организации или между различными организациями в деловых целях. Любая компания использует электронную почту для отправки и получения данных. Но электронной почтой чаще всего злоупотребляют, так как ее легко переслать кому угодно. Кроме того, иногда электронные письма содержат вирусы, которые могут узнать учетные данные целевого хоста, а затем хакер может легко получить доступ к электронной почте этого сотрудника организации из любого места. Они также могут использовать его для другого несанкционированного доступа.

Способ устранения: использование политик безопасности электронной почты и частая смена паролей системы через определенный промежуток времени - лучшее решение для этого.

- **Смартфоны и другие цифровые устройства.** Смартфоны и другие планшетные устройства могут работать как компьютер в дополнение к выполнению различных задач, таких как интеллектуальные вызовы, видеозвонки, большой объем памяти, камера с высоким разрешением и огромная система поддержки приложений. Риск утечки конфиденциальных данных также высок, поскольку сотрудник организации, использующий смартфон, может щелкнуть изображение секретного бизнес- предложения или расценок и может отправить их любому, кто пользуется мобильной сетью 4G.

Способ устранения: необходимо внедрить политики, которые могут контролировать доступ к устройству при входе в среду сетевой системы и выходе из нее.

- **Слабая безопасность учетных данных:** использование слабых паролей в сетевой системе легко подвергает сеть различным вирусным атакам.

Способ устранения: пароль, используемый для сетевой безопасности, должен быть надежным, как уникальная комбинация буквенно-цифровых символов и символов. Кроме того, нельзя использовать один и тот же пароль в течение длительного времени, необходимо регулярно менять системный пароль для получения лучших результатов.

- **Плохая конфигурация и использование устаревшего брандмауэра:** брандмаэр играет очень важную роль в процессе управления сетевой безопасностью. Если администратор неправильно настроит брандмаэр на различных уровнях сети, она станет уязвимой для атак. Помимо этого, программный патч брандмауэра должен постоянно обновляться для правильного функционирования брандмауэра. Использование устаревшего оборудования и программного обеспечения брандмауэра бесполезно.

Способ устранения: Регулярное обновление программного обеспечения межсетевого экрана и правильная реализация.

Этапы оценки уязвимости:

- **Сбор данных:** первым шагом оценки является сбор всех необходимых данных, касающихся ресурсов, используемых в системе, таких как IP-адреса системы, используемые носители, используемое оборудование, тип антивируса, используемого системой, и т. д.;
- **Выявление возможных сетевых угроз:** теперь, имея входные данные, мы можем определить возможные причины и лазейки сетевых угроз в сети, которые могут нанести вред нашей системе. Здесь нам также необходимо определить риски и приоритетность угроз (The impact of Risk, The threshold of Risk, Risk strategy, business impact);
- **Анализ паролей маршрутизаторов и Wi-Fi:** необходимо проверить, что пароли, используемые для входа в маршрутизатор, и пароль, используемый для доступа в Интернет, достаточно надежны, чтобы их было сложно взломать. Кроме того, здесь важно убедиться, что пароль меняется через регулярные промежутки времени, чтобы система стала более устойчивой к атакам;
- **Анализ стойкости (strength) сети организации:** Следующим шагом является оценка стойкости сети системы по отношению к обычным атакам, включая распределенную атаку типа «отказ в обслуживании» (DDoS), атаку «человек посередине» (MITM) и сетевое вторжение. Это, в свою очередь, даст нам четкое представление о том, как наша система будет реагировать в случае этих атак, и способна ли она спастись или нет;
- **Оценка безопасности сетевых устройств:** теперь проанализируйте реакцию сетевых устройств, таких как коммутатор, маршрутизатор, модем и ПК, на сетевые атаки. Здесь будет подробно рассказываться о реакции устройств со ссылкой на угрозы;
- **Сканирование на выявленные уязвимости:** последний шаг оценки - сканирование системы на предмет известных угроз и уязвимостей, которые уже присутствуют в сети. Это делается с помощью различных инструментов сканирования;
- **Создание отчета:** Очень важна документация по процессу оценки уязвимости сети. Она должна содержать все действия, выполненные от начала до конца, и угрозы, обнаруженные во время тестирования, а также процесс их устранения;
- **Повторное тестирование:** следует постоянно проверять и анализировать систему на предмет новых возможных угроз и атак и принимать все возможные меры для их смягчения;

Процесс оценки уязвимости выступает в качестве входных данных для политики сетевой безопасности (network security policy).

Классификации сканеров уязвимостей:

По типу активов/ресурсов (assets):

- **Сетевые сканеры** (Network-based scanners). Вы можете использовать сетевые сканеры для обнаружения неавторизованных устройств или неизвестных пользователей в сети. Эти сканеры позволяют сетевым администраторам определять, существуют ли в сети скрытые лазейки по периметру, такие как несанкционированный удаленный доступ. Сетевые сканеры не имеют прямого доступа к файловой системе. Таким образом, они не могут проводить проверки безопасности низкого уровня;
- **Хост-сканеры** (Host-based scanners). Как следует из названия, сканер на основе хоста находится на каждом хосте в отслеживаемой сети. Он обнаруживает и идентифицирует уязвимости на рабочих станциях, серверах или других сетевых узлах, обеспечивая большую видимость настроек конфигурации ваших активов;
- **Сканеры приложений** (Application scanners). Сканеры приложений находят уязвимости на веб-сайтах. Их режим работы аналогичен режиму работы поисковых систем - они «ползут» по веб-сайтам, отправляя ряд зондов на каждую веб-страницу на веб-сайте, чтобы найти слабые места;
- **Сканеры беспроводной сети** (Wireless network scanners). Сканеры беспроводных сетей, также называемые анализаторами беспроводных протоколов (wireless protocol analyzers), представляют собой инструменты, которые вы можете использовать для обнаружения открытых беспроводных сетей в вашей среде. Организации, которые запрещают использование беспроводных сетей, могут использовать эти сканеры беспроводных сетей для обнаружения любых неавторизованных сетей Wi-Fi;

- **Сканеры баз данных** (Database scanners). Вы можете использовать сканеры баз данных для выявления уязвимостей в вашей базе данных. Сканеры баз данных могут помочь вам предотвратить вредоносные взломы, такие как атаки с использованием SQL-инъекций;

По источнику:

- **Сканеры внешних уязвимостей.** С помощью внешних сканеров вы выполняете сканирование уязвимостей вне сети компании. Внешние сканеры обычно нацелены на ИТ-инфраструктуру, доступную в Интернете, включая открытые порты в сетевом брандмауэре и брандмауэрах веб-приложений;
- **Сканеры внутренних уязвимостей.** В отличие от внешних сканеров, внутренние сканеры проводят сканирование уязвимостей изнутри корпоративной сети. Это сканирование позволяет защитить и укрепить критически важные приложения, обнаруживая внутренние угрозы, например вредоносные программы, проникшие в сеть;

По авторизованности:

- **С аутентификацией.** Сканирование с проверкой подлинности - также называемое сканированием с учетными данными - позволяет администратору сети войти в систему как пользователь и определить слабые места сети с точки зрения доверенного пользователя. Поскольку вы вошли в систему, вы можете глубже проникнуть в сеть, чтобы обнаружить многочисленные угрозы;
- **Без аутентификации.** При сканировании без аутентификации вам не нужно входить в сеть для выполнения сканирования. Хотя вы можете получить представление о сети посторонним, вы, скорее всего, упустите большинство уязвимостей при использовании сканирования без аутентификации;

Тестирование на проникновение (Penetration testing):

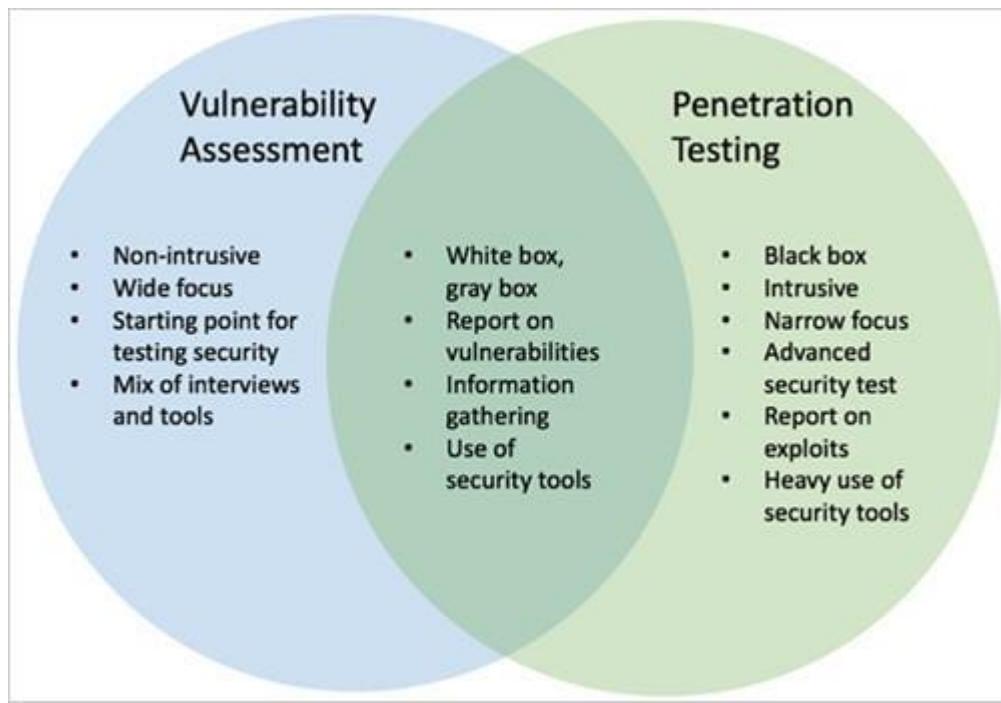
Это имитация атаки, в ходе которой компьютерная система, сеть или приложение проверяются на наличие слабых мест в системе безопасности. Эти тесты основаны на сочетании инструментов и методов, которые настоящие хакеры использовали бы для взлома. Это похоже на то, как банк нанимает кого-то, чтобы в роли грабителя попытаться проникнуть в их здание и получить доступ к хранилищу. Если «грабитель» добьется успеха и проникнет в банк или хранилище, банк получит ценную информацию о том, как им нужно усилить меры безопасности. Другие распространенные названия тестов на проникновение - это white hat attacks and ethical hacking. 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:
 - выявление 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 сессиями;
- Тест на физическое проникновение:
 - взлом физических барьеров;
 - проверка и взлом замков;
 - нарушения работы и обход датчиков;
 - вывод из строя камер видеонаблюдения;

Отличия Vulnerability Assessment от Penetration testing

VAPT = Vulnerability Assessment + Penetration testing



| Vulnerability Assessment | Penetration Testing |
|---|---|
| Это метод поиска и измерения уязвимости системы | Находит уязвимости и использует их |
| Конечным результатом является список уязвимостей, которые приоретизируются по их влиянию | Тест на проникновение более целенаправлен. Он помогает наметить путь, по которому злоумышленник проникнет в систему |
| В основном автоматизированно | В основном вручную |
| Быстрее | Дольше |
| Дешевле | Дороже |
| Направлено вширь | Направлено вглубь каждой уязвимости для достижения конкретных целей |
| Выполняется в критически важных системах в реальном времени | Выполняется на некритичных системах |
| Оценка уязвимости должна проводиться не реже одного раза в квартал, в основном после обновления оборудования или серьезных изменений в сети | Тестирование на проникновение следует проводить ежегодно после значительных изменений |
| Подробно описывается, какие уязвимости существуют и что изменилось с момента последнего анализа | Эффективно идентифицирует информацию, которая была скомпрометирована |

Источники:

- [Network Vulnerability Assessment And Management Guide](#)
- [What Is Vulnerability Assessment, and Why Is It Important?](#)

- [Penetration Testing vs Vulnerability Assessment](#)

Доп. материал:

- [Top 10 Most Powerful Vulnerability Assessment Scanning Tools In 2021](#)
- [A Complete Penetration Testing Guide With Sample Test Cases](#)
- [Взрослый разговор о пентесте и хакинге](#)
- [50 000 \\$ в месяц — не проблема, или Сколько на самом деле зарабатывают пентестеры](#)
- [TWAPT — пентестим по-белому в домашних условиях](#)

Фаззинг-тестирование (Fuzz testing)

FUZZ testing (fuzzing) - это тип тестирования безопасности, который обнаруживает ошибки кодирования и лазейки в программном обеспечении, операционных системах или сетях. Фаззинг включает в себя ввод огромного количества случайных данных, называемых fuzz, в тестируемое программное обеспечение, чтобы заставить его дать сбой или прорвать его защиту. Фаззинг часто выявляет уязвимости, которые могут быть использованы с помощью SQL-инъекции, переполнения буфера, отказа в обслуживании (DOS) и XSS. Fuzz-тестирование выполняется с помощью фаззера - программы, которая автоматически вводит полуслучайные данные в программу и обнаруживает ошибки. Fuzz-тестирование обычно выполняется автоматически.

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: Этот тип фаззера проще всего создать, поскольку он изменяет существующие образцы данных для создания новых тестовых данных. Это относится к тупому фаззеру, но его можно использовать с более интеллектуальными фаззерами. Мы можем сделать это, выполнив некоторый уровень анализа образцов, чтобы гарантировать, что он изменяет только определенные части или не нарушает общую структуру ввода;
- Generation-Based Fuzzers: Этот тип фаззера требует большего интеллекта для создания тестовых данных с нуля, т.е. новые тестовые данные создаются на основе входной модели. Обычно он разбивает протокол или формат файла на фрагменты, которые затем выстраиваются в допустимом порядке, и эти фрагменты случайным образом распределяются независимо друг от друга;
- 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.;

Источники:

- [Fuzz Testing Guide](#)

- [Fuzz Testing\(Fuzzing\) Tutorial: What is, Types, Tools & Example](#)

Доп. материал:

[Фаззинг тестирование веб-интерфейса. Расшифровка доклада](#)

Можно ли отнести тестирование безопасности или нагружочное тестирование к функциональным видам тестирования?

Данные виды принято относить к нефункциональным видам тестирования, однако если конкретно безопасность или производительность является основным функционалом приложения, а не его атрибутами, то можно отнести и к функциональным. Как пример я бы привел программу-шифровальщик флешки (можно обсудить в коммьюнити, накидают вариантов).

"<..> 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".

Источник: [azshoo](#)

Тестирование совместимости/взаимодействия (Compatibility/Interoperability testing)

Взаимодействие (**Interoperability**) - это способность одной системы взаимодействовать с другой системой. Это взаимодействие между двумя разными системами или двумя разными приложениями вместе. Часто взаимодействие путают с интеграцией, совместимостью и портируемостью.

Interoperability = Inter + operable

Inter - означает «между собой», «друг между другом», «взаимно».

Operable - означает «способный выполнить поставленную задачу».

Пример №1: Возьмем пример бронирования вашего рейса. Считайте, что вам нужно поехать из Нью-Дели в Нью-Йорк. Сейчас у вас нет прямого рейса. Вы должны лететь из Нью-Дели в Лондон, а затем лететь стыковочным рейсом из Лондона в Нью-Йорк. Поскольку у вас есть некоторые ограничения по времени, вы бронируете свой рейс из Нью-Дели в Лондон на авиалинии «Jet Airways» и из Лондона в Нью-Йорк на «Virgin Atlantic». Это означает, что все данные о ваших пассажирах были переданы от Jet Airways до Virgin Atlantic. Итак, здесь Jet Airways и Virgin Atlantic, оба являются независимыми приложениями вместе, и при бронировании вашего рейса ваши данные о бронировании передаются от Jet Airways в Virgin Atlantic в полном объеме, без предварительного уведомления.

Пример №2. Аналогичным образом представьте себе систему управления больницей, где записи пациентов обмениваются между одним отделением и другим отделением. Итак, здесь можно связать отдел с приложением. Информация о пациенте передается из одного приложения в другое без предварительного уведомления.

Уровни Interoperability testing:

- Физический (Physical Interoperability);
- Типы данных (Data-type Interoperability);
- Уровень спецификации (Specification level Interoperability);
- Семантический (Semantic Interoperability);

Как провести Interoperability testing?

Мы можем следовать колесу Деминга (Deming wheel или цикл PDCA), чтобы провести Interoperability testing:

Plan: планирование - это самый важный этап определения стратегии выполнения практически любых задач при разработке программного обеспечения. Прежде чем мы на самом деле спланируем определение процедуры выполнения IOT, необходимо понять каждое приложение или систему, развернутую в сети. Мы должны знать обо всех приложениях - их функциональность, поведение, вводимые данные и раскрываемые результаты вывода. Я также рекомендовал бы, чтобы каждое приложение было полностью функционально протестировано и было без дефектов, прежде чем готовить его к interoperability testing. Поэтому, когда вы планируете, не думайте только об одном или двух приложениях, думайте обо всех приложениях как о едином блоке. Планируя этот метод тестирования, вы должны смотреть с высоты птичьего полета. Излишне говорить - задокументируйте свой план. Мы можем использовать план тестирования и немного адаптировать его в соответствии с требованиями к документированию планирования IOT. После того, как ваш план тестирования составлен, переходите к определению условий тестирования (test conditions). Основное внимание при получении условий тестирования не должно ограничиваться отдельными приложениями; вместо этого он должен быть основан на потоке данных через все приложения. Условия должны быть спроектированы таким образом, чтобы проходились если не все, но большинство приложений в сети. После определения условий тестирования переходите к разработке или написанию сценария (в случае, если вы планируете автоматизировать) ваших тест-кейсов. Вы можете создать RTM (матрицу прослеживаемости требований), чтобы сопоставить ваши тест-кейсы с условиями тестирования и ваши условия тестирования с условиями / требованиями приемочного тестирования. Когда вы работаете в сети, также важно спланировать нефункциональное тестирование. Это может быть нигде не записано или не задокументировано, но обязательно для проверки нефункциональных аспектов системы в целом. Эти нефункциональные области будут включать производительность и безопасность. При необходимости вы можете составить отдельный план для функционального тестирования, тестирования производительности и тестирования безопасности; или создайте единый план и разные документы с условиями тестирования для каждого из этих типов тестирования;

Do: это промежуток времени, в течение которого вы прогоняете тест-кейсы. Соответственно планируйте свое время для выполнения функционального и нефункционального тестирования. Мы следуем циклу тестирования (testing cycle) на этом этапе выполнения кейсов, логируем дефекты, команда разработчиков их устраняет, после чего мы выполняем повторное тестирование и регрессионное тестирование системы в целом, и предоставляем отчет о результатах тестирования;

Check - это этап, на котором мы пересматриваем результаты наших тестов и пытаемся сопоставить их с RTM и проверить, выполнены ли все ожидаемые требования и все ли приложения пройдены. Мы проверяем, что данные передаются и обмениваются правильно и плавно между приложениями / системами. Нам также нужно будет убедиться, что данные, которые мы просматриваем, не изменяются. Также подумайте о том, чтобы сделать ретроспективу всего процесса interoperability testing. Определите области, которые хорошо сработали, те, которые не удались, и любые элементы действий, о которых необходимо позаботиться.

Act - действовать по ретроспективным элементам. Пункты, которые были определены как «good practices», продолжают выполняться, а для пунктов, над которыми можно было бы лучше поработать, определяются шаги по их исправлению. Имейте в виду одну вещь: области или шаги, которые не сработали, НЕ должны повторяться. В конце концов, мы должны учиться на своих ошибках, а не повторять их.

Совместимость (Compatibility, Coexistence) - это метод, с помощью которого проверяется совместимость 2 или более приложений в одной среде. MS Word и Калькулятор - это два разных приложения, и они показывают ожидаемое поведение независимо в одной и той же операционной системе. Итак, мы говорим, что эти 2 приложения совместимы друг с другом. Другой пример: если сайт Google.com совместим, он должен

открываться во всех браузерах и операционных системах. Тестирование совместимости - это нефункциональное тестирование для обеспечения удовлетворенности клиентов. Оно предназначено для определения того, может ли программное обеспечение или продукт работать в различных браузерах, базах данных, оборудовании, операционной системе, мобильных устройствах и сетях. На приложение также может влиять различные версии, разрешения, скорости интернета, конфигурации и т. д. Следовательно, важно тестировать приложение всеми возможными способами, чтобы уменьшить сбои и избежать затруднений, связанных с утечкой ошибок (bug's leakage). Тест на совместимость всегда должен выполняться в реальной среде, а не в виртуальной. Протестируйте совместимость приложения с различными браузерами и операционными системами, чтобы гарантировать 100% покрытие.

Типы тестирования совместимости:

- Тестирование совместимости браузера (Browser compatibility testing): очень популярно при тестировании совместимости. Это необходимо для проверки совместимости программного приложения с различными браузерами, такими как Chrome, Firefox, Internet Explorer, Safari, Opera и т. д.;
- Аппаратное обеспечение (Hardware): Это необходимо для проверки совместимости приложения / программного обеспечения с различными конфигурациями оборудования;
- Сети (Networks): Это для проверки приложения в разных сетях, таких как 3G, WIFI и т. д.;
- Мобильные устройства (Mobile Devices): Это необходимо для проверки совместимости приложения с мобильными устройствами и их платформами, такими как android, iOS, windows и т. д.;
- Операционная система (Operating System): Это необходимо для проверки совместимости приложения с различными операционными системами, такими как Windows, Linux, Mac и т. д.;
- Версии (Versions): Важно тестировать программные приложения в разных версиях программного обеспечения. Существует два разных типа проверки версий:
 - Тестирование обратной совместимости (Backward Compatibility Testing) - тестирование приложения или программного обеспечения со старыми или предыдущими версиями. Это также известно как обратная совместимость (downward compatible);
 - Тестирование прямой совместимости (Forward Compatibility Testing) - тестирование приложения или программного обеспечения с новыми или будущими версиями. Это также известно как прямая совместимость (forward compatible);

Источники:

- [A Simple Guide To Interoperability Testing \(With Examples\)](#)
- [What Is Software Compatibility Testing?](#)

Конфигурационное тестирование (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.).

Configuration = performance + compatibility:

- performance аспект: определить оптимальную конфигурацию оборудования, обеспечивающую требуемые характеристики производительности и времени реакции тестируемой системы;
- compatibility аспект: проверить объект тестирования на совместимость с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм;

Уровни конфигурационного тестирования для клиент-серверных приложений (для некоторых типов приложений может быть актуален только один):

- Серверный: Основной упор здесь делается на тестирование с целью определения оптимальной конфигурации оборудования, удовлетворяющего требуемым характеристикам качества (эффективность, портативность, удобство сопровождения, надежность). 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.);
- Клиентский: 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 видеoadаптера (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);

Prerequisites:

- создать матрицу покрытия (Coverage Matrix, BCM - Basic Configuration Matrix - это таблица, в которую заносят все возможные конфигурации);
- провести 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.

Примечание: в ISTQB вообще не говорится о таком виде тестирования как конфигурационное: "configuration testing: See portability testing."

Тестирование переносимости (Portability testing) - это вид тестирования, который определяет легкость или сложность, с которой программный компонент или приложение могут быть перемещены из одной среды в другую. Результаты тестирования, полученные в результате тестирования переносимости, помогают выяснить, насколько легко программный компонент из одной среды можно использовать в другой среде. Термин «среда» относится к переходу от одной операционной системы к другой, от одного браузера к другому или от одной версии базы данных к другой версии базы данных. Измерение переносимости - это усилия, необходимые для перемещения программного компонента из одной среды в другую. Одна единица измерения переносимости - это стоимость адаптации программного обеспечения к новой среде по сравнению со стоимостью повторной разработки программного обеспечения. Переносимость может включать в себя Installability, Adaptability, Replaceability, Compatibility or Coexistence, Interoperability, Localization.

Portability vs. Compatibility:

- Совместимость касается того, могут ли два или более компонента работать в одной и той же среде одновременно, не влияя отрицательно на поведение друг друга. Пример: можно сказать, что текстовый процессор и калькулятор, работающие в одной ОС, такой как Windows 10, совместимы друг с другом, поскольку запуск одного приложения не влияет на поведение другого приложения;
- Переносимость касается перемещения компонента из одной среды в другую. Пример: игра, работающая в Windows XP, считается переносимой, если та же игра может быть запущена в Windows 7 без каких-либо изменений в ее поведении;

- Проще говоря, тестирование переносимости касается программного компонента в разных средах, в то время как тестирование совместимости касается тестирования разных приложений в одной среде;

Источники:

- [Configuration Testing Tutorial With Examples](#)
- [Portability Testing Guide With Practical Examples](#)

Тестирование на соответствие (Conformance/Compliance testing)

Compliance - официальное соответствие ПО различным стандартам, законам, сертификация и т.п.

Conformance - неофициальные, внутренние стандарты организации, добровольное обязательство делать что-либо признанным образом, либо стремление к Compliance, но которое не закончено / не подтверждено формально.

Источник:

[Compliance Vs. Conformance](#)

Тестирование удобства пользования (Usability testing)

Тестирование удобства пользования - это нефункциональный вид тестирования программного обеспечения, являющийся подмножеством тестирования пользовательского опыта - UX, "Ю-Экс", user experience. В целом оно подразделяется на понятность, обучаемость, работоспособность, привлекательность и соответствие (understandability, learnability, operability, attractiveness, and compliance). Юзабилити-тестирование предназначено для определения того, насколько программный продукт понятен, легок в освоении, прост в эксплуатации и привлекателен для пользователей при определенных условиях и требованиях. Этот тип тестирования обычно выполняется реальными пользователями.

Категории юзабилити-тестирования:

- **Исследовательская:** обычно мы рассматриваем эту категорию на ранних этапах процесса тестирования программного обеспечения. Чем раньше выполняется тестирование юзабилити в процессе тестирования, тем меньше риски в продукте. На этом этапе обычно рассматривается дизайн продукта и концепции, относящиеся к продукту или услуге;
- **Оценочная:** эта категория описывает оценку выполнения E2E теста, а также анализирует эффективность продукта и удовлетворенность пользователей;
- **Сравнительная:** в этой категории два или более схожих продукта сравниваются по разным атрибутам, таким как дизайн продукта, преимущества и недостатки, что помогает выбрать продукт, который обеспечивает лучший пользовательский опыт;

Методы юзабилити-тестирования:

- **Промежуточное** (? hallway) тестирование. Этот метод является одним из наиболее эффективных и экономичных по сравнению с другими доступными методами. При использовании этого метода веб-сайт или продукт для тестирования получают несколько случайных людей, а не обученные специалисты. Поскольку случайные люди тестируют службу без предварительного знания продукта, они тестируют ее более эффективно и предоставляют более точные результаты и честную обратную связь для улучшения, если таковые имеются;
- **Удаленное** тестирование. Как следует из названия, удаленное тестирование юзабилити проводится людьми, которые находятся в удаленных местах. Обратная связь может быть записана и отправлена случайными людьми, а не экспертом по технологиям. Иногда удаленное тестирование выполняется с помощью видеоконференцсвязи. Этот тип юзабилити-тестирования снижает стоимость по сравнению с другими типами тестирования;
- **Экспертная оценка.** Эксперта в данной области просят протестировать продукт или услугу и предоставить отзыв, а затем представить результаты. Обычно это быстро, но и стоит дорого. Эксперт находит лазейки и обнаруживает недостатки в продукте или услуге;

- **Бумажный прототип.** Тестирование бумажных прототипов - один из самых традиционных подходов к тестированию юзабилити. Этот метод включает в себя пробный запуск теста, ручной набросок, рисование моделей или прототипа. Обсуждение последовательности операций и их рисование на бумаге, а также рассмотрение всех возможных исходных данных, сценариев и условий - вот цель этого типа тестирования. Это один из основных типов тестирования, который чаще всего применяется во всех проектах для устранения основных проблем. Выполняя тестирование бумажного прототипа, можно получить больше ясности в процессе выполнения. Тестирование бумажного прототипа обычно проводится в проектной группе. Следовательно, это рассматривается на ранних этапах процесса тестирования. Это относительно более дешевый метод тестирования юзабилити, но не самый эффективный способ тестирования, поскольку он временами занимает больше времени, и существует более высокая вероятность того, что даже после тестирования мы можем пропустить несколько проблем;
- **Автоматизированное.** Как следует из названия, этот метод тестирования выполняется путем написания сценариев автоматизации. После выполнения теста результаты записываются и отправляются. Для этого типа метода тестирования компании необходимо нанять ресурс, который хорошо знаком с написанием сценариев и построением среды автоматизации. Это один из наиболее часто используемых методов тестирования;

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.

Источники:

- [Usability Testing Tutorial: A Complete Getting Started Guide](#)
- [What is Usability Testing? UX\(User Experience\) Testing Example](#)
- [Usability Testing : How To Perform, Test Cases, Checklist, Methods](#)

Доп. материал:

- [Вкусный и здоровый гайд по юзабилити-тестированием](#)
- [Юзабилити-тестирование на удаленке. Выводы и лайфхаки по итогам года работы](#)
- [User Interface Elements](#)
- [Design for Fingers, Touch, and People, Part 1](#)

- [Чеклист на соответствие гайдлайнам по доступности \(WCAG\)](#)
- [Web Content Accessibility Guidelines \(WCAG\)](#)

Тестирование доступности (Accessibility testing)

Тестирование доступности (accessibility testing) - это подмножество юзабилити-тестирования. Его цель - убедиться в том, что наш продукт удобен в использовании людям с различными видами ограничений, инвалидности или особенностями восприятия. Это могут быть проблемы со зрением, слухом или ограничения в подвижности рук. Что наиболее важно, существуют определенные законы и инструкции по тестированию доступности, которые также должны соблюдаться, например, Рекомендации по доступности веб-контента ([Web content accessibility guidelines](#)). 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?

Источники:

- [Accessibility Testing Tutorial \(A Complete Step By Step Guide\)](#)
- [Accessibility Testing Tutorial: What is, Tools & Examples](#)

Доп. материал:

- [Accessibility Testing: Color Blindness](#)
- [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](#)
- [Говорим о практике в области UX/UI-тестирования в Университете ИТМО — подкаст «ITMO Research»](#)
- [Accessibility Testing Tutorial: What is, Tools & Examples](#)

Инсталляционное тестирование (Installation Testing)

Тестирование инсталляции (установки) направлено на проверку успешной установки, настройки, обновления и удаления ПО, как десктопного, так и мобильного.

Примеры кейсов:

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 административными), 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;

Источник:

[Тестирование инсталляции](#)

[**Тестирование локализации, глобализации и интернационализации \(Localization/globalization/internationalization testing\)**](#)

Глобализированное ПО - это ПО, функционирующее одинаково качественно 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, 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.

Globalization = Internationalization + Localization.

Интернационализация ПО (Internationalization (I18N)) – 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 I18N тестируют работу продукта на одинаковую работу 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 интернационализации направлен на:

- Тестирование языковой совместимости: это включает проверку того, может ли продукт правильно работать в определенной языковой среде;
- Тестирование функциональности: это включает выполнение регрессионных тестов функциональности в различных языковых средах и ввод строк на родном языке. Это включает в себя проверку того, правильно ли отображается и принимается на ввод валюта, дата, время, индекс и т.п.;
- Проверка пользовательского интерфейса: пытаются выявить любые визуальные проблемы, такие как проблемы с графикой, наложение текста, усечение текста и т. д.;
- Тестирование совместимости: это включает тестирование программного обеспечения на целевых кросс-платформах, операционных системах, версиях приложений и т. д.;
- Тестирование юзабилити: проверяет простоту использования приложения;
- Тестирование установки: это включает попытку установить приложение на разных родных языках и проверить, правильно ли отображаются все сообщения об установке в языковых настройках;

Локализация ПО (Localization (L10N)) – деятельность по модификации ПО в соответствии с определенными региональными настройками (языком, географической территорией, культурными особенностями). 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 (например, онлайн-сервис по заказу бургеров не будет показывать корову на главной странице в Индии или свинью в мусульманских странах). На что обратить внимание:

- Длина переведенных слов;
- Параметры шрифта пользовательского интерфейса;
- Ввод текста в разных локализациях;
- RTL-языки (справа-налево) или вертикальные;
- 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.;

Примеры проверок:

- **Языковой словарь:** Глобализированный продукт поддерживает множество языков. Чем больше языков он поддерживает, тем больше потребность в тестировании. Вы можете использовать языковые переводчики и по одному проверять, использует ли приложение правильный словарный запас для каждого языка;
- **Пользовательский интерфейс:** Как вы знаете, у каждого языкового сценария свой стиль письма (некоторые пишутся слева направо, а некоторые - справа налево), и пространство, необходимое для

слов, может варьироваться от одного языка к другому. Таким образом, необходимо протестировать макет пользовательского интерфейса на каждом языке, чтобы убедиться, что пользовательский интерфейс чистый и отсутствуют такие проблемы, как перекрытие текста, несовпадение текста, проблемы с навигацией и т. д.;

- **Обозначение даты и времени:** Форматы отображения даты и времени зависят от региона. Например, наиболее распространенный формат даты в США - мм / дд / гггг. В отличие от этого, наиболее распространенный формат даты в Европе - дд / мм / гггг. С другой стороны, Канада принимает как ДД / ММ / ГГГГ, так и ММ / ДД / ГГГГ. Точно так же некоторые страны используют 24-часовую нотацию, в то время как другие используют 12-часовую нотацию. Поэтому очень важно убедиться, что дата и время отображаются в соответствующем формате при переключении на другие регионы / страны;
- **Корректность даты / времени:** Это не только формат, но и фактическая дата и время варьируются от региона к региону в зависимости от часового пояса. Например, 11:53 субботы по индийскому стандартному времени (IST) - 1:23 субботы по восточному времени (ET). Значит, необходимо проверить правильность отображения даты и времени в приложении при переключении в разные страны;
- **Формат валюты и обработка курсов конвертации:** Если ваше приложение включает электронную коммерцию, проверка валюты становится критически важной. Числовые форматы валют варьируются от страны к стране. Итак, вам следует позаботиться о форматировании. Еще одна важная вещь - отображать правильный символ валюты вместе с единицами измерения. Например, если цена товара составляет 100 рупий, но в приложении он упоминается как «100», это может сбить с толку покупателя, так как это 100 рупий или 100 долларов. Следующим важным тестом должно быть подтверждение того, позаботились ли о коэффициентах конверсии. Также рекомендуется отображать обменный курс для пользователя, чтобы сделать его более удобным и полезным;
- **Формат номера телефона, адреса и почтового индекса:** Порядок отображения адресов зависит от языка. Например, на японском языке порядок адресов - это почтовый индекс, штат, город, а на английском языке порядок адресов - это имя, город, штат, почтовый индекс и т. д. Итак, вам необходимо проверить, нормально ли работает отображение порядка адресов при переключении между разными языками, поддерживаемыми вашим приложением. Точно так же длина и формат телефонного номера также различаются от страны к стране. В наши дни у нас также есть [рекомендация E.164](#) для форматирования чисел в соответствии с общей международной нотацией,

Примечание автора: частный случай задачи на тестирование локализации в android/ios приложениях может быть и в контексте файлов strings, в которых приложение хранит все текстовые строки. Строки могут быть с динамически подставляемыми параметрами чтобы содержимое строки динамически изменялось в зависимости от чего-либо. Например: "Вы сможете запросить код повторно через %s" или "Закрыто. До открытия %1\$d ч.". В данном случае потребуется проверить переводы на предмет того, что динамические аргументы в строках не были сломаны переводчиками. Лично я для этого писал скрипт на python, он есть в другом репозитории.

Источник:

[What Is Globalization Testing \(A Complete Guide\)](#)

Доп. материал:

- [Sample International Test Cases](#)
- [Страх и ненависть локализации в больших проектах. Доклад Яндекса](#)
- [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](#)
- [Internationalization & localization testing](#)
- [Android Developers - Docs - Reference - Formatter](#)

Исследовательское тестирование (Exploratory testing)

Исследовательское Тестирование — одновременно является и техникой, и видом тестирования. В общем виде мы так или иначе всегда используем комбинацию сценарного и исследовательского подходов.

Exploratory testing подразумевает под собой одновременно изучение проекта, функционала, тест-дизайн в уме и тут же исполнение тестов, после чего данный цикл может повторяться необходимое количество раз, каждый раз улучшая создаваемые кейсы и документируя пройденные сессии.

Джеймс Бах указал на важную характеристику исследовательского тестирования - тестировщик участвует когнитивно. Он активно, целенаправленно, с любопытством исследует тестируемое программное обеспечение, всегда принимая на себя ответственность каждую минуту решать, какой путь к тому, что он выбрала для исследования, является наиболее многообещающим. Нет никаких искусственных ограничений на разведку. Тестировщик может свободно использовать любые доступные источники информации, включая спецификации, записи службы технической поддержки, реализации сопоставимого программного обеспечения конкурентами и (конечно) эксперименты (тесты), которые эмпирически раскрывают информацию. Нет никаких ограничений на методы тестирования, которые могут использовать исследователи - например, любая степень автоматизации подойдет. Однако исследователь не просто перезапускает старые тесты, а тестирует чтобы учиться. Вероятно, он будет внимательно изучать поведение программы во время ее тестирования, ища новые идеи о том, как она может выйти из строя, как ее можно было бы в дальнейшем протестировать или измерить, и насколько полезны эти тесты на данном этапе разработки. Выполнение тестов можно автоматизировать, а мышление - нет. Антитезой исследования является тестирование по сценарию, в котором тестировщик (или машина) следует набору процедур, изложенных давно, сравнивая наблюдаемое поведение с любыми результатами, которые разработчик тестов считал актуальными или интересными в то время. Познание произошло тогда, а не сейчас. Объем исследования такой же, как и объем самого тестирования. Разница в том, что исследователь выполняет их в любой полезной последовательности, смешивая исследование, дизайн, выполнение, интерпретацию и общение, чтобы постоянно открывать новую информацию и идти в ногу с текущими изменениями на рынке, платформе, дизайне и реализации тестируемого программного обеспечения.

Подход к тестированию:

- Используйте эвристики для управления тестированием;
- Выполнение и создание тест-кейсов идут рука об руку;
- Тест-кейсы продолжают развиваться на основе наблюдений и обучения тестировщиков;
- К ЕТ могут применяться различные методы тестирования, такие как анализ граничных значений, классы эквивалентности и т. д.;
- ЕТ можно использовать сессионно , чтобы сделать его более структурированным и сфокусированным;
- Тестировщики могут развивать свои идеи, но никогда не отклоняться от своей миссии;
- Тестирование ЕТ не использует сценарии, а зависит от интуиции, навыков и опыта тестировщика;

Туры 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.

| | 1 | Наиболее вероятные типы ошибок | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----------------------|--|--------------------------------|--------------|-----------------|-----------|------------------------|--------------------------------------|---|-----------------------------|---------------|
| | | Функциональны е | Интерфейсные | Лингвистические | Юзабилити | Производитель ность | Эффективность расхода ресурсов | Ошибки в документации на ПО, в справке | | Другие |
| Туры и районы | | | | | | | | | | |
| 1 | Бизнес-центр (Tours of the Business District) | | | | | | | | | |
| 2 | Тур по путеводителю (Guidebook Tour) | | | да | да | | | да | да | |
| 3 | Денежный тур (Money Tour) | да | | да | | | | да | да | |
| 4 | Тур по ориентирам (Landmark Tour) | да | | (критические) | | | | | | |
| 5 | Тур интеллектуала (The Intellectual Tour) | да (крахи) | | | да | да | | | | |
| 6 | Тур службы доставки (FedEx Tour) | да | да | да | да | | | | | |
| 7 | Тур после работы, внеурочный тур (The After-Hours Tour) | да | | | | мб | да | | | |
| 8 | Тур мусоросборника (The Garbage Collector's Tour) | да | да | да | да | да | да | да | да (любые) | |
| 9 | Исторический район (Tours Through the Historical District) | | | | | | | | | |
| 10 | Тур по нерекоменданным местам (The Bad-Neighborhood Tour) | да | | | | | | | | |
| 11 | Музейный тур (The Museum Tour) | да (крахи) | | | | | да | | да (несоотв. стандартам) | |
| 12 | Тур предыдущей версии (The Prior Version Tour) | да | | | да | | | | | |
| 13 | Район развлечений (Tours Through the Entertainment District) | | | | | | | | | |
| 14 | Тур актера второго плана (The Supporting Actor Tour) | да | да | да | | | | | | |
| 15 | Тур по темным переулкам (The Back Alley Tour) | да | да | да | | | | | | |
| 16 | Тур любителя ночной жизни (The All-Nighter Tour or Clubbing Tour) | да | | | | да | да | | | |
| 17 | Туристический район (Tours Through the Tourist District) | | | | | | | | | |
| 18 | Тур коллекционера (The Collector's Tour) | да | | | | | | | | |
| 19 | Тур одиночного бизнесмена (The Lonely Businessman Tour) | да | да | да | да | да | да | да | да | |
| 20 | Тур супермодели (The Supermodel Tour) | да | | | да | | | | | |
| 21 | Тур шопоголика (The TOGOF Tour - Test One Get One Free) | да | | | | да | | | | |
| 22 | Тур по шотландским пабам (The Scottish Pub Tour) | да | да | да | да | да | да | да | да (любые) | |
| 23 | Район отелей (Tours Through the Hotel District) | | | | | | | | | |
| 24 | Тур, отмененный из-за дожда (The Rained-Out Tour) | да | | | | да | | | | |
| 25 | Тур лежебоки (The Couch Potato Tour) | да | | | | да | | | | |
| 26 | Недлагополучный район (Tours Through the Seedy District) | | | | | | | | | |
| 27 | Тур саботажника, диверсанта (The Saboteur Tour) | да (крахи) | | | да | | | | | |
| 28 | Антисоциальный тур (The Antisocial Tour) | да (крахи) | | | да | | | | | да (зашщщен.) |
| 29 | Обсессивно-компульсивный тур или тур нервотика (The Obsessive-Compulsive Tour) | да | | | | | да | да | | |
| 30 | | | | | | | | | | |

Источники:

- [What Is Exploratory Testing In Software Testing \(A Complete Guide\)](#)
- [BBST - Exploratory Testing](#)
- [Исследовательское тестирование и исследовательские туры Виттакера](#)

Доп. материал:

- [Rapid Software Testing Methodology](#)
- [Exploratory Testing](#)
- [Sample test cases of Exploratory Testing of the IRCTC website](#)
- [Exploratory Testing an Indispensable Nonsystematic Software Testing Technique](#)
- [Исследовательское тестирование: пустая трата времени или мощный инструмент?](#)
- [Исследовательское тестирование — полезно или вредно для проекта?](#)
- [Exploratory Testing](#)
- [Exploratory Testing Dynamics](#)
- [A Tutorial in Exploratory Testing](#)
- [Plan your next exploratory testing session](#)
- [Исследовательское тестирование – обезьяняла работа?](#)
- [Exploratory Testing](#)
- [How To Use Tours To Ensure Complete And Thorough Exploratory Testing](#)
- [Туры в исследовательском тестировании. Личный перевод из книги Д. Виттакера «Исследовательское тестирование ПО»](#)
- [Переводы туров для исследовательского тестирования](#)

Свободное / Интуитивное тестирование (Adhoc, Ad-hoc Testing)

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 тестировщик должен иметь полные знания и осведомленность о тестируемой системе, особенно если проект очень сложный и большой. Поэтому нужно хорошее представление о целях проекта, его назначении, основных функциях и возможностях.

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. Сразу после того, как разработчик завершает модульное тестирование, тестировщик и разработчик вместе работают над модулем. Этот вид тестирования позволяет обеим сторонам рассматривать эту функцию в более широком масштабе. Разработчик получит представление обо всех различных тестах, выполняемых тестером, а тестировщик получит представление о том, какова внутренняя конструкция, которая поможет ему избежать разработки недействительных сценариев;
- Pair testing – в этом тестировании два тестировщика (лучше с разным опытом) работают вместе над одним модулем. Идея, лежащая в основе этой формы тестирования состоит в том, чтобы заставить двух тестировщиков провести мозговой штурм идей и методов, чтобы выявить ряд дефектов. Оба могут разделять работу по тестированию и делать необходимую документацию по всем сделанным наблюдениям;
- 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);

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;

| Adhoc Testing | Exploratory Testing |
|--|--|
| Начинается с изучения приложения, а затем - с фактического процесса тестирования | Начинается с тестирования приложения, а затем его понимания посредством исследования |
| Самостоятельный вид тестирования | Разновидность Adhoc Testing |
| Не требуется никакой документации | Обязательно наличие документации по деталям тестирования. |
| Adhoc Testing проводят тестировщики, обладающие глубокими знаниями о приложении | Для изучения приложения не обязательно иметь эксперта. |
| Тестирование начинается после того, как будут собраны все данные для проведения тестирования | Сбор данных и тестирование происходят одновременно. |
| Это работает для отрицательных сценариев тестирования | В основном это касается положительных сценариев |
| Ориентировано на улучшение процесса | Ориентировано на изучение приложения |

| | |
|--|---|
| тестирования | |
| Зависит от творческих способностей и интуиции тестировщика | Зависит от любопытства и понимания тестировщика |
| Нет ограничений по времени | Это ограниченный по времени метод |

Источники:

- [Ad-Hoc Testing: How To Find Defects Without A Formal Testing Process](#)
- [Adhoc Testing Guide - What You Should Know](#)

Тестирование поддержки (Maintenance testing)

Maintenance является последней стадией SDLC. По мнению многих экспертов, по мере того, как изменения после релиза вносятся в существующее приложение, каждое изменение может рассматриваться как начало нового цикла SDLC, но точнее будет сказать, что проект в это время находится в жизненном цикле обслуживания программного обеспечения (SMLC - Software Maintenance Life Cycle). Многие проекты проводят большую часть своего времени именно в SMLC после релиза, а не в SDLC перед ним.

Maintenance testing (тестирование поддержки/обслуживания/эксплуатации/сопровождения) - это модификация программного продукта после его выпуска с целью исправления дефектов, улучшения производительности или других характеристик или для адаптации продукта к изменившемуся окружению. [IEEE 1219]. После того, как программное обеспечение или приложение задеплоены, оно начинает эксплуатироваться годами и даже десятилетиями. В это время система и ее операционная среда часто исправляются, изменяются или расширяются. Пользователю могут потребоваться некоторые добавленные или новые функции в текущем программном обеспечении, которые требуют внесения изменений в текущее программное обеспечение, и эти изменения должны быть протестированы. Конечным пользователям может потребоваться миграция программного обеспечения на другую самую последнюю аппаратную платформу или изменение среды, например версии ОС, варианта базы данных и т. д., что требует тестирования всего приложения на новых платформах и в новой среде. После того, как продукт релизится, он время от времени требует некоторого обслуживания в целях профилактики сбоев.

Основные активности (principal activities):

- Динамическое обслуживание (Dynamic maintenance)
- Корректирующее обслуживание (Corrective maintenance)
- Адаптивное обслуживание (Adaptive maintenance)

Задача выполнения Maintenance testing становится более эффективной, когда программное обеспечение имеет хорошую характеристику обслуживаемости (maintainability).

Reliability, maintainability в ISO 9126 определяется как «легкость, с которой программный продукт может быть изменен для исправления дефектов, модифицирован для соответствия новым требованиям, модифицирован для облегчения будущего обслуживания или адаптирован к изменившейся среде».

Maintainability состоит из:

- Анализируемость: это относится к усилиям, требуемым (обычно разработчиками) для диагностики дефектов или выявления частей программной системы, требующих изменений;
- Изменяемость: это касается усилий, необходимых для фактического исправления дефектов или внесения улучшений;
- Стабильность: вероятность возникновения непредвиденных побочных эффектов в результате внесения изменений в программное обеспечение. Это то, что мы имеем в виду, когда иногда говорим, что программное обеспечение хрупкое (brittle);

- Тестируемость: описывает усилия, необходимые для тестирования измененного программного обеспечения. Это один из основных атрибутов качества программного обеспечения, который напрямую влияет на нашу работу;

Причины плохой Maintainability:

| Основные риски (Maintainability Risks) | Последствия |
|--|--|
| Усилия, необходимые для исправления дефектов и внесения изменений, могут быть больше, чем планировалось | Если размер группы поддержки (maintenance team), выполняющей эти задачи, будет фиксированным (обычная ситуация), это также приведет к увеличению временных затрат |
| Время, затрачиваемое на задачи обслуживания, превышает фиксированные окна обслуживания (maintenance windows) | Это может отрицательно сказаться на производстве (например, сотрудники, прибывающие на работу, обнаруживают, что сервер приложений недоступен из-за проскальзывания окна планового ночного обслуживания) |
| | Поддержка (Maintainers) могут быть вынуждены сокращать сроки, чтобы не выходить за рамки согласованных периодов обслуживания. Им может потребоваться сделать предположения относительно реализации необходимых изменений (возможно, из-за плохой документации) |
| | Если применяются Соглашения об уровне обслуживания, могут налагаться штрафы |
| Долгосрочное накопление плохой поддерживаемости в результате кумулятивного эффекта плохой практики разработки программного обеспечения | Уровень надежности постепенно снижается |
| | Увеличивается количество функциональных дефектов, вносимых изменениями (регрессии) |
| | На исправление дефектов уходит больше времени |
| | На персонал поддержки оказывается все большее давление, что может даже привести к дальнейшему ухудшению ситуации |

Виды Maintenance testing:

- **Подтверждающее тестирование (Confirmation Maintenance Testing):** тестирование измененной функциональности. Вы должны тщательно протестировать все модификации (небольшие или большие), внесенные в программное обеспечение, и убедиться, что нет проблем с функциональностью и простоев. Тестовая среда должна быть копией реальной среды вместе с тестовыми данными;
- **Регрессионное тестирование (Regression Maintenance Testing):** тестирование существующей функциональности на предмет регрессии. Это делается после фазы подтверждающего тестирования. Вы должны протестировать всю систему, чтобы убедиться, что измененная функциональность (работы по обслуживанию) не должна влиять на функциональность существующего программного обеспечения;

Для поддерживающего релиза (maintenance release) может потребоваться поддерживающее тестирование на нескольких уровнях тестирования с использованием различных типов тестов в зависимости от его объема.

Объем технического обслуживания зависит от:

- Степень риска изменения, например, степень, в которой измененная область программного обеспечения общается с другими компонентами или системами;
- Размер существующей системы;
- Размер изменения;

Триггеры для Maintenance. Существует несколько причин, по которым проводится обслуживание программного обеспечения, а значит, и тестирование обслуживания, как для запланированных, так и для незапланированных изменений. Мы можем классифицировать триггеры для обслуживания следующим образом:

- Модификации, такие как запланированные улучшения (например, на основе выпуска), корректирующие и срочные изменения, изменения операционной среды (например, плановые обновления операционной системы или базы данных), обновления программного обеспечения COTS и исправления для дефектов и уязвимостей;
- Миграция, например, с одной платформы на другую, которая может потребовать операционных тестов новой среды, а также измененного программного обеспечения, или тестов преобразования данных, когда данные из другого приложения будут перенесены в поддерживаемую систему;
- Вывод из эксплуатации, например, когда срок поддержки приложения подходит к концу. Когда приложение или система выводятся из эксплуатации, это может потребовать тестирования миграции или архивирования данных, если требуются длительные периоды хранения данных;
- Также может потребоваться тестирование процедур восстановления / извлечения после архивирования в течение длительного периода хранения;
- Регрессионное тестирование может потребоваться, чтобы убедиться, что все функциональные возможности, которые остаются в эксплуатации, по-прежнему работают;

Например, для систем Интернета вещей (IoT) тестирование обслуживания может быть инициировано введением в общую систему совершенно новых или модифицированных вещей, таких как аппаратные устройства и программные услуги. При тестировании обслуживания таких систем особое внимание уделяется интеграционному тестированию на разных уровнях (например, уровень сети, уровень приложений) и аспектам безопасности, в частности, тем, которые относятся к персональным данным.

Impact Analysis для Maintenance. Анализ воздействия оценивает изменения, которые были внесены в поддерживаемую версию, чтобы определить предполагаемые последствия, а также ожидаемые и возможные побочные эффекты (side effects) изменения, а также определить области в системе, на которые это изменение повлияет. Анализ влияния также может помочь определить влияние изменения на существующие тесты. Побочные эффекты и затронутые области в системе необходимо протестировать на предмет регрессии, возможно, после обновления любых существующих тестов, затронутых изменением. Анализ воздействия может быть проведен до внесения изменения, чтобы помочь решить, следует ли вносить изменение, исходя из возможных последствий в других областях системы.

Источники:

- [Importance of Maintainability to a Good Software & Role of Maintenance Testing](#)
- [Maintenance Testing Guide](#)
- [Maintenance Testing](#)

Доп. материал:

- [IEEE Guide to the Software Engineering Body of Knowledge](#). Chapter 5.

Регрессионные виды тестирования (Regression testing)

Регресс - это противоположность прогресса. Любое ПО по мере прогресса в функционале неизбежно усложняется, увеличиваются взаимосвязи в функциях и т.п., и чтобы убедиться в том, что в существующей системе не начинается регресс, полезно иногда проводить ее полное тестирование. И уж тем более логично перетестировать всё, что можно, если в систему были внесены какие-то существенные изменения. Но этого недостаточно. По-сути, проблема намного серьезнее - мы каждый раз не знаем, что принесет с собой новая функциональность в системе. Нам каждый раз надо предположить/узнать/протестировать новые

взаимодействия в системе, а не тестировать только новые функции в изоляции от остальных. Старый функционал с новым если начинают пересекаться - надо заново расчехлять аналитику, выявлять новые ситуации, которые могут возникнуть, писать новые тест-кейсы, которые затрагивают уже не столько функциональные, сколько интеграционные аспекты. Поэтому выяснение "не наступил ли регресс" (внимание, не путать с "не наступила ли регрессия") - постоянная задача, которую также необходимо решать в контексте maintenance testing.

Регрессионное тестирование (Regression Testing) - собирательное название для всех видов тестирования программного обеспечения связанных с изменениями, направленных на обнаружение ошибок в уже протестированных участках исходного кода, на проверку того, что новая функциональность не зааффектила (affect) старую. Такие ошибки — когда после внесения изменений в программу перестаёт работать то, что должно было продолжать работать, — называют регрессионными ошибками (regression bugs). Регрессионные тесты должны быть частью релизного цикла (Release Cycle) и учитываться при тестовой оценке (test estimation).

При корректировках программы необходимо гарантировать сохранение качества. Для этого используется регрессионное тестирование - дорогостоящая, но необходимая деятельность в рамках maintenance testing, 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. Регрессионное тестирование обычно проводится перед релизом новой версии приложения. Это происходит следующим образом: в течение какого-то времени делаются какие-то фичи и другие задачи, они тестируются по отдельности и сливаются в общую ветку (мастер/девелоп - чаще всего эта ветка называется в зависимости от процессов в проекте). Дальше, когда время подходит к релизу от ветки девелопа создается ветка релиза, из которой собирается релиз-кандидат и на нем уже проводят регресс.

Главной задачей maintenance testing 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 (Impact Analysis, анализ влияния изменений). Итоговая область регрессии называется Regression Scope / Scope of Regression.

Классификация регрессионного тестирования:

- Проверить всё (Retest All): Как следует из названия, все тест-кейсы в наборе тестов повторно выполняются, чтобы гарантировать отсутствие ошибок, возникших из-за изменения кода. Это дорогостоящий метод, поскольку он требует больше времени и ресурсов по сравнению с другими методами;
- Минимизация набора тестов (test suite minimization) стремится уменьшить размер тестового набора за счет устранения избыточных тестовых примеров из тестового набора;

- Задача выбора теста (test case selection) связана с проблемой выбора подмножества тестов, которые будут использоваться для проверки измененных частей программного обеспечения. Для этого требуется выбрать подмножество тестов из предыдущей версии, которые могут обнаруживать неисправности, основываясь на различных стратегиях. Большинство задокументированных методов регрессионного тестирования сосредоточены именно на этой технике. Обычная стратегия состоит в том, чтобы сосредоточить внимание на отождествлении модифицированных частей SUT (system under test) и для выбора тестовых случаев, имеющих отношение к ним. Например, техника полного повторного тестирования (retest-all) – один из наивных типов выбора регрессионного теста путем повторного выполнения всех видов тестов от предыдущей версии на новой. Она часто используется в промышленности из-за её простого и быстрого внедрения. Тем не менее, её способность обнаружения неисправностей ограничена. Таким образом, значительный объём работ связан с разработкой эффективных и масштабируемых селективных методов;
- Задача определения приоритетов теста (test case prioritization). Ее цели заключаются в выполнении заказанных тестов на основе какого-либо критерия. Например, на основе истории, базы или требований, которые, как ожидается, приведут к более раннему выявлению неисправностей или помогут максимизировать некоторые другие полезные свойства;
- Гибридный: Гибридный метод представляет собой комбинацию выборочного и приоритезации. Вместо того, чтобы выбирать весь набор тестов, выберите только те тест-кейсы, которые повторно выполняются в зависимости от их приоритета;

Типы регрессии по Канеру:

- 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;

Регрессия в Agile:

В Agile продукт разрабатывается в рамках короткой итерации, называемой спринтом, которая длится 2–4 недели. В Agile существует несколько итераций, поэтому это тестирование играет важную роль, поскольку в итерациях добавляется новая функциональность или изменения кода. Набор регрессионных тестов должен быть подготовлен на начальном этапе и обновляться с каждым спринтом. В Agile проверки регрессии делятся на две категории:

- Регрессия уровня спрингта (Sprint Level Regression): выполняется в основном для новых функций или улучшений, внесенных в последний спрингт. Тест-кейсы из набора тестов выбираются в соответствии с новыми добавленными функциями или сделанными улучшениями;
- Сквозная регрессия (End to End Regression): включает в себя все тест-кейсы, которые должны быть повторно выполнены для сквозного тестирования всего продукта, охватывая все основные функции;

Смоук тестирование (Smoke testing)

Smoke testing, BVT - Build Verification Testing, BAT - Builds Acceptance Testing, Breath Testing, Shakeout/Shakedown Testing, Intake test, а также в русскоязычных вариантах дымовое, на дым, дымное, тестирование сборки и т.п. - это подмножество регрессионного тестирования, короткий цикл тестов, выполняемый для каждой новой сборки для подтверждения того, что ПО после внесенных изменений стартует и выполняет основные функции без критических и блокирующих дефектов. В случае отсутствия таких дефектов Smoke testing объявляется проходным, и команда QA может начинать дальнейшее тестирование полного цикла, в противном случае, сборка объявляется дефектной, что делает дальнейшее тестирование пустой тратой времени и ресурсов. В таком случае сборка возвращается на доработку и исправление. Smoke testing обычно используется для Integration, Acceptance and System Testing.

Если мы говорим про сайт интернет-магазина, то сценарий может быть 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 случайному контакту

Небольшая шпаргалка по степени важности:

- **smoke** - самое важное. Тест-кейсы играют очень важную роль на этом уровне тестирования, поэтому предел метрик (metric limit) часто соответствует 100% или примерно 100%;
- **critical path** - повседневное. Тесты критического пути запускаются для проверки функциональности, используемой типичными пользователями в их повседневной деятельности. Есть много пользователей, которые обычно используют определенную часть функциональности приложения, которую необходимо проверить, как только smoke этап будет успешно завершен. Здесь лимит метрик немного ниже, чем у smoke, и соответствует 70-80-90% в зависимости от цели проекта;
- **extended** - все. Выполняется для изучения всей функциональности, указанной в требованиях. Проверяется даже функциональность с низким приоритетом. При этом в этом тестировании нужно понимать, какой функционал наиболее ценный, а какой менее важный. При условии, что у вас достаточно времени или других ресурсов, тесты на этом уровне можно использовать для требований с низким приоритетом;

Примечание. 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.

Санити тестирование (Sanity testing)

Sanity testing также является подмножеством регрессионного тестирования и выполняется до или вместо полной регрессии, но после smoke. Эти два подвида похожи (где-то вообще не заморачиваются и используют как синонимы), но в целом Sanity используется на более стабильных билдах для определения работоспособности определенной части приложения после внесения изменений.

Примечание. Санитарным это тестирование в русскоязычной среде называлось по совершенно непонятным причинам, но гуглится только так. На самом же деле дословно переводится как тестирование на вменяемость / разумность / работоспособность / согласованность или по версии ISTQB “Тест работоспособности”.

Подтверждающее, повторное тестирование ([confirmation testing, re-testing](#))

Повторное тестирование - это тип тестирования, выполняемый в новой сборке по проваленному на старой сборке тест-кейсу с тем же окружением и данными, для проверки того, что этот дефект теперь устранен. Ретест выполняется перед sanity-тестированием, приоритет ре-теста выше регрессионных проверок, поэтому оно должно выполняться перед ними.

Тестирование N+1 (N+1 testing)

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.

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;
- 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;
- 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;

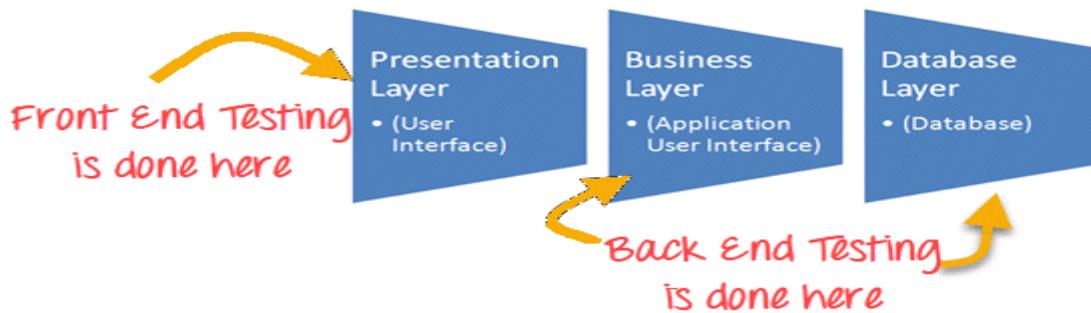
Источники:

- [Maintenance, Regression testing and Re-testing](#)
- [Регрессионное тестирование](#)
- [Тестировщики нужны - пост “Регресс для самых маленьких”](#)
- [QA Outsourcing: Smoke Testing, Critical Path Testing, Extended Testing](#)
- [What Is Regression Testing? Definition, Tools, Method, And Example](#)
- [В чём разница Smoke, Sanity, Regression, Re-test и как их различать?](#)
- [Difference Between Retesting and Regression Testing](#)
- [Top 150 Software Testing Interview Questions and Answers for Freshers and Experienced](#)

Дополнительный материал:

- [Лекция 11: Регрессионное тестирование: цели и задачи, условия применения, классификация тестов и методов отбора](#)
- [Black Box Software Testing PART 11 - REGRESSION TESTING by Cem Kaner + 2005 year version](#)
- [Епифанов Н. А. - Методы реализации регрессионного тестирования по расширенным тестовым наборам](#)
- [Anti-Regression Approaches: Impact Analysis and Regression Testing Compared and Combined – Part I: Introduction and Impact Analysis](#)
- [Anti-Regression Approaches – Part II: Regression Prevention and Detection Using Static Techniques](#)
- [Как сохранить нервы тестировщика или ускорить регресс с 8 до 2 часов](#)
- [Регрессионное тестирование или Regression Testing](#)
- [QA Outsourcing: Smoke Testing, Critical Path Testing, Extended Testing](#)

Тестирование клиентской части и серверной (Frontend testing Vs. Backend testing)



Frontend testing - это тип тестирования, который проверяет уровень представления (Presentation layer) в 3-уровневой архитектуре (3 Tier Architecture). 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.

Источник:

[Frontend Testing Vs. Backend Testing: What's the Difference?](#)

Доп. материал:

- [Как найти границы на клиенте и сервере](#)
- [Как Иван ошибку в бэкенде локализовывал](#)
- [Круглый стол "Почему не стоит тестировать бэкенд руками"](#)

Тестирование графического интерфейса/визуальное тестирование (GUI - Graphical User Interface testing)

Интерфейс - это то, с помощью чего происходит “общение” между ПО и окружением. Существует два типа интерфейсов:

- **Интерфейс командной строки** (CLI - Command Line Interface), где вы вводите текст в терминал, и компьютер отвечает на эту команду;

- **Графический интерфейс пользователя** (GUI - Graphical User Interface), где вы взаимодействуете с компьютером, используя графическое представление, а не текст;

Тестирование графического интерфейса пользователя (GUI) проводят с целью проверить функциональность и корректность отображения интерфейса пользователя (меню, панели инструментов, цвета, шрифты, размеры, значки, контент, кнопки и т. д., как они реагируют на ввод пользователя).

Техники тестирования GUI:

- **Manual testing:** При таком подходе тестеры вручную проверяют графические экраны в соответствии с требованиями, изложенными в документе бизнес-требований (business requirements document);
- **Capture & replay testing или Record and Replay:** Мы также можем провести тестирование графического интерфейса пользователя, используя некоторые инструменты автоматизации, разработанные специально для этого. Идея состоит в том, чтобы запустить приложение и записать взаимодействие, которое должно происходить между пользователем и самим приложением (движения мыши и т. д.), после чего эти тесты будут прогоняться, а фактический результат сравниваться с ожидаемым;
- **Model based testing:** Модель - это графическое описание поведения системы. Это помогает нам понять и спрогнозировать поведение системы. Модели помогают в создании эффективных тестовых примеров с использованием системных требований. Процесс:
 - Построение модели;
 - Определение входных данных для модели;
 - Расчет ожидаемых результатов для модели;
 - Запуск тестов;
 - Сравнение фактических результатов с ожидаемыми;
 - Решение о дальнейших действиях по модели;

Некоторые методы моделирования, на основе которых могут быть получены тестовые примеры:

- Графики - отображает состояние системы и проверяет состояние после некоторого ввода;
- Таблицы решений - таблицы, используемые для определения результатов для входных данных;

Тестирование на основе моделей - это развивающийся метод создания тестовых примеров на основе требований. Его главное преимущество по сравнению с двумя вышеупомянутыми методами заключается в том, что он может определять нежелательные состояния, которых может достичь ваш графический интерфейс.

Примеры проверок:

- **Тип и размер шрифта:** шрифт одинаковый на всех экранах или хотя бы одного семейства, одинаковый размер шрифта заголовков, основного текста и т. д.;
- **Цвета:** должны быть сочетаемы. Придерживайтесь одних цветов и следуйте гайдлайнам. Вы не можете использовать 4 разных варианта оранжевого (если только он не является частью дизайна). Посмотрите на гиперссылки, фон, кнопки, основной текст и т. д.;
- **Стили значков:** вам не следует выбирать 5 разных стилей значков, если вы выбираете «плоские» значки, оставайтесь с плоскими значками;
- **Визуальные несоответствия:** постоянство всегда является ключевым моментом. Внешний вид во всем приложении должен быть одинаковым. Помимо внешнего вида, аббревиатуры также должны быть последовательными;
- **Несоответствия диалоговых окон:** если вы используете «выход» в одних диалоговых окнах, вы должны использовать «выход» в других;
- **Обязательные поля:** всегда лучше указать, что поле является обязательным, добавив к нему звездочку и предоставив пользователю своего рода предупреждение, если данные не указаны;
- **Ошибки типов данных:** всегда проверяйте, что указан правильный тип данных (даты, возраст, вес и т. д.);

- **Один и тот же документ, несколько открытий:** когда документ открывается / загружается более одного раза, вместо перезаписи вы можете переименовать его, добавив номер к имени файла;
- **Ширина полей:** очевидно, если разрешено определенное количество символов и введенные данные не должны превышать определенное число, вы должны прояснить это;
- **Экранные инструкции:** экраны (непонятные) должны содержать какие-то экранные инструкции, которые помогут / направят пользователя;
- **Индикаторы выполнения:** когда ждете результатов, индикаторы выполнения хороши, чтобы пользователи понимали, что им нужно чего-то ждать и что процесс все еще продолжается;
- **Подтверждение сохранения:** если вы можете вносить изменения в приложение без необходимости сохранения, всегда полезно убедиться, что пользователь не хочет сохранять, прежде чем перейти к другому экрану;
- **Подтверждение удаления:** поскольку мы подтверждаем сохранение, всегда полезно подтвердить, что пользователь хочет удалить элемент. Я уверен, что многие из вас (как и я) удаляли что-то на странице, не желая этого;
- **Ввод перед Drop down list:** когда у вас есть сотни вариантов на выбор в выпадающем меню, гораздо лучше иметь возможность сначала вводить текст, чем просматривать весь список;
- **Недопустимые параметры:** иногда чтобы выбрать какие-то параметры вам необходимо подтвердить другие. Эта опция должна отображаться как доступная, когда все требования выполнены;
- **Пункты меню:** показывать только те пункты меню, которые доступны в данный момент, вместо отображения всех пунктов, даже если они недоступны;
- **Сообщения об ошибках:** сообщения об ошибках должны быть информативными;
- **Рабочие шорткаты:** если в вашем приложении есть шорткаты, убедитесь, что все они работают, независимо от того, какие браузеры используются;
- **Разные разрешения:** проверить корректность верстки при масштабировании и на разных разрешениях;
- **Полосы прокрутки;**
- **Изображения:** сжатие, выравнивание и т.п.;
- **Проверка орфографии;**

Источники:

- [A Guide To GUI Testing](#)
- [GUI Testing Tutorial: User Interface \(UI\) TestCases with Examples](#)
- [GUI Testing Tutorial: A Complete User Interface \(UI\) Testing Guide](#)

Доп. материал:

- [Эффективное тестирование верстки](#)
- [#9 Артем, Сева и Визуальное тестирование](#)
- [Кросбраузерное визуальное тестирование - выбор подходящего инструмента для дизайн-системы NewsKit](#)
- [О бедном мокапе замолвите слово](#)
- [A Pattern Library for Interaction Design](#)

Тестирование API (API - Application Programming Interface)

Каждый день используя любимые мобильные приложения и веб-ресурсы вы незаметно взаимодействуете с API, скрытым под интерфейсом пользователя.

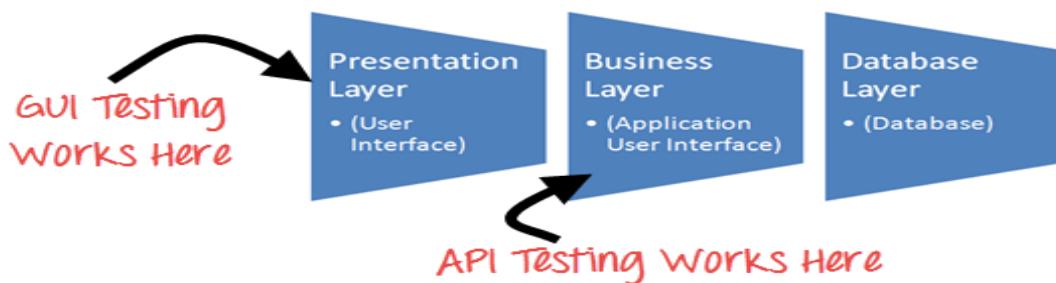
API действует как интерфейс между двумя программными приложениями и позволяет им связываться друг с другом на оговоренных правилах и не залезая в реализацию предоставляемых функций, при том правила - не договоренность, а контракт. Простой пример: вы можете встроить на свою главную страницу сайта маленький 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.

Тестирование API - это тип тестирования (хотя правильнее наверное сказать не тип или вид, а еще один вариант взаимодействия с системой) 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.



Типы тестирования API:

- Unit testing: Для проверки функциональности отдельной операции;
- Functional testing: Чтобы проверить функциональность более широких сценариев с помощью блока результатов unit-тестирования, протестируемых 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;
- Usability testing: проверяет, является ли API функциональным и удобным для пользователя и хорошо ли интегрируется с другой платформой;
- Documentation testing: команда тестирования должна убедиться, что документация соответствует требованиям и предоставляет достаточно информации для взаимодействия с API. Документация должна быть частью окончательного результата;

Примеры проблем, которые обнаруживает тестирование API:

- Некорректная обработка условий ошибки;

- Неиспользуемые флаги;
- Отсутствующие или повторяющиеся функции;
- Вопросы надежности;
- Сложность подключения и получения ответа от API;
- Проблемы с безопасностью;
- Проблемы с многопоточностью;
- Проблемы с производительностью. Время отклика API очень велико;
- Неправильные ошибки / предупреждение вызывающему абоненту;
- Неправильная обработка допустимых значений аргументов;
- Данные ответа неправильно структурированы (JSON или XML);

Контрактное тестирование API

В общем случае контрактное тестирование или Consumer Driven Contract (CDC) является связующим звеном между модульным и интеграционным тестированием.

Каждый интерфейс имеет поставщика (supplier) и потребителя (consumer). Само собой, сервисы поставщика и потребителя распределены между разными командами, мы оказываемся в ситуации, когда четко прописанный интерфейс между ними (или контракт) просто необходим. Обычно многие подходят к этой проблеме следующим образом:

- Пишут подробное описание спецификации интерфейса - контракт;
- Реализуют сервис поставщика согласно спецификации;
- Передают спецификацию интерфейса потребителю;
- Ждут реализации от другой стороны;
- Запускают ручные системные тесты, чтобы всё проверить;
- Держат кулаки, что обе стороны будут вечно соблюдать описанный интерфейс;

Сегодня многие компании заменили последние два шага на автоматизированные контрактные тесты, которые регулярно проверяют соответствие описания и реализации у поставщика и потребителя определенного контракта. Что является набором регрессионных тестов, которые обеспечивают раннее обнаружение отклонения от контракта.

Разберемся во взаимодействии на примере REST архитектуры: поставщик создает API с некоторым endpoint, а потребитель отправляет запрос к API, например, с целью получения данных или выполнения изменений в другом приложении. Это контракт, который описывается с помощью DSL (domain-specific language). Он включает API описание в форме сценариев взаимодействия между потребителем и поставщиком. С помощью CDC выполняется тестирование клиента и API с использованием заглушек, которые собираются на основе контракта. Основной задачей CDC является сближение восприятия между командами разработчиков API и разработчиков клиента. Таким образом, участники команды потребителей пишут CDC тесты (для всех данных проекта разработки), чтобы команда поставщика смогла запустить тесты и проверить API. В итоге команда поставщика с легкостью разработает свой API, используя тесты CDC. Результатом прогона контрактных тестов является понимание, что поставщик уверен в исправной работе API у потребителя. Следует обратить внимание, что команда потребителя должна регулярно осуществлять поддержку CDC-тестов при каждом изменении, и вовремя передавать всю информацию команде поставщика. Если регулярно фиксируем неудачно выполненные CDC-тесты, то следует пойти (в буквальном смысле слова, к пострадавшей стороне теста и узнать, в рамках какой задачи были изменения (что привело к падению теста).

Из того, что было описано выше, можно выделить следующие тезисы для выполнения контрактного тестирования:

- Команда разработчиков (тестировщиков) со стороны потребителей пишет автоматизированные тесты с ожидаемыми параметрами со стороны потребителей.
- Тесты передаются команде поставщика.

- Команда поставщика запускает контрактные тесты и проверяет результат их выполнения. Если происходит падение тестов, то команды должны зафиксировать сбой и перепроверить документацию (согласованность разработки).

Минусы CDC:

- CDC тесты не заменяют E2E тесты. По факту я склонен отнести CDC к заглушкам, которые являются моделями реальных компонентов, но не являются ими, т.е. это еще одна абстракция, которую нужно поддерживать и применять в нужных местах (сложно реализовать сложные сценарии);
- CDC тесты не заменяют функциональные тесты API. Лично придерживаюсь золотого правила – если убрать контракт и это не вызывает ошибки или неправильную работу клиента, то значит он не нужен. Пример: Нет необходимости проверять все коды ошибок через контракт, если клиент обрабатывает их (ошибки) одинаково. Таким образом контракт то, что важно для клиента сервиса, а не наоборот;
- CDC тесты дороже в поддержке, чем функциональные тесты;
- Для реализации CDC-тестов нужно использовать (изучать) отдельные инструменты тестирования – Spring Cloud Contract, PACT;

Отличие API от SDK:

SDK (software development kit) - это набор функционала (библиотек) и утилит для разработки. Собственно SDK и предоставляет реализацию некоторого API, это оболочка API's, которая упрощает работу для разработчиков.

- API: набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением для использования во внешних программных продуктах. Это интерфейс, похоже на спецификацию телефонной системы или электропроводки в вашем доме. Это список того, что можно вызывать и какого ждать результата;
- SDK: набор реальных инструментов внедрения. Это как чемодан деталей и инструментов, который позволяет вам подключиться к телефонной системе или электрической проводке. Это библиотеки, в которых реализованы вызываемые функции + файлы необходимые для подключения этих библиотек;

Тестирование 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 endpoint - 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.

Источники:

- [API Testing Tutorial: What is API Test Automation? How to Test](#)
- [A Comprehensive API Testing Guide](#)
- [API Testing Tutorial: A Complete Guide For Beginners](#)
- [Spring Cloud Contract. Что такое контрактное тестирование и с чем его едят](#)
- [Чем отличается api от sdk?](#)
- [Тестирование API без документации](#)

Доп. материал:

- [Курс Тестирование ПО. Занятие 29. Тестирование API - QA START UP](#)
- [Эвристики и мнемоники в тестировании: шаблоны для тестирования API](#)

- [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](#)
- [Тестирование Web API — From Zero To Hero](#)
- [Стратегия тестирования REST API: что именно вам нужно тестировать?](#)
- [Test Design and Automation for Rest API. Part 1. Иван Катунов. Comqa Spring 2018 + Test Design and Automation for Rest API. Part 2. Иван Катунов. Comqa Spring 2018 + pdf](#)
- [What is the Difference Between an API and an SDK?](#)
- [Introduction to API Testing](#)
- [19:05 «Контрактное тестирование Rest API» + презентация](#)
- [Организация контрактного тестирования микросервисов и графического портала](#)
- [Introduction To Contract Testing With Examples](#)
- [Микросервисы для разработчиков Java: Контрактное тестирование](#)
- [Black box API testing with server logs](#)

A/B тестирование (A/B Testing)

Для проведения A/B Testing (split testing,bucket testing) мы создаем и анализируем два варианта чего-либо (экрана приложения, страницы сайта, элементов GUI, механики работы, воронки продаж и т.п.), чтобы найти, какой вариант работает лучше с точки зрения пользовательского опыта, потенциальных клиентов, конверсий или любой другой цели. 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 А / В-тестирование. Мы создадим интерфейс нового варианта и перенаправим часть трафика пользователей на него. Например - мы можем распределить пользователей в соотношении 50:50 или 80:20 между двумя вариантами - А и В. После этого в течение определенного периода времени мы будем наблюдать за статистикой и коэффициентами конверсии обоих вариантов. Таким образом, тестирование A/B помогает принять решение о выборе лучшего варианта.

Исходный вариант (А) называется контроль (control), а альтернативный (В) - вариация (variation). При проведении А / В-тестирования мы получаем данные / статистику от чемпионов, претендентов и вариаций (champions, challengers, and variations). Эти версии дают представление о коэффициентах конверсии ваших посетителей.

Терминология:

- **Вариант (Variant):** разные версии веб-страниц или любой другой маркетинговый актив, может быть несколько форм одной и той же страницы с небольшими изменениями;
- **Чемпион (Champion):** чемпионом будет веб-страница, которая хорошо работает или показывала хорошие результаты в прошлом. Обычно чемпион сравнивается с другими вариантами, и вариант с более высоким коэффициентом конверсии становится вариантом чемпиона;
- **Претендент (Challenger):** это новый вариант вашей веб-страницы, который сравнивается с вашим существующим чемпионом;
- **Трафик (Traffic):** он относится к пользователям, посещающим ваш сайт, и измеряется количеством пользователей, проводящих время на вашем сайте;
- **Коэффициент конверсии (Conversion Rate):** коэффициент конверсии - это процент пользователей, которые совершают желаемое действие, запланированное на веб-сайте, например, подписываются на список рассылки для оплаты продукта, деленное на количество посетителей;
- **Оптимизация коэффициента конверсии (CRO - Conversion Rate Optimization):** это практика оптимизации работы веб-сайта или целевой страницы на основе поведения посетителей веб-сайта. Это помогает владельцу сайта понять, как посетители сайта совершают желаемые действия

- (конверсии), такие как нажатие «добавить в корзину», покупка продукта, подписка на услугу, заполнение формы или нажатие на ссылку, становясь клиентов на соответствующей странице и что им мешает достичь своих целей;
- **Гипотеза (Hypothesis):** хорошо структурированная гипотеза поможет вам понять, является ли она успешной, неудачной или неубедительной. Пример: больше людей будут нажимать кнопку, если она синяя, потому что она контрастирует с другими цветами на странице;
 - **Интуиция (Insights):** это комбинация аргументированных идей и выводов, сделанных на основе систематического сбора и анализа данных;
 - **Копирайт (Copy):** ad copy or sales copy представляет собой письменный контент, который направлен на повышение узнаваемости бренда, чтобы убедить человека или группу совершить определенное действие;

Что тестируется с помощью A/B Testing:

- **Лэндинги (Landing pages):** это веб-страница, на которую пользователь попадает после нажатия на объявление, ссылку в вашей почтовой кампании или в любом другом цифровом месте. Обычно на этих одностраничниках есть четкий призыв к действию (CTA - call to action), чтобы купить продукт или просто присоединиться к вашему списку. Ваша целевая страница должна быть оптимизирована для привлечения пользователей к любому предложению, которое вы им представляете. Перед запуском A / B-тестирования для сбора данных и гипотез используйте тепловую карту, т. е. Визуальное представление внимания, вовлеченности и взаимодействий с посетителями, это поможет вам сосредоточиться на том, что требует внимания;
- **Заголовки (Headlines):** могут иметь очень важное значение, поскольку это первое, на что смотрит любой пользователь, заходя на ваш сайт. Если ваш заголовок не привлекает их внимания, не ожидайте, что они задержатся на вашем сайте. Вы должны быть осторожны при создании сору, шрифта, размера, цвета и сообщения;
- **Макет страницы (Page layout):** должен быть простым, но мощным, чтобы посетители могли получить к нему доступ. Достаточно правильного дизайна и макета с четкой информацией, понятным содержанием, четким призывом к действию, а также медиа, улучшающих визуальный аспект страницы. Будь то ваша домашняя страница, целевая страница, сообщение в блоге или страница с описанием продукта, убедитесь, что дизайн UI / UX не загроможден и ориентирован на конверсии;
- **Навигация (Navigation):** посетитель должен беспрепятственно перемещаться по вашему сайту. Их не должны перегружать или сбивать с толку разные элементы вашего веб-сайта. Разместите панель навигации в верхнем левом углу и логотип вашего сайта в верхнем правом углу, нажав на нее, вы вернетесь на главную страницу. Следуйте этим основным форматам, которые используют большинство веб-сайтов, чтобы пользователь мог быстро найти то, что ищет. В худшем случае ваш посетитель заблудится на вашем сайте;
- **Формы (Forms):** это способ получить информацию о вашем потенциальном клиенте. Формы - лучший способ связаться с вашим потенциальным клиентом. Убедитесь, что они подходящего размера. Если они слишком длинные, ваш посетитель может просто отказаться от них, если они слишком короткие, вы можете не собрать информацию, чтобы убедиться, что посетители являются потенциальными клиентами;
- **Длина страницы, глубина содержимого (Page length, Content depth):** некоторые посетители хотели бы иметь четкий и полезный контент, в то время как некоторые посетители предпочитают контент для глубокого погружения. С помощью A / B-тестирования вы можете определить предпочтения вашей целевой аудитории и удовлетворить их потребности;
- **Призыв к действию (Call To Action):** это самая важная вещь, которая напрямую влияет на коэффициент конверсии. СТА побуждает вашего посетителя совершить любое действие - купить ваш продукт / услугу, подписаться на рассылку электронной почты, послушать ваш подкаст, это может быть что угодно. Но на вашей странице должен быть только один призыв к действию, чтобы посетители не запутались;
- **Медиа (Media):** играют огромную роль в вашей воронке продаж. У вас может быть канал на YouTube, подкаст, Instagram и т. д., Которые направляют вашу аудиторию на ваш сайт. Этот фото / аудио / видео контент может дать вам представление о том, сколько людей из вашей аудитории посмотрели ваш

сайт и сколько из них обратилось к вашему клиенту. Таким образом, вы можете оптимизировать свой контент, чтобы привлечь трафик на ваш сайт;

Многовариантное тестирование (Multivariate Testing): несколько элементов на странице изменяются в комбинации, она сравнивается с текущей версией веб-страницы. Цель проведения многовариантного тестирования - измерить эффективность каждой комбинации дизайна. Многовариантное тестирование дает более быстрые результаты, но может быть сложным для новичка.

| A/B Testing | Multivariate Testing |
|--|---|
| позволяет экспериментировать с одним или несколькими вариантами веб-страницы друг против друга. | помогает вам поэкспериментировать с несколькими вариантами нескольких элементов на веб-странице одновременно |
| Здесь ваш трафик будет разделен на две разные веб-страницы: версия А и версия В. | Здесь ваш трафик будет разделен на несколько веб-страниц с несколькими вариантами. |
| Вам не понадобится большой трафик, чтобы попробовать это тестирование. | Для реализации многовариантного тестирования ваш сайт должен иметь огромные объемы трафика. |
| Пример: изменение цвета кнопки покупки. Версия А: цвет по умолчанию Версия В: красный цвет | Пример: изменение основных элементов целевой страницы. Версия А: Headline_3, Img_2, CTA_1 Версия В: Заголовок_1, Img_3, CTA_2 Версия С: Headline_2, Img_4, CTA_1 Версия D: Headline_4, Img_1, CTA_3 |

Тестирование разделением URL (Split URL Testing): вы создаете два или более вариантов для своей веб-страницы. Затем протестируйте эти несколько версий вашего веб-сайта по разным URL-адресам. Здесь варианты представляют собой полностью разработанные веб-страницы и хранятся на сервере, доступ к которому осуществляется через разные URL-адреса;

| A/B Testing | Split URL Testing |
|---|--|
| Это маркетинговый инструмент, позволяющий выяснить, что лучше всего работает для повышения конверсии, путем разделения трафика между контрольным и вариантом. | Это маркетинговый инструмент, позволяющий выяснить, что лучше всего работает для повышения конверсии, путем разделения трафика по разным URL-адресам. |
| Для создания вариантов будут внесены лишь незначительные изменения в элементы в HTML. | Здесь варианты обычно представляют собой полностью разработанную веб-страницу, которая хранится на сервере и к которой можно получить доступ через другой URL-адрес. |
| Лучше всего использовать при оптимизации отдельных веб-страниц и часто используется для проведения быстрых тестов. | Можно использовать для больших изменений, например, для новых редизайнов |

Мультистраничное тестирование (Multi-Page Testing): При многостраничном тестировании мы экспериментируем, изменяя определенный элемент на нескольких страницах. Здесь вы должны показывать пользователям сочетание и совпадение вариантов вместо того, чтобы показывать согласованные варианты по набору страниц. Таким образом, мы можем тщательно протестировать один вариант против другого;

Источник:

[A/B Testing Guide: How To Perform AB Testing](#)

Доп. материал:

- [How to Do A/B Testing: A Checklist You'll Want to Bookmark](#)
- [Ошибки в дизайне А/В тестов, которые я думала, что никогда не совершу](#)
- [A/B Тестирование: Основы](#)
- [Как выбрать уровень статистической значимости для АБ-теста и как интерпретировать результат](#)
- [Время — деньги: анализируй А/В-тесты разумно](#)
- [Взгляд на А/В-тестирование со стороны тестировщика](#)

Деструктивное и недеструктивное тестирование (DT - Destructive testing and NDT – Non Destructive testing)

Destructive testing (негативное) - тип тестирования ПО для поиска точек отказа в ПО, 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 Destructive testing является предотвращение сбоя приложений из-за некорректных входных данных. Просто проводя положительное тестирование, мы можем только убедиться, что наша система работает в нормальных условиях, но помимо этого мы должны убедиться, что наша система может справиться и с непредвиденными условиями. Типичные примеры: ввести неправильно составленный e-mail и номер телефона, загрузить файл не предусмотренного расширения или размера.

Для деструктивного тестирования существует множество способов его проведения:

- 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 ВА (Business Analyst);
- Экспертная проверка тестировщика: проанализируйте или дайте на ревью ваши Test cases коллеге-тестировщику, который менее знаком с системой/функцией ;
- Бизнес-анализ тест-кейсов: конечные пользователи или эксперты могут подумать о многих допустимых сценариях, которые иногда тестировщики могут их не учитывать или упустить, так как все их внимание будет сосредоточено на тестировании требований;
- Проведение предварительного тестирования 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;

Non Destructive testing (позитивное) - это тип тестирования программного обеспечения, который включает в себя правильное взаимодействие с ПО. Другими словами, неразрушающее тестирование (NDT) также можно назвать позитивным тестированием или тестированием «счастливого пути» (Happy path). Оно 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

Источник:

[What is Destructive Testing? Methods, Techniques and Examples](#)

Доп. материал:

- [Destructive Testing And Non Destructive Testing Tutorial](#)
- [Top 10 негативных кейсов](#)

Выборочное/хаотическое тестирование (Random/monkey testing)

В ISTQB и некоторых других источниках эти понятия разделяются:

- Выборочное тестирование (random testing): Разработка тестов методом черного ящика, при котором тестовые сценарии выбираются для соответствия функциональному разрезу, обычно с помощью алгоритма псевдослучайного выбора. Этот метод может использоваться для тестирования таких нефункциональных атрибутов, как надежность и производительность.
- Хаотическое тестирование (monkey testing): Тестирование случайным выбором из большого диапазона входов, случайным нажатием кнопок, без соотнесения с тем, как в реальности будет использоваться система.

В других же источниках они используются как синонимы. В любом случае, отдельно по Random testing мне не удалось найти хорошего большого материала, так что оставлю пока как есть.

Monkey Testing - это метод тестирования черного ящика, при котором тестировщик предоставляет случайные входные данные и применяет случайные действия в программном приложении для проверки поведения системы. Это помогает нам оценить, дает ли система сбой при получении таких неожиданных входных данных. Здесь входными данными могут быть данные, которые вводятся в приложение, или нажатие кнопки для следующего действия, или нажатие на ссылку для перехода на другую страницу.

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.

Типы Обезьян:

- Тупая обезьяна (Dumb Monkey): тестировщики не имеют представления о системе и ее функциональных возможностях, флоу, валидности ввода. Затруднено воспроизведение ошибок;
- Умная обезьяна (Smart Monkey): тестировщик имеет четкое представление о системе, ее назначении и функциональности. Тестировщик перемещается по системе и предоставляет действительные данные для выполнения тестирования. Всё это полезно при воспроизведении ошибок. Так же умная обезьяна больше сосредоточена на попытках сломать приложение, чем на поиске случайных ошибок.
- Выдающаяся обезьяна (Brilliant Monkey): тестировщики обладают глубокими знаниями о приложении, выполняют тестирование в соответствии с поведением пользователя и могут указать некоторые вероятности возникновения ошибок в будущем;

Gorilla testing проводится в соответствии с методом ручного тестирования, при котором тестировщик многократно тестирует модуль, чтобы проверить его надежность. Здесь разработчик и тестировщик объединяются, чтобы протестировать конкретный модуль во всех аспектах. При тестировании Gorilla каждый модуль приложения берется по одному, и для проверки этих модулей вводится диапазон допустимых и недопустимых входных данных. Эти входные значения берутся случайным образом. Тестирование Gorilla направлено на изучение возможностей отдельных модулей. При тестировании Gorilla каждый второстепенный код приложения проверяется до тех пор, пока он не начнет разваливаться или не даст ожидаемых результатов.

- Поскольку Gorilla testing фокусируется на тестировании отдельных модулей, оно может обнаруживать ошибки на ранней стадии, тем самым снижая затраты при высоком покрытии;
- Gorilla testing гарантирует, что даже заблудший пользователь не столкнется с какими-либо сбоями;
- Gorilla testing помогает команде понять уровень толерантности системы.

| Monkey Testing | Gorilla Testing |
|--|---|
| Не использует никаких тестовых примеров для тестирования приложения, это просто случайные входные данные | Гарантирует, что у модуля нет проблем, выполняя повторяющиеся задачи по вставке случайных входных данных в модуль |

| | |
|---|---|
| Синоним Random testing | Также известно как повторяющееся тестирование или Тестирование на пытки или Тестирование на отказоустойчивость (repetitive testing or Torture Testing or Fault Tolerance Testing) |
| Проверяет выполнение всего приложения, используя случайные входные данные, чтобы гарантировать, что система не выйдет из строя из-за неожиданных значений | стремится тщательно протестировать отдельный модуль |
| Может быть выполнено любым стейкхолдером проекта | Для проведения тестирования gorilla требуется разработчик или тестировщик с хорошим знанием приложения |
| Фокусируется на сбое (crashing) всей системы из-за случайного ввода | фокусируется на тестировании функциональности конкретного модуля |

| Monkey Testing | Ad-hoc Testing |
|---|--|
| Фокусируется на ломании приложения случайным вводом | Фокусируется на поиске ошибок, которые не были обнаружены существующими кейсами |
| Ошибки обнаруживаются на основе случайных входных значений | Ошибки обнаруживаются на основе неисследованных областей приложения |
| Тестировщики не знают приложение, они testируют приложение, случайным образом щелкая или вводя данные, чтобы проверить, не приводит ли это к ошибке | Тестировщик должен хорошо разбираться в приложении и понимать его функции |
| От тестировщика не требуется быть экспертом в данной области или иметь какие-либо глубокие знания о приложении. | Тестировщик будет знать точный рабочий процесс приложения вместе со знанием предметной области |
| Может быть выполнено любым стейкхолдером | Обычно выполняется тестировщиком, который знает приложение |

Если мы говорим о ручном тестировании, он может быть менее эффективным, чем другие методы черного ящика. Но если мы добавим тулы автоматизации, она станет мощным инструментом. Просто представьте, что кейсы с различными наборами входных данных генерируются, выполняются и оцениваются автоматически в непрерывном цикле, что позволяет вам запускать тысячи и миллионы кейсов в течение разумного времени.

Источник: [Monkey Testing Guide](#)

Тестирование рабочего процесса/воркфлоу (Workflow testing)

Это тип тестирования программного обеспечения, который проверяет, что каждый software workflow точно отражает данный бизнес-процесс путем тестирования Software Workflows в документе с бизнес-требованиями (BRD - Business Requirements Document). Workflow - это серия задач для получения желаемого результата, которая обычно включает несколько этапов или шагов. Тестирование рабочего процесса также будет включать в себя части системных и интеграционных тестов. Test Model включает тестирование таких артефактов, как: test cases, test procedures, test components, test sub-system, etc.

Процесс Workflow testing:

- Начальная фаза (Inception phase): эта фаза включает начальное планирование испытаний и тестирование прототипа;
- Фаза разработки (Elaboration phase): эта фаза включает baseline архитектуры 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;

Кто проводит Workflow testing:

- 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;

Источник:

[What is Workflow Testing in Software Testing? with Examples](#)

Тестирование документации (Documentation testing)

Плохая документация может повлиять на качество продукта. 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)

Техники тестирования требований:

Взаимный просмотр (peer review). Взаимный просмотр («рецензирование») является одной из наиболее активно используемых техник тестирования требований и может быть представлен в одной из трех следующих форм (по мере нарастания его сложности и цены):

- Беглый просмотр (walkthrough) может выражаться как в показе автором своей работы коллегам с целью создания общего понимания и получения обратной связи, так и в простом обмене результатами работы между двумя и более авторами с тем, чтобы коллега высказал свои вопросы и замечания. Это самый быстрый, дешевый и часто используемый вид просмотра. Для запоминания: аналог беглого просмотра — это ситуация, когда вы в школе с одноклассниками проверяли перед сдачей сочинения друг друга, чтобы найти описки и ошибки.
- Технический просмотр (technical review) выполняется группой специалистов. В идеальной ситуации каждый специалист должен представлять свою область знаний. Тестируемый продукт не может считаться достаточно качественным, пока хотя бы у одного просматривающего остаются замечания. Для запоминания: аналог технического просмотра — это ситуация, когда некий договор визирует юридический отдел, бухгалтерия и т.д.
- Формальная инспекция (inspection) представляет собой структурированный, систематизированный и документируемый подход к анализу документации. Для его выполнения привлекается большое количество специалистов, само выполнение занимает достаточно много времени, и потому этот вариант просмотра используется достаточно редко (как правило, при получении на сопровождение и доработку проекта, созданием которого ранее занималась другая компания). Для запоминания:

аналог формальной инспекции — это ситуация генеральной уборки квартиры (включая содержимое всех шкафов, холодильника, кладовки и т.д.).

Вопросы. Следующей очевидной техникой тестирования и повышения качества требований является (повторное) использование техник выявления требований, а также (как отдельный вид деятельности) — задавание вопросов. Если хоть что-то в требованиях вызывает у вас непонимание или подозрение — задавайте вопросы. Можно спросить представителей заказчика, можно обратиться к справочной информации. По многим вопросам можно обратиться к более опытным коллегам при условии, что у них имеется соответствующая информация, ранее полученная от заказчика. Главное, чтобы ваш вопрос был сформулирован таким образом, чтобы полученный ответ позволил улучшить требования;

Тест-кейсы и чек-листы. Мы помним, что хорошее требование является проверяемым, а значит, должны существовать объективные способы определения того, верно ли реализовано требование. Продумывание чек-листов или даже полноценных тест-кейсов в процессе анализа требований позволяет нам определить, насколько требование проверяемо. Если вы можете быстро придумать несколько пунктов чек-листа, это ещё не признак того, что с требованием всё хорошо (например, оно может противоречить каким-то другим требованиям). Но если никаких идей по тестированию требования в голову не приходит — это тревожный знак. Рекомендуется для начала убедиться, что вы понимаете требование (в том числе прочесть соседние требования, задать вопросы коллегам и т.д.). Также можно пока отложить работу с данным конкретным требованием и вернуться к нему позднее — возможно, анализ других требований позволит вам лучше понять и это конкретное. Но если ничего не помогает — скорее всего, с требованием что-то не так. Справедливости ради надо отметить, что на начальном этапе проработки требований такие случаи встречаются очень часто — требования сформированы очень поверхностно, расплывчато и явно нуждаются в доработке, т.е. здесь нет необходимости проводить сложный анализ, чтобы констатировать непроверяемость требования. На стадии же, когда требования уже хорошо сформулированы и протестированы, вы можете продолжать использовать эту технику, совмещая разработку тест-кейсов и дополнительное тестирование требований.

Исследование поведения системы. Эта техника логически вытекает из предыдущей (продумывания тест-кейсов и чек-листов), но отличается тем, что здесь тестированию подвергается, как правило, не одно требование, а целый набор. Тестировщик мысленно моделирует процесс работы пользователя с системой, созданной по testable требованиям, и ищет неоднозначные или вовсе неописанные варианты поведения системы. Этот подход сложен, требует достаточной квалификации тестировщика, но способен выявить нетривиальные недоработки, которые почти невозможно заметить, тестируя требования по отдельности.

Рисунки (графическое представление). Чтобы увидеть общую картину требований целиком, очень удобно использовать рисунки, схемы, диаграммы, интеллект-карты и т.д. Графическое представление удобно одновременно своей наглядностью и краткостью (например, UML-схема базы данных, занимающая один экран, может быть описана несколькими десятками страниц текста). На рисунке очень легко заметить, что какие-то элементы «не стыкуются», что где-то чего-то не хватает и т.д. Если вы для графического представления требований будете использовать общепринятую нотацию (например, уже упомянутый UML), вы получите дополнительные преимущества: вашу схему смогут без труда понимать и дорабатывать коллеги, а в итоге может получиться хорошее дополнение к текстовой форме представления требований.

Прототипирование. Можно сказать, что прототипирование часто является следствием создания графического представления и анализа поведения системы. С использованием специальных инструментов можно очень быстро сделать наброски пользовательских интерфейсов, оценить применимость тех или иных решений и даже создать не просто «прототип ради прототипа», а заготовку для дальнейшей разработки, если окажется, что реализованное в прототипе (возможно, с небольшими доработками) устраивает заказчика.

Подробный разбор примера тестирования требований можно прочитать в книге Святослава Куликова «Тестирование программного обеспечения. Базовый курс» в разделе 2.2.7. «Пример анализа и тестирования требований».

Источники:

- [What is Documentation Testing in Software Testing](#)
- [Святослав Куликов “Тестирование программного обеспечения. Базовый курс”](#). Глава 2.

Доп. материал:

- [Тестирование требований: как я нахожу ошибки в бизнес-логике фичи прежде, чем их закодят](#)
- [Чек-лист тестирования требований](#)

Мутационное тестирование (Mutation testing)

Юнит тесты помогают нам удостовериться, что код работает так, как мы этого хотим. Одной из метрик тестов является процент покрытия строк кода (Line Code Coverage). Но насколько корректен данный показатель? Имеет ли он практический смысл и можем ли мы ему доверять? Ведь если мы удалим все assert строки из тестов, или просто заменим их на assertSame(1, 1), то по-прежнему будем иметь 100% Code Coverage, при этом тесты ровным счетом не будут тестировать ничего. Насколько вы уверены в своих тестах? Покрывают ли они все ветки выполнения ваших функций? Тестируют ли они вообще хоть что-нибудь? Ответ на этот вопрос даёт мутационное тестирование.

Мутационное тестирование (MT, Mutation Testing, Mutation Analysis, Program mutation, Error-based testing, Fault-based testing strategy) - это вид тестирования ПО методом белого ящика, основанный на всевозможных изменениях (мутациях) частей исходного кода и проверке реакции на эти изменения набора автоматических юнит тестов. Изменения в мутантной программе сохраняются крайне небольшими, поэтому это не влияет на общее исполнение программы. Если тесты после изменения кода не падают (failed), значит либо этот код недостаточно покрыт тестами, либо написанные тесты бесполезны. Критерий, определяющий эффективность набора автоматических тестов, называется Mutation Score Indicator (MSI).

Введем некоторые понятия из теории мутационного тестирования:

Для применения этой технологии у нас, очевидно, должен быть исходный код (source code), некоторый набор тестов (для простоты будем говорить о модульных — unit tests). После этого можно начинать изменять отдельные части исходного кода и смотреть, как реагируют на это тесты. Одно изменение исходного кода будем называть **Мутацией** (Mutation). Например, изменение бинарного оператора "+" на бинарный "-" является мутацией кода. Результатом мутации является **Мутант** (Mutant) — то есть это новый мутированный код. Каждая мутация любого оператора в вашем коде (а их сотни) приводит к новому мутанту, для которого должны быть запущены тесты. Кроме изменения "+" на "-", существует множество других **мутационных операторов** (Mutation Operator, Mutator, faults or mutation rules), каждый из которых имеет свою цель и применение:

- **Мутация значений** (Value mutation): изменение значения параметра или константы;
- **Мутация операторов** (Statement mutation): реализуется путем редактирования, удаления или перестановки оператора;
- **Мутация решения** (Decision Mutation): изменение логических, арифметических и реляционных операторов;

В зависимости от результата теста мутанты делятся на:

- **Выжившие мутанты** (Survived Mutants): мутанты, которые все еще живы, то есть не обнаруживаются при выполнении теста. Их также называют live mutants;
- **Убитые мутанты** (Killed Mutants): мутанты, обнаруженные тестами;
- **Эквивалентные мутанты** (Equivalent Mutants): мутанты, которые изменив части кода не привели к какому-либо фактическому изменению в выводе программы, т.е. они эквивалентны исходному коду;
- **Нет покрытия** (No coverage): в этом случае мутант выжил, потому что для этого мутанта не проводились тесты. Этот мутант находится в части кода, не затронутой ни одним из ваших тестов. Это означает, что наш тестовый пример не смог его охватить;
- **Тайм-аут** (Timeout): выполнение тестов с этим активным мутантом привело к тайм-ауту. Например, мутант привел к бесконечному циклу в вашем коде. Не обращайте внимание на этого мутанта. Он считается «обнаруженным». Логика здесь в том, что если этот мутант будет внедрен в ваш код, ваша CI-сборка обнаружит его, потому что тесты никогда не завершатся;

- Ошибка выполнения (Runtime error): выполнение тестов привело к ошибке (а не к провалу теста). Это может произойти, когда средство запуска тестов не работает. Например, когда средство выполнения теста выдает ошибку OutOfMemoryError или для динамических языков, когда мутант привел к неразборчивому коду. Не тратьте слишком много внимания на этого мутанта. Он не отображается в вашей оценке мутации;
- Ошибка компиляции (Compile error): это состояние возникает, когда это компилируемый язык. Мутант привел к ошибке компиляции. Он не отражается в оценке мутации, поэтому вам не нужно уделять слишком много внимания изучению этого мутанта;
- Игнорируется (Ignored): мы можем видеть это состояние, когда пользователь устанавливает конфигурации для его игнорирования. Он будет отображаться в отчетах, но не повлияет на оценку мутации;
- Тривиальные мутанты (Trivial Mutants): фактически ничего не делают. Любой тестовый пример может убить этих мутантов. Если в конце тестирования остались тестовые примеры, значит, это недопустимый мутант (invalid mutant);

Метрики:

- Обнаруженные (Detected): это количество мутантов, обнаруженных нашим тестом, то есть убитых мутантов. $\text{Detected} = \text{Killed mutants} + \text{Timeout}$;
- Необнаруженные (Undetected): это количество мутантов, которые не были обнаружены нашим тестом, то есть выживших мутантов. $\text{Undetected} = \text{Survived mutants} + \text{No Coverage}$;
- Покрытые (Covered): это количество мутантов покрытых тестами. $\text{Covered} = \text{Detected mutants} + \text{Survived mutants}$;
- Действительные (Valid): это количество действительных мутантов, не вызвавших ошибки компиляции или рантайма. $\text{Valid} = \text{Detected mutants} + \text{Undetected mutants}$;
- Недействительные (Invalid): это количество всех недействительных мутантов, т.е. они не могли быть протестированы, так как вызывали ошибку компиляции или выполнения. $\text{Invalid} = \text{Runtime errors} + \text{Compile errors}$;
- Всего мутантов (Total mutants): содержит всех мутантов. $\text{Total} = \text{Valid} + \text{Invalid} + \text{Ignored}$;
- Оценка мутации на основе покрытого кода (Mutation score based on covered code): оценивает общий процент убитых мутантов на основе покрытия кода. $\text{Mutation score based on covered code} = \text{Detected} / \text{Covered} * 100$;
- Неправильный синтаксис (Incorrect syntax): их называют мертворожденными мутантами, это представлено как синтаксическая ошибка. Обычно эти ошибки должен обнаруживать компилятор;
- **Оценка мутации (Mutation score)**: это оценка, основанная на количестве мутантов. В идеале равна 1 (100%). $\text{Mutation score} = \text{Detected} / \text{Valid} * 100 (\%)$;

Источники:

- [Мутационное тестирование](#)
- [Mutation Testing Guide: What You Should Know](#)

Доп. материал:

- [Mutation testing](#)
- [What Is Mutation Testing: Tutorial With Examples](#)

Разница тестирования ПО и железа (Software Vs. Hardware testing)

Программное обеспечение (Software) - это управляющий набор инструкций и данных. В информатике и разработке программного обеспечения программное обеспечение - это вся информация, обрабатываемая компьютерными системами, включая программы и данные. Программное обеспечение включает программы, библиотеки и связанные с ними неисполняемые данные, такие как онлайн-документация или цифровые носители. Программное обеспечение и оборудование требуют друг друга, и ни одно из них не может реально использоваться по отдельности. На самом низком уровне программирования исполняемый код состоит из инструкций машинного языка, поддерживаемых отдельным процессором - обычно центральным процессором (ЦП) или графическим процессором (ГП). Машинный язык состоит из групп двоичных значений,

обозначающих инструкции процессора, которые изменяют состояние компьютера по сравнению с его предыдущим состоянием.

Аппаратное обеспечение (Hardware) - это физическое электронное устройство, промышленное оборудование или компонент компьютера. Примерами оборудования компьютера являются CPU, МВ, устройства ввода/вывода и хранения данных (HDD или SSD). Без оборудования программному обеспечению не на чем будет работать. Аппаратное и программное обеспечение взаимодействуют друг с другом, и программное обеспечение сообщает аппаратному обеспечению, какие задачи оно должно выполнять.

Разница в тестировании ПО и железа:

В глобальном смысле нам не сильно важна природа объекта тестирования, т.к. принципы особо не меняются. Однако, как говорится, есть нюансы:

- “Железный” баг может легко физически безвозвратно сломать единственный рабочий прототип. Критерии отбора тестов должны учитывать и анализ рисков;
- Само взаимодействие с железом, как и снятие показателей результатов / метрик может потребовать дополнительное оборудование, физические модификации, а также почти наверняка инженерный ум и прямые руки из нужного места;
- Тесты программного обеспечения в отличии от тестов железа виртуальны, их можно копировать и переиспользовать и прогонять сколько потребуется. ПО можно легко изменить и развить с помощью нескольких релизов, в то время как оборудование требует более высоких затрат на изменение и не может быть подвергнуто рефакторингу после производства. Тестирующие должны понимать, что когда оборудование создается, они не могут добавлять к нему новые возможности. Таким образом, тесты подходят только для этой линейки оборудования, и для следующего продукта необходимо будет создать другой набор критериев оценки. Конструкции оборудования также значительно более ограничены из-за конкретных деталей или отраслевых рекомендаций;
- В результате первого различия существуют в тест-кейсах. При использовании кейсов для ПО для выполнения всех запланированных тестов может потребоваться от 50 до 100 шагов, это результат того, что нужно учесть бесчисленное множество вещей и высокого уровня автоматизации тестирования, связанного с гибкой разработкой программного обеспечения. Команды могут использовать инструменты тестирования качества, чтобы отслеживать эти операции и гарантировать, что все идет так, как ожидалось. С другой стороны, этапы тестирования оборудования намного короче и проще и включают всего несколько этапов, чтобы проверить, работает ли продукт. Во-первых, прошивка может быть проверена на исправность. Затем оборудование оценивается, чтобы убедиться, что оно хорошо интегрируется с другими системами и должным образом работает с необходимыми приложениями и операционными системами. Наконец, вся система оценивается по тому, насколько хорошо она соответствует требованиям заказчика и высокоуровневым спецификациям, таким как соответствие (compliance). Аппаратное обеспечение нельзя сильно менять перед выпуском, поэтому важно, чтобы группы выполняли полные тесты для выявления любых слабых мест, которые могут быть присущи продукту.
- Что касается программного обеспечения, управляющего оборудованием, здесь резко может возрасти необходимый технический уровень для тестирования (низкоуровневое понимание работы железа, связки с прошивкой, программирование микроконтроллеров, ассемблер и вот это всё):
 - Embedded operating systems;
 - Device driver and service testing;
 - Multi threaded communications;
 - Synchronising data from multiple external sources;
 - Using data scopes to isolate errors (Hardware vs Software);

Источники:

- [Difference between Hardware and Software](#)
- [The Difference between Software Testing and Hardware Testing](#)
- [software vs hardware testing](#)

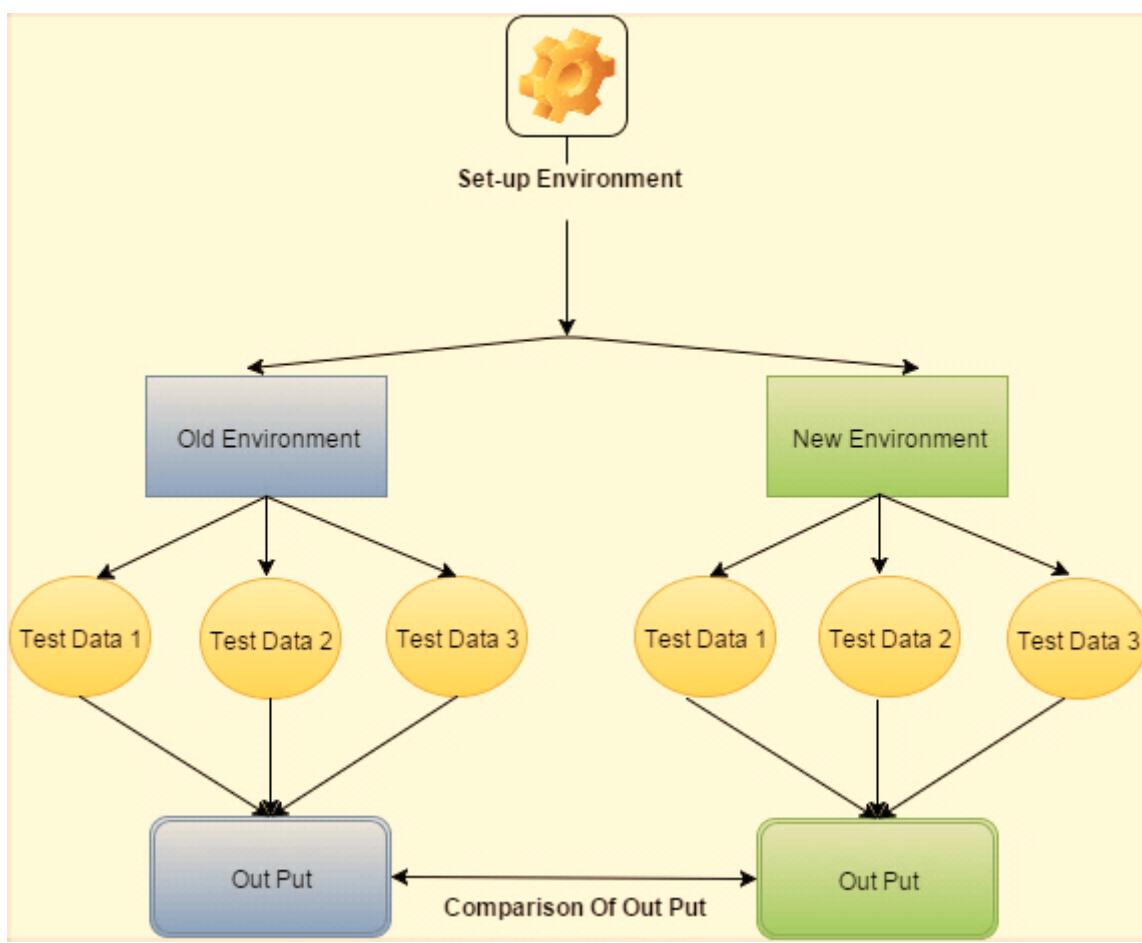
Доп. материал:

- [Hardware vs Software Product Testing](#)
- [Siemens hardware and software testing solutions](#)

Параллельное тестирование (Parallel testing)

Обычно большинство команд тестирования выполняют свои тесты по одному. Этот процесс называется последовательным тестированием или серийным тестированием (sequential testing or serial testing). В процессе последовательного тестирования каждый тестовый пример запускается один за другим и следующий тест не начинается, пока не завершится предыдущий. Последовательный процесс тестирования требует много времени, усилий и ресурсов. Чтобы выпускать качественные продукты в кратчайшие сроки, более эффективным будет параллельное тестирование.

Параллельное тестирование включает в себя в основном автоматизированное тестирование нескольких версий одного и того же приложения или разных компонентов приложения одновременно и с одинаковыми входными данными, чтобы сократить общее время выполнения теста путем интеграции фреймворка автоматизации с облачными решениями и виртуализацией.



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».

Источники:

- [Parallel Testing Guide: How To Perform Parallel Testing](#)
- [What is Parallel Testing? Definition, Approach, Example](#)

Доп. материал:

- [Parallel testing: get feedback earlier, release faster](#)

- [What Is Parallel Testing And Why Is It Important?](#)

Тестирование качества данных (Data Quality Testing)

Сегодняшний мир переживает очередную технологическую революцию, одним из аспектов которой является использование всевозможными компаниями накопленных данных для раскрутки собственного маховика продаж, прибылей и пиара. Представляется, что именно наличие хороших (качественных) данных, а также умелых мозгов, которые смогут из них делать деньги (правильно обработать, визуализировать, построить модели машинного обучения и т. п.), стали сегодня ключом к успеху для многих. Естественно, это потребовало технической организации этого процесса — подсоединиться к источнику данных, выкачать их, проверить, что они загружены в полном объёме и т. п. Количество таких процессов стало расти, и на сегодняшний день мы получили огромную потребность в Data Quality инженерах - тех, кто следил бы за потоком данных в системе (data pipelines), за качеством данных на входе и на выходе, делал бы выводы об их достаточности, целостности и прочих характеристиках. Обязанности Data Quality инженера не ограничиваются только рутинными ручными/автоматическими проверками на «nulls, count и sums» в таблицах БД, а требуют глубокого понимания бизнес нужд заказчика и, соответственно, способностей трансформировать имеющиеся данные в пригодную бизнес-информацию.

Data Quality — один из этапов Data Management и первый пункт плана управления данными ([data management plan](#)).

Аспекты качественных данных ([dimensions](#)):

- Точность (Accuracy): насколько хорошо информация отражает реальность?
- Полнота (Completeness). Соответствует ли вашим ожиданиям от того, что является всеобъемлющим?
- Согласованность (Consistency): соответствует ли информация, хранящаяся в одном месте, релевантным данным, хранящимся в другом месте?
- Своевременность (Timeliness): доступна ли ваша информация тогда, когда она вам нужна?
- Срок действия, соответствие (Validity aka Conformity): имеет ли информация определенный формат, тип или размер? Соответствует ли она бизнес-правилам / передовой практике?
- Целостность (Integrity): можно ли правильно объединить разные наборы данных, чтобы отразить общую картину? Хорошо ли определены и реализованы отношения?

Сам процесс тестирования не подразумевает строгое копирование этих признаков в тест-кейсы и их проверку. В Data Quality, как и в любом другом виде тестирования, необходимо прежде всего отталкиваться от требований по качеству данных, согласованных с участниками проекта, принимающими бизнес-решения.

Пример самого обобщенного перечня активностей Data Quality инженера:

- Подготовить тестовые данные (валидные\невалидные\большие\маленькие) через автоматизированный инструмент.
- Загрузить подготовленный набор данных в исходный источник и проверить его готовность к использованию.
- Запустить ETL-процессы по обработке набора данных из исходного хранилища в окончательное или промежуточное с применением определённого набора настроек (в случае возможности задать конфигурируемые параметры для ETL-задачи).
- Верифицировать обработанные ETL-процессом данные на предмет их качества и соответствие бизнес-требованиям.

При этом основной акцент проверок должен приходиться не только на то, что поток данных в системе в принципе отработал и дошёл до конца (что является частью функционального тестирования), а по большей части на проверку и валидацию данных на предмет соответствия ожидаемым требованиям, выявления аномалий и прочего.

Источники + доп. материал:

- [Тестировщик больших и маленьких данных: тренды, теория, моя история](#)
- [Data Quality Testing: Ways to Test Data Validity and Accuracy](#)

- [Data Quality Testing – A Quick Checklist to Measure and Improve Data Quality](#)

Подкожный тест (Subcutaneous test)

Впервые упоминание встречается в 2010 году у Jimmy Bougard в [блоге](#): "В то время как модульный тест фокусируется на мелкомасштабном дизайне, подкожный тест не касается дизайна, а вместо этого проверяет основные входы и выходы системы в целом", затем в докладе Matt Davies and Rob Moore - "[Microtesting - How We Set Fire To The Testing Pyramid While Ensuring Confidence](#)", а позже и в презентации Daniel Lafeir and Michael Lawrie. Термин «подкожный» означает автоматизированный тест непосредственно под слоем пользовательского интерфейса. В приложении MVC это будут тесты для всего, что находится непосредственно под контроллером. Для веб-службы все, что находится ниже ендпоинта (endpoint).

Идея состоит в том, что самый верхний уровень в приложении не выполняет никакой реальной бизнес-логики, а просто соединяет внешние интерфейсы с базовыми службами. Как подчеркивает Martin Fowler [в своем сообщении о подкожном тестировании](#), такие тесты особенно полезны при попытке выполнить функциональные тесты, когда вы хотите реализовать сквозной сценарий, избегая при этом некоторых трудностей, связанных с тестированием через сам пользовательский интерфейс:

- UI-тесты медленные. От этого никуда не деться. Их можно запускать параллельно, допиливать напильником и делать чуть-чуть быстрее, но они останутся медленными;
- UI-тесты нестабильные. Отчасти потому, что они медленные. А еще потому, что Web-браузер и интерфейс пользователя не были созданы для того, чтобы ими управлял компьютер (в настоящее время данный тренд меняется, но не факт, что это хорошо);
- UI-тесты — это наиболее сложные тесты в написании и поддержке. Они просто тестируют слишком много. (Это усиливается тем фактом, что, зачастую, люди берут «ручные» тест-кейсы и начинают их автоматизировать как есть, без учета разницы в ручном и автоматическом тестировании);
- Нам говорят, что, якобы, UI-тесты эмулируют реального пользователя. Это не так. Пользователь не будет искать элемент на странице по ID или XPath локатору. Пользователь не заполняет форму со скоростью света, и не «упадет» если какой-то элемент страницы не будет доступен в какую-то конкретную миллисекунду. И даже теперь, когда браузеры разрабатываются с учетом того, что браузером можно программно управлять — это всего-лишь эмуляция, даже если очень хорошая;
- Кто-то скажет, что некоторый функционал просто нельзя протестировать иначе. Я скажу, что если есть функционал, который можно протестировать только UI тестами (за исключением самой UI логики) — это может быть хорошим признаком архитектурных проблем в продукте.

Таким образом определение можно сформулировать так: "Если модульный тест тестирует наименьший тестируемый компонент кода приложения, то подкожный тест представляет собой единый рабочий процесс (workflow), который можно идентифицировать и протестировать в коде приложения. Подкожное тестирование рассматривает рабочий процесс как тестируемую единицу".

Стоит помнить, что это не прямая замена автоматизации тестирования через UI, а просто более целенаправленное тестирование функциональности на нужном уровне приложения. Это позволяет команде создавать более совершенные сквозные тесты с использованием существующего кода, фреймворка и / или библиотек, доступных для внешнего приложения. Основная цель этого вида тестирования - уменьшить нестабильность теста и сосредоточиться на функциональности. Это позволяет группе различать функциональные сбои из-за проблем с кодом и сбои приложений из-за проблем совместимости или проблем с внешними зависимостями. Поскольку подкожные тесты можно запускать с той же тестовой средой, что и модульные тесты, они не имеют доступа к внутреннему коду или вызовам API во время работы. Изоляция этих тестов и использование имитирующих инструментов, которые могут имитировать вызовы API и заглушки для различных взаимодействий служб, могут дать целенаправленную, изолированную оценку функциональности во внешнем стеке. Такое тестирование функциональности позволяет сделать любую автоматизацию через браузер и пользовательский интерфейс более легкими. Это означает, что традиционные тесты пользовательского интерфейса могут быть очень минимальными и проверять только то, что связи работают между уровнями приложения. Вместо использования дорогостоящих тестов графического интерфейса и пользовательского интерфейса для получения информации о функциональности они могут быть поверхностными проверками работоспособности или дымовыми тестами. Избегая использования

автоматизации пользовательского интерфейса, утверждающей ожидание чего-то функционально работающего, тесты могут утверждать, правильно ли приложение обменивается данными на своих уровнях интеграции.

Поскольку при подкожном тестировании используются все компоненты на странице, создание интегрированных тестов, которые тестируют только несколько компонентов, или тестирование одного компонента на уровне модуля может не потребоваться. Модульные тесты могут быть нацелены на более сложную логику, а не пытаться протестировать всю логику вокруг одного компонента, тестирование базовой логики облегчает нагрузку на модульное тестирование. Подкожное тестирование использует ту же структуру тестирования (например, Jest), что и модульные тесты. Это сохраняет функциональные тесты внутренними и ближе к коду, что дает команде больше шансов на более быструю обратную связь и более быструю настройку теста, чем тестовая среда, поддерживаемая отдельно. Это означает, что командам не нужно выполнять дополнительную работу по сопровождению нескольких фреймворков, репозиториев, а иногда и языков для выполнения функционального тестирования пользовательского интерфейса. Теперь, когда подкожное тестирование позволяет проводить функциональное тестирование кода, а не через пользовательский интерфейс, любые тесты пользовательского интерфейса могут быть сокращены до небольшой части того, что было необходимо ранее. UI-тесты можно использовать как дымовые тесты. Таким образом они могут доказать, что приложение и все его уровни находятся в работоспособном состоянии связи.

Поскольку подкожные тесты ориентированы на поведение высокого уровня (high-level behavior), а не на дизайн, они идеально подходят для стратегий тестирования на основе сценариев (scenario-based testing strategies), таких как BDD или паттерн [Testcase Class per Fixture](#).

К сожалению, наряду со всеми плюсами subcutaneous подхода мы можем получить и снижение покрытия (coverage), в частности glue code ([Связующий код](#) — программный код, который служит исключительно для «склеивания» разных частей кода, и при этом не реализует сам по себе никакую иную прикладную функцию). Насколько важна\существенна потеря покрытия в данном случае? Зависит от ситуации. Мы потеряли немного glue code, который может быть (а может и не быть) важным (рекомендую в качестве упражнения определить, какой код потерялся). Оправдывает ли данная потеря покрытия введения тяжеловесного тестирования на уровне UI? Это тоже зависит от ситуации. Мы можем, например:

- Добавить один UI-тест для проверки glue code, или
- Если мы не ожидаем частых изменений glue code — оставить его без автотестов, или
- Если у нас есть какой-то объем «ручного» тестирования — есть отличный шанс, что проблемы с glue code будут замечены тестировщиком, или
- Придумать что-то еще (тот же канареичный релиз, Canary deployment)

Один из способов избежать потери покрытия - "[Feature Tests Model](#)"

Источники:

- [Introduction To Subcutaneous Testing](#)
- [Пишем автотесты эффективно — Subcutaneous tests](#) + [англ. версия](#) + [видеоверсия](#)
- [Subcutaneous Testing in ASP.NET Core](#)

Тест дизайн

Тест-дизайн и техники тест-дизайна (Test Design and Software Testing Techniques)

Тест-дизайн - важный этап STLC, а именно деятельность по получению и определению тестовых примеров из условий тестирования (test conditions), где условие тестирования - это «аспект основы тестирования (test basis), который важен для достижения конкретных целей тестирования», а основа тестирования - это «совокупность знаний, используемая в качестве основы для анализа и дизайна тестов (test analysis and test design)». Проще говоря, цель тест-дизайна - создать максимально эффективный набор кейсов, покрывающий наиболее важные аспекты тестируемого ПО, т.е. минимизировать количество тестов, необходимых для нахождения большинства серьезных ошибок.

Одним из наиболее важных аспектов теста является то, что он проверяет, выполняет ли система то, что она должна делать. Copeland говорит: “По сути, тестирование - это процесс сравнения того, что есть с тем, что должно быть”. Если мы просто введем какие-то данные и подумаем, что это было весело, я предполагаю, что с системой, вероятно, все в порядке, потому что она не крашнулась, но действительно ли мы ее тестируем? Beizer называет это «детским тестированием» (kiddie testing). Мы можем не знать каждый раз, какой правильный ответ в деталях, и иногда мы все равно можем получить некоторую выгоду от этого подхода, но на самом деле это не проверка. Чтобы знать, что система должна делать, нам нужен источник информации о правильном поведении системы - это называется «оракул» или тестовый оракул (test oracle). После того, как заданное входное значение было выбрано, тестирующему необходимо определить, каким будет ожидаемый результат ввода этого входа, и задокументировать его как часть тестового примера.

Давайте проясним. Требования или пользовательские истории с критериями приемлемости (формы test basis) определяют, что вы должны тестировать (test objects and test conditions), и исходя из этого, вы должны выяснить способ тестирования, то есть спроектировать тестовые примеры. Один из наиболее важных вопросов заключается в следующем: какие факторы влияют на успешный дизайн теста? Если вы читаете разные блоги, статьи или книги, вы найдете примерно следующее:

- Время и бюджет, доступные для тестирования;
- Соответствующие знания и опыт вовлеченных людей;
- Определен целевой уровень покрытия (измерение уровня достоверности (measuring the confidence level));
- Способ организации процесса разработки программного обеспечения (например, водопад или гибкая разработка);
- Устанавливается соотношение методов создания тестов (например, ручных и автоматических);

Это неправда! Без достаточного времени и бюджета вы, вероятно, вообще не начнете ни одного проекта. Если у вас нет квалифицированных специалистов по тестированию программного обеспечения, включая дизайн тестов, то, вероятно, вы тоже не начнете проект. Однако, хороший дизайн теста включает три предварительных условия:

- Полная спецификация (Complete specification)(test bases);
- Анализ рисков и сложности (Risk and complexity analysis);
- Исторические данные ваших предыдущих разработок;

Требуются некоторые пояснения. Полная спецификация не означает безошибочную спецификацию, так как во время разработки теста можно найти и исправить множество проблем (предотвращение дефектов). Это только означает, что у нас есть все необходимые требования или в Agile разработке у нас есть все эпики, темы и пользовательские истории с критериями приемлемости (acceptance criteria). Существует минимальная ценность в одновременном рассмотрении затрат на тестирование и затрат на исправление дефектов, и цель хорошего тест-дизайна - выбрать подходящие методы тестирования, приближающиеся к этому минимуму. Это можно сделать, проанализировав сложность, риски и используя исторические данные. Таким образом, анализ рисков неизбежен для определения тщательности тестирования. Чем выше риск использования функции / объекта, тем более тщательное тестирование необходимо. То же самое можно сказать и о

сложности кода. Для более рискованного или сложного кода мы должны сначала применить больше НЕкомбинаторных методов проектирования тестов вместо одного чисто комбинаторного.

Наше другое и правильное представление о дизайне тестирования состоит в том, что если у вас есть соответствующая спецификация (тестовая база) и надежный анализ рисков и сложности, то, зная ваши исторические данные, вы можете выполнить дизайн теста оптимальным образом. Вначале у вас нет исторических данных, и вы, вероятно, не достигнете оптимума. Нет проблем, давайте сделаем предварительную оценку. Например, если риск и сложность низкие, используйте только исследовательское тестирование. Если они немного выше, используйте исследовательское тестирование и простые методы, основанные на спецификациях, такие как классы эквивалентности с анализом граничных значений. Если риск высок, вы можете использовать исследовательское тестирование, комбинационное тестирование, предотвращение дефектов, статический анализ и обзоры (reviews).

Еще одно важное замечание. Критерии выбора тестов и адекватности тестовых данных различны. Первый - неотъемлемая часть любой техники тест-дизайна. Второй проверяет набор тестов. В результате процесса разработки тестов создаются независимые от реализации тестовые примеры, которые проверяют требования или пользовательские истории. Напротив, тесты, которые создаются на основе отсутствия покрытия по выбранным критериям адекватности тестовых данных, подтверждают проблемы, зависящие от реализации; однако это НЕ дизайн теста, это создание теста. Очень важно использовать метод «сначала тестирование» (test-first method), т. е. дизайн теста должен быть отправной точкой разработки. Дизайн тестов также очень эффективен для предотвращения дефектов, если он применяется до внедрения.

Итак, хороший **процесс тест-дизайна** выглядит так:

- Сбор информации, чтобы понять требования пользователей;
- Получение всех важных бизнес-сценариев;
- Создание тестовых сценариев для каждого производного критически важного бизнес-сценария;
- Назначение всех запланированных тестовых сценариев различным тестовым случаям;

Затем вам нужно будет выбрать технику тест-дизайна для каждого требования. На этом этапе, если все реализовано правильно, вы можете внести значительные изменения, которые чрезвычайно повлияют на ваш ROI.

Роли, ответственные за тест дизайн:

- **Тест аналитик** (test analyst) - определяет "ЧТО тестировать?":
 - Исследует продукт:
 - Понимание цели создания продукта;
 - Какими способами цель должна достигаться;
 - Какие и основные и вспомогательные возможности предоставляет продукт пользователям;
 - Оценка, правильно ли понял разработчик заказчика.
 - Составляет логическую карту продукта: Интеллект - карта - это техника представления любого процесса, события, мысли или идеи в систематизированной визуальной форме;
 - Разбивает программный продукт на основные части:
 - Система расчленяется только по одному, постоянному для всех уровней признаку (Они должны отвечать на один и тот же вопрос, по отношению к своему родителю);
 - Вычисленные подсистемы должны взаимно исключать друг друга, а в сумме - характеризовать систему;
 - На каждом уровне рекомендуется использовать не более 7 подсистем;
 - Расставляет приоритеты для тестирования:
 - Требования клиента;
 - Степень риска;
 - Сложность системы;
 - Временные ограничения;
- **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 выделяются, а доверяется обычным тестировщикам, что не всегда положительно оказывается на качестве тестов, тестировании и, как из этого следует, на качестве ПО (конечного продукта).

Техники тест-дизайна (Software testing techniques)

- Статические (Static):
 - Reviews:
 - 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, Structure-Based)
 - 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;
 - Attack Testing;
 - Черный ящик (Black-box, Specification-based):
 - Эквивалентное Разделение (Equivalence Partitioning - EP)
 - Анализ Границых Значений (Boundary Value Analysis - BVA)
 - Комбинаторные техники (Combinatorial Test Techniques)
 - Переходы между состояниями (State transition)
 - Случай использования (Use case testing)
 - Domain testing
 - Decision Table Testing
 - Classification Tree Method
 - State Transition Testing
 - Cause-Effect Graphing
 - Scenario Testing
 - Random Testing
 - Syntax Testing
 - Check List Based Testing
 - Risk-Based Testing
 - User Journey Test

Источники:

- [What is Test design? or How to specify test cases?](#)
- [What Is Test Design Actually?](#)
- [Максим Дорофеев - Lecture 12. Test Design](#)

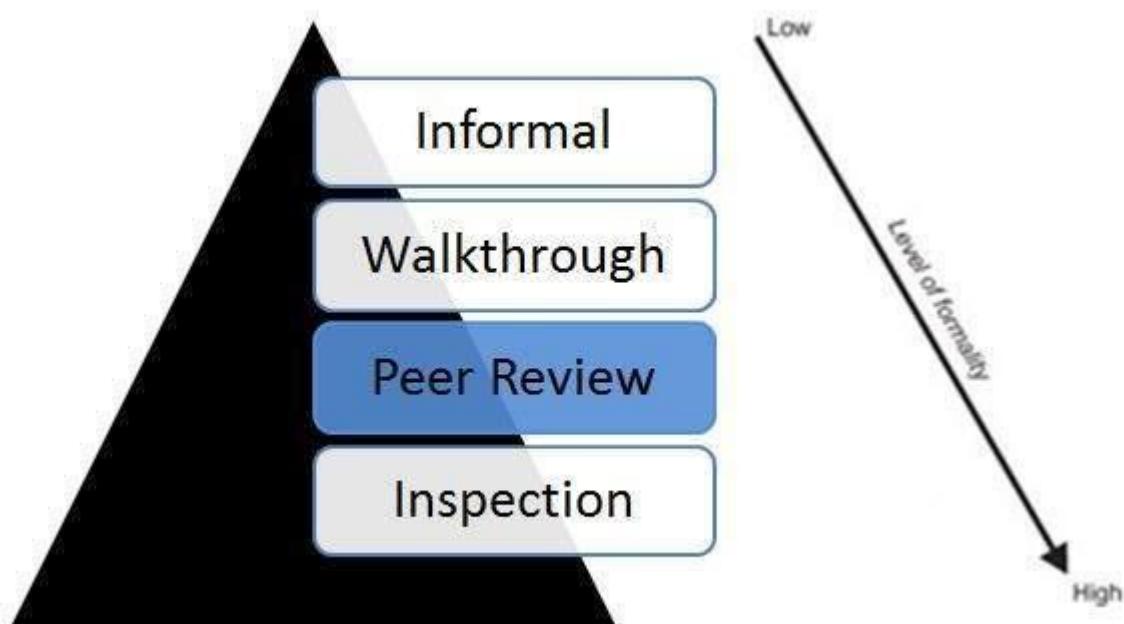
Доп. материал:

- [Copeland Lee - "A Practitioner's Guide to Software Test Design"](#) + [перевод](#)
- [Rikard Edgren - The little black book on test design](#) + [перевод](#)
- [Test Design: A Survey of Black Box Software Testing Techniques](#) (видеолекции + доп.материалы)
- Борис Бейзер - "Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем"
- [Hillel - Техники тест-дизайна](#) (доклад с примерами)
- [Лекция 3: Критерии выбора тестов](#)
- [NIXMultiConf: Игорь Ямшанов - Приоритизация: начни с самого важного](#)
- [Armando1514/Software-testing-techniques](#)
- [The ABCs of Acceptance Test Design](#)

Static -> Reviews

Методы статического тестирования делятся на две основные категории, одной из которых являются ревью.

Ранжирование по уровню формальности:



Экспертные обзоры (Peer Reviews): Рецензирование - это стандартизованный метод проверки правильности исходного кода (методом белого ящика) при разработке программного обеспечения, который проводится для выявления дефектов на ранних этапах жизненного цикла и которые не могут быть обнаружены с помощью методов тестирования черного ящика.

Прохождение/просмотр/пошаговый разбор (walkthrough): это метод проведения неформального группового / индивидуального просмотра. В walkthrough автор описывает и объясняет рабочий продукт на неформальной встрече своим коллегам или руководителю, чтобы получить обратную связь. Здесь проверяется применимость предложенного решения для рабочего продукта. Либо рабочий продукт проверяется на наличие дефектов несколькими лицами, кроме человека, который его фактически произвел;

Технический обзор (Technical Review): Это метод более высокого уровня по сравнению с inspection или walkthrough, поскольку он также включает в себя управление. Этот метод используется для оценки (assess and evaluate) продукта путем проверки его соответствия стандартам разработки, руководствам и спецификациям. У него нет определенного процесса, и большая часть работы выполняется модератором, как описано ниже:

- Модератор собирает и раздает материал и документацию всем членам команды;
- Модератор также готовит набор показателей для оценки продукта в соответствии со спецификациями и уже установленными стандартами и гайдлайнами:
 - последовательность;

- документация;
- соблюдение стандартов;
- полнота;
- определение проблемы и требования (? problem definition and requirements);
- Результаты фиксируются в документе, который включает как дефекты, так и предложения;
- Наконец, устраняются дефекты и учитываются предложения по улучшению продукта;

Инспекция: Инспекция определяется как наиболее формальная, тщательная, глубокая групповая проверка, направленная на выявление проблем как можно ближе к их исходной точке. Процесс проверки выполняется на ранних этапах SDLC и применяется к определенной части продукта, такой как SRS, код, дизайн продукта. и т. д. Это включает в себя ручное изучение различных компонентов продукта на более ранних этапах.

Инспекционная деятельность следует определенному процессу, и участники играют четко определенные роли. Инспекционная группа состоит из трех-восьми человек, которые играют роли модератора, автора, читателя, записывающего и инспектора. Например, разработчик может выступать в качестве инспектора во время проверки кода, в то время как представитель по обеспечению качества может действовать как исполнитель стандартов.

Software inspection process:

- Планирование встречи: на этом этапе основное внимание уделяется определению продукта, подлежащего инспекции, и цели этой инспекции. На этом этапе назначается модератор, который управляет всем процессом. Назначенный модератор проверяет, готов продукт к инспекции или нет. Модератор также выбирает инспекционную группу и назначает им их роли. Модератор также планирует инспекционную встречу и раздает необходимые материалы инспекционной группе;
- Обзор: на этом этапе инспекционной группе предоставляется вся справочная информация для инспекционного совещания. Автор, который является программистом или дизайнером, ответственным за разработку продукта, представляет свою логику и рассуждения о продукте, включая функции продукта, его предполагаемое назначение и подход или концепцию, использованные при его разработке. Удостоверяется, что каждый член инспекционной группы понял и знаком с задачами и целью инспекционного совещания, которое должно быть проведено;
- Индивидуальная подготовка участников: на этом этапе члены инспекционной группы индивидуально готовятся к инспекционной встрече, изучая материалы, предоставленные на более ранних этапах. Члены команды выявляют потенциальные ошибки или недочеты в продукте и записывают их в журнал. Журнал передается модератору. Затем модератор собирает все журналы, полученные от участников, и отправляет их автору. Инспектор - лицо, ответственное за проверку и выявление ошибок и несоответствий в документах или программах, проверяет продукт и записывает все обнаруженные в нем проблемы (как общие, так и специфические). Инспектор записывает проблемы или issues в журнал вместе со временем, затраченным на подготовку. Модератор просматривает логи, чтобы проверить, готова ли команда к инспекционной встрече или нет. Наконец, модератор отправляет автору все скомпилированные логи;
- Инспекционная встреча (Inspection Meeting): на этом этапе автор обсуждает вопросы, поднятые членами команды в скомпилиированном журнале. Участники приходят к решению, является ли поднятый вопрос ошибкой или нет. Модератор завершает встречу и подводит итоги встречи - это список ошибок, обнаруженных в продукте, которые должен устранить автор.
- Переделка: доработка проводится автором согласно сводному списку, представленному модератором на предыдущем этапе. Автор исправляет все ошибки и сообщает модератору;
- Follow – up: модератор проверяет, все ли ошибки устраниены или нет. Затем модератор готовит отчет. Если все ошибки исправлены и устраниены, модератор выпускает документ. В противном случае в отчет добавляются нерешенные вопросы и назначается еще одно инспекционное собрание;

REVIEWS

| | Informal | Walkthrough | Technical review | Inspection |
|-------------------------|------------------|------------------|-------------------|-------------------|
| Documented | no | yes | yes | yes |
| Led by author | yes | yes | no | no |
| Use checklists | no | optional | optional | yes |
| Moderator (facilitator) | no | yes | ideally trained | trained |
| Scribe | no | mandatory | mandatory | mandatory |
| Individual preparation | no | optional | mandatory | mandatory |
| Reviewers | colleague, buddy | different people | technical experts | different experts |
| Potential defects logs | may be | may be | mandatory | mandatory |

Источники:

- [Software Testing Techniques](#)
- [Types of Static Testing](#)
- [Explain peer review](#)
- [Static Testing Techniques](#)

Static -> Static Analysis

Статический анализ - это анализ программных артефактов, таких как программный код (или требования, дизайн), выполняемый статически, т.е. без запуска и, очевидно, методом белого ящика. Основная цель этого анализа - как можно раньше найти ошибки, независимо от того, могут ли они вызывать отказы (failures). Как и в случае с обзорами (reviews), статический анализ обнаруживает ошибки (bugs), а не отказы. Обычно статический анализ проводят до формальной проверки, даже до unit testing, путём добавления этих проверок специалистами DevOps в пайплайн проекта. Статический анализ не связан с динамическими свойствами требований, дизайна и кода, такими как покрытие тестами (test coverage). Существует множество инструментов для статического анализа, которые в основном используются разработчиками до или во время тестирования компонентов или интеграции (чаще новые и измененные классы и функции), а также дизайнерами во время моделирования программного обеспечения. Инструменты могут отображать не только структурные атрибуты, такие как глубина вложенности или число цикломатической сложности и проверка на соответствие стандартам кодирования, но также графические изображения потока управления, взаимосвязи данных и количество отдельных путей от одной строки кода к другой. Информация может использоваться вплоть до формальных методов, которые математически подтверждают свойства данной программы.

Инструменты помогают в выявлении следующих дефектов:

- Неиспользуемые переменные;
- Части кода, которые никогда не выполняются;
- Бесконечные циклы;
- Переменная с неопределенным значением;
- Неправильный синтаксис;
- Несогласованные интерфейсы между модулями и компонентами, такие как неправильное использование объекта, метода или функции, включая неправильные параметры;

- Уязвимости безопасности, такие как проблемы безопасности, связанные с переполнением буфера, возникающим из-за невозможности проверить длину буфера перед копированием в буфер;
- Различные типы нарушения стандартов программирования, как нарушения, создающие риск фактического сбоя, так и нарушения, которые усложняют тестирование, анализ и поддерживаемость кода;

Методы статического анализа:

- **Анализ управления (Control Analysis):** фокусируется на изучении элементов управления, используемых в структуре вызовов, анализе потока управления и анализе переходов состояний (calling structure, control flow analysis and state transition analysis). Структура вызова связана с моделью путем идентификации вызовов и их структуры. Вызывающая структура может быть процессом, подпрограммой, функцией или методом. Анализ потока управления проверяет последовательность передачи управления и может выявить неэффективные конструкции в модели. Создается граф модели (CFG - Control Flow Graph), в котором условные ветви и стыки модели представлены узлами. По итогам также можно рассчитать цикломатическую сложность программы. Для анализа потока управления могут быть использованы: Абстрактная интерпретация, Удовлетворение ограничений, Типизация данных;
- **Анализ данных (Data Analysis):** обеспечивает правильную работу с объектами данных, такими как структуры данных и связанные списки. Кроме того, этот метод также обеспечивает правильное использование определенных данных. Анализ данных включает два метода, а именно: зависимость данных и анализ потока данных (data dependency and data flow analysis). Зависимость данных необходима для оценки точности синхронизации между несколькими процессорами. Анализ потока данных проверяет определение и контекст переменных. Виды анализа потока данных:
 - Reaching Definitions;
 - Available Expressions;
 - Constant Propagation;
 - Very Busy Expressions;
 - Live Variables;
 - Use-Definition & Definition-Use;
- Анализ неисправностей / отказов (Fault/Failure Analysis): анализирует неисправности (некорректный компонент) и отказ (некорректное поведение компонента модели) в модели. Этот метод использует описание преобразования ввода-вывода для определения условий, являющихся причиной сбоя. Для определения отказов в определенных условиях проверяется проектная спецификация модели (model design specification);
- Анализ интерфейса (Interface Analysis): проверяет взаимодействующие и распределенные модели для проверки кода (This software verifies and verifies interactive and distribution simulations to check the code). Существует два основных метода анализа интерфейса, и анализ пользовательского интерфейса исследует интерфейсы подмоделей и определяет точность структуры интерфейса. Анализ пользовательского интерфейса исследует модель пользовательского интерфейса и меры предосторожности, предпринимаемые для предотвращения ошибок во время взаимодействия пользователя с моделью. Этот метод также фокусируется на том, насколько точно интерфейс интегрирован в общую модель и симуляцию.

Анализ потока управления (Control Flow Analysis) и анализ потока данных (Data Flow Analysis) взаимозависимы: чтобы получить точные результаты для анализа потока данных, необходимо учитывать поток управления (поскольку порядок операций влияет на возможные значения данных в конкретном месте программы). Чтобы получить точные результаты для анализа потока управления, необходимо учитывать поток данных, поскольку поток динамического управления (решение, принимаемое во время выполнения) зависит от значений данных в конкретных местах программы. Однако эти два анализа преследуют разные цели.

Граф потока управления (Control Flow Graph)

Граф потока управления (CFG) - это графическое представление потока управления или вычислений во время выполнения программ или приложений. Графы потока управления в основном используются в статическом анализе, а также в приложениях-компиляторах, поскольку они могут точно представлять поток внутри программного модуля. Характеристики графа потока управления:

- Граф потока управления процессно-ориентированный (process oriented);
- Граф потока управления показывает все пути, которые можно пройти во время выполнения программы;
- Граф потока управления - это ориентированный граф;
- Рёбра в CFG изображают пути потока управления, а узлы в CFG изображают базовые блоки.

Полное описание возможных элементов графа.

Цикломатическая сложность (Cyclomatic Complexity)

Цикломатическая сложность - это метрика для измерения сложности кода, основанная на графике потока управления. Независимый путь определяется как путь, имеющий хотя бы одно ребро, которое ранее не проходило ни в одном другом пути.

Определение из книги Ли Копланда - "A Practitioner's Guide to Software Test Design", Главы 10:

Цикломатическая сложность - это конечное минимальное количество независимых, нециклических маршрутов (называемых основными маршрутами), которые могут образовывать все возможные линейные пути в программном модуле.

Цикломатическая сложность может быть рассчитана относительно функций, модулей, методов или классов в программе как вручную, так и с помощью автоматизированных инструментов.

Математически цикломатическая сложность структурированной программы определяется с помощью ориентированного графа, узлами которого являются блоки программы, соединенные ребрами, если управление может переходить с одного блока на другой. Тогда сложность определяется как

$$M = E - N + 2P,$$

где:

- M = цикломатическая сложность,
- E = количество ребер в графике,
- N = количество узлов в графике,
- P = количество компонент связности.

В другой формулировке используется график, в котором каждая точка выхода соединена с точкой входа. В этом случае график является сильносвязным, и цикломатическая сложность программы равна цикломатическому числу этого графа (также известному как первое число Бетти), которое определяется как

$$M = E - N + P.$$

Это определение может рассматриваться как вычисление числа линейно независимых циклов, которые существуют в графике, то есть тех циклов, которые не содержат в себе других циклов. Так как каждая точка выхода соединена с точкой входа, то существует по крайней мере один цикл для каждой точки выхода.

Для простой программы, или подпрограммы, или метода P всегда равно 1. Однако цикломатическая сложность может применяться к нескольким таким программам или подпрограммам (например, ко всем методам в классе), в таком случае P равно числу подпрограмм, о которых идет речь, так как каждая подпрограмма может быть представлена как независимая часть графа.

Может быть показано, что цикломатическая сложность любой структурированной программы с только одной точкой входа и одной точкой выхода эквивалентна числу точек ветвления (то есть, операторов `if` или условных циклов), содержащихся в этой программе, плюс один.

Цикломатическая сложность может быть распространена на программу с многочисленными точками выхода; в этом случае она равна

$$\pi - s + 2,$$

где:

- π — число точек ветвления в программе,
- s — число точек выхода.

Применение:

- Ограничение сложности при разработке: одно из первоначально предложенных Маккейбом применений состоит в том, что необходимо ограничивать сложность программ во время их разработки. Он рекомендует, чтобы программистов обязывали вычислять сложность разрабатываемых ими модулей и разделять модули на более мелкие всякий раз, когда цикломатическая сложность этих модулей превысит 10. Эта практика была включена НИСТ-ом в методику структурного тестирования с замечанием, что со временем исходной публикации Маккейба выбор значения 10 получил весомые подтверждения, однако в некоторых случаях может быть целесообразно ослабить ограничение и разрешить модули со сложностью до 15. В данной методике признается, что иногда могут существовать причины для выхода за рамки согласованного лимита. Это сформулировано как рекомендация: «Для каждого модуля следует либо ограничивать цикломатическую сложность до согласованных пределов, либо предоставить письменное объяснение того, почему лимит был превышен»;
- Применение при тестировании программного обеспечения: определение количества тестов, необходимых для полного покрытия кода. Цикломатическая сложность M имеет два свойства, для конкретного модуля:
 - M — оценка сверху для количества тестов, обеспечивающих покрытие условий (точек ветвления);
 - M — оценка снизу для количества маршрутов через граф потока управления и, таким образом, количества тестов для полного покрытия путей.
- В составе других метрик: используется в качестве одного из параметров в индексе удобства сопровождения (англ. maintainability index).

Источники:

- [Types of Static Analysis Methods](#)
- [Software Testing - Static Testing](#)
- [Static program analysis](#)
- [What is Static Analysis](#)
- [Software Engineering - Control Flow Graph \(CFG\)](#)
- [Цикломатическая сложность](#)

Доп. материал:

- [Михаил Моисеев - “Формальные методы обеспечения качества ПО: Введение в статический анализ”](#)
- [Y.N. Srikant - “Control Flow Analysis”](#)
- [Levels in Data Flow Diagrams \(DFD\)](#)
- [Control Flow Graph \(CFG\)](#)
- [Static analysis tools](#)

Dynamic -> White box

Динамическое тестирование методом белого ящика - это стратегия, основанная на внутренних путях, структуре и реализации тестируемого программного обеспечения. Тесты здесь выполняются динамически, т.е. с запуском объекта тестирования и основаны на различных видах покрытии кода (путей исполнения программы).

Глобально основных техник динамического тестирования методом белого ящика всего две:

1. Тестирование потока управления (Control Flow Testing);
2. Тестирование потока данных (Data Flow Testing).

Фактически, это динамическая часть одного цельного тестирования, статическая часть которого - анализ и построение графа, описывается в предыдущей теме про статический анализ, а на этом определяется целевое покрытие (Coverage Target), создаются соответствующие тест-кейсы, тесты исполняются и результаты выполнения тестов анализируются.

1. Уровни тестового покрытия в тестировании потока управления

Под "покрытием" имеется в виду отношение объема кода, который уже был проверен, к объему, который осталось проверить. В тестировании потока управления покрытие определяется в виде нескольких различных уровней. Заметим, что эти уровни покрытия представлены не по порядку. Это потому, что в некоторых случаях проще определить более высокий уровень покрытия, а затем определить более низкий уровень покрытия в условиях высокого.

100% покрытие операторов (Statement/node coverage). Оператор (statement) - это сущность языка программирования, обычно являющаяся минимальным неделимым исполняемым блоком (ISTQB). Покрытие операторов - это метод проектирования тестов методом белого ящика, который включает в себя выполнение всех исполняемых операторов (if, for и switch) в исходном коде как минимум один раз. Процентное отношение операторов, исполняемых набором тестов, к их общему количеству является метрикой покрытия операторов. Борис Бейзер написал: "тестирование, меньшее чем это [100% покрытие операторов], для нового программного обеспечения является недобросовестным и должно быть признано преступлением.". Несмотря на то, что это может показаться разумной идеей, на таком уровне покрытия может быть пропущено много дефектов и затруднен анализ покрытия некоторых управляющих структур. Покрытие операторов позволяет найти:

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

100% покрытие альтернатив/ветвей (Decision/branch/all-edges/basis path/DC/C2/ decision-decision-path/edge coverage). «Решение» - это программная точка, в которой control flow имеет два или более альтернативных маршрута (ветви). На этом уровне достаточно такого набора тестов, в котором каждый узел с ветвлением (альтернатива), имеющий TRUE или FALSE на выходе, выполняется как минимум один раз, таким образом, для покрытия по веткам требуется как минимум два тестовых примера. На данном уровне не учитываются логические выражения, значения компонент которых получаются вызовом функций. В отличие от предыдущего уровня покрытия данный метод учитывает покрытие условных операторов с пустыми ветками. Покрытие альтернатив не гарантирует покрытие всех путей, но при этом гарантирует покрытие всех операторов;

Для более полного анализа компонент условий в логических операторах существуют следующие три метода, учитывающих структуру компонент условий и значения, которые они принимают при выполнении тестовых примеров.

100% покрытие условий (Condition/Toggle Coverage). Рассматриваются только выражения с логическими operandами, например, AND, OR, XOR. На этом уровне достаточно такого набора тест-кейсов, в котором каждое условие, имеющее TRUE и FALSE на выходе, выполнено как минимум один раз. Покрытие условий обеспечивает лучшую чувствительность к control flow, чем decision coverage. Для обеспечения полного покрытия по данному методу каждая компонента логического условия в результате выполнения тестовых примеров должна принимать все возможные значения, но при этом не требуется, чтобы само логическое условие принимало все возможные значения, т.е. Condition Coverage не дает гарантии полного decision coverage;

100% покрытие условий + альтернатив (Decision + Condition coverage). На этом уровне тест-кейсы создаются для каждого условия и для каждой альтернативы, т.е. данный метод сочетает требования предыдущих двух методов - для обеспечения полного покрытия необходимо, чтобы как логическое условие, так и каждая его компонента приняла все возможные значения;

100% покрытия множественный условий (Multiple condition coverage). Для выявления неверно заданных логических функций был предложен метод покрытия по всем условиям. При данном методе покрытия должны быть проверены все возможные наборы значений компонент логических условий: условий, альтернатив и условий/альтернатив. Т.е. в случае n компонент потребуется 2^n тестовых примеров, каждый из которых проверяет один набор значений. Тесты, необходимые для полного покрытия по данному методу, дают полную таблицу истинности для логического выражения. Несмотря на очевидную полноту системы тестов, обеспечивающей этот уровень покрытия, данный метод редко применяется на практике в связи с его сложностью и избыточностью. Еще одним недостатком метода является зависимость количества тестовых примеров от структуры логического выражения. Кроме того, покрытие множественных условий не гарантирует покрытие всех путей;

Покрытие бесконечного числа путей. Если, в случае зацикливания, количество путей становится бесконечным, то имеет смысл существенно их сократить, ограничив количество циклов выполнения, что позволит уменьшить количество тестовых случаев. Первый вариант - не выполнять цикл совсем; второй - выполнить цикл один раз; третий - выполнить цикл n раз, где n - это небольшое значение, представляющее символическое количество повторений цикла; четвертый - выполнить цикл m раз, где m - максимальное количество повторений цикла. Кроме того, можно выполнить цикл m-1 и m+1 раз. Перед тем, как начинать тестирование потока управления, должен быть выбран соответствующий уровень покрытия;

100% покрытие путей (Path coverage). Проверяет каждый линейно независимый путь в программе, что означает, что число тестовых примеров будет эквивалентно цикломатической сложности программы. Для кода модулей без циклов количество путей, как правило, достаточно мало, поэтому на самом деле можно построить тест-кейсы для каждого пути. Для модулей с циклами количество путей может быть огромным, что представляет неразрешимую проблему тестирования.

Путь на самом деле является направлением, потоком выполнения, который следует за последовательностью инструкций. Он охватывает функцию от входа до точки выхода. Он охватывает statement, branch/decision coverage. Покрытие пути можно понять в следующих терминах:

- **Loop coverage:** используется для проверки того, что все циклы были выполнены и сколько раз они были выполнены. Цель этого метода покрытия - убедиться, что циклы соответствуют предписанным условиям и не повторяются бесконечно и не завершаются ненормально. Цикл тестирования направлен на мониторинг от начала до конца цикла. Ценным аспектом этой метрики является определение того, выполняются ли циклы while и for более одного раза, т.к. эта информация не сообщается другими метриками;
- **Function coverage:** показывает, вызывали ли вы каждую функцию или процедуру;
- **Call coverage:** показывает, выполняли ли вы каждый вызов функции. Гипотеза состоит в том, что ошибки обычно возникают в интерфейсах между модулями (вызывающая функция и вызываемая функция). Также известен как покрытие пары вызовов (call pair coverage);

7 вышеперечисленных уровней описываются в книге Копленда "A Practitioner's Guide to Software Test Design", но можно найти и другие

FSM coverage (Finite State Machine Coverage)

Конечные автоматы (FSM) имеют конечное число состояний, условий, которые приводят к внутренним переходам между состояниями, и соответствующее поведение ПО в каждом состоянии автомата. Автомат обычно моделирует поведение управляющей логики.

Покрытие FSM - 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. Конечные автоматы могут иметь множество ветвей и несколько функциональных путей, а также любой скрытый путь (функциональный путь, пропущенный при проверке, или путь, непреднамеренно введенный на этапе реализации) в дизайне может вызвать серьезное нарушение функциональности, а также может создать тупик (система не может самостоятельно выйти из определенного состояния, даже если намеченный стимул присутствует).

Basis Path testing

Цель тестирования базового пути - в отличии от D-D Path (Decision-to-decision path) получить полное покрытие тех путей, которые находятся между точками принятия решений (decisions points) с высоким бизнес-риском и высокой бизнес-ценностью, т.к. проверять все возможные пути обходится слишком дорого. Это гибрид branch testing и path testing

LCSAJ coverage

LCSAJ (linear code sequence and jump) «линейная последовательность кода и переход». Каждый LCSAJ представляет собой сегмент кода, который выполняется последовательно от начальной точки до конечной точки, а затем прерывает последовательный поток для передачи потока управления. Каждая строка кода имеет плотность (density), то есть количество раз, когда номер строки появляется в LCSAJ.

Один LCSAJ состоит из трех компонентов:

- Начало сегмента, который может быть ветвью или началом программы;
- Конец сегмента, который может быть концом ветви или концом программы;
- Конкретная целевая линия;

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 кода программы. При использовании в этом смысле, LCSAJ также называют JJ-путь (jump-to-jump path). 100% LCSAJ означает 100% Statement Coverage, 100% Branch Coverage, 100% procedure или Function call Coverage, 100% Multiple condition Coverage (в ISTQB говорится только о 100% Decision coverage).

Определенные метрики используются для проверки покрытия кода. Эти показатели могут помочь нам определить, достаточно ли тестирования или нет. Эти показатели называются коэффициентом эффективности тестирования (TER - Test Effectiveness Ratio):

- TER-1: количество операторов, выполненных с помощью тестовых данных, деленное на общее количество операторов;
- TER-2: количество ветвей потока управления, выполненных тестовыми данными, деленное на общее количество ветвей потока управления;
- TER-3: количество LCSAJ, выполненных тестовыми данными, деленное на общее количество LCSAJ;

Исследователи ссылаются на коэффициент покрытия путей длиной n LCSAJ как на коэффициент эффективности теста (TER) $n + 2$.

2. 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. Рапс и Вьюкер, популяризаторы данного метода, писали: "Мы уверены, что, как нельзя чувствовать себя уверенными в программе без выполнения каждого ее оператора в рамках какого-то тестирования, так же не следует быть уверенными в программе без видения результатов использования значений, полученных от любого и каждого из вычислений".

Когда «поток данных» через информационную систему представлен графически, он известен как диаграмма потока данных (Data Flow Diagram). Она также используется для визуализации обработки данных. Но не нужно путать это с графом потока данных (Data Flow Graph), который используется в Data Flow Testing. Граф потока данных похож на граф потока управления тем, что показывает поток обработки через модуль. Дополнительно к этому, он детализирует определение, использование и уничтожение каждой из переменных модуля. Мы построим эти диаграммы и убедимся, что шаблоны определение-использование-уничтожение являются подходящими. Сначала мы проведем статический анализ. Под "статическим" мы имеем в виду, что мы исследуем диаграмму (формально через проверки или неформально беглыми просмотрами). Потом мы проведем динамические тесты модуля. Под "динамическими" мы понимаем, что мы создаем и исполняем тестовые сценарии.

Так как тестирование потока данных основано на потоке управления модуля, то, предположительно, поток управления в основном верный. Процесс тестирования потока данных сводится к выбору достаточного количества тестов, таких как:

- каждое "определение" прослеживается для каждого его "использования";
- каждое "использование" прослеживается из соответствующего ему "определения";

Чтобы сделать это, перечислим маршруты в модуле. Порядок выполнения такой же, как и в случае с тестированием потока управления: начинаем с точки входа в модуль, строим самый левый маршрут через весь модуль и заканчиваем на выходе из него. Возвращаемся в начало и идём по другому направлению в первом разветвлении. Прокладываем этот путь до конца. Возвращаемся в начало и идём по другому направлению во втором разветвлении, потом в третьем и т.д., пока не пройдем все возможные пути. Затем создадим хотя бы один тест для каждой переменной, чтобы покрыть каждую пару определение-использование.

Существуют условные обозначения, которые могут помочь в описании последовательных во времени пар в жизненном цикле переменной:

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

Таким образом, $\sim d$, du , kd , ud , uk , uu , $k \sim$, $u \sim$ являются вполне допустимыми комбинациями, когда $\sim u$, $\sim k$, dd , dk , kk , ku , $d \sim$ 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.

Источники:

- [Software Testing Techniques](#)
- Ли Копланд - "A Practitioner's Guide to Software Test Design", Главы 10 и 11
- [НОУ Интuit - Лекция 4: Тестирование программного кода \(покрытия\)](#)
- [Code Coverage Tutorial: Branch, Statement, Decision, FSM](#)
- [Statement, Branch, and Path Coverage Testing](#)
- [What is the difference between dd-path testing and basis path testing \(both aims branch coverage\)?](#)
- [Linear code sequence and jump](#)
- [What is LCSAJ Testing?](#)

Доп. материал:

- [dynamic analysis tools](#)

- [НОУ ИНТУИТ - Лекция 4: Метод MC/DC для уменьшения количества тестовых примеров при 3-м уровне покрытия кода + Анализ покрытия](#)

Dynamic -> Black box

Все specification-based или Black Box testing techniques могут быть удобно описаны и систематизированы с помощью следующей таблицы:

| Группа | Техника | Когда используется |
|--|----------------------------|---|
| Элементарные техники: <ul style="list-style-type: none"> • сосредоточены на анализе входных / выходных параметров; • можно комбинировать для лучшего покрытия; • обычно не используют и не зависят от других методик; | 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: <ul style="list-style-type: none"> • объединяют возможные значения нескольких параметров ввода / вывода; • могут использовать элементарные приемы для уменьшения количества возможных значений; | All Combinations | Количество возможных комбинаций входных значений достаточно мало, или каждая отдельная комбинация входных значений приводит к определенному выходному значению |
| | Pairwise Testing | Количество входных комбинаций чрезвычайно велико и должно быть сокращено до приемлемого набора кейсов |
| | Each Choice Testing | У вас есть функции, при которых скорее конкретное значение параметра вызывает ошибку, нежели комбинация значений |
| | Base Choice Testing | Вы можете выделить набор значений параметров, который имеет наибольшую вероятность использования |
| Продвинутые техники: <ul style="list-style-type: none"> • помогают проанализировать Систему с точки зрения бизнес-логики, иерархических отношений, сценариев и т. д.; • анализ основан на данных, организованных в таблицы, диаграммы и шаблоны; • может полагаться на элементарные и | Decision Table Testing | Существует набор комбинаций параметров и их выходных данных, описываемых бизнес-логикой 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, переходы которых регулируются правилами (например, потоки) |

| | | |
|--|-----------------------|---|
| комбинаторные методы для разработки тестовых примеров; | Cause-Effect Graphing | Причины (входы) и следствия (выходы) связаны большим количеством сложных логических зависимостей |
| | Scenario Testing | В функционале есть четкие сценарии |
| Другие техники | Random Testing | Вам необходимо имитировать непредсказуемость реальных вводных данных, или функциональность имеет несистематические дефекты |
| | Syntax Testing | Функциональность имеет сложный синтаксический формат для входных данных (например, коды, сложные имена электронной почты и т. д.) |

Эквивалентное разделение (Equivalence Partitioning (ISTQB/Myers 1979) / Equivalence Class Testing (Lee Copeland))

Класс эквивалентности представляет собой набор данных, которые либо одинаково обрабатываются модулем, либо их обработка выдает одинаковые результаты. При тестировании любое значение данных, входящее в класс эквивалентности, аналогично любому иному значению класса.

Эквивалентное разделение - это разделение всего набора данных ввода / вывода на такие разделы. Таким образом, вам не нужно выполнять тесты для каждого элемента подмножества, и достаточно одной проверки, чтобы охватить все подмножество. Хитрость заключается в том, чтобы увидеть и идентифицировать разделы, т.к. далеко не всегда они представляют собой числа.

Пример: Мы пишем модуль для системы отдела кадров, который определяет, в каком порядке нужно рассматривать заявления о приеме на работу в зависимости от возраста кандидата.

Правила нашей организации таковы:

- от 0 до 16 - не принимаются;
- от 16 до 18 - могут быть приняты только на неполный рабочий день;
- от 18 до 55 - могут быть приняты как сотрудники на полный рабочий день;
- от 55 до 99 - не принимаются;

Что в коде выглядит как:

- If (applicantAge >= 0 && applicantAge <=16)
 - hireStatus="NO";
- If (applicantAge >= 16 && applicantAge <=18)
 - hireStatus="PART";
- If (applicantAge >= 18 && applicantAge <=55)
 - hireStatus="FULL";
- If (applicantAge >= 55 && applicantAge <=90)
 - hireStatus="NO";

Из чего очевидно, что вместо 100 кейсов нам понадобится 4 по числу эквивалентных классов, все остальные кейсы внутри своих классов будут давать одинаковый результат тестов и являются избыточными.

Теперь мы готовы начать тестирование? Вероятно, нет. Что насчет таких входных данных как 969, -42, FRED или &#!? Должны ли мы создавать тестовые сценарии для некорректных входных данных? Для того, чтобы понять ответ, мы должны проверить подход, который пришел из объектно-ориентированного мира, названный "проектирование-по-контракту".

В подходе "проектирование-по-контракту" модули (в парадигме объектно-ориентированного программирования они называются "методами", но "модуль" является более общим термином) определены в терминах предусловий и постусловий. Постусловия определяют, что модуль обещает сделать (вычислить значение, открыть файл, напечатать отчет, обновить запись в базе данных, изменить состояние системы и т.д.). Предусловия описывают требования к модулю, при которых он переходит в состояние, описываемое постусловиями.

Например, если у нас есть модуль "openFile", что он обещает сделать? Открыть файл. Какие будут разумные предусловия для этого модуля?

- файл должен существовать,
- мы должны предоставить имя (или другую идентифицирующую информацию),
- файл должен быть "открываемым", т.е. он не может быть открыт в другом процессе,
- у нас должны быть права доступа к файлу и т.д.

Предусловия и постусловия основывают контракт между модулем и всеми, кто его вызывает. Тестирование-по-контракту основывается на философии проектирования-по-контракту. При использовании данного подхода мы создаем только те тест-кейсы, которые удовлетворяют нашим предусловиям. Например, мы не будем тестировать модуль "openFile", если файл не существует. Причина проста. Если файл не существует, то openFile не обещает работать. Если не существует требования работоспособности в определенных условиях, то нет необходимости проводить тестирование в этих условиях.

В этот момент тестировщики обычно возражают. Да, они согласны, что модуль не претендует на работу в этом случае, но что делать, если предусловия нарушаются в процессе разработки? Что делать системе? Должны ли мы получить сообщение об ошибке на экране или дымящуюся воронку на месте нашей компании? Другим подходом к проектированию является оборонительное проектирование. В этом случае модуль предназначен для приема любого входного значения. Если выполнены обычные предусловия, то модуль достигнет своих обычных постусловий. Если обычные предварительные условия не выполняются, то модуль сообщит вызывающему, возвратив код ошибки или бросив исключение (в зависимости от используемого языка программирования). На самом деле, это уведомление является еще одним из постусловий модуля.

На основе этого подхода мы могли бы определить оборонительное тестирование: подход, который анализирует как обычные, так и необычные предварительные условия.

Нужно ли нам делать проверку с такими входными значениями, как -42, FRED и &#!@? Если мы используем проектирование-по-контракту и тестирование-по-контракту, то ответ "Нет". Если мы используем оборонительное проектирование и, поэтому, оборонительное тестирование, то ответ "Да". Спросите ваших проектировщиков, какой подход они используют. Если их ответом будет «контрактный» либо «оборонительный», то вы знаете, какой стиль тестирования использовать. Если они ответят "Хм?", то это значит, что они не думают о том, как взаимодействуют модули. Они не думают о предусловиях и постусловиях контрактов. Вам стоит ожидать, что интеграционное тестирование будет главным источником дефектов, будет более сложным и потребует больше времени, чем ожидалось.

Несмотря на то, что тестирование классов эквивалентности полезно, его величайшим вкладом является то, что оно приводит нас к тестированию граничных значений.

Анализ граничных значений (BVA - Boundary Value Analysis (Myers 1979)/range checking)

Тестирование классов эквивалентности - это самая основная методика тест-дизайна. Она помогает тестировщикам выбрать небольшое подмножество из всех возможных тестовых сценариев и при этом обеспечить приемлемое покрытие. У этой техники есть еще один плюс. Она приводит к идеи о тестировании граничных значений - второй ключевой технике тест-дизайна.

Пример. Выше описывались правила, которые указывали, каким образом будет происходить обработка заявок на вакансии в зависимости от возраста соискателя.

Обратите внимание на проблемы на границах - это "края" каждого класса. Возраст "16" входит в два различных класса эквивалентности (как и "18", и "55"). Первое правило гласит не нанимать шестнадцатилетних. Второе правило гласит, что шестнадцатилетние могут быть наняты на неполный рабочий день. Тестирование граничных значений фокусируется на границах именно потому, что там спрятано очень много дефектов. Опытные тестировщики сталкивались с этой ситуацией много раз. У неопытных тестировщиков может появиться интуитивное ощущение, что ошибки будут возникать чаще всего на границах. Эти дефекты могут быть в требованиях, или в коде, если программист ошибется с указанием границ в коде (включительно/не включительно, индекс +1).

Попробуем исправить приведенный выше пример:

- от 0 до 15 - не принимаются;
- от 16 до 17 - могут быть приняты только на неполный рабочий день;
- от 18 до 54 - могут быть приняты как сотрудники на полный рабочий день;
- от 55 до 99 - не принимаются;

А что насчет возраста -3 и 101? Обратите внимание, что требования не указывают, как должны быть рассмотрены эти значения. Мы можем догадаться, но "угадывание требований" не является приемлемой практикой. Следующий код реализует исправленные правила:

- if (applicantAge >= 0 && applicantAge <= 15)
 - hireStatus = "NO";
- if (applicantAge >= 16 && applicantAge <= 17)
 - hireStatus = "PART";
- if (applicantAge >= 18 && applicantAge <= 54)
 - hireStatus = "FULL";
- if (applicantAge >= 55 && applicantAge <= 99)
 - hireStatus = "NO";

В этом примере интересными значениями на границах или вблизи них являются {-1, 0, 1}, {15, 16, 17}, {17, 18, 19}, {54, 55, 56} и {98, 99, 100}. Другие значения, например {-42, 1001, FRED, %\$#@} могут быть включены в зависимости от предусловий документации модуля.

Для создания тест-кейсов для каждого граничного значения определите классы эквивалентности, выберите одну точку на границе, одну точку чуть ниже границы и одну точку чуть выше границы. Стоит отметить, что точка чуть выше границы может входить в другой класс эквивалентности. В таком случае не нужно дублировать тест. То же самое может быть верно по отношению точки чуть ниже границы.

Тестирование граничных значений является наиболее подходящим там, где входные данные являются непрерывным диапазоном значений.

Тестирование таблиц решений (Decision Table testing)

Этот простой, но эффективный метод заключается в документировании бизнес-логики в таблице как наборы правил, условий выполнения действий и самих действий. Тестирование таблиц принятия решений может быть использовано, когда система должна реализовывать сложные бизнес-правила, когда эти правила могут быть представлены в виде комбинации условий и когда эти условия имеют дискретные действия, связанные с ними.

Пример. Компания по автострахованию дает скидку водителям, которые состоят в браке и/или хорошо учатся.

| - | Правило 1 | Правило 2 | Правило 3 | Правило 4 |
|---------|-----------|-----------|-----------|-----------|
| Условия | - | - | - | - |

| | | | | |
|------------------|----|-----|-----|-----|
| Состоит в браке? | Да | Да | Нет | Нет |
| Хороший студент? | Да | Нет | Да | Нет |
| - | - | - | - | - |
| Действия | - | - | - | - |
| Скидка (\$) | 60 | 25 | 50 | 0 |

эта таблица содержит все комбинации условий. Задав два бинарных условия ("да" или "нет"), возможные комбинации будут: ("да", "да"), ("да", "нет"), ("нет", "да") и ("нет", "нет"). Каждое правило представляет собой одну из этих комбинаций. Нам, тестировщикам, нужно будет проверить, что определяются все комбинации условий. Пропущенное сочетание может привести к разработке такой системы, которая не сможет правильно обработать определенный набор исходных данных. Каждое правило является причиной "запуска" действия. Каждое правило может задать действие, уникальное для этого правила, или правила могут иметь общие действия. Для каждого правила с помощью таблицы решений можно указать более одного действия. Опять же, эти правила могут быть уникальными или быть общими. В такой ситуации выбрать тесты просто - каждое правило (вертикальная колонка) становится тест-кейсом. Условия указывают на входные значения, а действия - на ожидаемые результаты.

Если тестируемая система имеет сложные бизнес-правила, а у ваших бизнес-аналитиков или проектировщиков нет документации этих правил, то тестировщикам следует собрать эту информацию и представить ее в виде таблицы решений. Причина проста: представляя поведение системы в такой полной и компактной форме, тест-кейсы могут быть созданы непосредственно из таблицы решений. При тестировании для каждого правила создается как минимум один тест-кейс. Если состояния этого правила бинарные, то должно быть достаточно одного теста для каждого сочетания. С другой стороны, если состояние является диапазоном значений, то тестирование должно учитывать и нижнюю, и высшую границы диапазона. Таким образом мы объединяем идею тестирования граничных значений с тестированием таблиц решений.

Чтобы создать тестовую таблицу, просто измените заголовки строк и столбцов: правила станут тест-кейсами, условия входными значениями, а действия ожидаемыми результатами.

Комбинаторные техники тест-дизайна (Combination Strategies)

Тестовые примеры выбираются на основе некоторого понятия покрытия, и цель стратегии комбинирования состоит в том, чтобы выбрать тестовые примеры из набора тестов таким образом, чтобы было достигнуто 100% покрытие.

- 1-wise coverage (each-used) - это самый простой критерий покрытия. Для 100% each-used покрытия требуется, чтобы каждое значение каждого параметра было включено хотя бы в один тестовый пример в наборе тестов.
- 2-wise (pair-wise) coverage требует, чтобы каждая возможная пара значений любых двух параметров была включена в некоторый тестовый пример. Обратите внимание, что один и тот же тестовый пример часто охватывает более одной уникальной пары значений.
- Естественным продолжением 2-wise coverage является t-wise coverage, которое требует включения всех возможных комбинаций интересных значений параметров t в какой-либо тестовый пример в наборе тестов.
- Самый тщательный критерий покрытия, N-wise coverage, требует набора тестов, который содержит все возможные комбинации значений параметров в input parameter model (IPM).

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.

Тестирование каждого выбора (EC - Each choice testing): эта стратегия требует, чтобы каждое значение каждого параметра было включено по крайней мере в один тестовый пример (Ammann & Offutt, 1994). Это также определение 1-wise coverage.

Тестирование базового выбора (BC - Base choice testing): алгоритм стратегии комбинирования базового выбора начинается с определения одного базового тестового примера. Базовый тестовый пример может быть определен по любому критерию, включая простейший, наименьший или первый. Критерий, предложенный Амманном и Оффуттом (Ammann & Offutt, 1994), - это «наиболее вероятное значение» с точки зрения конечного пользователя. Это значение может быть определено тестировщиком или основано на рабочем профиле, если таковой существует. Из базового тестового примера создаются новые тестовые примеры, изменяя интересующие значения одного параметра за раз, сохраняя значения других параметров фиксированными в базовом тестовом примере. Базовый выбор включает каждое значение каждого параметра по крайней мере в одном тестовом примере, поэтому он удовлетворяет 1-wise coverage.

Попарное тестирование (Pairwise testing)

Pairwise testing — техника тест-дизайна, а именно метод обнаружения дефектов с использованием комбинационного метода из двух тестовых случаев. Он основан на наблюдениях о том, что большинство дефектов вызвано взаимодействием не более двух факторов (дефекты, которые возникают при взаимодействии трех и более факторов, как правило менее критичны). Следовательно, выбирается пара двух тестовых параметров, и все возможные пары этих двух параметров отправляются в качестве входных параметров для тестирования. Pairwise testing сокращает общее количество тест-кейсов, тем самым уменьшая время и расходы, затраченные на тестирование. Захватывающей надеждой попарного тестирования является то, что путем создания и запуска 1-20% тестов вы найдете 70-85% от общего объема дефектов.

Пример: По ТЗ сайт должен работать в 8 браузерах, используя различные плагины, запускаться на различных клиентских операционных системах, получать страницы от разных веб-серверов, работать с различными серверными, операционными системами. Итого:

- 8 браузеров;
- 3 плагина;
- 6 клиентских операционных систем;
- 3 сервера;
- 3 серверных операционных системы;

= 1296 комбинаций. Количество комбинаций настолько велико, что, скорее всего, у нас не хватит ресурсов, чтобы спроектировать и пройти тест-кейсы. Не следует пытаться проверить все комбинации значений для всех переменных, а нужно проверять комбинации пар значений переменных.

Использование всех пар для создания тест-кейсов основывается на двух техниках:

- ортогональные массивы (OA - Orthogonal Array): это двумерный массив символов. На примере выше мы составляем таблицу, где столбцы представляют собой переменные (браузер, плагин, клиентская операционная система, веб-сервер и серверная операционная система, а строки - значения каждой переменной (Chrome/Opera, Windows 8/10/11 и т.п.). После чего нужно определить ортогональный массив, у которого будет столбец для каждой переменной (каждый столбец ортогонального массива имеет столько же вариантов значений, сколько имеет ваша переменная). Используя ортогональный массив для примера выше, все пары всех значений всех переменных могут быть покрыты всего лишь 64-мя тестами.
- алгоритм Allpairs: генерирует пары непосредственно, не прибегая к таким к ортогональным массивам. "Несбалансированный" характер алгоритма выбора всех пар требует только 48 тестов для примера. Следует отметить, что комбинации, выбранные методом ортогонального массива, могут быть не такими же, как те, которые выбраны Allpairs. Но это не важно. Важно лишь то, чтобы были выбраны все парные комбинации параметров. Это будут комбинации, которые мы хотим проверить.

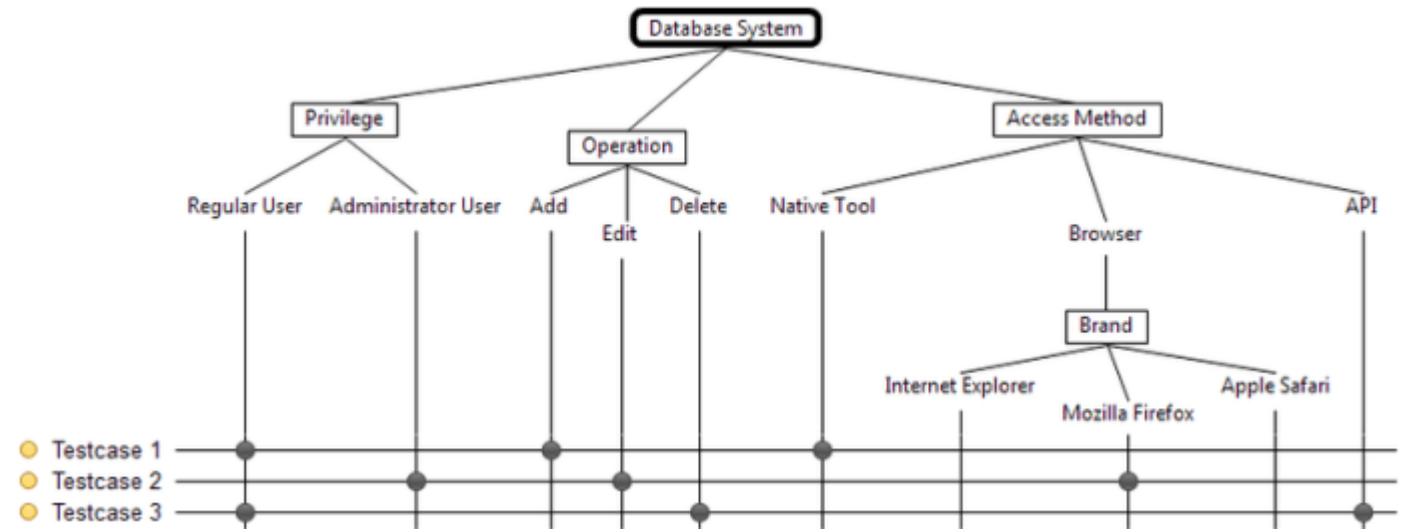
Подробнее с разбором примера см. у Копленда в главе 6.

На практике же вручную эти массивы никто не формирует, всю механику реализуют автоматизированные инструменты, самый популярный из них PICT. Тестировщику остается лишь подготовить и скормить данные.

Classification tree method

Дерево классификации (Classification tree): структура, показывающая иерархически упорядоченные классы эквивалентности, которое используется для разработки тестовых примеров в методе дерева классификации (Classification tree method). Не путать с [Decision tree](#).

Метод дерева классификации: вид комбинаторной техники, в которой тестовые примеры, описанные с помощью дерева классификации, предназначены для выполнения комбинаций представителей входных и / или выходных доменов.



Чтобы рассчитать количество тестовых примеров, нам необходимо проанализировать требования, определить соответствующие тестовые функции (классификации) и их соответствующие значения (классы).

Обычно для создания Classification tree используется инструмент Classification Tree Editor. Если же взять лист бумаги и ручку, то 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 классы 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 (all combinations, each choice и т. д.), и создаем все необходимые комбинации.

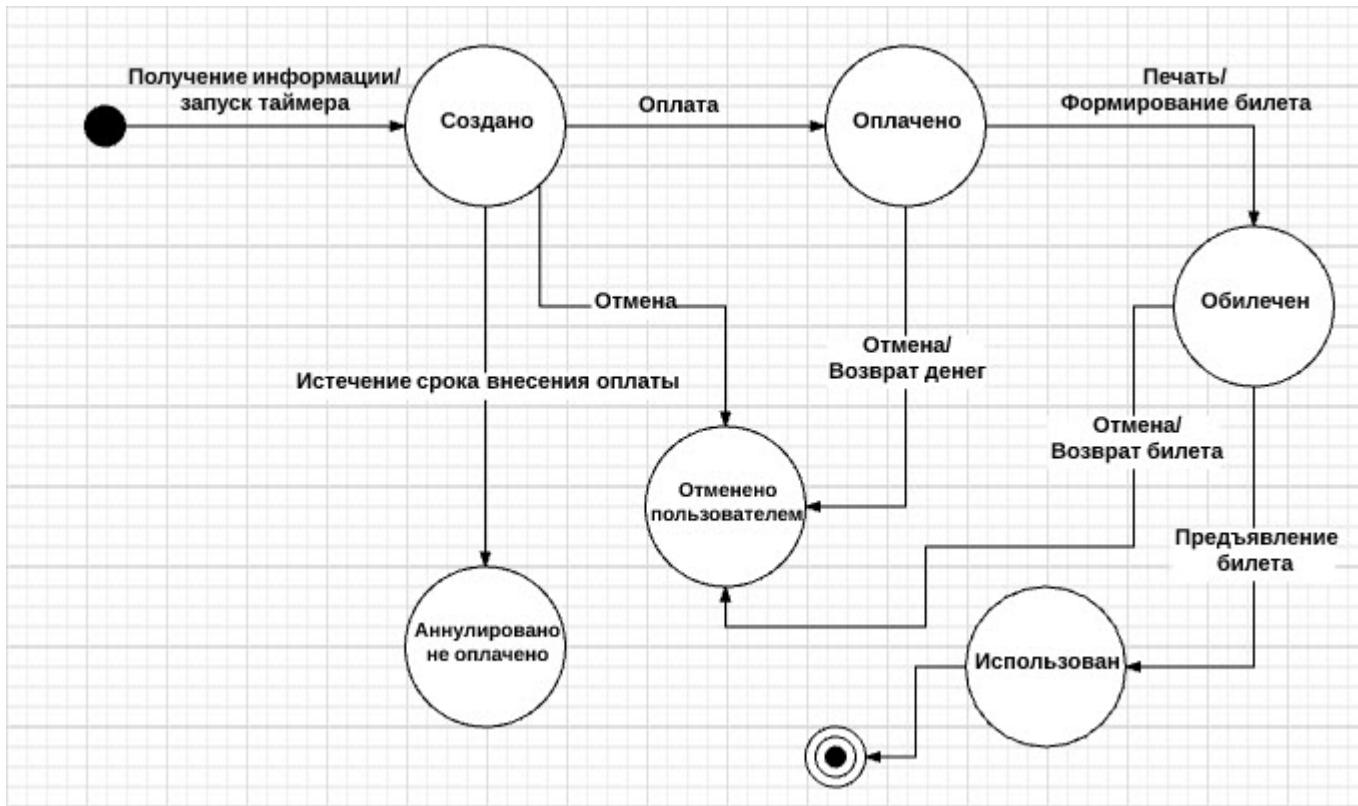
Максимальное количество тестовых примеров - это декартово произведение всех классов всех классификаций в дереве, быстро приводящее к большим числам для реалистичных тестовых задач.

Минимальное количество тестовых примеров - это количество классов в классификации с наиболее содержащимися классами. На втором этапе тестовые примеры составляются путем выбора ровно одного класса из каждой классификации дерева классификации.

Тестирование переходов между состояниями (State Transition testing)

Тестирование переходов между состояниями определяется как метод тестирования ПО, при котором изменения входных условий вызывают изменения состояния в тестируемом приложении (AUT). 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. Для наглядности возьмем классический пример покупки авиабилетов:



- 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;

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 пользователя, оплата билета, получение билета на руки, и т. д.) отображаются в блок-схеме.

На основании полученной схемы составляется набор тестов.

Определим четыре разных уровня покрытия:

1. Набор тестов, в котором все состояния будут посещены как минимум один раз. Этому требованию удовлетворяет набор из трех тестов, показанный ниже. Обычно это низкий уровень тестового покрытия.
2. Набор тестов, в котором все события выполняются как минимум один раз. Следует отметить, что тест-кейсы, которые покрывают каждое событие, могут быть точно теми же, которые покрывают каждое состояние. Опять же, это низкий уровень покрытия.
3. Набор тестов, в котором все пути будут пройдены как минимум один раз. Несмотря на то, что этот уровень является наиболее предпочтительным из-за его уровня покрытия, это может быть неосуществимо. Если диаграмма состояний и переходов содержит петли, то количество возможных путей может быть бесконечным.
4. Набор тестов, в котором все переходы будут осуществлены как минимум один раз. Этот уровень тестирования обеспечивает хороший уровень покрытия без порождения большого количества тестов. Этот уровень, как правило, один из рекомендованных.

Диаграмма состояний и переходов - не единственный способ документирования поведения системы.

Диаграммы, возможно, легче в понимании, но таблицы состояний и переходов могут быть проще в использовании на постоянной и временной основе. Таблицы состояний и переходов состоят из четырех столбцов - "Текущее состояние", "Событие", "Действие" и "Следующее состояние". Преимущество таблицы состояний и переходов в том, что в ней перечисляются все возможные комбинации состояний и переходов, а не только допустимые. При крайне необходимом тестировании систем с высокой степенью риска, например авиационной радиоэлектротехники или медицинских устройств, может потребоваться тестирование каждой пары состояния-переход, включая те, которые не являются допустимыми. Кроме того, создание таблицы состояний и переходов часто извлекает комбинации, которые не были определены, задокументированы или рассмотрены в требованиях. Очень полезно обнаружить эти дефекты до начала кодирования.

Использование таблицы состояний и переходов может помочь обнаружить дефекты в реализации, которые позволяют недопустимые пути из одного состояния в другое. Недостатком таких таблиц является то, что, когда количество состояний и событий возрастает, они очень быстро становятся огромными. Кроме того, в таблицах, как правило, большинство клеток пустые.

Подробнее с разбором примера см. у Копленда в главе 7.

Domain testing

В главах по тестированию классов эквивалентности и граничных значений мы рассмотрели тестирование одиночных переменных, которые требовали оценки в указанных диапазонах. В этой главе мы рассмотрим тестирование нескольких переменных одновременно. Существуют две причины, по которым стоит обратить на это внимание:

- у нас редко будет достаточно времени на создание тест-кейсов для каждой переменной в нашей системе. Их просто слишком много;
- часто переменные взаимодействуют. Значение одной переменной ограничивает допустимые значения другой. В этом случае, если проверять переменные поодиночке, можно не обнаружить некоторые дефекты;

Domain-тестирование - это техника, которая может применяться для определения эффективных и действенных тест-кейсов, когда несколько переменных (например, поля ввода) должны проверяться вместе - либо для эффективности, либо по причине их логического взаимодействия. Она использует и обобщает тестирование классов эквивалентности и граничных значений в п одномерных измерениях. Подобно этим техникам, мы ищем случаи, где граница была неверно определена или реализована. Несмотря на то, что эта техника лучше всего подходит для числовых значений, она может быть обобщена и на другие типы - boolean, string, enumeration и т.д.

В двухмерном измерении (с двумя взаимодействующими параметрами) могут возникнуть следующие дефекты:

- сдвиг границы - граница, перемещённая вертикально или горизонтально;
- направление границы - граница, повёрнутая под неправильным углом;
- пропущенная граница;
- лишняя граница.

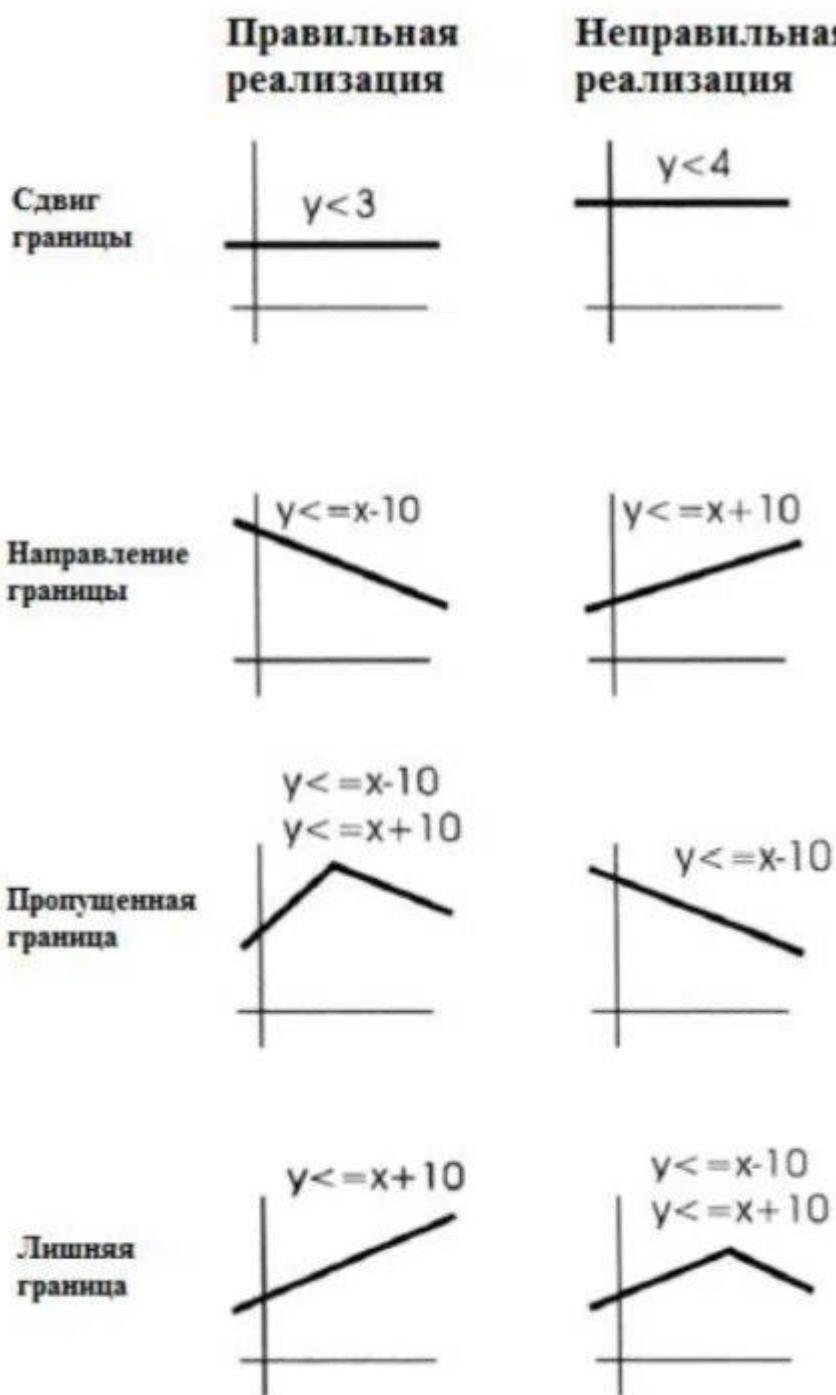


Рисунок 8-1: Дефекты двухмерного пространства.

Подробнее с разбором примера см. у Копленда в главе 8.

Use case-based Testing

До сих пор мы исследовали техники разработки тестовых сценариев для частей системы — входные переменные с их диапазонами и границами, бизнес-правила, представленные в виде таблиц решений, а также поведения системы, представленные с помощью диаграмм состояний и переходов. Теперь пришло

время рассмотреть тестовые сценарии, которые используют системные функции с начала и до конца путем тестирования каждой из их индивидуальных операций.

Сегодня самым популярным подходом определения выполняемых системой операций является [диаграмма вариантов использования](#) (диаграмма прецедентов, Use case diagram). Как и таблицы решений и диаграммы состояний и переходов, диаграммы вариантов использования обычно создаются разработчиками для разработчиков. Но, как и другие техники, диаграммы вариантов использования содержат много полезной информации и для тестировщиков. Варианты использования были созданы Иваром Якобсоном и объяснены в его книге "Объектно-ориентированная разработка программ: подход, основанный на вариантах использования". Якобсон определил

Вариант использования (Use Case) - это сценарий, который описывает использование системы действующим лицом для достижения определенной цели (Ивар Якобсон - "Объектно-ориентированная разработка программ: подход, основанный на вариантах использования").

- Действующее лицо (или актер) - это пользователь, играющий роль с уважением к системе, старающегося использовать систему для достижения чего-то важного внутри конкретного контекста. Действующими лицами в основном являются люди, хотя действующими лицами также могут выступать другие системы;
- "Сценарий" - это последовательность шагов, которые описывают взаимодействия между актером и системой. Заметьте, что варианты использования определены с точки зрения пользователя, а не системы. Заметьте также, что операции, выполняемые внутри системы, хоть и важны, но не являются частью определения вариантов использования. Набор вариантов использования составляет функциональные требования системы.

Прежде чем использовать сценарии 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.

Польза вариантов использования в том, что они:

- позволяют выявить функциональные требования системы с точки зрения пользователя несмотря на техническую перспективу и независимо от того, какая парадигма разработки использовалась;
- могут быть использованы для активного вовлечения пользователей в процесс сбора требований и определений;
- предоставляют базис для идентификации ключевых компонентов системы, структур, баз данных и связей;
- служат основанием для разработки тест-кейсов системы на приемочном уровне.

Подробнее с разбором примера см. у Копленда в главе 8.

Как создать хорошие сценарии (Сэм Канер):

1. Напишите истории жизни для объектов в системе.
2. Перечислите возможных пользователей, проанализируйте их интересы и цели.
3. Подумайте об отрицательных пользователях: как они хотят злоупотреблять вашей системой?
4. Перечислите «системные события». Как система справляется с ними?
5. Перечислите «особые события». Какие приспособления система делает для них?
6. Перечислите преимущества и создайте сквозные задачи, чтобы проверить их.

7. Интервью пользователей об известных проблемах и сбоях старой системы.
8. Работайте вместе с пользователями, чтобы увидеть, как они работают и что они делают.
9. Читайте о том, что должны делать подобные системы.
10. Изучите жалобы на предшественника этой системы или ее конкурентов.
11. Создать фиктивный бизнес. Относитесь к нему как к реальному и обрабатывайте его данные.
12. Попробуйте преобразовать реальные данные из конкурирующего или предшествующего приложения.

Cause/Effect, Cause-Effect (CE)

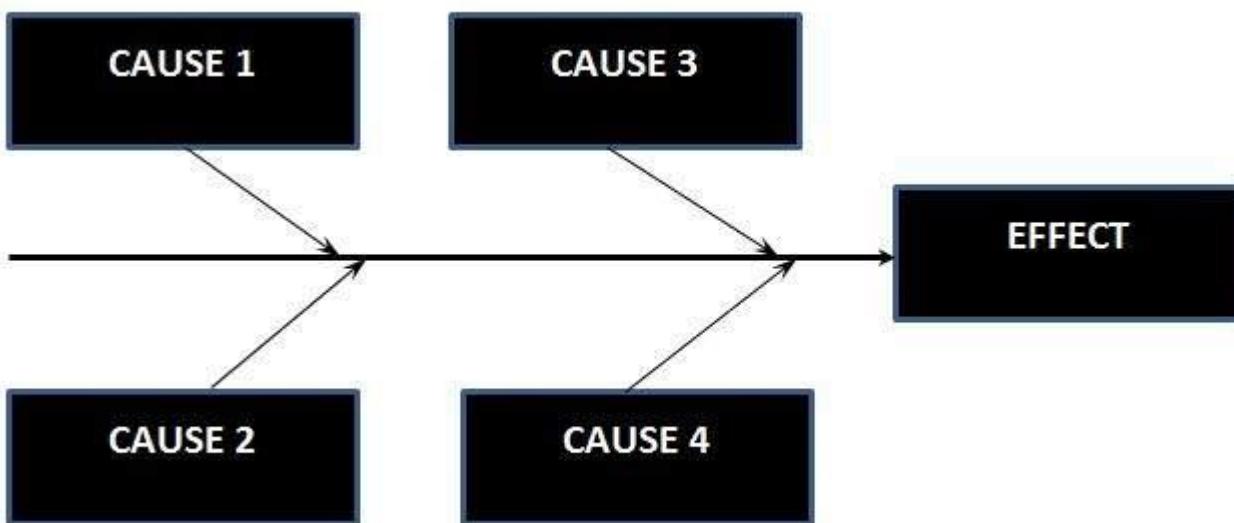
Тестовые примеры должны быть разработаны так, чтобы проявлять принципы, которые характеризуют взаимосвязь между входными и выходными данными компонента, где каждый принцип соответствует единственной возможной комбинации входных данных компонента, которые были выражены как логические значения. Для каждого тестового примера следует уточнить:

- Логическое состояние для каждого эффекта;
- Логическое состояние (истина или ложь) по любой причине;

Граф причинно-следственных связей (Cause-Effect Graph) использует такую модель логических взаимосвязей между причинами и следствиями для компонента. Каждая причина выражается как условие, которое может быть истинным, ложным на входе или комбинацией входных данных компонента. Каждый эффект выражается в виде логического выражения, представляющего результаты или комбинацию результатов для произошедшего компонента (*?Every effect is expressed as a Boolean expression representing results, or a combination of results, for the component having occurred*). Модель обычно представлена как логический граф, связывающий производные логические выражения ввода и вывода с использованием логических операторов:

- И (AND);
- ИЛИ (OR);
- Истина, если не все входные данные верны («не оба») ([NAND](#));
- Истина, когда ни один из входов не является истиной ("ни один") ([NOR](#));
- НЕ (NOT).

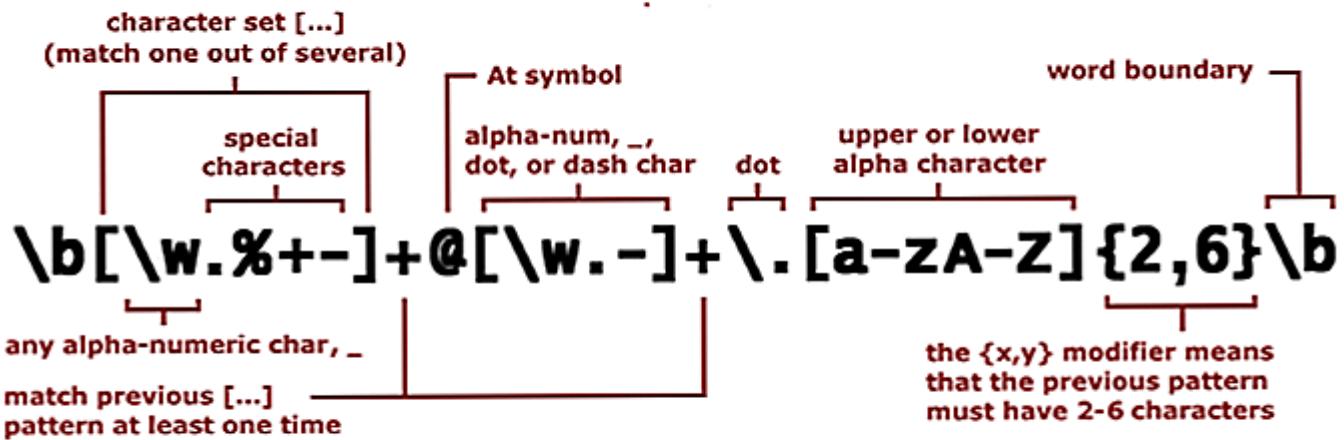
Cause-Effect Graph также известен как диаграмма Исикавы, поскольку он был изобретен Каору Исикава, или как диаграмма рыбьей кости из-за того, как он выглядит.



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.

Syntax testing

Синтаксическое тестирование используется для проверки формата и правильности входных данных в случаях символьных текстовых полей, проверки соответствия формату файла, схеме базы данных, протоколу и т.д., при этом данные могут быть формально описаны в технических или установленных и определенных обозначениях, таких как BNF ([Форма Бэкуса - Найра](#)).



Parse: `username@domain.TLD (top level domain)`

Как правило, синтаксические тесты автоматизированы, так как они предполагают создание большого количества кейсов. Синтаксис тестируется с использованием двух условий:

- Валидные: Проверка нормального состояния с использованием покрывающего набора путей синтаксического графа для минимально необходимых требований (?Testing the normal condition using the covering set of paths of the syntax graph, for the minimum necessary requirements). Иными словами находим возможные варианты значений, допускаемые отдельными элементами определения BNF, а затем разрабатываем кейсы, чтобы просто охватить эти варианты;
- Невалидные: Проверка мусорных условий (garbage condition)* с использованием недопустимого набора входных данных.

Примечание *: Мусорные условия - это метод проверки устойчивости системы к неверным или грязным данным. Условие выполняется путем предоставления в систему грязных данных (недопустимых данных), которые не поддерживаются указанным форматом и грамматикой синтаксиса. Для создания таких данных 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).

Check List Based Testing

Тестирование на основе контрольного списка (чеклиста) выполняется с использованием предварительно подготовленного опытными тестировщиками чеклиста, который продолжает обновляться с учетом любых новых дефектов, обнаруженных при выполнении контрольных примеров контрольного списка. При любых изменениях в продукте прогоняется быстрый чеклист, чтобы убедиться, что из-за изменений не возникло новых дефектов. Этот контрольный список не имеет отношения к пользовательским историям.

Risk-Based Testing

Тест-дизайн на основе риска можно рассмотреть как часть вида/подхода к тестированию.

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 помочь 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.

Тестирование на основе рисков помогает обнаруживать наиболее важные и критические ошибки на ранней стадии. 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. Если вовремя не выявить какой-либо риск, это может помешать завершению проекта. Самым первым шагом является определение рисков, а затем их оценка, т.е. определение приоритетов, а затем обработка рисков. После того, как риски идентифицированы, необходимо оценить риск, чтобы определить риски с точки зрения их воздействия, то есть того, какой ущерб он может нанести.

Приоритет любого риска зависит от вероятности возникновения и его воздействия на продукт, то есть от того, насколько серьезным является воздействие: Priority = Probability * Severity.

- Управление рисками (Risk Management) должно начинаться на ранней стадии жизненного цикла, чтобы с рисками можно было справиться на ранней стадии. Категории рисков:
 - Project Risks: бюджет, ресурсы, график, проблемы, связанные с клиентами и т. д.
 - Technical Risks: Технический риск включает риски на этапах проектирования, внедрения, тестирования и разработки. Большинство технических рисков возникает либо на этапе требований, либо во время кодирования, поскольку неясность требований может привести к серьезным рискам во время разработки. Во-вторых, на этапе разработки, если разработчики недостаточно хорошо знакомы с Продуктом, это также может привести к серьезным рискам.
 - Business Risks: Бизнес-риски включают проблемы с бюджетом и созданием проекта, который не соответствует требованиям заказчика, потерю клиента.
- Оценка рисков (Risk Assessment): риск оценивается на основе того, насколько серьезным является воздействие.
- Контроль рисков (Risk Control): Риском можно управлять с помощью трех стратегий:
 - Предотвращение риска (Risk Avoidance): избежание риска - это избежание риска проекта с согласия клиента. Например. Чтобы избежать риска задержки реализации проекта из-за нехватки ресурсов, ресурсы можно оставить на скамейке запасных для замены, если какой-либо ресурс уйдет в отпуск. Объем работ можно сократить, чтобы избежать рискову
 - Снижение риска (Risk Reduction): снижение риска включает в себя планирование управления рисками и минимизации их последствий. Например. Чтобы снизить риск, для реализации Проекта следует использовать инкрементальный подход;
 - Передача риска (Risk Transfer): эта стратегия включает покупку страховки или любого компонента, разработанного третьей стороной, чтобы избежать возникновения рисков;

Процесс Risk-based Testing:



Риск идентифицируется и анализируется, т. е. рассчитывается влияние (impact) и серьезность (severity) риска, и принимаются меры в соответствии с приоритетом риска. Как только приоритет определен, начинается тестирование, т. е. создаются объем (test scope) и план тестирования, а затем выполняются тесты.

User Journey Test

User Journey test, как следует из названия, охватывает полное путешествие пользователя по системе. Он охватывает сквозные тесты, из-за которых процент покрытия тестами больше по сравнению с другими методами. Этот метод помогает уменьшить количество тестовых примеров, поскольку тестовые примеры являются исчерпывающими и охватывают большую часть функциональности в одном сценарии. Сценарии написаны для самого сложного путешествия. Тесты взаимодействия с пользователем не связаны с пользовательскими историями (user stories).

User Story Testing (Agile)

Пользовательская история - это краткое и простое описание требований клиентов или конечного пользователя. Пользовательские истории написаны владельцем продукта (Product owner), поскольку именно он получает от клиента информацию о продукте, который будет создан. Если пользовательская история большая, она разбивается на несколько более мелких историй. Истории пользователей записываются на учетных карточках и вывешиваются на стене для обсуждения. Обсуждая важные аспекты функции, выберите те, которые в дальнейшем используются в пользовательской истории. Приемочные испытания - это заключительный этап, на котором продукт принимает заказчик после того, как он соответствует всем критериям выхода. Критерии приемлемости определяются владельцем продукта, заказчик на поставку также может привлекать разработчиков, определяя то же самое.

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 почти всегда не представляется возможным, из-за огромного количества входных значений.

Источники:

- Ли Копланд - "A Practitioner's Guide to Software Test Design", Главы 3-9
- [Test Design Techniques overview](#)
- [An Evaluation and comparison of practical results of Combination Strategies for Test Case Selection](#)
- [Handling constraints in the input space when using combination strategies for software testing](#)
- [State Transition testing](#)
- [Classification Tree Method](#)
- [Black Box Test Techniques. Cause-Effect Graphing](#)
- [Syntax Testing](#)
- [Popular Software Testing Techniques With Examples](#)

Доп. материал:

- [Тестирование областей определения или классы эквивалентности + анализ граничных значений + pairwise](#)
- [James Bach, Patrick J. Schroeder - "Pairwise Testing: A Best Practice That Isn't"](#)
- [Paul Ammann & Jeff Offutt - "Input Space Partition Testing"](#)
- [Jeff Offutt & Sten F. Andler - "Combination Testing Strategies"](#)
- [Комбинаторное тестирование: примеры с PICT](#)
- [Pairwise Testing - Обзор веб инструментов для попарного тестирования](#)
- [Cem Kaner - "Introduction to Domain Testing"](#)
- [Шаблон Requirement Traceability Matrix](#)
- [Cause-Effect Graph example](#)
- [Leadership in test: risk-based testing](#)

Dynamic -> Experience based

Error Guessing;

Предположение об ошибках (EG - error guessing): Метод проектирования тестов, когда опыт тестировщика используется для предугадывания того, какие дефекты могут быть в testeируемом компоненте или системе в результате сделанных ошибок, а также для разработки тестов специально для их выявления. (ISTQB)

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.

Некоторые факторы использующиеся при Error Guessing:

- Уроки, извлеченные из прошлых релизов;
- Исторические знания;
- Интуиция;
- Тикеты с прода;
- Review checklist;
- Пользовательский интерфейс приложения;
- Отчеты о рисках программного обеспечения;
- Тип данных, используемых для тестирования;
- Общие правила тестирования;
- Результаты предыдущих тестов;
- Знание об AUT (тестируемое приложение);

Исследовательское тестирование (Exploratory testing);

См. в видах тестирования

Ad-hoc testing;

См. в видах тестирования

Attack Testing;

Атака (attack): Направленная и нацеленная попытка оценить качество, главным образом надежность, объекта тестирования за счет попыток вызвать определенные отказы. См. также негативное тестирование. (ISTQB)

Тестирование на основе атак (attack-based testing): Методика тестирования на основе опыта, использующая программные атаки с целью провоцирования отказов, в частности - отказов, связанных с защищенностью. (ISTQB)

Источник:

[How Error Guessing Testing Benefit in Software Testing](#)

Тестовая документация/артефакты (Test Deliverables/test artifacts)

Виды тестовой документации

Артефакт (artifact) - это один из многих видов материальных побочных продуктов, возникающих в процессе STLC. Это не только документация, а в принципе всё, что создаётся для того, чтобы быть задействованным в тестировании.

Результаты тестирования (Test Deliverables) - это артефакты, которые передаются заинтересованным сторонам проекта программного обеспечения в течение жизненного цикла разработки программного обеспечения. На каждом этапе жизненного цикла разработки программного обеспечения существуют разные результаты тестирования. Некоторые результаты тестирования предоставляются до этапа тестирования, некоторые - на этапе тестирования, а некоторые - после завершения циклов тестирования.

Наличие или отсутствие документации, ее актуальность, как и используемые виды варьируются от компании к компании и даже от проекта к проекту. Создание и ведение документации требует весомого количества времени (и компетенций), а потому важно знать основные документы и их роль в процессах, учитывать требования всех заинтересованных лиц, нормативную и законодательную базу, политику и стандарты компании и особенности проекта чтобы понимать, какие из них необходимы (и обоснованы для бизнеса) в каждом случае. Существует огромное количество вариантов документов, часть из которых вы можете никогда и не встретить в реальной работе.

По Куликову документацию можно разделить на два больших вида в зависимости от времени и места ее использования:

- Продуктная документация (product documentation, development documentation) используется проектной командой во время разработки и поддержки продукта. Она включает:
 - План проекта (project management plan) и в том числе тестовый план (test plan);
 - Требования к программному продукту (product requirements document, PRD) и функциональные спецификации (functional specifications document, FSD; software requirements specification, SRS);
 - Архитектуру и дизайн (architecture and design);
 - Тест-кейсы и наборы тест-кейсов (test cases, test suites);
 - Технические спецификации (technical specifications), такие как схемы баз данных, описания алгоритмов, интерфейсов и т.д.;
- Проектная документация (project documentation) включает в себя как продуктную документацию, так и некоторые дополнительные виды документации и используется не только на стадии разработки, но и на более ранних и поздних стадиях (например, на стадии внедрения и эксплуатации). Она включает:
 - Пользовательскую и сопроводительную документацию (user and accompanying documentation), такую как встроенная помощь, руководство по установке и использованию, лицензионные соглашения и т.д.;
 - Маркетинговую документацию (market requirements document, MRD), которую представители разработчика или заказчика используют как на начальных этапах (для уточнения сути и концепции проекта), так и на финальных этапах развития проекта (для продвижения продукта на рынке).

Можно встретить и другие классификации. Внутренняя документация подробно описывает процесс разработки продукта, например стандарты, проектную документацию, заметки о деловой переписке и т. д. Внешняя документация относится к документам, которые подробно описывают сам продукт, например, Системная документация и Пользовательская документация. К внешней документации можно отнести Test policy, Test strategy, различные отчеты, Defect Report, Замечание, Запрос на изменение (улучшение), к внутренней всё от чеклиста до плана тестирования, тестовые данные и т.п. Пользовательская документация (User documentation) - это вся документация, которая будет передана конечному пользователю в комплекте с ПО.

Виды документации:

- **Политика качества** (Quality policy): отражает видение компании в отношении производства и поставки качественного продукта;
- **Политика тестирования** (Test policy): документ высокого уровня, в котором описаны принципы, методы и все важные цели тестирования в организации;
- **Стратегия тестирования** (Test strategy): статический документ высокого уровня (high-level), обычно разрабатываемый менеджером проекта (project manager). Это документ, который отражает подход к тестированию продукта и достижению целей. Обычно он выводится из Спецификации бизнес-требований (BRS - Business Requirement Specification). На основе стратегии тестирования готовится План тестирования;
- **План тестирования** (Test plan): документ, который содержит план всех действий по тестированию, которые необходимо выполнить для получения качественного продукта. План тестирования является производным от описания продукта (Product Description), SRS (Software requirements specification) или сценариев использования (Use Case) для всех будущих действий проекта. Обычно его готовят руководитель тестирования или менеджер по тестированию (Test Lead or Test Manager);
- **Отчет об оценке усилий** (Effort Estimation Report): в этом отчете группы тестирования оценивают усилия для завершения процесса тестирования;
- **Сценарий тестирования** (Test Scenario): элемент или событие программной системы, которое может быть проверено одним или несколькими тестовыми случаями;
- **Тестовый набор/комплект** (Test Suite): некоторый набор формализованных Test case, объединенных между собой по общему логическому признаку;
- **Тестовый случай/пример** (Test case): набор положительных и отрицательных исполняемых шагов тестового сценария, который имеет набор предварительных условий, тестовых данных, ожидаемого результата, пост-условий и фактических результатов;
- **Тест сурвей** (Test Survey): в рунете только [один источник](#) о нем, но есть упоминания в истории чатов коммьюнити. Test Survey по детализации занимает место посередине между чек-листом и тест-кейсом, а именно содержит в себе только summary и expected result. Т.е. подробнее чек-листов, где только заголовки, но с ожидаемым результатом и без шагов и прочего как в тест-кейсах;
- **Чек-лист** (Check List): перечень формализованных Test case в упрощенном виде удобном для проведения проверок, часто только список из заголовков кейсов;
- **Матрица прослеживаемости требований** (Requirements Traceability Matrix): документ, который соотносит требования с тестовыми примерами;
- **Тестовые данные** (Test Data): данные, необходимые при тестировании. Например, тестовая строка символов или учетные данные тестового пользователя;
- **Отчет о дефектах** (Defect Report): цель документа заключается в том, чтобы зафиксировать факт ошибки и передать разработчикам подробную информацию о ней;
- **Отчет о выполнении теста** (Test Execution Report): содержит результаты тестирования и сводку действий по выполнению тестов;
- **Сводный отчет о тестировании** (Test summary report): представляет собой документ высокого уровня, в котором резюмируются проведенные действия по тестированию, а также результаты тестирования;
- **Графики и метрики** (Graphs and Metrics): предназначены для мониторинга и управления процессом и продуктом. Это помогает без отклонений вести проект к намеченным целям. Метрики отвечают на разные вопросы. Важно решить, на какие вопросы вы хотите получить ответы;
- **Отчет о тестовых инцидентах** (Test incident report): содержит все инциденты, разрешенные или неразрешенные, обнаруженные во время тестирования;
- **Отчет о завершении тестирования** (Test closure report): содержит подробный анализ обнаруженных ошибок, удаленных ошибок и несоответствий, обнаруженных в программном обеспечении;
- **Отчет о статусе тестирования** (Test status report): предназначен для отслеживания статуса тестирования. Его готовят периодически или еженедельно. В нем указаны работы, выполненные до настоящего времени, и работы, которые еще не завершены;
- **Еженедельный отчет о статусе** (менеджер проекта для клиента): Weekly status report похож на отчет о статусе тестирования, но генерируется еженедельно;

- **Отчет об улучшении** (?Enhancement report): описание неявных/некритичных косвенных требований, которые не были учтены при планировании/реализации продукта, но несоблюдение, которых может вызвать неприятие у конечного потребителя;
- **Запрос на модификацию** (Modification Request): запрос клиента на изменение существующей функциональности;
- **Примечания к выпуску** (Release Note): примечания к выпуску будут отправлены клиенту, заказчику или заинтересованным сторонам вместе со сборкой. Он содержит список новых выпусков, исправления ошибок;
- **Руководство по установке / настройке** (Installation/configuration guide): это руководство помогает установить или настроить компоненты, из которых состоит система, и ее аппаратные и программные требования;
- **Руководство пользователя** (User guide): это руководство помогает конечному пользователю понять как пользоваться продуктом;
- **Различные документы требований.**

Источник:

- [Test Deliverables in Software Testing – Detailed Explanation](#)

Доп. материал:

- [Podlodka#223 – Техническая документация](#)
- [Пользовательская документация](#)

Политика качества и политика тестирования (Quality policy and Test policy)

Политика качества - это заявления, сделанные организациями для передачи своих долгосрочных стратегических целей, задач, видения в отношении производства и поставки качественного продукта. В этих политиках излагаются основные принципы организации, которые помогают им следовать установленным процедурам при разработке и тестировании продукта и постоянно стремиться к улучшению как продукта, так и процесса. Политика в области качества отражает основные ценности организации, что помогает понять их представления об атрибуте качества, о том, что для них означает качество, подходах, ключевых областях внимания и приоритетах при обеспечении качества для своих клиентов. Наличие четко определенной политики в области качества в соответствии со стандартами ISO 9001 является обязательным требованием для организации. Quality policy составляют CEO и Quality Manager.

При написании политики подчеркиваются следующие ключевые области:

- Внимание клиентов: потребности и ожидания клиентов являются важнейшим ключевым критерием, который помогает в достижении перспектив качества продукта. Таким образом, основное внимание следует уделять информированию о текущих и будущих потребностях клиента, а также их выполнению;
- Лидерство: должна быть в состоянии моделировать и создавать мотивационную и восторженную среду, чтобы извлекать максимум из каждого человека для достижения качества;
- Постоянное улучшение: должна стремиться к постоянному совершенствованию процедуры и подхода, чтобы улучшить качество;
- Процесс: должна отражать соблюдение и следование всем стандартным методам и процессам, что способствует повышению качества;
- Отношения: должна быть направлена на укрепление отношений с клиентом / покупателем;
- Создание и распространение осведомленности: информирование людей как внутри (персонал), так и за пределами (целевые клиенты) организации о стандартах, принципах и практиках, которым следует организация.

Кроме того, она должна обеспечивать прочную основу для достижения целей в области качества и периодически пересматриваться и обновляться, чтобы постоянно соответствовать существующим требованиям и ожиданиям. Вкратце можно сказать, что политика в области качества, определяемая организациями, действует как зеркало и отражает их виртуальный образ в реальном мире, на основе

которого внешние организации могут воспринимать и понимать свои основные принципы и обязательства по отношению к вкладу в качество.

Политика тестирования (test policy): Документ высокого уровня, описывающий принципы, подход и основные цели организации в отношении тестирования. (ISTQB)

Политика тестирования объясняет философию тестирования компании в целом и указывает направление, которого отдел тестирования должен придерживаться и которому следует следовать. Это должно относиться как к новым проектам, так и к проектам на поддержке. Установление старшими менеджерами соответствующей политики тестирования обеспечивает прочную основу, в которой могут работать специалисты-практики. Это поможет обеспечить максимальную стратегическую ценность каждого проекта.

Политика тестирования является частью политики качества, если она есть, в таких случаях политика качества разъяснит общую цель менеджмента в отношении качества. В ином случае этот документ верхний в иерархии тестовой документации. Политика тестирования содержит следующее:

- Обозначение преимуществ тестирования и коммерческой ценности, которые оправдывают [затраты на качество](#);
- Определяет [цели тестирования](#), такие как укрепление доверия, обнаружение дефектов и снижение рисков для качества;
- Описывает методы измерения эффективности тестирования и результативности выполнения задач тестирования;
- Обобщает [процессы](#), используемые при тестировании;
- Описывает для организации способы [улучшения процессов](#) тестирования.

Политика тестирования также должна включать действия по тестированию, необходимые для поддержки текущего проекта, а также разработки новых проектов.

Источники:

- [Quality Policy](#)
- [What is Test Policy? What does it contain?](#)

Доп. материал:

- [ISO 9001 Quality Policy - How to Write & Communicate your Policy Statement](#)

Стратегия тестирования (Test strategy)

Стратегия тестирования (test strategy): Высокоуровневое описание уровней тестирования, которые должны быть выполнены, и тестирования, входящего в эти уровни, для организации или программы из одного или более проектов. (ISTQB)

Стратегия тестирования - это статический документ высокого уровня, обычно разрабатываемый менеджером проекта. Это документ, который отражает подход к тестированию продукта и достижению целей, и дает четкое представление о том, что команда тестирования будет делать для всего проекта. Обычно он выводится из Спецификации бизнес-требований (BRS). Как только стратегия тестирования готова, группа тестирования начинает писать подробный план тестирования и продолжает дальнейшие этапы тестирования. В мире Agile некоторые компании не тратят время на подготовку плана тестирования из-за минимального времени для каждого выпуска, но они поддерживают документ стратегии тестирования. Это один из важных документов в test deliverables, которым команда тестирования делится с заинтересованными сторонами для лучшего понимания объема проекта, рисков, подходов к тестированию и других важных аспектов.

Содержание стратегии будет разным в зависимости от проекта, поэтому нет единого для всех шаблона. Можно найти эвристики в помощь, множество зарубежных статей на тему составления стратегии и некоторые общие пункты, которые чаще используются:

- **Обзор и объем** (Scope and overview): объем работ по тестированию (что тестировать и зачем тестировать) и обзор тестируемого продукта;
- **Подход к тестированию** (Test Approach):
 - Уровни тестирования (Test levels);
 - Виды тестирования (Test Types);
 - Роли и обязанности (Roles and responsibilities);
 - Требования к окружениям (Environment requirements);
- **Инструменты тестирования** (Testing tools): инструменты, необходимые для проведения тестов (TMS, багтрекинговая система, стек автоматизации);
- **Отраслевые стандарты**, которым необходимо следовать (Industry standards to follow): В этом разделе описывается отраслевой стандарт для производства высококачественной системы, которая соответствует ожиданиям клиентов или превосходит их. Обычно менеджер проекта определяет модели и процедуры тестирования, которым необходимо следовать для достижения целей проекта;
- **Результаты тестирования** (Test deliverables): документация, которую необходимо создать до, во время и по окончании тестирования;
- **Метрики тестирования** (Testing metrics): метрики, которые следует использовать в проекте для анализа статуса проекта;
- **Матрица отслеживания требований** (RTM);
- **Риски и способы их снижения** (Risk and mitigation): все риски тестирования и план по их снижению;
- **Инструмент отчетности** (Reporting tool): как будут отслеживаться дефекты и проблемы;
- **Результаты тестов** (Test Summary): виды сводных отчетов о тестах, которые будут создаваться, с указанием периодичности. Сводные отчеты о тестах будут генерироваться ежедневно, еженедельно или ежемесячно, в зависимости от критичности проекта.

Источники:

- [The Complete Guide To Writing Test Strategy](#)

Доп. материал:

- [Большая качественная подборка материалов по теме](#)
- [Practical test strategy using heuristics](#)
- Примеры: [раз](#), [два](#), [три](#)

План тестирования (Test plan)

“План тестирования (test plan): Документ, описывающий цели, подходы, ресурсы и график запланированных тестовых активностей. Он определяет объекты тестирования, свойства для тестирования, задания, ответственных за задания, степень независимости каждого тестировщика, тестовое окружение, метод проектирования тестов, определяет используемые критерии входа и критерии выхода и причины их выбора, а также любые риски, требующие планирования на случай чрезвычайных обстоятельств.” (IEEE 829)

В то время как стратегия излагает общие принципы или теорию, план детально описывает практические аспекты того, как проект будет протестирован в реальности.

Хотя есть рекомендации по составлению тест плана (IEEE 829 ([1](#), [2](#)), [RUP](#)), нет единственно правильного шаблона или формата для написания тест-планов. В обзорных статьях можно встретить и свои варианты:

Такой:

- Перечень планируемых тестовых активностей ([Test Activities](#));
- Тестовая логистика ([Test Logistics](#));
- Необходимые ресурсы ([Resources](#));
- Необходимые коммуникации ([Your Support Network](#));
- Оценки трудозатрат ([Estimates](#));
- Зависимости и риски ([Dependencies, Risks and Assumptions](#));

- Порядок обсуждений и отчетности в процессе работы (Communication, Commitment and Progress Reporting);

Или:

- Какие ресурсы требуются и когда;
- Когда задачи нужно начинать и заканчивать, и кто их будет выполнять;
- Навыки, необходимые для выполнения задач;
- Инструменты и технологии, поддерживающие план;
- Результаты и когда они будут доставлены;
- Затраты на усилия и необходимые ресурсы;
- Процесс продвижения проекта / процесса по стадиям;
- Риски, угрожающие доставке.

В какой-то момент можно заметить, что все они предлагают плюс-минус похожую структуру и пункты, а итоговый вариант всё равно будет уникальным для каждого конкретного проекта. Весомая часть литературы по данной теме предполагает работу по водопадной модели разработки и эта информация не так актуальна в наше время. Это не значит, что в гибких методологиях не бывает тест-планов. Даже в Agile необходимо предварительное планирование для структурирования работы, распределения ресурсов и планирования - по крайней мере, на высоком уровне - процесса выпуска на ближайшие месяцы. Но итерация за итерацией, а часто и день за днем, общий план постоянно корректируется с учетом событий и новой информации, которая появляется на свет. Планирование - это непрерывное обучение, а не задача с конечным результатом.

В гибких методологиях всё чаще говорят о концепции одностороннего тест-плана, а в случае необходимости дополнений и уточнений просто создаются ссылки на внешние страницы/документы. Такой план может быть и в гугл-таблицах, в виде дашборда, mind map, и как вам самим вздумается. Тест-план призван отвечать на те вопросы, ради которых его создают. Порой весомую часть пользы от данной активности можно получить на этапе самого планирования и составления плана, а не от самого документа. Если команда понимает, что никакой практической "боли" этот документ и его создание не решает, на него нет времени, то можно прекрасно обойтись и без его формализации, т.к. в некоей словесной форме он всё равно будет существовать всегда.

"В зависимости от размеров команды, сложности продукта, количества зависимостей и строгости критериев качества эти вопросы могут быть иными. Если процесс тестирования имеет большое количество зависимостей, например разные команды должны выполнять разные этапы тестирования в строго определенном порядке - это необходимо фиксировать. Без этого ты не только не сможешь планировать работу команд, но и несколько раз выстрелишь себе в ногу, когда команды будут блокировать друг друга из-за того, что заранее не проговорили зависимости. Чем более комплексным является объект тестирования (и как результат само тестирование), тем более подробного описания требует методология тестирования, применяемые подходы и практики - просто за счёт увеличения объема того, что необходимо проверить. Без этого сложно оценивать объемы работ, давать эстимейты и строить планы по релизам. Чем более точно и строго необходимо оценивать уровень качества, тем более детально должны быть описаны критерии прохождения тестирования, ключевые метрики и [quality gate](#)'ы. Поэтому что без их формализации нельзя будет однозначно оценить результаты тестирования. Люди, находящиеся за пределами команды тестирования (а иногда и команды разработки в целом) хотят понимать, что вообще происходит на этапе тестирования и как обеспечивается качество продукта. Иногда это связано с регуляторикой отрасли, иногда для согласования объемов работы с заказчиком, иногда из-за высокой степени рисков или просто потому, что работа этих людей напрямую зависит от результатов процесса обеспечения качества." (c) [Shoo and Endless Agony: What's the plan?](#)

Виды тест-планов:

- **Мастер Тест-План ([Master Test Plan](#))**: "Главный план тестирования (*master test plan, project test plan*): План тестирования, обычно охватывающий несколько уровней тестирования." (*ISTQB*). Это может

быть как единственный базовый план, так и главный в иерархии нескольких планов, самый статичный и высокоуровневый. Нужен когда:

- продукт имеет множество релизов или итераций, между которыми сохраняется общая информация, которую нет смысла повторять;
- различные тестовые команды работают над одним продуктом, выполняя различные задачи, которые необходимо объединить в рамках одного документа;
- **Детальный Тест-план** (Phase Test plan): “Уровневый план тестирования (*level test plan*): План тестирования, обычно относящийся к одному уровню тестирования.” (ISTQB). Детальный план составляется на каждый релиз/итерацию или для каждой команды в рамках проекта и является динамическим, т.е. может претерпевать изменения по необходимости. Его основная цель - кратко и доходчиво отразить задачи тестирования. Детальных планов может быть несколько для отдельных модулей ПО или команд тестирования. Кроме того, могут быть созданы планы для отдельных уровней тестирования (Level Test Plan) или видов тестирования. В Agile проектах могут быть планы итерационного тестирования ([iteration testing plans](#)) для каждой итерации;
- **План приемочных испытаний** (Acceptance Test Plan, ПСИ): план приемочного тестирования отличают от обычного плана тестирования факторы, которые приводят к принятию бизнес-решения. План приемочного тестирования - это один из жизненно важных документов, который содержит руководство по выполнению приемочного тестирования для конкретного проекта. Пишется на основе бизнес-требований (Business Requirements). Ревью этого плана обычно выполняется by Managers/Business Analysts/Customers.

Источники:

- [Leadership in test: test planning](#)
- [Acceptance Testing Documentation With Real-Time Scenarios](#)

Доп. материал:

- [Тест-план не для галочки, или 8 вопросов к заказчику на старте проекта](#)
- [Blog: What Should A Test Plan Contain?](#)
- [The One Page Test Plan](#)
- [TEST PLAN: What is, How to Create \(with Example\)](#)
- Примеры: [раз](#), [два](#); [Acceptance Test Plan Template](#)

Тестовый сценарий (Test scenario)

Сценарий выполнения (*test scenario*): См. спецификация процедуры тестирования. (ISTQB)

Спецификация процедуры тестирования (*test procedure specification*): Документ, описывающий последовательность действий при выполнении теста. Также известен как ручной сценарий тестирования. (IEEE 829) См. также спецификация теста

Спецификация теста (*test specification*): Документ, состоящий из спецификации проектирования теста, спецификации тестовых сценариев и/или спецификации процедуры тестирования (ISTQB)

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. Иными словами, это последовательность шагов, которые пользователь может предпринять, чтобы использовать ваше программное обеспечение во время тестирования. Сценарии тестирования должны учитывать все возможные способы выполнения задачи (функции) и охватывать как положительные, так и отрицательные тестовые примеры, потому что конечные пользователи могут не обязательно предпринимать шаги, которые вы от них ожидаете. Используя тестовые сценарии, мы оцениваем работу приложения с точки зрения конечного пользователя. Фактически при успешном прохождении всего тестового сценария мы можем сделать заключение о том, что продукт может выполнять ту или иную возложенную на него функцию.

Как писать сценарии:

- Тщательно ознакомьтесь с требованиями (Спецификация бизнес-требований (BRS), Спецификация требований к программному обеспечению (SRS), Спецификация функциональных требований (FRS)) тестируемой системы (SUT), use cases, книгами, руководствами и т. д.;
- Для каждого требования выясните, как пользователь может использовать программное обеспечение всеми возможными способами;
- Составьте список сценариев тестирования для каждой функции тестируемого приложения (AUT);
- Создайте матрицу прослеживаемости и свяжите все сценарии с требованиями. Это позволит вам определить, сопоставлены ли все требования с тестовыми сценариями или нет;
- Отправьте сценарии тестирования руководителю, чтобы он рассмотрел и оценил их. Даже сценарии тестирования дополнительно проверяются всеми заинтересованными сторонами.

Не стоит путать Test scenario с **Test Suite** (набор тестов, тест-свит).

Набор тестов (test suite): Комплект тестовых наборов для исследуемого компонента или системы, в котором обычно постуловие одного теста используется в качестве предусловия для последующего. (ISTQB)

Test Suite - это некоторый набор формализованных Test case, объединенных между собой по общему логическому признаку, которые позволяют проверить один из частей или 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.

Источники:

- [How To Create Test Scenarios With Examples](#)
- [Каких ответов я жду на собеседовании по тестированию](#)

Доп. материал:

- [Test Scenarios Registration Form](#)
- [Test Scenarios of GMail](#)
- [Шаблон сценария](#)

Тест-кейс (Test case)

Тестовый сценарий (test case): Набор входных значений, предусловий выполнения, ожидаемых результатов и постусловий выполнения, разработанный для определенной цели или тестового условия, таких как выполнения определенного пути программы или же для проверки соответствия определенному требованию. (IEEE 610)

Тестовый сценарий высокого уровня (high level test case): Тестовый сценарий без конкретных (уровня реализации) значений входных данных и ожидаемых результатов. Использует логические операторы, а экземпляры реальных значений еще не определены и/или доступны. (ISTQB)

Тестовый сценарий низкого уровня (low level test case): Тестовый сценарий с конкретными (уровня реализации) значениями входных данных и ожидаемых результатов. Логические операторы из тестовых сценариев высокого уровня заменяются реальными значениями, которые соответствуют целям этих логических операторов. (ISTQB)

Test case (тест-кейс, тестовый пример/случай) - 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.

Содержание тест-кейса:

- Идентификатор набора тестов (**Test Suite ID**): Идентификатор набора тестов, в которых входит этот кейс;

- Идентификатор тестового кейса (**Test Case ID**): Идентификатор самого кейса;
- Заголовок кейса (**Test Case Summary**): Краткое и емкое название проводимой проверки;
- Связанное требование (**Related Requirement**): Идентификатор требования, к которому относится / отслеживается данный тестовый пример;
- Предварительные условия (**Prerequisites**): Любые предпосылки или предварительные условия, которые должны быть выполнены перед выполнением теста;
- Шаги выполнения (**Test Script / Procedure**): Шаги выполнения теста;
- Тестовые данные (**Test Data**): Тестовые данные или ссылки на тестовые данные, которые должны использоваться при проведении теста;
- Ожидаемый результат (**Expected Result**): результат, который мы ожидаем получить после выполнения шагов теста;
- Статус пройден или не пройден (**Status**): Другие статусы могут быть «Не выполнено», если тестирование не проводится, и «Заблокировано», если тестирование заблокировано;
- Заметки (**Remarks**): Любые комментарии к тесту или выполнению теста;
- Создано (**Created By**): Имя автора тестового примера;
- Дата создания (**Date of Creation**): Дата создания тестового примера (опционально модификации);
- Выполнено (**Executed By**): Имя человека, выполнившего тест;
- Дата выполнения (**Date of Execution**): Дата выполнения теста;
- Тестовое окружение (**Test Environment**): оборудование / программное обеспечение / сеть, в которых выполнялся тест, т.е. все необходимые сведения об окружении, чтобы можно было воспроизвести полученный результат.

В иностранной литературе часто делят кейсы на две категории:

- **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 smoke. 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.

Нужно ли вообще писать кейсы? Ответ тот же, что и для любого документа - если написание кейсов решает определенную задачу и это обоснованно, то писать. Если вы один, не путаетесь в небольшом проекте, пользуетесь чек листами/mind map/.., можете и без TMS/test runs reports наглядно предоставлять актуальные сведения о протестированности/качестве заинтересованным лицам, то не писать.

Может ли быть несколько ожидаемых результатов? Может, если это необходимо, но сразу после каждого шага.

Можно ли объединять позитивные и негативные тест-кейсы? Позитивные можно, негативные нельзя, поскольку сложно будет понять, что именно влияет на результат.

Источники:

- [Test Case](#)

Доп. материал:

- [Тест-кейсы: полная лекция из ШНАТ](#)
- [Составление тест-кейсов](#)
- [12 характеристик высокоэффективных тестов](#)
- [Blog: Evaluating Test Cases, Checks, and Tools](#)

- [How to write Test Cases for a Login Page](#)
- Примеры: [раз](#), [два](#)

Чек-лист (Check List)

Контрольный список/лист проверок - это список проверок, которые помогают тестировщику протестировать приложение или отдельные функции. Основная цель чеклиста состоит в том, чтобы вы не забыли проверить всё, что планировали. Классический чеклист состоит из:

- 1-й столбец: заголовки тест-кейсов, структурированные по разделам/функционалу, или любые определенные составителем пункты;
- 2-й столбец для отметки: пусто (еще не проверялось)/успех/ошибка;
- 3-й столбец опционально под заметки.

Если привести простой пример из жизни, то когда вы планируете дела на день или готовитесь к важному мероприятию, вы записываете, или хотя бы составляете мысленно список дел. Это и есть пример чек-листа.

Чек-лист не обязательно является некоторой заменой тест-кейсов, это более глобальная сущность, в виде которой можно записывать множество планов и предстоящих действий: критерии начала и окончания тестирования, проверки перед началом каждой фазы, действия по их завершении, подспорье при исследовательском тестировании, накидать проверок с mind map функционала продукта, шеринг опыта с коллегами и т.п.

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 создание и поддержка кейсов требует времени, сил и является рутиной. Помимо прочего, очевидно, тест-кейс часто подразумевает только один конкретный тест, когда в чек-листе подразумевается целый перечень разных проверок.

Сила чек-листа в том, что он простой. Там нет глубокой детализации, это просто памятка. К тому же, он довольно наглядный с точки зрения отчетности. Минус в том, что другому человеку может быть сложно вникнуть в суть проверок без деталей и шагов. Чек-листы стали популярнее с приходом гибких моделей разработки, когда писать детальные кейсы может не быть времени и смысла, т.к. всё меняется слишком быстро, к тому же команда может быть небольшой и расписывать кейсы просто не для кого.

Доп. материал:

- [Чек-листы: полная лекция](#)
- [Составление чек-листов](#)
- Примеры: [раз](#)

Баг-репорт (Defect/bug report)

Отчет о дефекте (defect report): Документ, содержащий отчет о любом недостатке в компоненте или системе, который может привести компонент или систему к невозможности выполнить требуемую функцию. (IEEE 829)

«Смысл написания отчета о проблеме (отчета об ошибке) состоит в том, чтобы исправить ошибки» - Джем Канер. Если тестировщик неправильно сообщает об ошибке, то программист, скорее всего, отклонит эту ошибку, заявив, что она невоспроизводима. Или потратит кучу лишнего времени на то, чтобы сделать вашу работу за вас. Едва ли такой тестировщик будет выгоден бизнесу, приятен коллегам и долго задержится на своем месте.

Главное при написании отчета - он должен быть сразу и однозначно понят читающим, а дефект однозначно воспроизведен по указанным шагам в указанном окружении.

Основные поля баг-репорта:

- Уникальный идентификатор (ID);
- Описание (Summary): краткое, емкое и понятное описание ошибки;

- Окружение (**Environment**): ссылка на билд/коммит/версия ПО и всего окружения;
- Шаги воспроизведения (**Steps to reproduce**): полный перечень шагов для воспроизведения;
- Ожидаемый результат (**Expected result**): какой результат должен был быть без ошибки;
- Фактический результат (**Actual result**): какой результат получился на самом деле;
- Вложения (**Attachments**): логи, скриншоты, видео - всё что необходимо для понимания ошибки.

Дополнительные:

- Предварительные условия (Prerequisites);
- Тестовые данные (Test Data);
- Серьезность дефекта (Defect Severity);
- Комментарии (Remarks);
- Проект (Project);
- Продукт (Product);
- Версия релиза (Release Version);
- Модуль (Module);
- Обнаружено в версии (Detected Build Version);
- Вероятность возникновения дефекта (Defect Probability);
- Приоритет дефекта (Defect Priority);
- Автор отчета (Reported By);
- Назначено на (Assigned To);
- Статус (Status);
- Fixed Build Version.

В случаях использования TMS поля будут настроены лицом/менеджером и в зависимости от размеров проекта могут быть пункты вроде milestone, epic, feature и т.п.

Помимо прочего, баг-репорты могут создаваться не только тестировщиками, но и любыми членами команды, приходить от пользователей или техподдержки. Во втором случае необходимо будет воспроизвести ошибку, составить баг-репорт по всем правилам или дополнить присланный, затем провести ретроспективу на тему того, как ошибка попала в прод и как этого избежать в будущем.

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

- В одном отчете один баг;
- Воспроизведите его 2-3 раза;
- Убедитесь, что используете актуальную версию ПО и окружения;
- Проверьте по поиску багтрекинговой системы наличие отчета о таком же дефекте;
- Локализуйте ошибку, чтобы выяснить ее первопричину;
- Напишите подробные шаги и полное окружение для воспроизведения ошибки;
- Напишите хорошее summary дефекта по формуле “Что? Где? При каких условиях?”;
- Следите за словами в процессе написания сообщения об ошибке, они не должны обвинять, оскорблять людей, содергать какую-либо точку зрения по поводу произошедшего. В общем, только факты по делу;
- Проиллюстрируйте проблему с помощью правильных скриншотов, видео и логов;
- Перед отправкой перепроверьте ваш отчет об ошибке. А потом еще раз;

Источники:

- [How To Write Good Bug Report](#)

Требования (Requirements)

Требование (requirement): Условия или возможности, необходимые пользователю для решения определенных задач или достижения определенных целей, которые должны быть достигнуты для выполнения контракта, стандартов, спецификации, или других формальных документов. (IEEE 610)

Требования являются отправной точкой для определения того, что проектная команда будет проектировать, реализовывать и тестировать. Вне зависимости от того, какая модель разработки ПО используется на проекте, чем позже будет обнаружена проблема, тем сложнее и дороже будет ее решение. Если проблема в требованиях будет выяснена на начальной стадии, ее решение может свестись к исправлению пары слов в тексте, в то время как недоработка, вызванная пропущенной проблемой в требованиях и обнаруженная на стадии эксплуатации, может даже полностью уничтожить проект.

Источники и пути выявления требований

Требования начинают свою жизнь на стороне заказчика. Их сбор (gathering) и выявление (elicitation) осуществляются с помощью следующих основных техник:

- **Интервью.** Самый универсальный путь выявления требований, заключающийся в общении проектного специалиста (как правило, бизнес-аналитика) и представителя заказчика (или эксперта, пользователя и т.д.). Интервью может проходить в классическом понимании этого слова (беседа в виде «вопрос — ответ»), в виде переписки и т.п. Главным здесь является то, что ключевыми фигурами выступают двое — интервьюируемый и интервьюер (хотя это и не исключает наличия «аудитории слушателей», например, в виде лиц, поставленных в копию переписки).
- **Работа с фокусными группами.** Может выступать как вариант «расширенного интервью», где источником информации является не одно лицо, а группа лиц (как правило, представляющих собой целевую аудиторию, и/или обладающих важной для проекта информацией, и/или уполномоченных принимать важные для проекта решения).
- **Анкетирование.** Этот вариант выявления требований вызывает много споров, т.к. при неверной реализации может привести к нулевому результату при объемных затратах. В то же время при правильной организации анкетирование позволяет автоматически собрать и обработать огромное количество ответов от огромного количества респондентов. Ключевым фактором успеха является правильное составление анкеты, правильный выбор аудитории и правильное преподнесение анкеты.
- **Семинары и мозговой штурм.** Семинары позволяют группе людей очень быстро обменяться информацией (и наглядно продемонстрировать те или иные идеи), а также хорошо сочетаются с интервью, анкетированием, прототипированием и моделированием — в том числе для обсуждения результатов и формирования выводов и решений. Мозговой штурм может проводиться и как часть семинара, и как отдельный вид деятельности. Он позволяет за минимальное время сгенерировать большое количество идей, которые в дальнейшем можно не спеша рассмотреть с точки зрения их использования для развития проекта.
- **Наблюдение.** Может выражаться как в буквальном наблюдении за некоторыми процессами, так и во включении проектного специалиста в эти процессы в качестве участника. С одной стороны, наблюдение позволяет увидеть то, о чём (по совершенно различным соображениям) могут умолчать интервьюируемые, анкетируемые и представители фокусных групп, но с другой — отнимает очень много времени и чаще всего позволяет увидеть лишь часть процессов.
- **Прототипирование.** Состоит в демонстрации и обсуждении промежуточных версий продукта (например, дизайн страниц сайта может быть сначала представлен в виде картинок, и лишь затем сверстан). Это один из лучших путей поиска единого понимания и уточнения требований, однако он может привести к серьезным дополнительным затратам при отсутствии специальных инструментов (позволяющих быстро создавать прототипы) и слишком раннем применении (когда требования еще не стабильны, и высока вероятность создания прототипа, имеющего мало общего с тем, что хотел заказчик).
- **Анализ документов.** Хорошо работает тогда, когда эксперты в предметной области (временно) недоступны, а также в предметных областях, имеющих общепринятую устоявшуюся регламентирующую документацию. Также к этой технике относится и просто изучение документов, регламентирующих бизнес-процессы в предметной области заказчика или в конкретной организации, что позволяет приобрести необходимые для лучшего понимания сути проекта знания.
- **Моделирование процессов и взаимодействий.** Может применяться как к «бизнес-процессам и взаимодействиям» (например: «договор на закупку формируется отделом закупок, визируется бухгалтерией и юридическим отделом...»), так и к «техническим процессам и взаимодействиям»

(например: «платежное поручение генерируется модулем “Бухгалтерия”, шифруется модулем “Безопасность” и передаётся на сохранение в модуль “Хранилище”»). Данная техника требует высокой квалификации специалиста по бизнес-анализу, т.к. сопряжена с обработкой большого объема сложной (и часто плохо структурированной) информации.

- **Самостоятельное описание.** Является не столько техникой выявления требований, сколько техникой их фиксации и формализации. Очень сложно (и даже нельзя!) пытаться самому «придумать требования за заказчика», но в спокойной обстановке можно самостоятельно обработать собранную информацию и аккуратно оформить ее для дальнейшего обсуждения и уточнения.

Уровни и типы требований

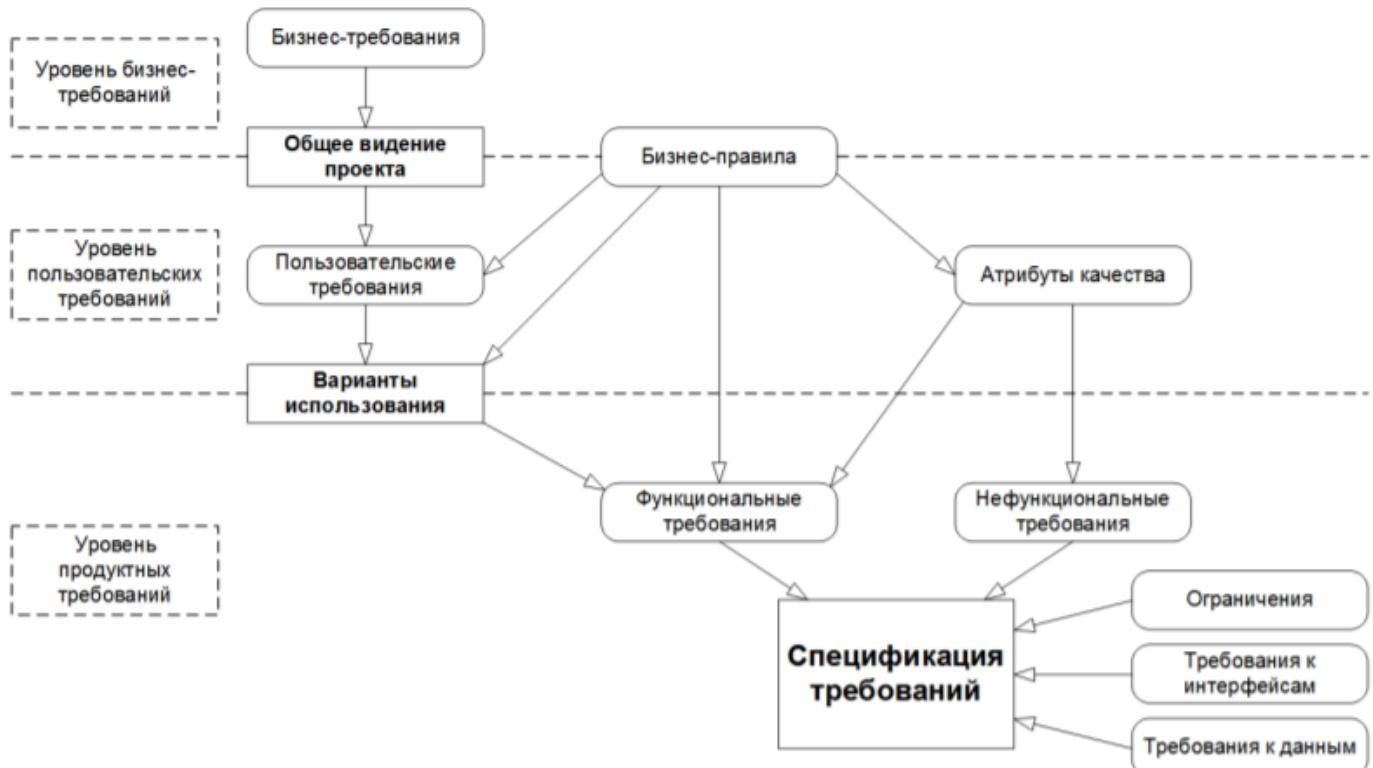


Рисунок 2.2.е — Уровни и типы требований

- **Бизнес-требования** (business requirements) выражают цель, ради которой разрабатывается продукт (зачем вообще он нужен, какая от него ожидается польза, как заказчик с его помощью будет получать прибыль). Результатом выявления требований на этом уровне является общее видение (vision and scope) — документ, который, как правило, представлен простым текстом и таблицами. Здесь нет детализации поведения системы и иных технических характеристик, но вполне могут быть определены приоритеты решаемых бизнес-задач, риски и т.п. Несколько простых, изолированных от контекста и друг от друга примеров бизнес-требований:
 - Нужен инструмент, в реальном времени отображающий наиболее выгодный курс покупки и продажи валюты;
 - Необходимо в два-три раза повысить количество заявок, обрабатываемых одним оператором за смену;
 - Нужно автоматизировать процесс выписки товарно-транспортных накладных на основе договоров.
- **Пользовательские требования** (user requirements) описывают задачи, которые пользователь может выполнять с помощью разрабатываемой системы (реакцию системы на действия пользователя, сценарии работы пользователя). Поскольку здесь уже появляется описание поведения системы, требования этого уровня могут быть использованы для оценки объема работ, стоимости проекта, времени разработки и т.д. Пользовательские требования оформляются в виде вариантов использования (use cases), пользовательских историй (user stories), пользовательских сценариев (user

scenarios). Несколько простых, изолированных от контекста и друг от друга примеров пользовательских требований:

- При первом входе пользователя в систему должно отображаться лицензионное соглашение;
- Администратор должен иметь возможность просматривать список всех пользователей, работающих в данный момент в системе;
- При первом сохранении новой статьи система должна выдавать запрос на сохранение в виде черновика или публикацию.
- **Бизнес-правила** (business rules) описывают особенности принятых в предметной области (и/или непосредственно у заказчика) процессов, ограничений и иных правил. Эти правила могут относиться к бизнес-процессам, правилам работы сотрудников, нюансам работы ПО и т.д. Несколько простых, изолированных от контекста и друг от друга примеров бизнес-правил:
 - Никакой документ, просмотренный посетителями сайта хотя бы один раз, не может быть отредактирован или удален;
 - Публикация статьи возможна только после утверждения главным редактором;
 - Подключение к системе извне офиса запрещено в нерабочее время.
- **Атрибуты качества** (quality attributes) расширяют собой нефункциональные требования и на уровне пользовательских требований могут быть представлены в виде описания ключевых для проекта показателей качества (свойств продукта, не связанных с функциональностью, но являющихся важными для достижения целей создания продукта — производительность, масштабируемость, восстанавливаемость). Атрибутов качества очень много, но для любого проекта реально важными является лишь некоторое их подмножество. Несколько простых, изолированных от контекста и друг от друга примеров атрибутов качества:
 - Максимальное время готовности системы к выполнению новой команды после отмены предыдущей не может превышать одну секунду;
 - Внесенные в текст статьи изменения не должны быть потеряны при нарушении соединения между клиентом и сервером;
 - Приложение должно поддерживать добавление произвольного количества неиероглифических языков интерфейса.
- **Функциональные требования** (functional requirements) описывают поведение системы, т.е. ее действия (вычисления, преобразования, проверки, обработку и т.д.). В контексте проектирования функциональные требования в основном влияют на дизайн системы. Стоит помнить, что к поведению системы относится не только то, что система должна делать, но и то, что она не должна делать (например: «приложение не должно выгружать из оперативной памяти фоновые документы в течение 30 минут с момента выполнения с ними последней операции»). Несколько простых, изолированных от контекста и друг от друга примеров функциональных требований:
 - В процессе инсталляции приложение должно проверять остаток свободного места на целевом носителе;
 - Система должна автоматически выполнять резервное копирование данных ежедневно в указанный момент времени;
 - Электронный адрес пользователя, вводимый при регистрации, должен быть проверен на соответствие требованиям RFC822.
- **Нефункциональные требования** (non-functional requirements) описывают свойства системы (удобство использования, безопасность, надежность, расширяемость и т.д.), которыми она должна обладать при реализации своего поведения. Здесь приводится более техническое и детальное описание атрибутов качества. В контексте проектирования нефункциональные требования в основном влияют на архитектуру системы. Несколько простых, изолированных от контекста и друг от друга примеров нефункциональных требований:
 - При одновременной непрерывной работе с системой 1000 пользователей, минимальное время между возникновением сбоев должно быть более или равно 100 часов;
 - Ни при каких условиях общий объем используемой приложением памяти не может превышать 2 ГБ;
 - Размер шрифта для любой надписи на экране должен поддерживать настройку в диапазоне от 5 до 15 пунктов.

Следующие требования в общем случае могут быть отнесены к нефункциональным, однако их часто выделяют в отдельные подгруппы (здесь для простоты рассмотрены лишь три таких подгруппы, но их может быть и гораздо больше; как правило, они проистекают из атрибутов качества, но высокая степень детализации позволяет отнести их к уровню требований к продукту).

- **Ограничения** (*limitations, constraints*) представляют собой факторы, ограничивающие выбор способов и средств (в том числе инструментов) реализации продукта. Несколько простых, изолированных от контекста и друг от друга примеров ограничений:
 - Все элементы интерфейса должны отображаться без прокрутки при разрешениях экрана от 800x600 до 1920x1080;
 - Не допускается использование Flash при реализации клиентской части приложения;
 - Приложение должно сохранять способность реализовывать функции с уровнем важности «критический» при отсутствии у клиента поддержки JavaScript.
- **Требования к интерфейсам** (*external interfaces requirements*) описывают особенности взаимодействия разрабатываемой системы с другими системами и операционной средой. Несколько простых, изолированных от контекста и друг от друга примеров требований к интерфейсам:
 - Обмен данными между клиентской и серверной частями приложения при осуществлении фоновых AJAX-запросов должен быть реализован в формате JSON;
 - Протоколирование событий должно вестись в журнале событий операционной системы;
 - Соединение с почтовым сервером должно выполняться согласно RFC3207 («SMTP over TLS»).
- **Требования к данным** (*data requirements*) описывают структуры данных (и сами данные), являющиеся неотъемлемой частью разрабатываемой системы. Часто сюда относят описание базы данных и особенностей её использования. Несколько простых, изолированных от контекста и друг от друга примеров требований к данным:
 - Все данные системы, за исключением пользовательских документов, должны храниться в БД под управлением СУБД MySQL, пользовательские документы должны храниться в БД под управлением СУБД MongoDB;
 - Информация о кассовых транзакциях за текущий месяц должна храниться в операционной таблице, а по завершении месяца переноситься в архивную;
 - Для ускорения операций поиска по тексту статей и обзоров должны быть предусмотрены полнотекстовые индексы на соответствующих полях таблиц.

Свойства качественных требований (требования к самим требованиям)

- **Завершенность** (*completeness*). Требование является полным и законченным с точки зрения представления в нем всей необходимой информации, ничто не пропущено по соображениям «это и так всем понятно». Типичные проблемы с завершенностью:
 - Отсутствуют нефункциональные составляющие требования или ссылки на соответствующие нефункциональные требования (например: «пароли должны храниться в зашифрованном виде» — каков алгоритм шифрования?);
 - Указана лишь часть некоторого перечисления (например: «экспорт осуществляется в форматы PDF, PNG и т.д.» — что мы должны понимать под «и т.д.»?);
 - Приведённые ссылки неоднозначны (например: «см. выше» вместо «см. раздел 123.45.b»).
- **Атомарность, единичность** (*atomicity*). Требование является атомарным, если его нельзя разбить на отдельные требования без потери завершенности и оно описывает одну и только одну ситуацию. Типичные проблемы с атомарностью:
 - В одном требовании, фактически, содержится несколько независимых (например: «кнопка “Restart” не должна отображаться при остановленном сервисе, окно “Log” должно вмещать не менее 20-ти записей о последних действиях пользователя» — здесь зачем-то в одном предложении описаны совершенно разные элементы интерфейса в совершенно разных контекстах);
 - Требование допускает разночтение в силу грамматических особенностей языка (например: «если пользователь подтверждает заказ и редактирует заказ или откладывает заказ, должен выдаваться запрос на оплату» — здесь описаны три разных случая, и это требование стоит

- разбить на три отдельных во избежание путаницы). Такое нарушение атомарности часто влечёт за собой возникновение противоречивости;
- В одном требовании объединено описание нескольких независимых ситуаций (например: «когда пользователь входит в систему, ему должно отображаться приветствие; когда пользователь вошел в систему, должно отображаться имя пользователя; когда пользователь выходит из системы, должно отображаться прощание» — все эти три ситуации заслуживают того, чтобы быть описанными отдельными и куда более детальными требованиями).
 - **Непротиворечивость, последовательность** (consistency). Требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам. Типичные проблемы с непротиворечивостью:
 - Противоречия внутри одного требования (например: «после успешного входа в систему пользователя, не имеющего права входить в систему...» — тогда как он успешно вошёл в систему, если не имел такого права?);
 - Противоречия между двумя и более требованиями, между таблицей и текстом, рисунком и текстом, требованием и прототипом и т.д. (например: «712.а Кнопка “Close” всегда должна быть красной» и «36452.х Кнопка “Close” всегда должна быть синей» — так всё же красной или синей?);
 - Использование неверной терминологии или использование разных терминов для обозначения одного и того же объекта или явления (например: «в случае, если разрешение окна составляет менее 800x600...» — разрешение есть у экрана, у окна есть размер).
 - **Недвусмысленность** (unambiguousness, clarity). Требование должно быть описано без использования жаргона, неочевидных аббревиатур и расплывчатых формулировок, должно допускать только однозначное объективное понимание и быть атомарным в плане невозможности различной трактовки сочетания отдельных фраз. Типичные проблемы с недвусмысленностью:
 - Использование терминов или фраз, допускающих субъективное толкование (например: «приложение должно поддерживать передачу больших объемов данных» — насколько «больших»?) Вот лишь небольшой перечень слов и выражений, которые можно считать верными признаками двусмысленности: адекватно (adequate), быть способным (be able to), легко (easy), обеспечивать (provide for), как минимум (as a minimum), быть способным (be capable of), эффективно (effectively), своевременно (timely), применимо (as applicable), если возможно (if possible), будет определено позже (to be determined, TBD), по мере необходимости (as appropriate), если это целесообразно (if practical), но не ограничиваясь (but not limited to), быть способно (capability of), иметь возможность (capability to), нормально (normal), минимизировать (minimize), максимизировать (maximize), оптимизировать (optimize), быстро (rapid), удобно (user-friendly), просто (simple), часто (often), обычно (usual), большой (large), гибкий (flexible), устойчивый (robust), по последнему слову техники (state-of-the-art), улучшенный (improved), результативно (efficient). Вот утрированный пример требования, звучащего очень красиво, но совершенно нереализуемого и непонятного: «В случае необходимости оптимизации передачи больших файлов система должна эффективно использовать минимум оперативной памяти, если это возможно»;
 - Использование неочевидных или двусмысленных аббревиатур без расшифровки (например: «доступ к ФС осуществляется посредством системы прозрачного шифрования» и «ФС предоставляет возможность фиксировать сообщения в их текущем состоянии с хранением истории всех изменений» — ФС здесь обозначает файловую систему? Точно? А не какой-нибудь «Фиксатор Сообщений»?);
 - Формулировка требований из соображений, что нечто должно быть всем очевидно (например: «Система конвертирует входной файл из формата PDF в выходной файл формата PNG» — и при этом автор считает совершенно очевидным, что имена файлов система получает из командной строки, а многостраничный PDF конвертируется в несколько PNG-файлов, к именам которых добавляется «page-1», «page-2» и т.д.). Эта проблема перекликается с нарушением корректности.
 - **Выполнимость** (feasibility). Требование должно быть технологически выполнимым и реализуемым в рамках бюджета и сроков разработки проекта. Типичные проблемы с выполнимостью:

- Так называемое «озолочение» (gold plating) — требования, которые крайне долго и/или дорого реализуются и при этом практически бесполезны для конечных пользователей (например: «настройка параметров для подключения к базе данных должна поддерживать распознавание символов из жестов, полученных с устройств трёхмерного ввода»).
 - Технически нереализуемые на современном уровне развития технологий требования (например: «канализ договоров должен выполняться с применением искусственного интеллекта, который будет выносить однозначное корректное заключение о степени выгоды от заключения договора»).
 - В принципе нереализуемые требования (например: «система поиска должна заранее предусматривать все возможные варианты поисковых запросов и кэшировать их результаты»).
- **Обязательность, нужность** (obligatoriness) и актуальность (up-to-date). Если требование не является обязательным к реализации, оно должно быть просто исключено из набора требований. Если требование нужное, но «не очень важное», для указания этого факта используется указание приоритета (см. «проранжированность по...»). Также исключены (или переработаны) должны быть требования, утратившие актуальность. Типичные проблемы с обязательностью и актуальностью:
 - Требование было добавлено «на всякий случай», хотя реальной потребности в нём не было и нет;
 - Требованию выставлены неверные значения приоритета по критериям важности и/или срочности;
 - Требование устарело, но не было переработано или удалено.
- **Прослеживаемость** (traceability). Прослеживаемость бывает вертикальной (vertical traceability) и горизонтальной (horizontal traceability). Вертикальная позволяет соотносить между собой требования на различных уровнях требований, горизонтальная позволяет соотносить требование с тест-планом, тест-кейсами, архитектурными решениями и т.д. Для обеспечения прослеживаемости часто используются специальные инструменты по управлению требованиями (requirements management tool) и/или матрицы прослеживаемости (traceability matrix). Типичные проблемы с прослеживаемостью:
 - Требования не пронумерованы, не структурированы, не имеют оглавления, не имеют работающих перекрестных ссылок;
 - При разработке требований не были использованы инструменты и техники управления требованиями;
 - Набор требований неполный, носит обрывочный характер с явными «пробелами».
- **Модифицируемость** (modifiability). Это свойство характеризует простоту внесения изменений в отдельные требования и в набор требований. Можно говорить о наличии модифицируемости в том случае, если при доработке требований искомую информацию легко найти, а ее изменение не приводит к нарушению иных описанных в этом перечне свойств. Типичные проблемы с модифицируемостью:
 - Требования неатомарны (см. «атомарность») и непрослеживаются (см. «прослеживаемость»), а потому их изменение с высокой вероятностью порождает противоречивость (см. «непротиворечивость»);
 - Требования изначально противоречивы (см. «непротиворечивость»). В такой ситуации внесение изменений (не связанных с устранением противоречивости) только усугубляет ситуацию, увеличивая противоречивость и снижая прослеживаемость;
 - Требования представлены в неудобной для обработки форме (например, не использованы инструменты управления требованиями, и в итоге команде приходится работать с десятками огромных текстовых документов).
- **Проранжированность по важности, стабильности, срочности** (ranked for importance, stability, priority). Важность характеризует зависимость успеха проекта от успеха реализации требования. Стабильность характеризует вероятность того, что в обозримом будущем в требование не будет внесено никаких изменений. Срочность определяет распределение во времени усилий проектной команды по реализации того или иного требования. Типичные проблемы с проранжированностью состоят в ее отсутствии или неверной реализации и приводят к следующим последствиям:

- Проблемы с проранжированностью по важности повышают риск неверного распределения усилий проектной команды, направления усилий на второстепенные задачи и конечного провала проекта из-за неспособности продукта выполнять ключевые задачи с соблюдением ключевых условий;
- Проблемы с проранжированностью по стабильности повышают риск выполнения бессмысленной работы по совершенствованию, реализации и тестированию требований, которые в самое ближайшее время могут претерпеть кардинальные изменения (вплоть до полной утраты актуальности);
- Проблемы с проранжированностью по срочности повышают риск нарушения желаемой заказчиком последовательности реализации функциональности и ввода этой функциональности в эксплуатацию.
- **Корректность (correctness) и проверяемость (verifiability).** Фактически эти свойства вытекают из соблюдения всех вышеперечисленных (или можно сказать, что они не выполняются, если нарушено хотя бы одно из вышеперечисленных). В дополнение можно отметить, что проверяемость подразумевает возможность создания объективного тест-кейса (тест-кейсов), однозначно показывающего, что требование реализовано верно и поведение приложения в точности соответствует требованию. К типичным проблемам с корректностью также можно отнести:
 - опечатки (особенно опасны опечатки в аббревиатурах, превращающие одну осмысленную аббревиатуру в другую также осмысленную, но не имеющую отношения к некоему контексту; такие опечатки крайне сложно заметить);
 - наличие неаргументированных требований к дизайну и архитектуре;
 - плохое оформление текста и сопутствующей графической информации, грамматические, пунктуационные и иные ошибки в тексте;
 - неверный уровень детализации (например, слишком глубокая детализация требования на уровне бизнес-требований или недостаточная детализация на уровне требований к продукту);
 - требования к пользователю, а не к приложению (например: «пользователь должен быть в состоянии отправить сообщение» — увы, мы не можем влиять на состояние пользователя).

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.

Виды документов требований:

- **Спецификация требований к программному обеспечению (SRS - Software Requirement Specification):** представляет собой документ, подготовленный группой системных аналитиков (system analysts), который используется для описания программного обеспечения, которое будет разработано, основной бизнес-цели и функциональности определенного продукта, а также того, как он выполняет свои основные функции. В организациях, которые используют SRS, они обычно очень похожи на то, что описывается в PRD и FSD. SRS - это основа любого проекта, поскольку он состоит из структуры, которой будет следовать каждый член команды. SRS также является основой контракта с заинтересованными сторонами (пользователями / клиентами), который включает в себя все подробности о функциональности будущего продукта и о том, как он должен работать. SRS широко используется разработчиками программного обеспечения в процессе разработки продукта или программы. SRS включает как функциональные, так и нефункциональные требования, а также варианты использования. В идеальном документе SRS учитывается не только то, как программное обеспечение будет взаимодействовать с другим программным обеспечением или когда оно встроено в оборудование, но также потенциальных пользователей и способы их взаимодействия с

программным обеспечением. Он также содержит ссылки на таблицы и диаграммы, чтобы получить четкое представление обо всех деталях, связанных с продуктом. Документ SRS помогает членам команды из разных отделов оставаться в единстве и обеспечивать выполнение всех требований. Этот документ также позволяет минимизировать затраты и время на разработку программного обеспечения.;

- **Спецификация бизнес-требований** (BRS - Business Requirement Specification): BRS - это спецификация бизнес-требований, цель которой - показать, как удовлетворить бизнес-требования на более широком уровне. Документ BRS является одним из наиболее широко распространенных документов со спецификациями. Это очень важно, и BRS обычно создается в самом начале жизненного цикла продукта и описывает основные цели продукта или потребности, которые клиент хочет достичь с помощью определенного программного обеспечения или продукта. Он обычно создается бизнес-аналитиком (business analyst) на основе спецификаций других заинтересованных сторон и после тщательного анализа компании-клиента. Обычно окончательная версия документа проверяется клиентом, чтобы убедиться, что ожидания всех заинтересованных сторон бизнеса верны. BRS включает в себя все требования, запрошенные клиентом. Как правило, он состоит из цели продукта, пользователей, общего объема работ, всех перечисленных функций и возможностей, требований к удобству использования и производительности. В этот тип документа не включены варианты использования, а также диаграммы и таблицы. BRS используется в основном высшим и средним менеджментом, инвесторами в продукты, бизнес-аналитиками;
- **Спецификация функциональных требований** (FRS - Functional Requirement Specification): документ, в котором описаны все функции, которые должно выполнять программное обеспечение или продукт. Фактически, это пошаговая последовательность всех операций, необходимых для разработки продукта от начала до конца. FRS объясняет подробности того, как определенные программные компоненты будут вести себя во время взаимодействия с пользователем. Этот документ создан квалифицированными разработчиками и инженерами и считается результатом тесного сотрудничества между тестировщиками и разработчиками. Основное отличие от документа SRS заключается в том, что FRS не включает варианты использования. Он также может содержать диаграммы и таблицы, но это не обязательно. Этот документ является наиболее подробным, поскольку в нем подробно объясняется, как программное обеспечение должно функционировать (включая бизнес-асpekты, соответствие требованиям, требования безопасности), поскольку оно также должно удовлетворять всем требованиям, упомянутым в документах SRS и BRS. FRS помогает разработчикам понять, какой продукт они должны создать, а тестировщики программного обеспечения лучше разбираются в различных тестовых примерах и сценариях, в которых ожидается тестирование продукта;
- **Документ бизнес-требований** (BRD - Business Requirements Document, Business Needs Specification, Business Requirements): BRD фокусируется на определении бизнес задач проекта. BRD определяет одну или несколько бизнес задач стоящих перед пользователями, которые могут быть решены с помощью продукта компании. После этого предлагается решение – обычно это новый продукт или усовершенствование существующего продукта в нужной части. Он также может включать какой-то предварительный бизнес анализ – прогноз прибылей, анализ рынка и конкурентов, а также стратегию продаж и продвижения. Чаще всего он пишется Менеджером по продукту, Менеджером по маркетингу продукта или Бизнес аналитиком. В маленьких компаниях это может быть даже директор или владелец фирмы;
- **Документ требований рынка** (MRD - Market Requirements Document): MRD фокусируется на определении требований рынка к предлагаемому новому продукту. Если BRD определяет круг проблем и предлагает вариант их решения – то MRD более подробно описывает детали предлагаемого решения. Он может включать несколько или все нижеприведенные аспекты:
 - Функциональные возможности, необходимые для решения бизнес задач;
 - Анализ рынка и конкурентов;
 - Функциональные и нефункциональные требования;
 - Приоритетацию требований и функциональных возможностей;
 - Варианты использования;

Чаще всего он пишется Менеджером по продукту, Менеджером по маркетингу продукта или Бизнес аналитиком совместно с Системным аналитиком. Некоторые организации объединяют MRD и PRD в один документ и называют этот документ MRD. В этом случае MRD будет включать то, что описано в этой части и то, что описано в следующей – и может содержать более 50 страниц;

- **Документ требований к продукту** (PRD - Product Requirements Document): PRD фокусируется на определении требований к предлагаемому новому продукту. Если MRD фокусируется на требованиях с точки зрения нужд рынка, PRD фокусируется на требованиях с точки зрения самого продукта. Обычно он более детально описывает возможности и функциональные требования и может даже содержать скриншоты и лэйауты пользовательских интерфейсов. В организациях, где MRD не включает детализацию требований и варианты использования, PRD закрывает эту брешь. Обычно он пишется Менеджером по продукту, Бизнес аналитиком или Продуктовым аналитиком;
- **Функциональная спецификация** (FSD – Functional Specifications Document): FSD детально определяет функциональные требования к продукту с фокусировкой на реализации. FSD может определять продукт последовательно форму за формой и одну функциональную возможность за другой. Это документ, который уже может непосредственно использоваться командой разработчиков для создания продукта. Если MRD и PRD фокусируются на требованиях с точки зрения потребностей рынка и продукта, FSD фокусируется на определении деталей продукта, в форме, которая может быть использована разработчиками. FSD может также включать законченные скриншоты и детальное описание пользовательских интерфейсов (UI). Обычно он пишется Системным аналитиком, Архитектором решения или Главным разработчиком – т.е. автор обычно сам относится к разработчикам;
- **Спецификация продукта** (PSD – Product Specifications Document): PSD – это наименее популярная аббревиатура, но в тех организациях, которые используют эти документы, они обычно соответствуют по содержанию и объему Функциональной спецификации (Functional Specifications Document FSD) описанный выше;
- **Спецификация функционального дизайна** (FDS - Functional Design Specification);
- **Спецификация технического дизайна** (TDS - Technical Design Specification);
- ...

Техники тестирования требований см. в теме “Тестирование документации” в видах тестирования.

Источники:

- [Святослав Куликов “Тестирование программного обеспечения. Базовый курс”](#). Глава 2.
- [Требования к программному обеспечению](#)
- [Important Software Testing Documentation: SRS, FRS and BRS](#)
- [BRD, MRD, PRD, FSD и прочие ТБА](#)

Доп. материал:

- [IEEE Guide to the Software Engineering Body of Knowledge](#). Chapter 1.
- [Software Requirements Engineering: What, Why, Who, When, and How By Linda Westfall](#)
- Карл Вигерс «Разработка требований к программному обеспечению»
- [Святослав Куликов “Тестирование программного обеспечения. Базовый курс”](#). Раздел 2.2.8. “Типичные ошибки при анализе и тестировании требований”.
- [«File Converter» Project Requirements SAMPLE](#)
- [Визуализация ТЗ — диаграммы, схемы, картинки](#)
- [Системные требования и требования к программному обеспечению](#)
- [Ицыксон В.М. ПТПО - Управление требованиями](#)

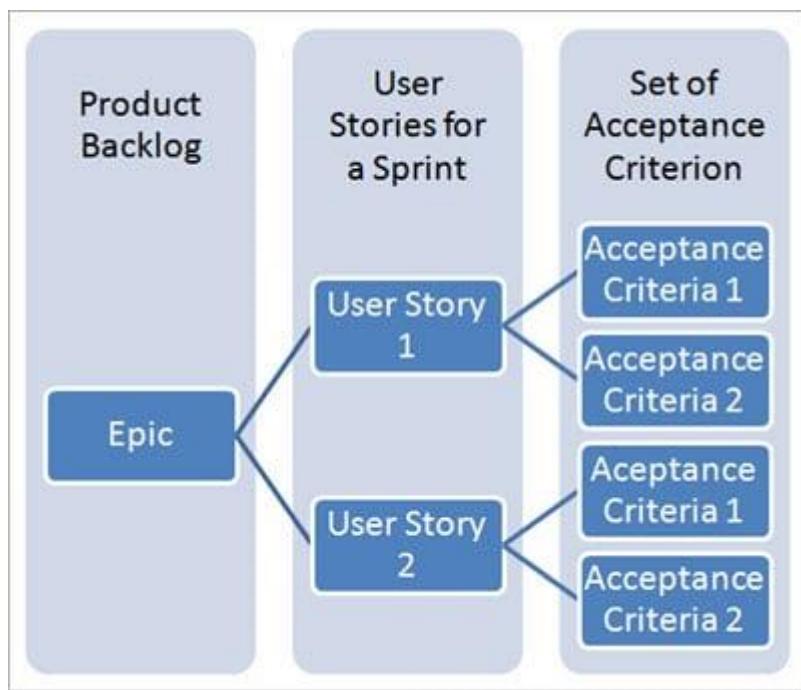
Пользовательские истории (User stories)

Пользовательская история (user story): Высокоуровневое пользовательское или бизнес-требование, обычно использующееся в гибких методологиях разработки программного обеспечения. Обычно состоит из одного или нескольких предложений на разговорном или формальном языке, описывающих

функциональность, необходимую пользователю, любые нефункциональные требования и включающих в себя критерии приемки. (ISTQB)

В индустрии разработки ПО слово «требование» определяет нашу цель, что именно нужно клиентам и что заставит нашу компанию развивать свой бизнес. Будь то продуктовая компания, которая производит программные продукты, или сервисная компания, которая предлагает услуги в различных областях программного обеспечения, основной базой для всех из них является требование, а успех определяется тем, насколько хорошо эти требования выполняются. Термин «требование» имеет разные названия в разных методологиях проекта. В Waterfall это называется Requirement/Specification Document, в Agile или SCRUM требования документируются в виде «Epic» и «User Story». В модели Waterfall документы с требованиями представляют собой огромные документы на сотни страниц, поскольку весь продукт реализуется за один этап. Но это не относится к Agile / SCRUM, потому что в этих случаях требования предъявляются к небольшим функциям или фичам, поскольку продукт готовится поэтапно.

Пользовательская история определяет требования к любой функциональности или фиче, в то время как критерии приемки (Acceptance Criteria) определяют критерии готовности (Definition of done) для пользовательской истории или требования.



Пользовательская история - это требование для любой функциональности или фичи, которое записано в 1-2 строки. Пользовательская история обычно является самым простым из возможных требований и касается одной-единственной функции/фичи.

Формат:

Как /роль пользователя или клиента/, я хочу /цель, которую нужно достичь/, чтобы я мог /причина цели/.

Например, “Как пользователь WhatsApp, я хочу, чтобы значок камеры в поле ввода чата позволял захватывать и отправлять изображения, чтобы я мог щелкнуть и поделиться своими фотографиями одновременно со всеми своими друзьями.”

Это стандартный формат, но далеко не обязательный или единственно-возможный. Главное в пользовательской истории - это ценность, которую пользователь получит от функции, т.е. User Story - это приём записи требований, который помогает команде разработки понять нужду клиента и после обсуждения выбрать, описать и утвердить то решение, которое удовлетворит эту нужду.

Job Stories

В целом Job Stories — схожая с US техника. Можно назвать их приёмом-субститутом, ведь обычно они не используются вместе и выполняют максимально похожую функцию. Job Stories представляют требование в виде действия, которое выполняет пользователь. Они не описывают саму функцию, а лишь концентрируют внимание команды на потребности. Job Stories концентрируются на психологической части фичи, на эмоциях, тревогах и прочем, что может возникнуть во время использования функции.

“Тело” JS делится на три части:

- Situation: дает контекст обо всей JS, который помогает dev-команде придумать возможное решение;
- Motivation: описывает невидимую руку, которая ведет юзера к использованию данной функции;
- Expected Outcome: описывает, что получит юзер после использования функции.

Job Stories могут писаться по двум форматам:

- В одну строку: When X I want to Y so I can Z" или "When X, actor is Y so that Z;
- В три строки:
 - When X
 - I want to Y
 - So I can Z.

Пример: When I want to withdraw money from my bank account, I want to know I have enough money in my account to withdraw some now so that I can go out to dinner with my friends.

Источники:

- [What Is User Story And Acceptance Criteria \(Examples\)](#)
- [Гайд по User Stories](#)

Доп. материал:

- [10 советов для написания хороших пользовательских историй](#)
- [Гайд по Job Stories в помощь к написанию user stories](#)
- [Юзер-стори идеальная, а багов 100500? Как мы тестируем документацию](#)
- [Job Stories Offer a Viable Alternative to User Stories](#)
- [25 sample user stories](#)
- [User Stories](#)

Критерии приемки (Acceptance Criteria)

Критерии приемки (acceptance criteria): Критерии выхода, которым должны соответствовать компонент или система, для того, чтобы быть принятыми пользователем, заказчиком или другим уполномоченным лицом. (IEEE 610)

Критерии приемки - это условия, которым должен удовлетворять программный продукт, чтобы быть принятным пользователем, заказчиком или, в случае функциональности системного уровня, потребляющей системой. Проще говоря - это список деталей (также известных как требования) о том, как новая функция (feature) программного обеспечения должна работать / выглядеть. Это гарантирует, что:

- Функция разработана хорошо. В противном случае важный или полезный аспект может быть упущен - и никто этого не заметит до самого конца.
- Это работает так, как было задумано. Если описание расплывчато, разработчикам, возможно, придется сделать предположения о том, как должна работать каждая область. С критериями приемки разработчики точно знают, какой дизайн и функциональность ожидаются.
- QA знает, чего ожидать во время тестирования. Даже если функция не выглядит сломанной, она может работать не так, как хотели менеджеры по продукту. Если критерии приемки отсутствуют, тестировщики не могут сообщать о подобных проблемах.

Хорошие критерии приемки должны быть простыми для понимания, но с достаточной детализацией, чтобы убедиться, что они не слишком расплывчаты. Это не всегда универсальный подход. Но они всегда должны

предоставлять достаточно информации для разработчиков, чтобы создать функцию, а для QA - для ее тестирования. Это не значит, что в процессе разработки программного обеспечения не возникнет вопросов. Но в целом функция должна быть понятной.

Формат / макет / шаблон критериев приемки (Acceptance Criteria Format/Layout/Template): существует два основных типа критериев приемки, основанные на сценариях и правилах:

- Критерии приемлемости, основанные на сценариях (Scenario-based acceptance criteria), используют шаблон для подробного описания конкретного поведения / последовательности действий пользователя;
- Критерии приемлемости на основе правил (Rule-based acceptance criteria) - это скорее простой список того, как функция должна выглядеть / работать;

Scenario-based acceptance criteria соответствует формату “Дано/Когда/Тогда” (“Given/When/Then”) (основан на BDD - [behavior driven development](#)):

- Given /какой-то аспект, связанный с поведением пользователя/
- When /пользователь выполняет определенное действие/
- Then /происходит определенный результат/

Между ними в случае нескольких условий можно добавлять “И” (“AND”).

Rule-Based Acceptance Criteria - это простой список «правил» о том, как функция должна выглядеть / работать:

- Все кнопки должны быть определенного цвета;
- Кнопка входа должна перенаправлять пользователя в определенный раздел;
- Кнопка регистрации должна находиться в определенной области;
- Все кнопки должны быть серыми, если не выполняются определенные требования;
- и многое другое;

Хотя критерии, основанные на правилах, имеют более простой формат, нет причин, по которым они не могут быть длинными и подробными.

Кто пишет критерии приемки? Обычно в создании критериев приемки участвуют несколько человек или команд. Тем не менее, это в первую очередь делает product manager (или “product owner”). Разработчики несут ответственность за обеспечение функциональности функции, а QA - за подтверждение ее удобства использования. Но критерии приемки создаются человеком или командой, ответственной за решение, какие новые функции добавить в продукт (независимо от типа приложения или веб-сайта).

Большая часть Agile включает внесение изменений по мере развития проекта. Так **могут ли критерии приемки измениться в середине спринта?** Ответ: «Это зависит от обстоятельств». Если спринт начался, но разработчики еще не завершили эту функцию, можно изменить требования. Но важно всегда сначала согласовывать с разработчиками и держать других (например, QA) в курсе. Тестировщики могли написать test cases, которые больше не актуальны после изменений. Кроме того, новый объем работы может оказаться слишком большим, чтобы разработчики могли завершить его вовремя.

User Stories vs Acceptance Criteria: пользовательские истории и критерии приемки идут рука об руку.

Пользовательская история описывает основную цель новой функции - обзор того, как она поможет пользователям. Критерии приемки перечисляют способы работы функции с технической точки зрения. Обычно в тикетах (например, в Jira или Trello) вверху указывается пользовательская история, за которой следуют критерии приемки

Definition of Done: чтобы заявка (или функция) считалась «выполненной», все критерии должны работать. Например, предположим, что пользовательская история была: “Как пользователь, я хочу иметь возможность войти в систему, чтобы получить доступ к панели управления моей учетной записи”. Как уже упоминалось, пользователь может войти в систему, чтобы получить доступ к панели управления своей учетной записи. Но тикет не считался бы «done», если бы он также содержал следующие критерии приемки: “Кнопка входа должна быть бирюзовой”, а фактически кнопка входа была бы, например, желтой. Иногда команда решает

запустить функцию даже с незначительными несоответствиями. Таким образом, они могут пометить тикет как выполненный (или создать отдельный для решения оставшихся аспектов), даже если не все критерии работают. Но с точки зрения технического определения, это не «готово», пока не пройдут все критерии приемки.

Источник: [What is Acceptance Criteria?](#)

Виды отчетов (Reports)

Отчет - это документ, содержащий информацию о выполненных действиях, результатах проведённой работы. Обычно он включает в себя таблицы, графики, списки, просто описывающую информацию в виде текста. Их пропорция и содержание определяют пользу и понятность отчета.

Нам важно понять, для кого, для чего и в каких условиях мы это делаем и на сколько это улучшит восприятие излагаемой нами информации. Надо помнить, что каждое действие преследует определенную цель. В случае отчета нам важно понять, для кого, для чего и в каких условиях мы это делаем.

Ниже перечислены наиболее известные варианты отчетов в тестировании.

Отчет по инциденту (incident report)

Отчет по инциденту (incident report): Документ, описывающий событие, которое произошло, например, во время тестирования, и которое необходимо исследовать. (IEEE 829)

Отчет об инцидентах можно определить как письменное описание инцидента, наблюдаемого во время тестирования. Чтобы лучше понять, давайте начнем с того, что такое «инцидент». Инцидент при тестировании программного обеспечения можно определить как наблюдаемое изменение или отклонение поведения системы от ожидаемого. Это может быть отклонение от функционального требования или от настроек среды. Очень часто инцидент называют дефектом или ошибкой, но это не всегда так. Инцидент - это в основном любое неожиданное поведение или реакция программного обеспечения, требующая расследования.

Инцидент необходимо расследовать, и на основании расследования инцидент может быть преобразован в дефект. Чаще всего это оказывается дефектом, но иногда это может произойти из-за различных факторов, например:

- Человеческий фактор;
- Требование отсутствует или неясно;
- Проблема среды, например отсутствие ответа от внутреннего сервера, вызывающее периодическое непредвиденное поведение или ошибку. Либо неправильная конфигурация среды;
- Ошибочные тестовые данные;
- Некорректный ожидаемый результат.

Incident report призван зафиксировать и сообщить об инциденте заинтересованным лицам, провести расследование. Составляется аналогично баг-репорту, возможно с упором на расследование, обсуждение, влияние (impact) и может быть назначен не на разработчиков для уточнения деталей.

Отчет о результатах тестирования (test result report)

Отчет о результатах тестирования - периодический отчет, в котором документируется подробная информация о выполнении теста и его результате. Также он содержит условия, предположения, ограничения теста, какой элемент теста кем тестируется. Помимо этого вносится подробная информация об оставшейся работе, чтобы показать, сколько еще работы необходимо выполнить в проекте.

Отчет о выполнении теста (Test Execution Report)

Отчет о выполнении теста содержит детали выполнения и результат выполнения теста. Обычно его готовят для отправки вышестоящему руководству от группы тестирования, чтобы показать состояние выполнения теста и ход тестирования. Когда мы доставляем программное обеспечение клиенту, мы вкратце отправим полную информацию о выполнении теста. Это даст клиенту лучшее понимание выполненного теста и покрытия.

Отчет о ходе тестирования (test progress report)

Отчет о ходе тестирования (test progress report): Документ, подводящий итог задачам и результатам, составляемый с определенной периодичностью с целью сравнения прогресса тестирования с базовой версией (например, с исходным планом тестирования) и извещения о рисках и альтернативах, требующих решения руководства. (ISTQB)

Аналитический отчет о тестировании (test evaluation report)

Аналитический отчет о тестировании (test evaluation report): Документ, создаваемый в конце процесса тестирования и подводящий итог тестовым активностям и результатам. Также в нем содержится оценка процесса тестирования и полученный опыт. (ISTQB)

Итоговый отчет о тестировании (test summary report)

Итоговый отчет о тестировании (test summary report): Документ, подводящий итог задачам и результатам тестирования, также содержащий оценку соответствующих объектов тестирования относительно критериев выхода. (IEEE 829)

Сводный отчет о тестировании содержит подробную информацию о тестировании, проведенном на протяжении жизненного цикла разработки программного обеспечения. Элементы в итоговом отчете по тестированию различаются от организации к организации, а также различаются для разных проектов. Информация в отчете об испытаниях основывается на аудитории отчета об испытаниях. Аудитория может быть клиентом, менеджментом, бизнес-аналитиком, разработчиками, членами команды тестирования, членами организации и т. д.

Отчет о пользовательском приемочном тестировании (User acceptance test report)

Отчет о пользовательском приемочном тестировании создается во время и после UAT. В нем указываются подробности проведенного пользователем приемочного теста и результат пользовательского приемочного теста. В нем также перечислены дефекты, не учтенные при UAT.

Источники:

- [Создание понятных отчетов о тестировании](#)
- [What Is Incident Report In Software Testing?](#)
- [Software Testing Artifacts – Test Reports](#)

Доп. материал:

- [48+ SAMPLE Test Report Templates](#)
- [Отчет по результатам тестирования сайта](#)
- [Отчет о тестировании релиза](#)
- [Test report templates](#)
- [Test Summary Reports Tutorial: Learn with Example & Template](#)

Базис тестирования (Test basis)

Базис тестирования (test basis): Документ, на основании которого определяются требования к компоненту или системе. Документация, на которой базируются тестовые сценарии. Если правка данного документа может быть осуществлена только в процессе формальной процедуры внесения изменения, то такой базис тестирования называется замороженным базисом тестирования. (ISTQB)

Базис тестирования определяется как источник информации или документ, необходимый для написания кейсов, а также как данные для начала анализа тестов:

- System Requirement Document (SRS);
- Functional Design Specification;
- Technical Design Specification;
- User Manual;

- Use Cases;
- Source Code;
- Business Requirement Document (BRD);
- ?User story;
- ?Vision;
- ?Mockup;
- ?Prototype.

Базис тестирования должен быть четко определен и должным образом структурирован, чтобы можно было легко определить условия тестирования, из которых можно получить тестовые примеры.

Тестовое условие (test condition): Объект или событие в компоненте или системе, которое должно быть проверено одним или несколькими тестовыми наборами. Например: функция, транзакция, параметр, атрибут качества или структурный элемент. (ISTQB)

Тестовое условие - тестируемый аспект в test basis.

Источники:

- [Test Basis in Software Testing](#)

Матрица трассируемости (RTM - Requirement Traceability Matrix)

Трассируемость (traceability): Способность идентифицировать связанные объекты в документации и программном обеспечении, например, требования со связанными с ними тестами. (ISTQB)

Матрица трассируемости (traceability matrix): Двумерная таблица, описывающая связь двух сущностей (например, требований и тестовых сценариев). Таблица позволяет производить прямую и обратную трассировку от одной сущности к другой, обеспечивая таким образом возможность определения покрытия и оценки влияния предполагаемых изменений. (ISTQB)

| | Тест-кейс 1 | Тест-кейс 2 | Тест-кейс 3 | Тест-кейс 4 | Тест-кейс 5 |
|-------------------|-------------|-------------|-------------|-------------|-------------|
| Требование 1. | + | | | | |
| Требование 2. | | + | | | |
| Требование 3. | | + | + | | |
| Требование 4. | | | | | + |
| Требование 5. ... | | | | + | |

В тестировании многое можно представить в виде удобной и наглядной матрицы (таблицы): Requirement Traceability Matrix, Test matrix, Compliance Matrix, Risk Matrix, RACI Matrix и т.д.

Матрица трассируемости (Requirement Traceability Matrix AKA Traceability Matrix or Cross Reference Matrix) используется для документирования связей между требованиями и тест-кейсами по этим требованиям и наглядного отображения трассируемости в виде простой таблицы.

Матрица трассируемости может служить одновременно в качестве матрицы покрытия. Наличие такой матрицы позволяет объективно оценить, какая часть продукта покрыта тестами, а какая нет.

Виды трассируемости:

- *Вертикальная трассируемость (vertical traceability): Отслеживание требований через уровни разработки к компонентам. (ISTQB)*
- *Горизонтальная трассируемость (horizontal traceability): Трассировка требований к уровню тестирования по отношению к уровням документации (например, план тестирования, спецификация проектирования теста, спецификация тестовых сценариев и спецификация процедуры тестирования или автоматизированный сценарий тестирования). (ISTQB)*

Другой источник:

- Прямая трассируемость (Forward Traceability): гарантирует, что проект продвигается в желаемом направлении и что каждое требование тщательно проверяется;
- Обратная трассируемость (Backward Traceability): гарантирует, что текущий разрабатываемый продукт находится на правильном пути. Это также помогает определить, что дополнительные неуказанные функции не добавляются и, таким образом, это не влияет на объем проекта;
- Двунаправленная трассируемость (Bi-Directional Traceability = Forward + Backward): содержит ссылки от тестовых примеров к требованиям и наоборот. Это гарантирует, что все тестовые примеры можно отследить до требований, и каждое указанное требование содержит точные и действительные тестовые примеры для них.

RTM актуальна на всех этапах программного проекта. Давайте разберемся с этим через водопадную модель SDLC:

- RTM начинается вместе с началом фазы сбора требований (Requirements Gathering phase);
- продолжается через управление требованиями (Requirements Management);
- проектирование (Design);
- разработку (Development);
- тестирование (Testing);
- внедрение (Implementation);
- и поддержку (Support).

При прохождении всех этих этапов трассируемость требований поддерживается с помощью этого документа. После того, как требования были внесены в таблицу, детали дизайна для этих требований будут сопоставлены с требованиями. На основе этих деталей проекта будет производиться разработка программного обеспечения / модуля. Детали репозитория кода из SVN, TFS, Bitbucket, Github будут сопоставлены. Теперь вы знаете, где находится дизайн и код каждого требования. Это трассируемость. Отслеживайте каждое требование от начала до его конечного результата по мере его использования пользователем приложения! На этапе поддержки RTM будет чрезвычайно полезен для понимания и решения проблем, пройдя через все соответствующие детали функции / требования. Улучшение функции стало бы возможным благодаря отслеживанию и пониманию логики, дизайна и кода. С точки зрения владения RTM, RTM принадлежит менеджерам проекта или бизнес-аналитикам. В организациях CMMi команда TQM также будет проверять это как стандартный результат в проектах программного обеспечения.

*Когда на основе требований к продукту составляются тест-сценарии и выполняется тестирование, это называется Requirement based testing.

Источники:

- [How To Create Requirements Traceability Matrix \(RTM\): Example And Sample Template](#)
- [What is the difference between Test matrix and Traceability matrix?](#)

Доп. материал:

- [Матрица трассабилити](#)
- [Reinventing the QA process](#)

Метрики тестирования (Software Test Metrics)

Метрика (metric): Шкала измерений и метод, используемый для измерений (ISO 14598)

“Вы не можете контролировать то, что не можете измерить” - Том Демакро.

Метрики тестирования предназначены для мониторинга и управления процессом и продуктом. Это помогает без отклонений вести проект к намеченным целям. Метрики отвечают на разные вопросы. Важно решить, на какие вопросы вы хотите получить ответы. Метрики тестирования программного обеспечения подразделяются на два типа:

- **Метрики процесса (Process metrics):** используются в процессе подготовки и выполнения тестирования.
 - **Продуктивность подготовки тест-кейсов (Test Case Preparation Productivity):** используется для расчета количества подготовленных тест-кейсов и усилий (Effort), затраченных на их подготовку.

Test Case Preparation Productivity = (No of Test Case) / (Effort spent for Test Case Preparation)

- **Охват тестового дизайна (Test Design Coverage):** процент покрытия тест-кейсами требований.

*Test Design Coverage = ((Total number of requirements mapped to test cases) / (Total number of requirements))*100*

- **Продуктивность выполнения тестов (Test Execution Productivity):** определяет количество тест-кейсов, которые могут быть выполнены в час.

Test Execution Productivity = (No of Test cases executed) / (Effort spent for execution of test cases)

- **Покрытие выполненных тестов (Test Execution Coverage):** предназначено для измерения количества выполненных тест-кейсов по сравнению с количеством запланированных тест-кейсов.

*Test Execution Coverage = (Total no. of test cases executed / Total no. of test cases planned to execute)*100*

- **Успешные тест-кейсы (Test Cases Passed):** для измерения процента пройденных успешно тест-кейсов.

*Test Cases Pass = (Total no. of test cases passed) / (Total no. of test cases executed) * 100*

- **Неуспешные тест-кейсы (Test Cases Failed):** для измерения процента заваленных тест-кейсов.

*Test Cases Failed = (Total no. of test cases failed) / (Total no. of test cases executed) * 100*

- **Заблокированные тест-кейсы (Test Cases Blocked):** для измерения процента блокированных тест-кейсов.

*Test Cases Blocked = (Total no. of test cases blocked) / (Total no. of test cases executed) * 100*

- **Метрики продукта (Product metrics):**

- **Уровень обнаружения ошибок (Error Discovery Rate):** для определения эффективности тест-кейсов.

*Error Discovery Rate = (Total number of defects found / Total no. of test cases executed)*100*

- **Процент выявления дефектов** (Defect Detection Percentage (DDP)): Количество дефектов, выявленных в фазе тестирования, поделенное на число дефектов, найденных в этой фазе тестирования, а также во всех последующих фазах. См. также ускользнувший дефект. (ISTQB)
- **Уровень исправления дефектов** (Defect Fix Rate): помогает узнать качество сборки (build) с точки зрения устранения дефектов.

$$\text{Defect Fix Rate} = (\text{Total no of Defects reported as fixed} - \text{Total no. of defects reopened}) / (\text{Total no of Defects reported as fixed} + \text{Total no. of new Bugs due to fix}) * 100$$

- **Плотность дефектов** (Defect Density): количество дефектов, обнаруженных в компоненте или системе, поделенное на размер компонента или системы (выраженный в стандартных единицах измерения, например строках кода, числе классов или функций). (ISTQB)

$$\text{Defect Density} = \text{Total no. of defects identified} / \text{Actual Size (requirements)}$$

- **Утечка дефектов** (Defect Leakage): используется для проверки эффективности процесса тестирования перед UAT.

$$\text{Defect Leakage} = ((\text{Total no. of defects found in UAT}) / (\text{Total no. of defects found before UAT})) * 100$$

- **Эффективность устранения дефектов** (Defect Removal Efficiency): позволяет сравнивать общую (дефекты, обнаруженные до и после поставки) эффективность устранения дефектов.

$$\text{Defect Removal Efficiency} = ((\text{Total no. of defects found pre-delivery}) / ((\text{Total no. of defects found pre-delivery}) + (\text{Total no. of defects found post-delivery}))) * 100$$

Источники:

- [Software Test Metrics – Product Metrics & Process Metrics](#)

Тестовый оракул (Test oracle)

Тестовый предсказатель (*test oracle*): Источник, при помощи которого можно определить ожидаемые результаты для сравнения с реальными результатами, выдаваемыми тестируемой системой. В роли тестового предсказателя могут выступать уже имеющаяся система (для эталонного тестирования), руководство пользователя, профессиональные знания специалиста, однако им не может быть программный код. (ISTQB)

Тестовый оракул - это механизм для определения того, прошел ли тест или нет. Использование оракулов включает сравнение (для заданных входных данных тестового примера) выходных данных тестируемой системы с выходными данными, которые, по определению оракула, должен иметь продукт. Термин «тестовый оракул» впервые был введен в статье Уильяма Э. Хаудена. Дополнительная работа над различными видами оракулов была исследована Элейн Вейкер.

Категории тестовых оракулов:

Определенные (Specified): Эти оракулы обычно связаны с формализованными подходами к моделированию программного обеспечения и построению программного кода. Они связаны с formal specification, model-based design, который может использоваться для создания тестовых оракулов, state transition specification, для которой могут быть получены оракулы, чтобы помочь model-based testing and protocol conformance testing, and design by contract, для которого эквивалентный тестовый оракул является утверждением (assertion). Указанные тестовые оракулы имеют ряд проблем. Формальная спецификация основана на абстракции, которая, в свою очередь, может иметь элемент неточности, поскольку все модели не могут зафиксировать все поведение;

Полученные (Derived): полученный тестовый оракул различает правильное и неправильное поведение, используя информацию, полученную из артефактов системы. Они могут включать в себя документацию, результаты выполнения системы и характеристики версий тестируемой системы. Regression test suites (or reports) являются примером производного тестового оракула - они построены на предположении, что

результат из предыдущей версии системы может быть использован в качестве помощника (оракула) для будущей версии системы. Ранее измеренные характеристики производительности могут быть использованы в качестве оракула для будущих версий системы, например, чтобы задать вопрос о наблюдаемом потенциальном ухудшении производительности. Текстовая документация из предыдущих версий системы может использоваться в качестве основы для определения ожиданий в будущих версиях системы. Псевдо-оракул попадает в категорию полученных тестовых оракулов. Псевдо-оракул, по определению Вейкуера, представляет собой отдельно написанную программу, которая может принимать те же входные данные, что и тестируемая программа или система, так что их выходные данные могут быть сопоставлены, чтобы понять, может ли быть проблема для исследования. Частичный оракул - это гибрид указанного тестового оракула и производного тестового оракула. Он определяет важные (но не полные) свойства тестируемой системы. Например, при метаморфическом тестировании (Metamorphic testing) такие свойства, называемые метаморфическими отношениями, используются при нескольких запусках системы.

Примеры:

- формальная спецификация, используемая в качестве входных данных для model-based design and model-based testing;
- документация, которая не является полной спецификацией продукта, такая как руководство по использованию или установке, или запись характеристик производительности или минимальных требований;
- оракул согласованности, сравнивающий результаты выполнения одного теста с другим на предмет сходства;
- псевдо-оракул: вторая программа, которая использует другой алгоритм для вычисления того же математического выражения, что и тестируемый продукт;
- Specified+derived: во время поиска Google у нас нет полного оракула, чтобы проверить правильность количества возвращенных результатов. Мы можем определить метаморфическое отношение так, что последующий суженный поиск будет давать меньше результатов.

Неявные (Implicit): Неявный тестовый оракул полагается на подразумеваемую информацию и предположения. Например, может быть какой-то подразумеваемый вывод из сбоя программы, то есть нежелательное поведение - оракул, чтобы определить, что может быть проблема. Существует несколько способов поиска и тестирования нежелательного поведения, независимо от того, называют ли это отрицательным тестированием, где есть специализированные подмножества, такие как фаззинг. У неявных тестовых оракулов есть ограничения, поскольку они полагаются на подразумеваемые выводы и предположения. Например, сбой программы или процесса может не быть приоритетной проблемой, если система является отказоустойчивой и поэтому работает в форме самовосстановления / самоуправления. Неявные тестовые оракулы могут быть подвержены ложным срабатываниям из-за зависимостей от среды;

Человек (Human): Если предыдущие категории оракулов не могут быть использованы, то потребуется участие человека. Это можно рассматривать как количественный и качественный подходы. Количественный подход направлен на поиск нужного количества информации, которую нужно собрать о тестируемой системе (например, результатов тестирования), чтобы заинтересованная сторона могла сделать решения о соответствии или выпуске программного обеспечения. Качественный подход направлен на определение репрезентативности и пригодности входных данных тестирования и контекста выходных данных тестируемой системы. Примером может служить использование реалистичных и репрезентативных данных испытаний и понимание результатов (если они реалистичны). При этом можно руководствоваться эвристическими подходами, такими как интуиция, эмпирические правила, вспомогательные контрольные списки и опыт, чтобы помочь адаптировать конкретную комбинацию, выбранную для тестируемой программы / системы.

Примеры:

- Качественный: эвристический оракул предоставляет репрезентативные или приблизительные результаты по классу тестовых входных данных;

- Качественный: статистический оракул использует вероятностные характеристики, например, с анализом изображений, где определен диапазон достоверности и неопределенности для того, чтобы тестовый оракул решил о совпадении.

Источники:

- [Test oracle](#)

Доп. материал:

- [Oracles from the inside out](#)

SDLC и STLC

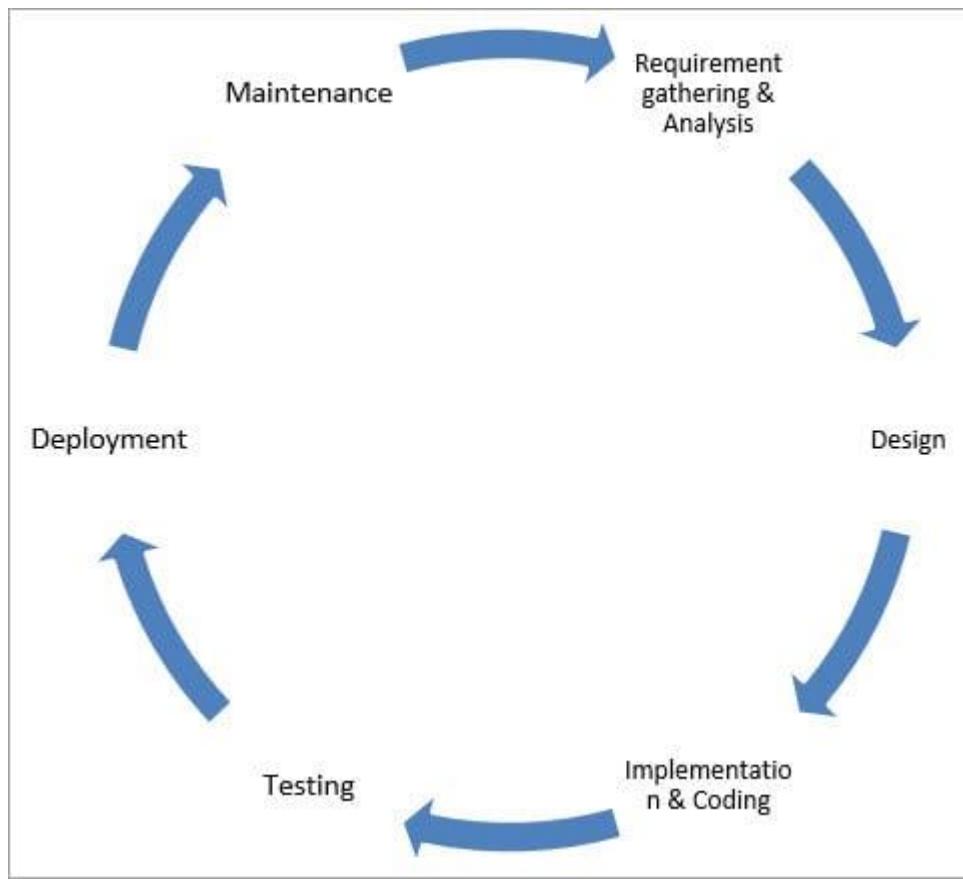
Жизненный цикл разработки ПО (SDLC - Software Development Lifecycle)

Жизненный цикл программного обеспечения (*software lifecycle*): Период времени, начинающийся с момента появления концепции программного обеспечения и заканчивающийся тогда, когда дальнейшее использование программного обеспечения невозможно. Жизненный цикл программного обеспечения обычно включает в себя следующие этапы: концепт, описание требований, дизайн, реализация, тестирование, инсталляция и наладка, эксплуатация и поддержка и, иногда, этап вывода из эксплуатации. Данные фазы могут накладываться друг на друга или проводиться итерационно. (ISTQB)

SDLC - это систематизированный процесс, этапы которого охватывают полный жизненный цикл программного обеспечения (Software Lifecycle) и который определяет различные этапы разработки программного обеспечения для создания высококачественного программного обеспечения, отвечающего ожиданиям клиентов и для улучшения эффективности разработки. Разработка системы должна быть завершена в заранее определенные сроки и стоимость. Каждая фаза жизненного цикла SDLC имеет свой собственный процесс и результаты, которые используются в следующей фазе.

Обычно он делится на шесть-восемь шагов, но менеджеры проектов могут объединять, декомпозировать или пропускать шаги, в зависимости от склона проекта.

В разных источниках фазы немного отличаются, но глобально суть везде одна и та же.



Фазы SDLC:

- **Сбор и анализ требований (Requirement Gathering and Analysis):** На этом этапе от клиента собирается вся необходимая информация для разработки продукта в соответствии с их ожиданиями. Любые неясности должны быть разрешены сразу на этом этапе. Бизнес-аналитик и менеджер проекта назначили встречу с заказчиком, чтобы собрать всю информацию, например, что заказчик хочет построить, кто будет конечным пользователем, какова цель продукта. Перед созданием продукта очень важно понимание или знание продукта. Например, клиент хочет иметь приложение, которое включает денежные транзакции. В этом случае требование должно быть четким, например, какие

транзакции будут выполняться, как они будут проводиться, в какой валюте они будут проводиться и т. д. После того, как сбор требований завершен, проводится анализ для проверки возможности разработки продукта. После четкого понимания требования создается документ SRS (Спецификация требований к программному обеспечению). Этот документ должен быть полностью понят разработчикам, а также должен быть рассмотрен заказчиком для использования в будущем;

- **Дизайн (Design):** На этом этапе требования, собранные в документе SRS, используются в качестве входных данных, и создается архитектура программного обеспечения, которая используется для реализации разработки системы. Создаются два вида дизайн-документов:
 - Высокоуровневый дизайн (HLD - High-Level Design):
 - Краткое описание и название каждого модуля;
 - Краткое описание функциональности каждого модуля;
 - Отношения интерфейсов и зависимости между модулями;
 - Таблицы базы данных, идентифицированные вместе с их ключевыми элементами;
 - Полные архитектурные схемы с подробными сведениями о технологиях.
 - Низкоуровневый дизайн (LLD - Low-Level Design):
 - Функциональная логика модулей;
 - Таблицы базы данных, которые включают тип и размер;
 - Полная детализация интерфейсов;
 - Решение всех типов проблем с зависимостями;
 - Список сообщений об ошибках;
 - Полные входные и выходные значения для каждого модуля.
- **Разработка (Implementation or Coding):** Реализация / кодирование начинается, как только разработчик получает Design document. Дизайн программного обеспечения переведен в исходный код. На этом этапе реализуются все компоненты программного обеспечения;
- **Тестирование (Testing):** Тестирование начинается после завершения кодирования и выпуска модулей для тестирования. На этом этапе разработанное программное обеспечение тщательно тестируется, и все обнаруженные дефекты передаются разработчикам для их исправления. Повторное тестирование, регрессионное тестирование проводится до тех пор, пока программное обеспечение не будет соответствовать ожиданиям клиента. Тестировщики обращаются к документу SRS, чтобы убедиться, что программное обеспечение соответствует стандарту заказчика;
- **Развертывание (Deployment):** После тестирования продукта он развертывается в производственной среде или выполняется первое UAT (пользовательское приемочное тестирование), в зависимости от ожиданий клиента. В случае UAT создается копия производственной среды, и заказчик вместе с разработчиками выполняет тестирование. Если клиент остается доволен, то предоставляется согласие на релиз;
- **Поддержка (Maintenance):** Основное внимание на этом этапе SDLC уделяется обеспечению того, чтобы потребности продолжали удовлетворяться и чтобы система продолжала работать в соответствии со спецификацией, упомянутой в первом этапе. После того, как система развернута и клиенты начинают использовать разработанную систему следует 3 вида активностей:
 - Исправление ошибок;
 - Обновление;
 - Улучшение.

Источники:

- [SDLC: Phases & Models of Software Development Life Cycle](#)
- [SDLC \(Software Development Life Cycle\) Phases, Process, Models](#)

Доп. материал:

- [IEEE Guide to the Software Engineering Body of Knowledge](#). Chapters 7-8.

Жизненный цикл тестирования ПО (STLC – Software Testing Lifecycle)

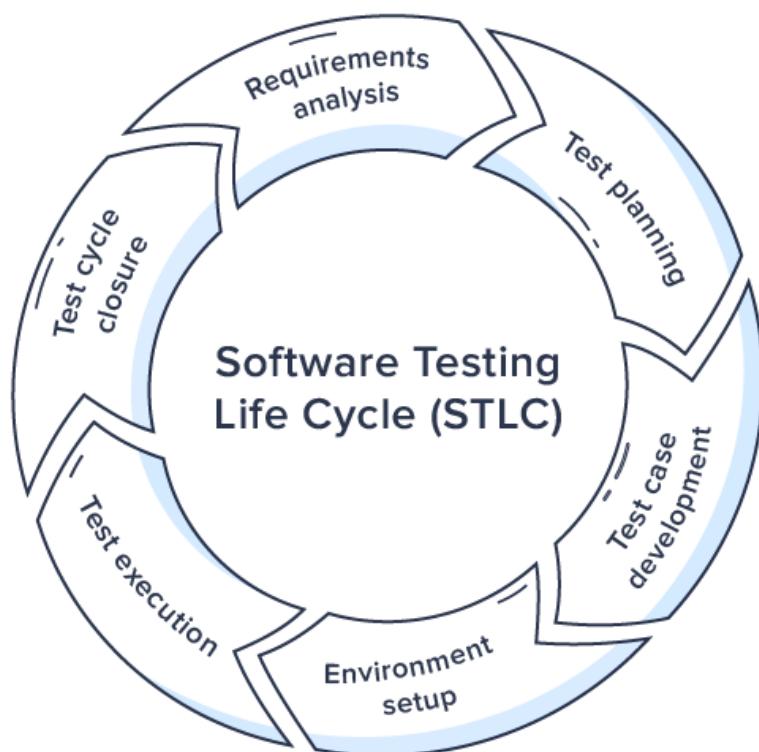
STLC - это процесс тестирования, который включает в себя определенную последовательность шагов, чтобы гарантировать достижение целей в области качества. В процессе STLC каждое действие выполняется

планомерно и систематически. Каждый этап имеет разные цели и результаты. У разных организаций разные этапы STLC, однако основа остается прежней.

Каждая фаза STLC имеет критерии начала и окончания:

- Критерии начала/входа (Entry Criteria): предварительные условия, которые должны быть выполнены перед началом фазы;
- Критерии окончания/выхода (Exit Criteria): условия, которые должны быть выполнены перед завершением фазы;

Фазы STLC



bamboo agile

STLC имеет несколько взаимосвязанных фаз и в целом очень похож на SDLC. Эти фазы являются последовательными и называются:

- **Анализ требований (Requirement Analysis)**: один из важнейших этапов, потому что именно на нем можно почти бесплатно исправить недостатки проекта. Этап анализа требований также определяет потенциальную потребность в автоматизированном тестировании и позволяет производить экономические расчеты затрат на рабочую силу на основе оценки проекта. На этом же этапе обсуждаются и документируются критерии начала и окончания тестирования.
 - Entry Criteria: BRS (Business Requirement Specification)
 - Deliverables: список всех проверяемых требований, технико-экономическое обоснование автоматизации (если применимо);
- **Планирование тестирования (Test Planning)**: на этом этапе формируется план тестирования, т.е. мы определяем действия и ресурсы, которые помогут достичь целей тестирования (участники и их роли, инструменты, окружение). Во время планирования мы также пытаемся определить метрики, метод сбора и отслеживания этих метрик. План составляют исходя из требований, тестовой стратегии и анализа рисков.
 - Entry Criteria: Requirements Documents;
 - Deliverables: Test Strategy, Test Plan, and Test Effort estimation document.

- **Разработка тест-кейсов** (Test Case Development): подразумевает использование ручного и автоматизированного тестирования для достижения полного охвата функциональности программного обеспечения, при этом процесс основан на заранее установленных требованиях. Чаще всего тест-кейсы для автоматического тестирования пишутся отдельно, так как кейсы для ручного тестирования описаны в виде шпаргалок (cheat sheets).
 - Entry Criteria: Requirements Documents (Updated version);
 - Deliverables: Test cases, Test Scripts (if automation), Test data.
- **Настройка тестовой среды** (Test Environment Setup): в плане тестирования четко указано, какую тестовую среду следует использовать. На этом этапе STLC настраиваются операционные системы и виртуальные машины, развертываются инструменты тестирования, такие как Selenium, Katalon Studio, а также тестовая среда и базы данных проекта. Мы также обращаемся с запросами к DevOps и администраторам, если требуется поддержка.
 - Entry Criteria: Test Plan, Smoke Test cases, Test Data;
 - Deliverables: Test Environment. Smoke Test Results.
- **Выполнение тестов** (Test Execution): тесты выполняются на основе готовой тестовой документации и правильно настроенной тестовой среды. Все результаты тестирования регистрируются в Системе управления тестированием. Отрицательно пройденные тесты, в которых фактический результат отличается от ожидаемого, регистрируются как ошибки и передаются команде разработчиков на доработку с последующей перепроверкой после исправления.
 - Entry Criteria: Test Plan document, Test cases, Test data, Test Environment;
 - Deliverables: Test case execution report, Defect report, RTM.
- **Завершение цикла испытаний** (Test Cycle Closure): окончательная генерация отчетов о тестировании для клиента. Они должны включать затраченное время, процент обнаруженных ошибок и положительных результатов тестирования, общее количество обнаруженных и исправленных ошибок. Что касается отдела тестирования, то это момент для анализа его работы, подведения итогов, анализа его продуктивности и возможности внести предложения по улучшению качества тестирования.
 - Entry Criteria: Test Case Execution report (убедитесь, что нет открытых high severity defects), Defect report;
 - Deliverables: Test Closure report, Test metrics.

Разница STLC и SDLC

STLC и SDLC тесно связаны друг с другом, но они одновременно преследуют разные задачи с одной и той же целью, а именно:

- сбор требований в желаемой форме и разработка заявленной функциональности (SDLC);
- анализ требований, помочь клиенту и команде разработчиков и подтверждение качества реализованной функциональности (STLC).

Общая цель - удовлетворение клиента и получение максимально возможного балла на этапах верификации и валидации.

Почему тестирование разбито на отдельные этапы?

Ответ из ISTQB Foundation Level Exam: У каждого этапа своя цель.

В большинстве случаев тестирование разбивается на несколько этапов в зависимости от развития самого кода и стремления к эффективному использованию ресурсов. Давайте рассмотрим пример приложения, которое включает в себя как службы REST, так и веб-интерфейс, в agile команде, которая предоставляет набор user stories, которые в конечном итоге будут развернуты как производственный выпуск.

Если команда следует Acceptance Test Driven Development (ATDD), то члены команды будут совместно работать над дизайном тестов историй. Это происходит до начала разработки (одна из характеристик ATDD). Допустим, Мэри - разработчик, который напишет код для служб REST, и допустим, что она практикует разработку через тестирование (TDD). Она строит модульные тесты, по одному, сначала позволяя тесту не пройти, а затем пишет достаточно кода для прохождения теста. Когда имеется достаточное количество тестов

для удовлетворения всех требований к истории и эти тесты проходят, тогда разработка и модульное тестирование завершаются. Затем Мэри может написать автоматизированные тесты, которые включают базу данных и, возможно, другие зависимости вне ее кода. Это интеграционные тесты. Поскольку команда использует ATDD, у нее уже есть набор приемочных тестов, поэтому она основывает свои автоматизированные тесты на них. Когда код Мэри проверяется и объединяется в общую ветвь разработки, в процессе сборки выполняются автоматические регрессионные тесты, чтобы убедиться, что существующие функциональные возможности не были нарушены работой Мэри. Эти тесты часто представляют собой просто набор автоматических приемочных испытаний, проводимых в течение месяцев или лет.

Если Сэм является разработчиком пользовательского интерфейса, возможно, он также пишет модульные тесты для некоторых частей своего кода. Когда его работа будет завершена, он может также написать автоматизированные интеграционные тесты, хотя в его тестах могут отсутствовать некоторые сценарии данных, поскольку они охватываются тестами Мэри, и он больше сосредоточится на использовании самого пользовательского интерфейса. Это все еще автоматизированные приемочные тесты, но цель состоит в том, чтобы избежать излишней избыточности существующих тестовых примеров. Как и в случае с кодом Мэри, когда код Сэма проверяется и объединяется в общую ветвь, автоматически выполняется набор регрессии пользовательского интерфейса.

После того, как Сэм и Мэри закончат, Джо, QA-инженер, может провести некоторое ручное приемочное тестирование всей истории, а также некоторое исследовательское тестирование, чтобы убедиться, что сумасшедшее поведение пользователя не приводит к сумасшедшему поведению приложения. Эта комбинация автоматических и ручных тестов составляет приемочное тестирование истории. По завершении ряда историй набор изменений внедряется в интегрированную тестовую среду для окончательного тестирования. Это может представлять собой приемочное тестирование более высокого уровня, а также дополнительное регрессионное тестирование. Дополнительное приемочное тестирование может выполняться инженерами по обеспечению качества, которые следят за тем, чтобы набор историй обеспечивал согласованный рабочий процесс для пользователей и, в конечном итоге, ценность некоторых требований более высокого уровня, которые ранее были разбиты на историю.

Заключительное регрессионное тестирование проводится поверх ранее запущенного пакета автоматизированной регрессии. Возможно, это приложение имеет некоторые функции, которые просто требуют человеческого вмешательства, или, возможно, это окончательная оценка другой группы тестировщиков, которые следят за соблюдением стиля и стандартов поведения.

Как видно из этого рабочего процесса, многие из этих шагов было бы неэффективно выполнить раньше, чем когда они описаны выше. Автоматические тесты являются исключением, так как они могут и часто выполняться многократно на протяжении всего процесса. Однако большинство других «этапов» тестирования - это просто естественное развитие, основанное на развитии выпуска.

Источники:

- [What is Software Testing Life Cycle \(STLC\)](#)
- [What Is Software Testing Life Cycle \(STLC\)?](#)
- [What is Software Testing Life Cycle \(STLC\) & STLC Phases](#)
- [Why is testing split into distinct stages?](#)

Модели разработки ПО

Чтобы лучше разобраться в том, как тестирование соотносится с программированием и иными видами проектной деятельности, для начала рассмотрим самые основы — модели разработки (lifecycle model) ПО (как часть жизненного цикла (software lifecycle) ПО). При этом сразу подчеркнем, что разработка ПО является лишь частью жизненного цикла ПО, и здесь мы говорим именно о разработке.

Модель разработки ПО (Software Development Model, SDM) — структура, систематизирующая различные виды проектной деятельности, их взаимодействие и последовательность в процессе разработки ПО. Выбор той или иной модели зависит от масштаба и сложности проекта, предметной области, доступных ресурсов и

множества других факторов. Выбор модели разработки ПО серьёзно влияет на процесс тестирования, определяя выбор стратегии, расписание, необходимые ресурсы и т.д.

Знать и понимать модели разработки ПО нужно затем, чтобы уже с первых дней работы осознавать, что происходит вокруг, что, зачем и почему вы делаете. Многие начинающие тестировщики отмечают, что ощущение бессмыслицы происходящего посещает их, даже если текущие задания интересны. Чем полнее вы будете представлять картину происходящего на проекте, тем яснее вам будет виден ваш собственный вклад в общее дело и смысл того, чем вы занимаетесь. Еще одна важная вещь, которую следует понимать, состоит в том, что никакая модель не является догмой или универсальным решением. Нет идеальной модели. Есть та, которая хуже или лучше подходит для конкретного проекта, конкретной команды, конкретных условий.

Водопадная модель (waterfall model) сейчас представляет скорее исторический интерес, т.к. в современных проектах практически неприменима, исключая авиастроение, военную или космическую отрасли, медицину и финансовый сектор. Она предполагает однократное выполнение каждой из фаз проекта, которые, в свою очередь, строго следуют друг за другом. Очень упрощенно можно сказать, что в рамках этой модели в любой момент времени команде «видна» лишь предыдущая и следующая фаза. В реальной же разработке ПО приходится «видеть весь проект целиком» и возвращаться к предыдущим фазам, чтобы исправить недоработки или что-то уточнить.



Рисунок 2.1.а — Водопадная модель разработки ПО

К недостаткам водопадной модели принято относить тот факт, что участие пользователей ПО в ней либо не предусмотрено вообще, либо предусмотрено лишь косвенно на стадии однократного сбора требований. С точки зрения же тестирования эта модель плоха тем, что тестирование в явном виде появляется здесь лишь с середины развития проекта, достигая своего максимума в самом конце.

V-образная модель (V-model)

V-модель (V-model): Модель, описывающая процессы жизненного цикла разработки программного обеспечения с момента составление спецификации требований до этапа сопровождения. V модель показывает интеграцию процессов тестирования в каждую фазу цикла разработки программного обеспечения. (ISTQB)

V-образная модель (V-model) является логическим развитием водопадной. Можно заметить (рисунок 2.1.b), что в общем случае как водопадная, так и v-образная модели жизненного цикла ПО могут содержать один и тот же набор стадий, но принципиальное отличие заключается в том, как эта информация используется в процессе реализации проекта. Очень упрощенно можно сказать, что при использовании v-образной модели на каждой стадии «на спуске» нужно думать о том, что и как будет происходить на соответствующей стадии «на подъёме». Тестирование здесь появляется уже на самых ранних стадиях развития проекта, что позволяет минимизировать риски, а также обнаружить и устранить множество потенциальных проблем до того, как они станут проблемами реальными.

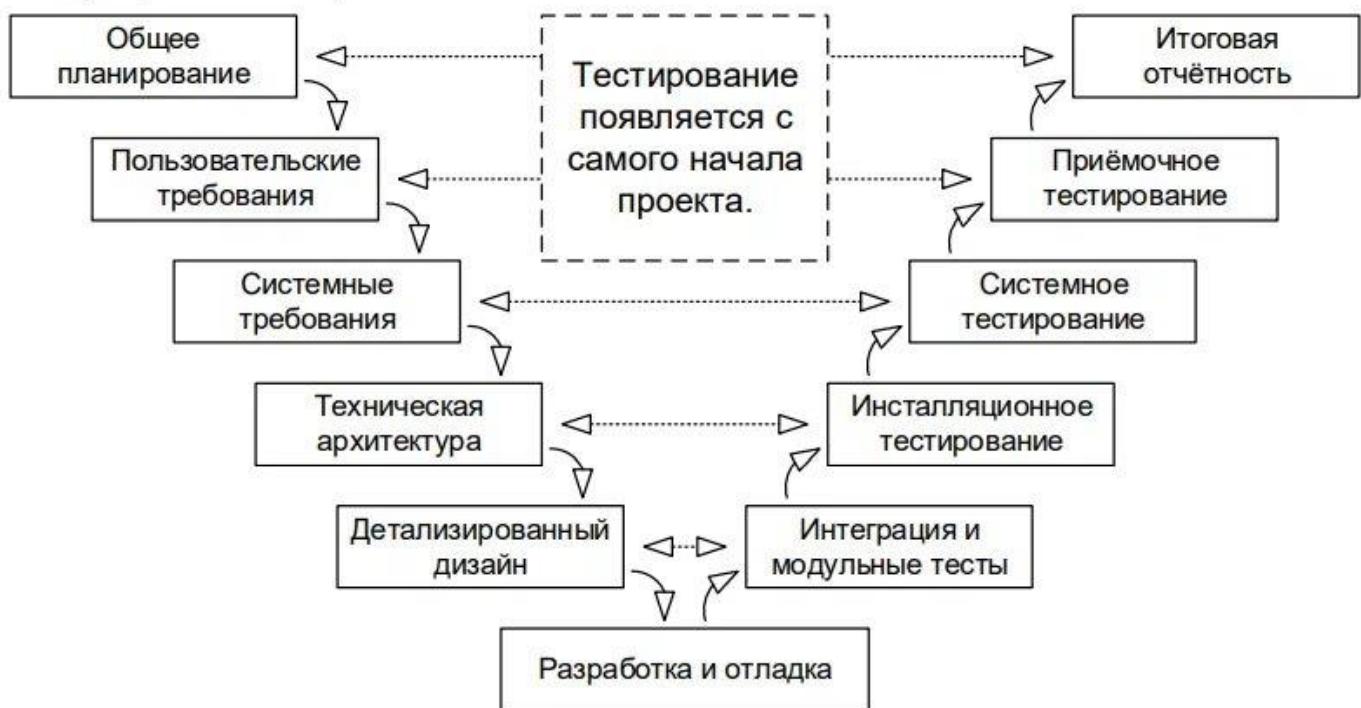


Рисунок 2.1.b — V-образная модель разработки ПО

Итерационная инкрементальная модель (iterative model, incremental model)

Инкрементная модель разработки (incremental development model): Модель жизненного цикла разработки, в которой проект разделен на серию приращений, каждое из которых добавляет часть функциональности в общих требованиях проекта. Требования приоритезированы и внедряются в порядке приоритетов. В некоторых (но не во всех) версиях этой модели жизненного цикла каждый подпроект следует «мини V-модели» со своими собственными фазами проектирования, кодирования и тестирования. (ISTQB)

Итеративная модель разработки (iterative development model): Модель жизненного цикла разработки, в которой проект разделен обычно на большое количество итераций. Итерация это полный цикл разработки, завершающийся выпуском (внутренним или внешним) рабочего продукта, являющегося частью конечного разрабатываемого продукта, который разрастается от итерации к итерации. (ISTQB)

Итерационная инкрементальная модель является фундаментальной основой современного подхода к разработке ПО. Ключевой особенностью данной модели является разбиение проекта на относительно небольшие промежутки (итерации), каждый из которых в общем случае может включать в себя все классические стадии, присущие водопадной и v-образной моделям. Итогом итерации является приращение (инкремент) функциональности продукта, выраженное в промежуточном билде (build).

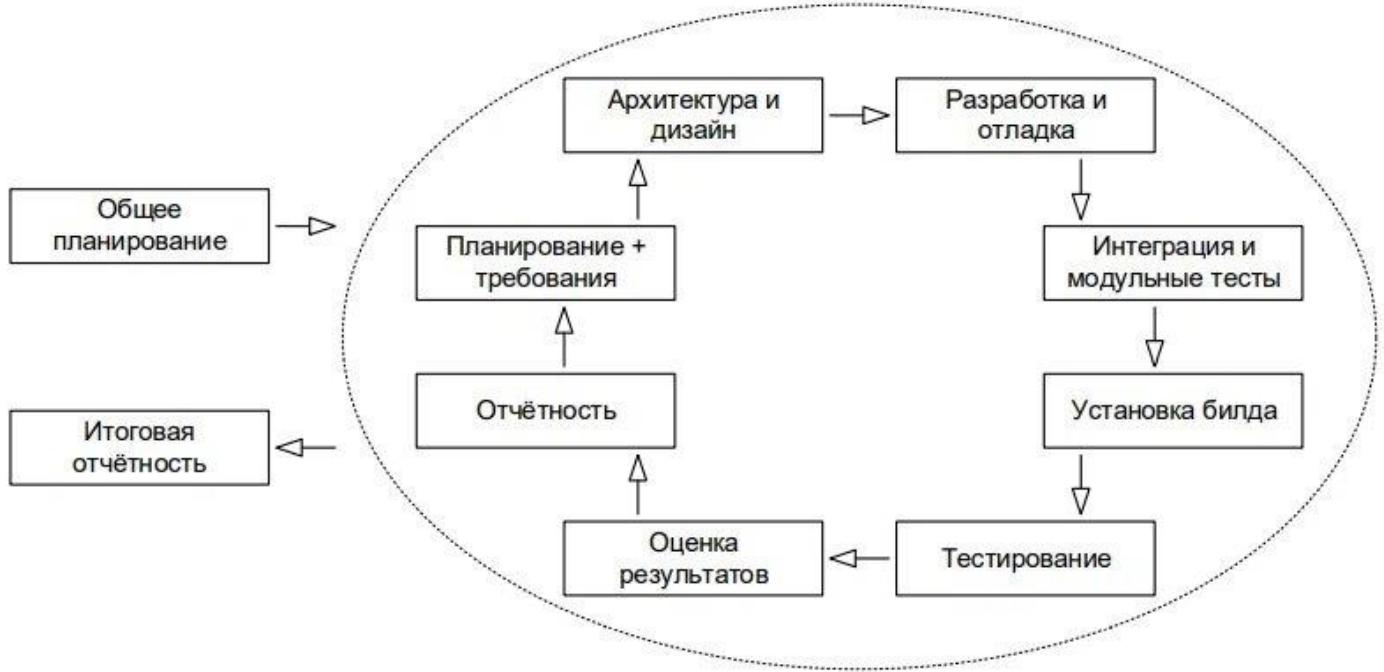


Рисунок 2.1.с — Итерационная инкрементальная модель разработки ПО

Длина итераций может меняться в зависимости от множества факторов, однако сам принцип многократного повторения позволяет гарантировать, что и тестирование, и демонстрация продукта конечному заказчику (с получением обратной связи) будет активно применяться с самого начала и на протяжении всего времени разработки проекта. Во многих случаях допускается распаралеливание отдельных стадий внутри итерации и активная доработка с целью устранения недостатков, обнаруженных на любой из (предыдущих) стадий. Итерационная инкрементальная модель очень хорошо зарекомендовала себя на объемных и сложных проектах, выполняемых большими командами на протяжении длительных сроков. Однако к основным недостаткам этой модели часто относят высокие накладные расходы, вызванные высокой «бюрократизированностью» и общей громоздкостью модели.

Сpirальная модель (spiral model)

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

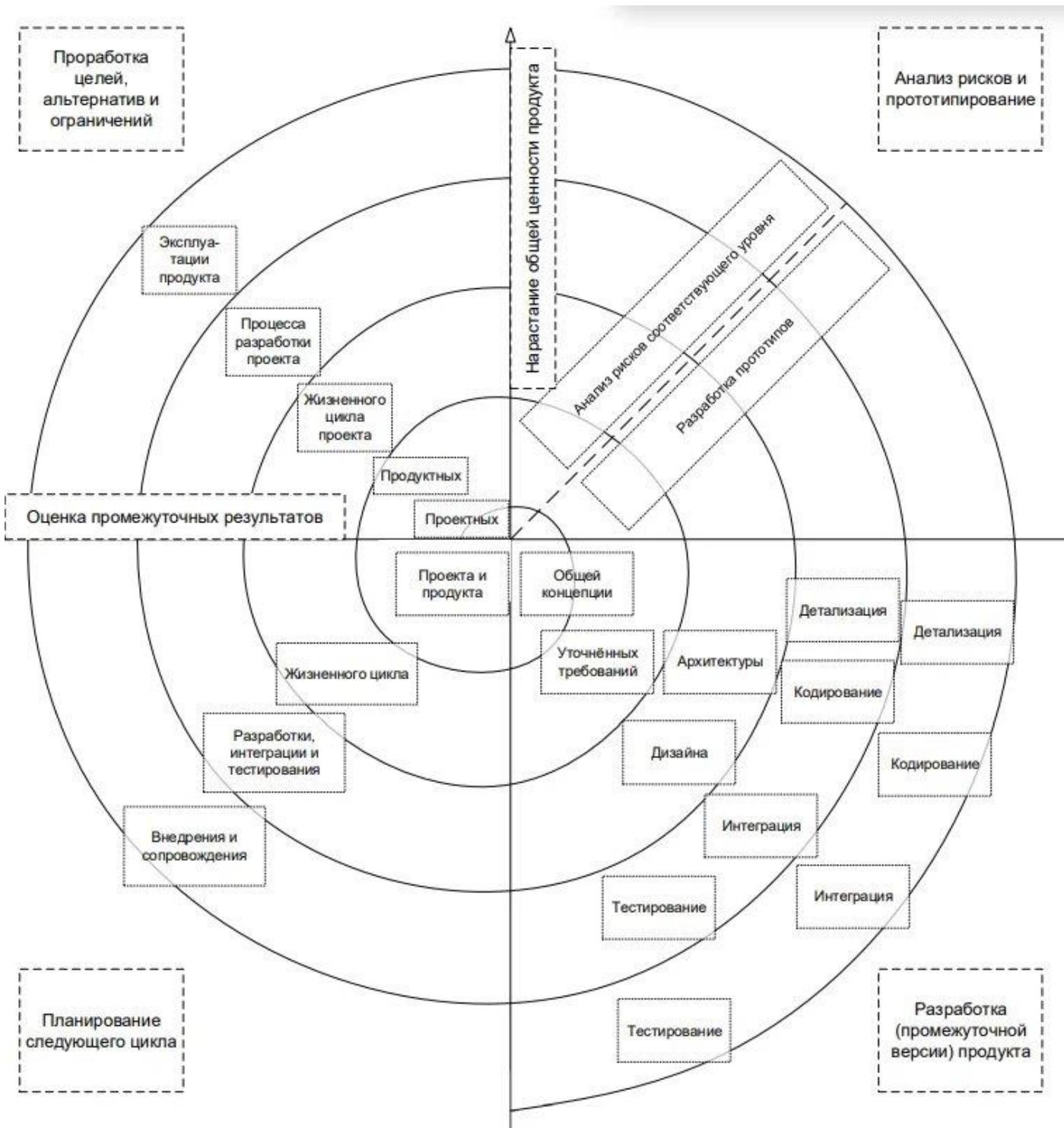


Рисунок 2.1.d — Спиральная модель разработки ПО

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

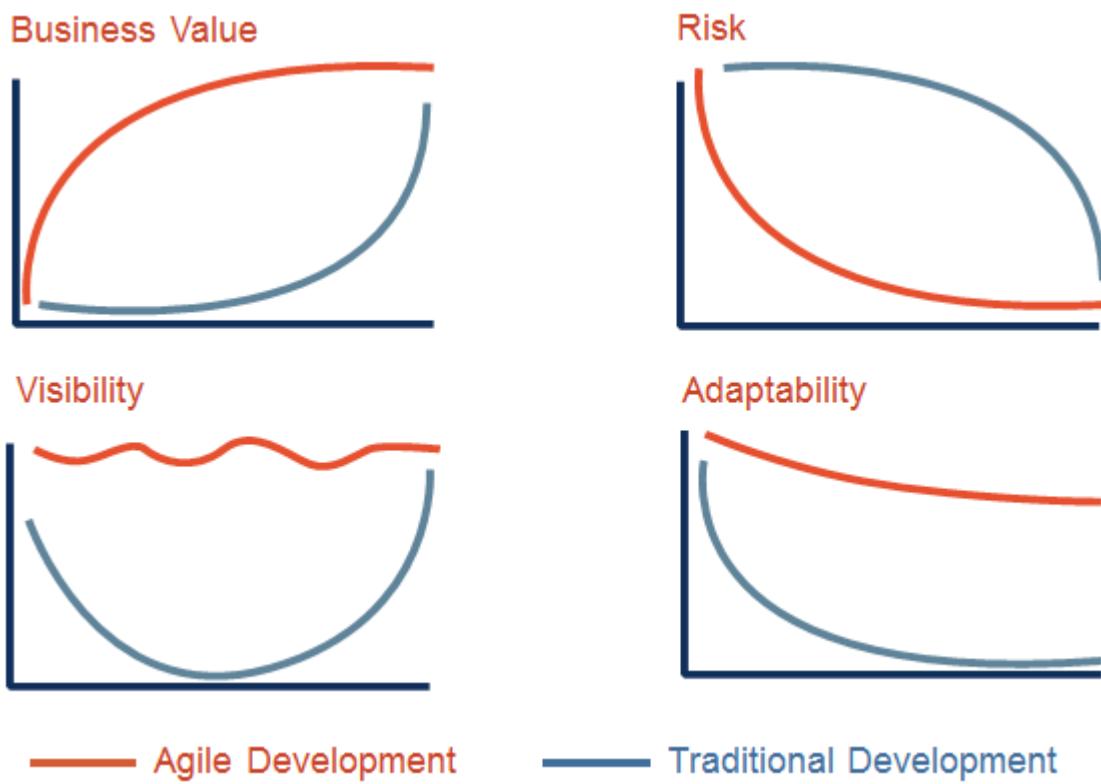
- проработка целей, альтернатив и ограничений;
- анализ рисков и прототипирование;
- разработка (промежуточной версии) продукта;
- планирование следующего цикла.

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

Гибкая модель (agile model)

Гибкая методология разработки программного обеспечения (agile software development): Группа методологий разработки программного обеспечения, основанных на итеративной поэтапной разработке, где требования и решения развиваются посредством сотрудничества между самоорганизующимися межфункциональными командами. (ISTQB)

Гибкая модель представляет собой совокупность различных подходов к разработке ПО и базируется на т.н. «agile-манифесте». Положенные в основу гибкой модели подходы являются логическим развитием и продолжением всего того, что было за десятилетия создано и опробовано в водопадной, v-образной, итерационной инкрементальной, спиральной и иных моделях. Причём здесь впервые был достигнут ощутимый результат в снижении бюрократической составляющей и максимальной адаптации процесса разработки ПО к мгновенным изменениям рынка и требований заказчика.



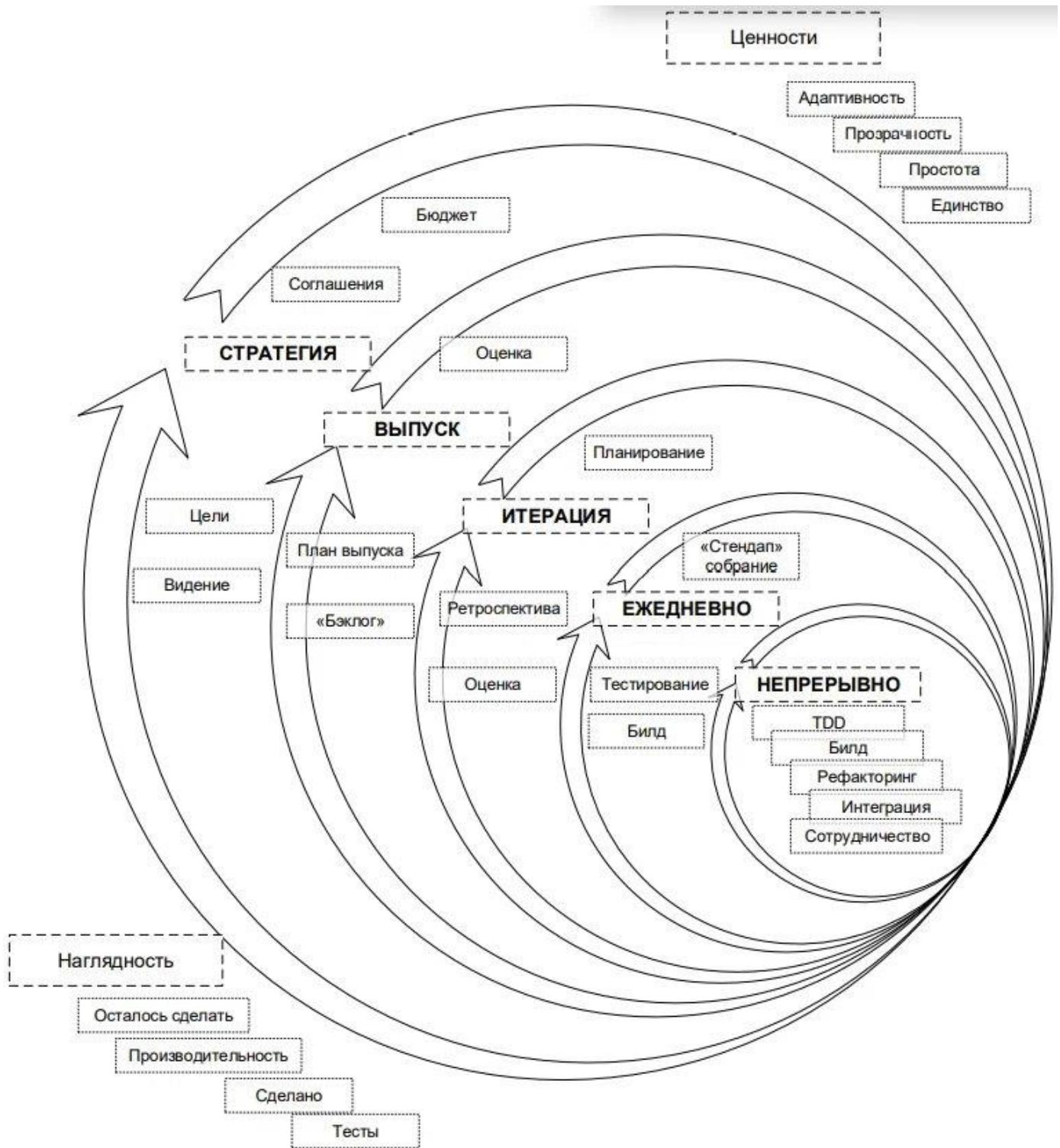


Рисунок 2.1.е — Суть гибкой модели разработки ПО

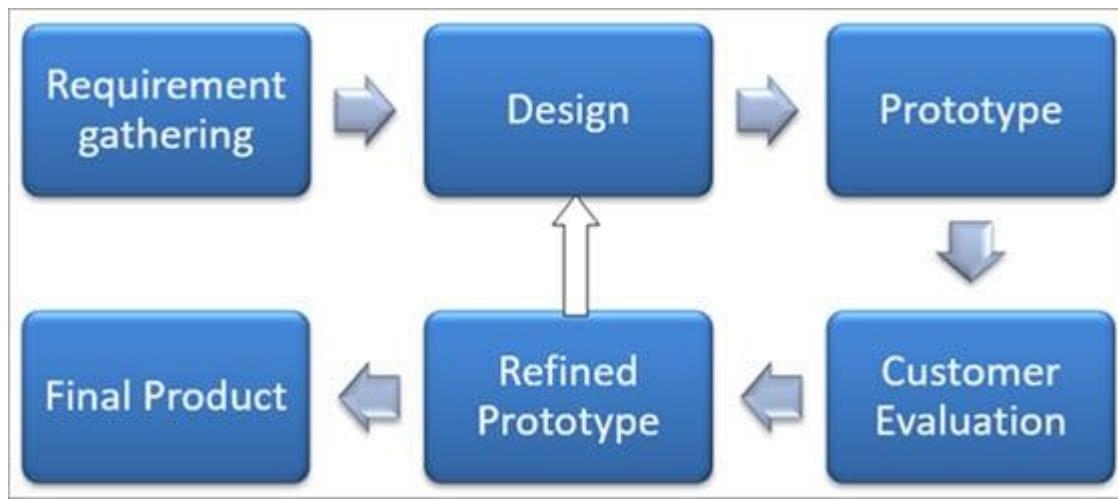
Очень упрощенно (почти на грани допустимого) можно сказать, что гибкая модель представляет собой облегченную с точки зрения документации смесь итерационной инкрементальной и спиральной моделей; при этом следует помнить об «agile-манифесте» и всех вытекающих из него преимуществах и недостатках.

Главным недостатком гибкой модели считается сложность ее применения к крупным проектам, а также частое ошибочное внедрение ее подходов, вызванное недопониманием фундаментальных принципов модели. Тем не менее можно утверждать, что всё больше и больше проектов начинают использовать гибкую модель разработки.

В некоторых источниках можно найти еще пару моделей:

Prototype Model

Prototype Model - это модель, в которой прототип разрабатывается раньше, чем фактическое программное обеспечение. Модели-прототипы обладают ограниченными функциональными возможностями и неэффективной производительностью по сравнению с реальным программным обеспечением. Фиктивные функции используются для создания прототипов. Это ценный механизм для понимания потребностей клиентов. Внедряются отзывы, и прототип снова проверяется заказчиком на предмет любых изменений. Этот процесс продолжается до тех пор, пока модель не будет принята заказчиком. После сбора требований создается быстрый дизайн и создается прототип, который представляется заказчику для оценки. Отзывы клиентов и уточненные требования используются для модификации прототипа и снова представляются заказчику для оценки. После того, как заказчик утверждает прототип, он используется в качестве требования для создания реального программного обеспечения. Фактическое программное обеспечение построено с использованием подхода модели водопада.



Преимущества прототипа модели: Модель прототипа снижает стоимость и время разработки, так как дефекты обнаруживаются намного раньше. Отсутствующие функции или функциональные возможности или изменение требований могут быть выявлены на этапе оценки и могут быть реализованы в доработанном прототипе. Вовлечение клиента на начальном этапе уменьшает путаницу в требованиях или понимании какой-либо функциональности.

Недостатки прототипа модели: Поскольку заказчик участвует на каждом этапе, заказчик может изменить требования к конечному продукту, что увеличивает сложность объема работ и может увеличить время доставки продукта.

Модель Большого Взрыва (Big Bang Model)

Big Bang Model не имеет определенного процесса. Деньги и усилия объединяются, поскольку вход и выход представляют собой разработанный продукт, который может совпадать, а может и не совпадать с тем, что нужно заказчику. Модель Большого Взрыва не требует особого планирования и составления графиков. Разработчик выполняет анализ требований и кодирование, а также разрабатывает продукт в соответствии с его пониманием. Эта модель используется только для небольших проектов. Нет команды тестирования и формального тестирования не проводится, и это может быть причиной провала проекта.

Преимущества модели большого взрыва: Это очень простая модель. Требуется меньше планирования и составления графиков. Разработчик может создавать собственное программное обеспечение.

Недостатки модели большого взрыва: Модели Большого взрыва нельзя использовать для крупных, текущих и сложных проектов. Высокий риск и неопределенность.

Источники:

- [Святослав Куликов “Тестирование программного обеспечения. Базовый курс”](#). Глава 2.
- [SDLC \(Software Development Life Cycle\) Phases, Process, Models](#)

Agile

Agile - это способность создавать и реагировать на изменения. Это способ справиться с неопределенной и неспокойной средой и в конечном итоге преуспеть в ней. Авторы Agile Manifesto выбрали «Agile» в качестве названия всей этой идеи, потому что это слово олицетворяет адаптивность и реакцию на изменения, которые так важны для их подхода. На самом деле речь идет об осмыслении того, как вы можете понять, что происходит в среде, в которой вы находитесь сегодня, определить, с какой неопределенностью вы сталкиваетесь, и выяснить, как вы можете адаптироваться к этому по мере продвижения.

Гибкая разработка программного обеспечения - это больше, чем такие фреймворки, как Scrum, Extreme Programming или Feature-Driven Development (FDD). Гибкая разработка программного обеспечения - это больше, чем такие практики, как парное программирование, разработка на основе тестирования, стендапы, сессии планирования и спринты.

Гибкая разработка программного обеспечения - это общий термин для набора структур и практик, основанных на ценностях и принципах, изложенных в Манифесте гибкой разработки программного обеспечения и 12 принципах, лежащих в его основе. Когда вы подходите к разработке программного обеспечения особым образом, обычно хорошо жить в соответствии с этими ценностями и принципами и использовать их, чтобы помочь понять, что делать в вашем конкретном контексте.

Одна вещь, которая отличает Agile от других подходов к разработке программного обеспечения, - это сосредоточение внимания на людях, выполняющих работу, и на том, как они работают вместе. Решения развиваются в результате сотрудничества между самоорганизующимися кросс-функциональными командами, использующими соответствующие методы для своего контекста. Сообщество Agile-разработчиков программного обеспечения уделяет большое внимание совместной работе и самоорганизующейся команде. Это не значит, что менеджеров нет. Это означает, что команды могут самостоятельно определять, как они собираются подходить к делу. Это означает, что эти команды кросс-функциональны. Этим командам не обязательно должны быть задействованы определенные роли, просто когда вы собираете команду вместе, вы убедитесь, что у вас есть все необходимые навыки в команде. Еще есть место для менеджеров.

Менеджеры следят за тем, чтобы члены команды имели или приобрели правильный набор навыков. Менеджеры создают среду, которая позволяет команде быть успешной. Менеджеры в основном отступают и позволяют своей команде выяснить, как они собираются выпускать продукты, но они вмешиваются, когда команды пытаются, но не могут решить проблемы. Когда большинство команд и организаций начинают заниматься гибкой разработкой, они сосредотачиваются на практиках, которые помогают в совместной работе и организации работы, и это здорово. Однако другой ключевой набор практик, которым не так часто следуют, но которые должны соблюдать, - это конкретные технические практики, которые напрямую связаны с разработкой программного обеспечения таким образом, чтобы помочь вашей команде справиться с неопределенностью. Эти технические приемы очень важны, и их нельзя упускать из виду.

В конечном итоге Agile - это образ мышления, основанный на ценностях и принципах Agile Manifesto. Эти ценности и принципы содержат указания о том, как создавать изменения и реагировать на них, а также как справляться с неопределенностью. Можно сказать, что первое предложение Agile Manifesto заключает в себе всю идею: «Мы открываем лучшие способы разработки программного обеспечения, делая это и помогая другим делать это». Когда вы сталкиваетесь с неуверенностью, попробуйте что-то, что, по вашему мнению, может сработать, получите обратную связь и внесите соответствующие корректизы. При этом помните о ценностях и принципах. Позвольте вашему контексту определять, какие рамки, практики и методы вы используете для сотрудничества со своей командой и предоставления ценности вашим клиентам.

Если Agile - это образ мышления, то что это говорит об идеи Agile-методологий? Чтобы ответить на этот вопрос, вам может быть полезно иметь четкое определение методологии. Алистер Кокберн предположил, что методология - это набор условностей, которым команда соглашается следовать. Это означает, что у каждой команды будет своя собственная методология, которая будет в малой или большой степени отличаться от методологии любой другой команды. Таким образом, Agile-методологии - это условности, которым команда решает следовать в соответствии с ценностями и принципами Agile. Вы, наверное, скажете: «Подождите, - я думал, что Scrum и XP - это Agile-методологии». Алистер применил термин "framework" к этим концепциям. Они, безусловно, родились на основе методологии одной команды, но они стали

фреймворками, когда были обобщены для использования другими командами. Эти фреймворки помогают понять, где команда начинает свою методологию, но они не должны быть ее методологией. Команде всегда необходимо адаптировать использование фреймворка, чтобы оно соответствовало его контексту.

Ключевые концепции Agile

- Пользовательские истории (**User Stories**): после консультации с заказчиком или владельцем продукта команда делит работу, которую необходимо выполнить, на функциональные этапы, называемые «пользовательскими историями». Ожидается, что каждая пользовательская история внесет свой вклад в ценность всего продукта;
- Ежедневные собрания (**Daily Meeting**): каждый день в одно и то же время группа собирается, чтобы ознакомить всех с информацией, которая имеет жизненно важное значение для координации: каждый член команды кратко описывает все «завершенные» вклады и любые препятствия, стоящие на их пути;
- Персонажи (**Personas**): когда этого требует проект - например, когда пользовательский опыт является основным фактором результатов проекта - команда создает подробные синтетические биографии фиктивных пользователей будущего продукта: они называются personas;
- Команда (**Team**): «Команда» в Agile понимании - это небольшая группа людей, назначенных на один и тот же проект или effort, почти все из них на постоянной основе. Незначительное меньшинство членов команды может работать неполный рабочий день или иметь конкурирующие обязанности;
- Инкрементальная разработка (**Incremental Development**): почти все Agile-команды отдают предпочтение стратегии инкрементального развития; в контексте Agile это означает, что можно использовать каждую последующую версию продукта, и каждая основывается на предыдущей версии, добавляя видимые для пользователя функциональные возможности;
- Итеративная разработка (**Iterative Development**): Agile-проекты являются итеративными, поскольку они намеренно позволяют «повторять» действия по разработке программного обеспечения и потенциально «пересматривать» одни и те же рабочие продукты;
- Ретроспектива (**Milestone Retrospective**): после того, как проект был запущен в течение некоторого времени или в конце проекта, все постоянные члены команды (не только разработчики) вкладывают от одного до трех дней в подробный анализ значимых событий проекта.

The Agile Manifesto:

- люди и взаимодействие важнее процессов и инструментов;
- работающий продукт важнее исчерпывающей документации;
- сотрудничество с заказчиком важнее согласования условий контракта;
- готовность к изменениям важнее следования первоначальному плану.

Основополагающие принципы Agile Manifesto:

- наивысшим приоритетом признается удовлетворение заказчика за счет ранней и бесперебойной поставки ценного программного обеспечения;
- изменение требований приветствуется даже в конце разработки (это может повысить конкурентоспособность полученного продукта);
- частая поставка работающего программного обеспечения (каждые пару недель или пару месяцев с предпочтением меньшего периода);
- общение представителей бизнеса с разработчиками должно быть ежедневным на протяжении всего проекта;
- проекты следует строить вокруг заинтересованных людей, которых следует обеспечить нужными условиями работы, поддержкой и доверием;
- самый эффективный метод обмена информацией в команде — личная встреча;
- работающее программное обеспечение — лучший измеритель прогресса;
- спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп на неопределённый срок;

- постоянное внимание к техническому совершенству и хорошему проектированию увеличивают гибкость;
- простота как искусство не делать лишней работы очень важна;
- лучшие требования, архитектура и проектные решения получаются у самоорганизующихся команд;
- команда регулярно обдумывает способы повышения своей эффективности и соответственно корректирует рабочий процесс.

Существуют методологии, которые придерживаются ценностей и принципов заявленных в Agile Manifesto, некоторые из них:

- Agile Modeling - набор понятий, принципов и приемов (практик), позволяющих быстро и просто выполнять моделирование и документирование в проектах разработки программного обеспечения. Не включает в себя детальную инструкцию по проектированию, не содержит описаний, как строить диаграммы на UML. Основная цель: эффективное моделирование и документирование; но не охватывает программирование и тестирование, не включает вопросы управления проектом, развертывания и сопровождения системы. Однако включает в себя проверку модели кодом.
- Agile Unified Process (AUP) упрощенная версия IBM Rational Unified Process (RUP), разработанная Скоттом Амблером, которая описывает простое и понятное приближение (модель) для создания программного обеспечения для бизнес-приложений.
- Agile Data Method — группа итеративных методов разработки программного обеспечения, в которых требования и решения достигаются в рамках сотрудничества разных кросс-функциональных команд.
- DSDM основан на концепции быстрой разработки приложений (Rapid Application Development, RAD). Представляет собой итеративный и инкрементный подход, который придаёт особое значение продолжительному участию в процессе пользователя/потребителя.
- Essential Unified Process (EssUP).
- Экстремальное программирование (Extreme programming, XP).
- Feature driven development (FDD) — функционально-ориентированная разработка. Используемое в FDD понятие функции или свойства (англ. feature) системы достаточно близко к понятию прецедента использования, используемому в RUP, существенное отличие — это дополнительное ограничение: «каждая функция должна допускать реализацию не более, чем за две недели». То есть если сценарий использования достаточно мал, его можно считать функцией. Если же велик, то его надо разбить на несколько относительно независимых функций.
- Getting Real — итеративный подход без функциональных спецификаций, использующийся для веб-приложений. В данном методе сперва разрабатывается интерфейс программы, а потом её функциональная часть.
- OpenUP — это итеративно-инкрементальный метод разработки программного обеспечения. Позиционируется как легкий и гибкий вариант RUP. OpenUP делит жизненный цикл проекта на четыре фазы: начальная фаза, фазы уточнения, конструирования и передачи. Жизненный цикл проекта обеспечивает предоставление заинтересованным лицам и членам коллектива точек ознакомления и принятия решений на протяжении всего проекта. Это позволяет эффективно контролировать ситуацию и вовремя принимать решения о приемлемости результатов. План проекта определяет жизненный цикл, а конечным результатом является окончательное приложение.
- Scrum устанавливает правила управления процессом разработки и позволяет использовать уже существующие практики кодирования, корректируя требования или внося тактические изменения. Использование этой методологии дает возможность выявлять и устранять отклонения от желаемого результата на более ранних этапах разработки программного продукта.
- Бережливая разработка программного обеспечения (lean software development) использует подходы из концепции бережливого производства.

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.

Особенности тестирования в Agile

В Agile, тестирование - это ответственность каждого. Критерий качества должен соблюдаться на протяжении всего цикла. В то время как бизнес-аналитики сосредотачиваются на создании подробных пользовательских историй, разработчики сосредотачиваются на разработке качественного кода, QA несет ответственность за уточнение критериев приемлемости для каждой пользовательской истории, тестирование завершенной функциональности в каждом спринте с точки зрения клиента и проверка всей ранее выполненной функциональности. Роли и ответственность QA не ограничиваются только тестированием в Agile, но также включают в себя следующее:

- QA работают в тесном сотрудничестве с владельцами продуктов, бизнес-аналитиками и командой разработчиков, чтобы понять разрабатываемый продукт, для кого он предназначен и каковы будут критерии успеха продукта.
- Они углубляются в критерии приемки, созданные бизнес-аналитиками, для написания тестовых примеров, визуализации рабочего процесса, тестирования стандартных элементов и проведения негативного тестирования.
- Опытный QA также хорошо осведомлен об объеме выпуска и соответственно устанавливает границы своего тестирования.
- Тестировщики должны общаться с командой и задавать вопросы во время тестирования, что позволяет им выявлять пробелы в требованиях или получать ответы на нерешенные вопросы. Коммуникация и сотрудничество с командой имеют решающее значение, поскольку они помогают сделать тестирование более точным и надежным. Это также помогает достичь необходимого темпа, чтобы двигаться быстрее, с ранними отзывами о тестировании и повышенным качеством.
- Как член Agile команды, тестировщик всегда должен быть синхронизирован с командой проекта, посещая сессии планирования спринтов для выявления возможных проблемных областей и ежедневных митингов для содействия сотрудничеству. Посещение ретроспективы спринта дает возможность выявить слабые места и определить решения внутри команды. Посещение митингов по обзору спринга или демонстрации продукта позволяет тестировщику увидеть, как работает новая функция, и дает им возможность задать важные вопросы разработчикам.
- Документирование сценариев тестирования и выполнения тестов с доказательствами важно для тестировщиков, но оно должно быть минимальным и кратким.

Какие стратегии использует QA для проведения Agile-тестирования?

В каждой организации есть разные подходы и стратегии, которые они используют для тестирования приложений. Методология Agile предполагает подготовку документации, достаточной только для удовлетворения насущных потребностей команды. Таким образом, QA подготовят достаточно высокоуровневой документации для стратегий тестирования и планов для руководства командой. Ниже приведены несколько стратегий, которым я следую при подготовке к тестированию в Agile:

- Чрезвычайно важно иметь четкий план до начала тестирования. Перед началом тестирования спланируйте свое время и тест-кейсы, и это позволит вам сразу же погрузиться в тестирование после развертывания приложения.
- Я использую сочетание ручного и автоматизированного тестирования. Автоматизированное тестирование помогает мне ускорить мои регрессионные тесты с помощью наборов тестов перед сборкой, а ручное тестирование помогает мне, когда мне нужно проводить более сфокусированное тестирование.
- Как тестировщик, я всегда верю в проведение смок-тестов основных или базовых функций сразу после развертывания приложения, что помогает мне выявлять любые ошибки критические раньше.
- Выполнение приемочных испытаний, основанных на критериях приемки, чтобы обеспечить лучшее покрытие тестами. Каждый критерий приемки может иметь одно или несколько приемочных испытаний и ориентирован на заданные условия.

- Тестирование эффективности потока помогает определить, могут ли пользователи беспрепятственно перемещаться по продукту. Это помогает определить, сбивает ли вас с толку какая-либо часть потока, и помогает определить, нужно ли вам добавлять или удалять какие-либо шаги.
- Проверка бизнес-правил и определений данных - важная часть тестирования.
- Проведение исследовательского тестирования позволяет тестировщикам идти по неопределенному пути и находить скрытые риски и дефекты в приложении.
- Проведение регрессионного тестирования - важная часть гибкого тестирования. Регрессионное тестирование гарантирует, что проверенные функции работают должным образом после введения новых функций.
- Использование кросбраузерного тестирования чрезвычайно важно для гибкого тестирования, так как тестировщик может быстро протестировать несколько устройств и браузеров за ограниченное время.
- Наличие приложений для обмена сообщениями в реальном времени, таких как Slack и Zoom, позволяет всем в команде быть на связи и быстро отвечать на важные вопросы.

Источники:

- [Agile 101](#)
- [Гибкая методология разработки](#)
- [Руководство тестировщика по Agile тестированию](#)

Доп. материал:

- [ISTQB Agile Tester Extension Exam Theory Study Material Part 1](#)
- [ISTQB Agile Tester Extension Exam Crash Course Part 1](#)
- [Agile Glossary](#)
- [Agile Software Development, lessons learned by Jerome Kehrli](#)
- [Организация процесса тестирования в Agile команде с помощью квадрантов тестирования. - презентация](#)
- [10 примеров эффективного общения для тестировщиков](#)
- [Как выживать тестировщику в Agile среде](#)
- [Стратегия тестирования краткосрочного проекта](#)

Scrum

Scrum – наиболее популярный Agile-фреймворк, для многих людей эти термины являются синонимами. Scrum – это фреймворк процесса, используемый для управления разработкой продукта и другой работой, связанной с знаниями. Скрам является эмпирическим в том смысле, что дает командам возможность установить гипотезу о том, как они думают, что что-то работает, опробовать это, проанализировать полученный опыт и внести соответствующие корректизы. То есть при правильном использовании фреймворка. Скрам структурирован таким образом, чтобы команды могли использовать практики из других фреймворков, которые имеют смысл для контекста команды.

Scrum лучше всего подходит в случае, когда кросс-функциональная команда работает в среде разработки продукта, где есть нетривиальный объем работы, которую можно разделить на более чем одну 2–4-недельную итерацию.

Ценности:

- Преданность (Commitment): Члены команды лично привержены достижению целей команды;
- Смелость (Courage): Члены команды поступают правильно и работают над сложными проблемами;
- Сфокусированность (Focus): Сконцентрируйтесь на работе, намеченной для спринта, и целях команды;
- Открытость (Openness): Члены команды и заинтересованные стороны открыто рассказывают обо всей работе и проблемах, с которыми сталкивается команда;
- Уважение (Respect): Члены команды уважают друг друга за способности и независимость.

Принципы:

- Прозрачность (Transparency): Команда должна работать в среде, где каждый знает, с какими проблемами сталкиваются другие члены команды. Команды выявляют проблемы внутри организации, часто возникающие в течение длительного времени, которые мешают успеху команды;
- Инспекция (Inspection): Частые контрольные точки встроены в структуру, чтобы дать команде возможность поразмышлять о том, как работает процесс. Эти контрольные точки включают в себя Daily Scrum meeting и the Sprint Review Meeting;
- Адаптация (Adaptation): Команда постоянно изучает, как идут дела, и проверяет те пункты, которые кажутся бессмысленными.

События:

- **Спринт (Sprint):** это временной интервал в 2-4 недели, в течение которого команда создает потенциально готовый к поставке инкремент продукта;
- **Планирование спринта (Sprint Planning):** Команда начинает спринт с обсуждения, чтобы определить, над какими элементами из бэклога продукта (product backlog) они будут работать во время спринта. Конечным результатом планирования спринта является бэклог спринта (Sprint Backlog). Планирование спринта обычно состоит из двух частей. В первой части владелец продукта и остальная часть команды согласовывают, какие элементы бэклога продукта будут включены в спринт. Во второй части планирования спринта команда определяет, как они будут успешно доставлять идентифицированные элементы Product Backlog как часть потенциально возможного инкремента продукта. Команда может определить конкретные задачи, необходимые для этого, если это одна из их практик. Элементы Product Backlog, определенные для доставки, и задачи, если применимо, составляют бэклог спринта. После того, как команда и владелец продукта установят объем спринта, как описано в элементах Product Backlog, никакие дополнительные элементы не могут быть добавлены в журнал Sprint Backlog. Это защищает команду от изменений в рамках этого спринта;
- **Ежедневная встреча (Daily Scrum/Meeting):** это короткое (обычно не более 15 минут) обсуждение, во время которого команда координирует свои действия на следующий день. Дейли не предназначен для обсуждения статуса или обсуждения проблем;
- **Обзор спринта (Sprint Review):** в конце спринта вся команда (включая владельца продукта) рассматривает результаты спринта с заинтересованными сторонами продукта. Цель этого обсуждения - обсудить, продемонстрировать и потенциально дать заинтересованным сторонам возможность использовать инкремент для получения обратной связи. Обзор спринта не предназначен для предоставления отчета о состоянии (status report). Отзывы об обзоре спринта помещаются в Product Backlog для дальнейшего рассмотрения;
- **Ретроспектива спринта (Sprint Retrospective):** в конце спринта после обзора спринта (sprint review) команда (включая владельца продукта) должна подумать о том, как дела шли во время предыдущего спринта, и определить корректировки, которые они могут внести в будущем. Результатом этой ретроспективы является как минимум одно действие, включенное в бэклог следующего спринта;
- **Упорядочение бэклога ([Grooming](#)):**

Артефакты:

- **Бэклог продукта (Product Backlog):** это упорядоченный список всех возможных изменений, которые могут быть внесены в продукт. Пункты в бэклоге продукта являются вариантами, а не обязательствами, и то, что они существуют в бэклоге продукта, не гарантирует, что они будут доставлены. Владелец продукта постоянно ведет бэклог продукта, включая его содержание, доступность и порядок;
- **Бэклог спринта (Sprint Backlog):** это набор элементов из бэклога продукта, выбранных для доставки в спринте. После того, как команда определяет задачи, эти задачи необходимо выполнить для достижения цели спринта (Sprint Goal);
- **Инкремент (Increment):** это набор элементов из бэклога продукта, которые соответствуют Definition of Done к концу спринта. Владелец продукта может решить выпустить дополнение или развить его в будущих Спринтах;

- **Критерии Готовности** (*Definition of Done*): это общее соглашение команды о критериях, которым должен соответствовать элемент бэклога продукта, прежде чем он будет считаться выполненным\$
- **Пользовательские истории** ([User Story](#));
- **Цель спринта** ([Sprint Goal](#));
- **Диаграмма сгорания задач** ([Burndown chart](#)).

Роли:

- **Владелец продукта** (*Product Owner*): роль, ответственная перед командой за управлением бэклогом продукта для достижения результатов, к которым стремится команда. Роль product owner-а существует в Scrum для решения проблем, когда команда разработки имеет множественные противоречивые направления движения или отсутствие направления вообще в отношении того, что создавать;
- **Скрам Мастер** (*Scrum Master*): роль, ответственная перед командой за то, чтобы команда придерживалась гибких ценностей и принципов и следовала процессам и практикам, которые команда согласилась использовать. Изначально это имя предназначалось для обозначения кого-то, кто является экспертом в Scrum и, следовательно, может обучать других. Роль обычно не имеет никаких фактических полномочий. Люди, выполняющие эту роль, должны руководить с позиции влияния, часто занимая позицию лидера-слуги ([servant-leadership](#));
- **Команда разработки** (*Development Team*): состоит из людей, которые создают инкремент продукта внутри спринта. Основная ответственность команды разработки - обеспечить инкремент, который приносит пользу каждому спринту. Как распределить работу, это остается на усмотрение команды в зависимости от условий на тот момент.

Жизненный цикл Scrum:

- Создайте бэклог продукта;
- Владелец продукта и команда разработчиков проводят планирование спринта. Определите объем спринта в первой части планирования спринта и план реализации этого объема во второй половине планирования спринта;
- По мере продвижения спринта команда разработчиков выполняет работу, необходимую для доставки выбранных элементов бэклога продукта;
- Команда разработчиков ежедневно координирует свою работу в рамках Daily Scrum;
- В конце спринта группа разработчиков предоставляет элементы бэклога продукта, выбранные во время планирования спринта. Команда разработчиков проводит обзор спринта, чтобы показать клиенту инкремент и получить обратную связь. Команда разработчиков и владелец продукта также размышляют о том, как прошел Sprint до сих пор, и соответствующим образом адаптируют свои процессы во время ретроспективы;
- Команда повторяет шаги 2–5 до тех пор, пока не будет достигнут желаемый результат.

Стори поинты (Story points)

Чтобы оценить объем работы над Элементом Бэклога Продукта, Скрам-команды обычно используют Стори Поинты ([1](#), [2](#)). Это условная величина, позволяющая давать Элементам Бэклога относительные веса. Чаще всего для оценки в Стори Поинтах используются числа Фибоначчи (1, 2, 3, 5, 8, 13, ...), что позволяет провести оценку достаточно быстро.

Источники:

- [What is Scrum?](#)

Доп. материал:

- [Scrum Guide](#)
- [Scrum Glossary](#)
- [Мини-справочник и руководство по Scrum](#)
- [Scrum-мем на злобу дня](#)

- [Когда Scrum не работает. Пять основных проблем его применения](#)
- [Лекция 7: Этапы и мероприятия Scrum](#)
- [Как провести ретроспективу](#)
- [Покер планирования - как сделать процесс постановки задач максимально прозрачным и четким](#)

Особенности тестирования в Scrum:

- [Тестирование в рамках SCRUM. Тернии, грабли и успехи](#)
- [Регрессионное тестирование на Scrum-проектах: руководство по проведению](#)
- [QA-процесс в Miro: отказ от водопада и ручного тестирования, передача ответственности за качество всей команде](#)
- [Тесты должна писать разработка \(?\)](#)
- [The Role of QA in Sprint Planning](#)
- [Код-ревью для тестировщиков](#)

Подходы к разработке/тестированию (... - driven development/testing)

Тут и там можно встретить упоминания различных подходов к разработке и тестированию на основе чего-то, здесь краткое описание самых часто встречающихся вариантов:

- **TDD - Test Driven Development:** разработка на основе тестов основывается на повторении коротких циклов разработки: изначально пишется тест, покрывающий желаемое изменение, затем пишется программный код, который реализует желаемое поведение системы и позволит пройти написанный тест. Затем проводится рефакторинг написанного кода с постоянной проверкой прохождения тестов. 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 известен как BDD - Behavior 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.
- **BDD - Behaviour Driven Development:** это разработка, основанная на описании поведения. Определенный человек(или люди) пишет описания вида "я как пользователь хочу когда нажали кнопку пуск тогда показывалось меню как на картинке" (там есть специально выделенные ключевые слова). Программисты давно написали специальные тулы, которые подобные описания переводят в тесты (иногда совсем прозрачно для программиста). А дальше классическая разработка с тестами (TDD);
- **TDD - Type Driven Development:** при разработке на основе типов ваши типы данных и сигнатуры типов являются спецификацией программы. Типы также служат формой документации, которая гарантированно обновляется. Типы представляют из себя небольшие контрольные точки, благодаря которым, мы получаем множество мини-тестов по всему нашему приложению;
- **DDD - Domain Driven Design:** Предметно-ориентированное проектирование не является какой-либо конкретной технологией или методологией. DDD — это набор правил, которые позволяют принимать правильные проектные решения. Это набор принципов и схем, направленных на создание оптимальных систем объектов. Процесс разработки сводится к созданию программных абстракций, которые называются моделями предметных областей. В эти модели входит бизнес-логика, устанавливающая связь между реальными условиями области применения продукта и кодом;
- **FDD - Features Driven Development:** представляет собой попытку объединить наиболее признанные в индустрии разработки программного обеспечения методики, принимающие за основу важную для заказчика функциональность (свойства) разрабатываемого программного обеспечения. Основной

целью данной методологии является разработка реального, работающего программного обеспечения систематически, в установленные сроки;

- **MDD - Model Driven Development:** разработка, управляемая моделями - это стиль разработки программного обеспечения, когда модели становятся основными артефактами разработки, из которых генерируется код и другие артефакты;
- **PDD - Panic Driven Development:** это своеобразный антипаттерн разработки, который, к сожалению, мы все время от времени практикуем. По сути это то, что получается, когда процессы плохо налажены и команда импровизирует в условиях горящих сроков (новые задачи приоритетнее старых, код решает конкретные срочные задачи, но копится технический долг, тестирование в конце и т.д.);
- **ADD - API Driven Development:** разработка на основе API - это практика сначала проектирования и создания API, а затем создания на их основе остальной части приложения;
- **BDT - Behavior Driven Testing:** в тестировании на основе поведения ваши тесты основаны на user stories, которые описывают некоторые конкретные ожидаемые действия приложения. Вместо проверки деталей реализации вы фактически проверяете то, что важно больше всего: правильно ли приложение выполняет user stories. Еще одним преимуществом является понятность тестов для менеджеров, аналитиков и т.п.;
- **MDT - Model Driven Testing:** Тестирование на основе моделей - 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;
- **DDT - Data Driven Testing** (table-driven testing or parameterized testing): в тестировании на основе данных тестовые данные хранятся в виде таблицы. Оно позволяет одним скриптом выполнять тесты для всех тестовых данных из таблицы и ожидать результатов теста в той же таблице;
- **VDT - Value Driven Testing:** тестирование на основе ценности - это подход, в основе которого лежит анализ ценности и экономической целесообразности тестирования;

Источники:

- [TDDx2, BDD, DDD, FDD, MDD и PDD, или все, что вы хотите узнать о Driven Development](#)
- [The Basics of API-Driven Development](#)
- [Behavior-Driven Testing для iOS используя Quick и Nimble](#)
- [Тестирование на основе моделей](#)
- [What is Data Driven Testing? Learn to create Framework](#)
- [Что нужно знать о Value Driven Testing. Анализируем ценность и экономическую целесообразность тестирования](#)

Доп. материал:

- [Leadership in test: modelling and coverage](#)

(Не обновлялось) Тестирование в разных сферах/областях (testing different domains)

Веб-тестирование

<https://www.softwaretestingmaterial.com/web-application-testing-tutorial/>

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 наличиебитых 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 такие 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.
- Стресстест 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.

Доп. материалы:

[Ничего не забыть: универсальная схема для тестирования веб-приложений](#)

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

Доп. материал:

[Тестирование систем банковского ритейла](#)

Тестирование электронной коммерции (eCommerce)

<https://www.softwaretestingmaterial.com/ecommerce-testing/>

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)

<https://www.softwaretestingmaterial.com/payment-gateway-testing/>

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 | <ul style="list-style-type: none"> • 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 |

| | |
|-------------------------------|---|
| | <ul style="list-style-type: none"> Проверьте, что система POS совместима с периферийными устройствами, такими как считыватель RFID, сканер штрих-кода и т. д. |
| Обработка платежного шлюза | <ul style="list-style-type: none"> Проверка правильности CVV кредитных карт Тест на использование карт с обеих сторон Убедитесь, что данные карты правильно зашифрованы и расшифрованы |
| Продажи | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> Проверка: система для различных скидок, таких как ветеранская скидка, сезонная скидка, скидка на покупку или перелет и т. д. |

| | |
|---|--|
| | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> 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. OSS-БСС тестирования
 - OSS-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

Доп. материал:

Чему я научился, разрабатывая биллинговую систему

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 нагружочное 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 многократных 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)

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

Тестирование сервис-ориентированной архитектуры (SOA - Service Oriented Architecture)

<https://www.softwaretestingmaterial.com/soa-testing/>

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?

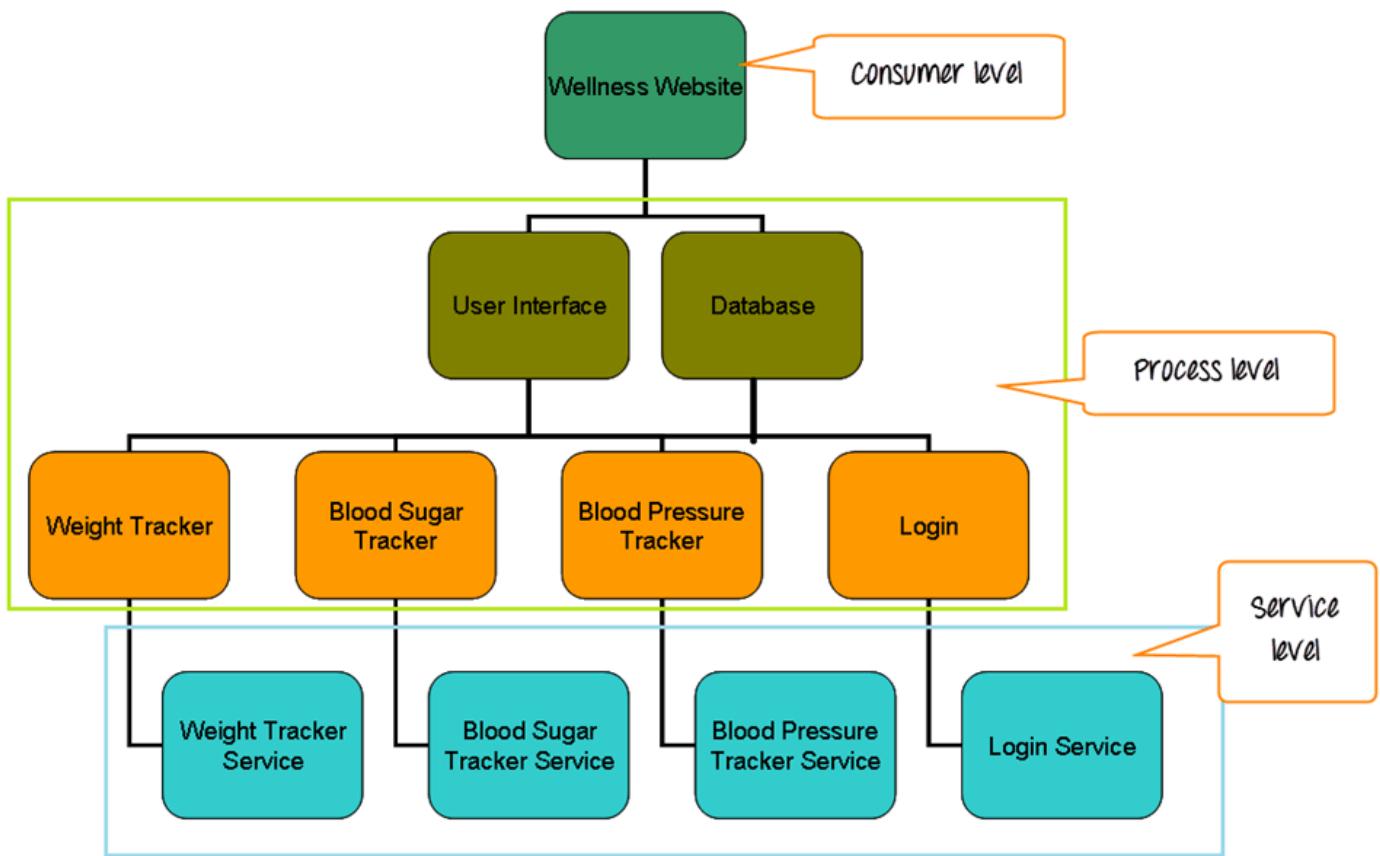
- Service могут быть функциональной единицей приложения или бизнес-процесса, которая может быть повторно использована или повторена любым другим приложением или процессом. (Например, Платежный шлюз - это сервис, который может быть повторно использован любым сайтом электронной коммерции. Каждый раз, когда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

Тестирование планирования ресурсов предприятия (ERP - Enterprise Resource Planning)

<https://www.softwaretestingmaterial.com/approach-the-testing-of-erp-applications/>

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) - это браузерная технология, предназначенная для передачи любых потоковых данных между браузерами или приложениями с использованием технологии двухточечной передачи (point-to-point transmission). Эта технология хороша тем, что можно создать видеочат без использования стороннего сервера, она позволяет устанавливать связь между пользователями, используя только браузер. 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 идеи писать автотесты (по большей части unit) и лишь некоторые доходят до написания бенчмарков. Возможно, в комментариях опытные коллеги поделятся своим опытом.

Доп. материал:

[Что такое WebRTC?](#)

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 А и компании В. После слияния их операции будут объединены, а их клиенты, сотрудники и другие данные будут храниться в единой централизованной базе данных. Предположим, что компания А использует базу данных Oracle для хранения всей информации, а компания В 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»](#)

[Как QA в управлении хранилища данных эволюционировал. Часть 2](#)

Тестирование VR программного обеспечения

<https://tproger.ru/articles/kak-proishodit-testirovaniye-vr-programmnogo-obespechenija/>

Тестирование мессенджера

<https://blog.gatestlab.com/2021/04/14/how-to-test-a-messenger-app/>

Тестирование чат-бота

<https://www.youtube.com/watch?v=9wpy2CwR-fU>

Тестирование микросервисной архитектуры

<https://martinfowler.com/articles/microservice-testing/>

<https://dou.ua/forums/topic/33851/>

Тестирование e-mail

<https://thegalead.com/topics/should-qas-care-about-email-testing/>

Почему важно делать подтверждение e-mail при регистрации?

- Основная причина — это провести double opt-in, т.е. убедиться, что был введен 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 сильно 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 email, то в этом поле можно будет написать все что угодно (в рамках проверки). Соответственно один и тот же пользователь будет регистрироваться множество раз, забывая свой предыдущий пароль, и логин.

Тестирование платформы электронного обучения

<https://blog.gatestlab.com/2021/05/05/e-learning-software-testing/>

Тестирование игр (Gamedev)

<https://m.habr.com/ru/company/otus/blog/556784/>

<https://habr.com/ru/company/otus/blog/557832/>

<https://www.softwaretestingmaterial.com/game-testing-interview-questions/>

<https://www.softwaretestingmaterial.com/game-testing/>

+

[Forwarded from Roman (rpwheeler)]

Особняком здесь стоит тестирование в азартных играх/казино (Gambling)

Гэмблинг — это термин, который на трейдерском жаргоне означает азартное бессистемное совершение сделок, которые зачастую совершаются в состоянии потери контроля над собой. Азартная игра, гэмблинг, это не игра, основанная на навыке или причастности. Это игра, основанная на чистом шансе. Это деятельность, когда человек рискует чем-то ценным, благодаря силам случая, находящимся полностью вне его контроля или вне любого рационального ожидания.

Казино занимается "классикой" -- рулетки, карточные игры (блэкджек и другие), слоты.

Не имеют отношения к казино ("оформляются отдельно"):

- Покер и турниры по оному

- Бинго

- Бинарные опционы

Это к тому что если реально работаешь на гэмблинг, то казино это одно, покер это другое, бинго это третье и бинарные опционы четвёртое (ну и ставки на спортивные состязания я забыл ещё).

[Forwarded from Roman (rpwheeler)]

К "и ещё много всего"

Во всё это (рулетку, покер, блэкджек, бинго...) может быть надо немножко уметь.

А также к вышеупомянутым играм, вебу, мобайлу и десктопу может быть веб-бэк, лайв-игры (то же казино, но с видео-трансляцией в веб или десктоп-клиент), тестирование физических слотов, изучение ограничений регулируемых рынков в разных странах и тестирование оных, бонусных и реферральных систем...

Vladislav Eremeev, [16.03.21 16:22]

[Forwarded from Roman (rpwheeler)]

Дело не только в специфике гэмблинга. Разработка слота или ну пусть покер-клиента -- может и геймдев.

Но гэмблинг _очень_ не сводится к разработке игр как таковых. Полно задач на тестирование фич, к играм прямого отношения не имеющих (кошельки-депозиты-бонусы-реферралы-порталы-регуляции), новых инстансов или апдейтов на эти инстансы, где подёргать игры, ну, 20% функционала. Могут попадаться задачи на тестирование времени загрузки с CDN.

Конечно, можно попасть в команду которая тестирует _только_ слоты. Но можно попасть и в команду портала, бэкенда или онлайн-кошелька, которая игр вообще не тестирует, зато им интересны SQL-запросы.

Vladislav Eremeev, [16.03.21 16:31]

[Forwarded from Ice Spirit]

Верно говоришь. Имел отношение в свое время к бекенду таких гемблинг проектов. Там от игр не было ничего от слова совсем. Апи, бд, внутренние алгоритмы, расчеты игр и финансов. От гемблинга там было только в лучшем случае нажать кнопку чтобы запустить процесс на бекенде.

Тестирование блокчейна (Blockchain)

Блокчейн - выстроенная по определённым правилам непрерывная последовательная цепочка блоков (связный список), содержащих информацию. Связь между блоками обеспечивается не только нумерацией, но и тем, что каждый блок содержит свою собственную хеш-сумму и хеш-сумму предыдущего блока. Изменение любой информации в блоке изменит его хеш-сумму. Сейчас блокчейн находит применение в таких областях, как финансовые операции, идентификация пользователей или создание технологий кибербезопасности. Блокчейн-технологии актуальны в первую очередь для банковских учреждений и государственных организаций.

- [Awesome Blockchain Testing](#)
- [Блокчейн изнутри: как устроен биткоин](#)

Тестирование банкомата (ATM)

<https://www.softwaretestingmaterial.com/how-to-write-test-cases-for-atm/>

Тестирование Salesforce

<https://www.softwaretestingmaterial.com/salesforce-testing/>

Тестирование SaaS

<https://www.softwaretestingmaterial.com/saas-testing/>

Мобильное тестирование

Особенности в тестировании мобильных приложений

Многие особенности очевидны из самого названия “мобильные” - смартфон имеет маленький дисплей, им пользуются на ходу и в условиях совместного использования с большим количеством других приложений и подключенных устройств, а высокий темп современной жизни заставит пользователя уйти к конкурентам, если экран приложения загружается дольше пары секунд, если с приложением сложно взаимодействовать или оно доставляет еще какие-либо неудобства.

Отличия от веба и десктопа:

- Постоянные прерывания в работе приложения: сворачивание, блокировка, входящий звонок или сообщение, уведомления, обновления приложений, выключение или перезагрузка, выгрузка системой из ОЗУ, разряд АКБ, подключение зарядки или других устройств, переход в энергосберегающий режим и режим ожидания, включение и отключение функций необходимых устройству (gps, bluetooth, связь), использование/отзыв разрешений, подключение/отключение карты памяти/симки/АКБ, платежи NFC, принудительная остановка);
- Работа в беспроводной сети с изменяющейся стабильностью и скоростью приема сигнала, переключение между сотовой связью и wi-fi;
- Использование вертикальной и горизонтальной ориентации, повороты;
- Распространение приложений через маркеты;
- Необходимо соответствие гайдлайнам систем;
- Большое внимание уделяется UI и UX;
- В случае Android большая фрагментация устройств и прошивок со своими особенностями;
- Тач-интерфейс, мультитач, жесты;
- Уведомления;
- Множество каналов ввода: стоковая клавиатура, сторонние клавиатуры, хардовые клавиатуры, голос, жесты и т. д.;
- Биометрия;
- Повышенные требования к энергопотреблению и использованию аппаратных ресурсов;
- Ситуации установки и обновления при нехватке памяти, переноса приложения на карту памяти и обратно;
- Важность инсталляционного тестирования, особенно обновлений, т.к. в условиях высококонкурентного мобильного рынка приложения обновляются часто, чтобы предоставить пользователям новые функции как можно скорее и при этом должны проходить бесшовно;
- GPS и локализация;
- Размер дисплея, чека и вырез под фронтальную камеру;

Типы мобильных приложений

Нативные приложения: написаны на родном 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, уведомления и т.п. (при наличии разрешений). В зависимости от предназначения нативного приложения, оно может всецело или частично обходиться без наличия интернет-соединения;

Веб-приложения: на самом деле не являются приложениями как таковыми. В сущности, они представляют собой сайты, которые адаптированы и оптимизированы под любой смартфон и выглядят похоже на нативное приложение. И для того, чтобы воспользоваться им, достаточно иметь на устройстве браузер, знать адрес и располагать интернет-соединением. 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. Очевидным

недостатком такого вида приложений является утрата работоспособности при потере интернет-соединения. Причем из этого выплывает и другой минус – их производительность, которая находится на среднем уровне, в сравнении с другими видами приложений и зависит от возможностей интернет-соединения провайдера услуг. Помимо вышеперечисленного, веб-приложения не могут получить доступ к функциям системы и самого устройства;

Гибридные приложения: это веб-приложение в обертке нативного приложения, что служит контейнером для отображения веб-приложения через встроенный упрощенный браузер (webview в Android и WKWebView в iOS). Нативный “фундамент” даёт преимущества нативных приложений: доступ к функционалу смартфона (API системы, пушки и т.п.), размещение в маркетах, иконка на рабочем столе и т.п., а сторона веб-приложений дает плюсы в виде кроссплатформенности и простоты обновления контента. Компания, имеющая веб-приложение, может практически “на коленке” собрать гибридные приложения для основных платформ и обеспечить себе присутствие в маркете и на рабочем столе клиентов;

Аналоги инсталлируемых приложений

- Progressive Web Apps (PWA);
- Google Play Instant (Android Instant Apps (AIA));
- Accelerated Mobile Pages (AMP).

Progressive Web Apps (PWA): термин «прогрессивное веб-приложение» не является официальным названием. Это просто сокращение, которое изначально использовалось Google для обозначения концепции создания гибкого, адаптируемого приложения с использованием только веб-технологий. PWA - это веб-приложения, которые постепенно улучшаются, чтобы функционировать как установленные нативные приложения на поддерживаемых plataформах, но при этом функционируют как обычные веб-сайты в других браузерах. Качества PWA сочетают в себе лучшее из Интернета и скомпилированных приложений. PWA запускаются в браузерах, как веб-сайты. Но PWA также имеют доступ к функциям приложений; Например:

- PWA все еще может работать, когда устройство отключено;
- PWA можно установить в операционной системе;
- PWA поддерживают push-уведомления и периодические обновления;
- PWA могут получить доступ к аппаратным функциям.

Кроссплатформенность таких веб-приложений не ограничивается мобильными устройствами. Например, на платформе Windows такие приложения тоже выглядят и существуют в системе точно также, как и нативные приложения: можно установить из стора, добавить ярлык в пуск, доступны File systems, Video, Audio, High-performance code, Databases, USB, Bluetooth и т.п., при этом все фактическое содержимое является веб-сайтом.

На самом деле, скорее всего, вы раньше часто посещали PWA, даже не осознавая этого. Если вы когда-нибудь просматривали Instagram, Pinterest, Spotify или Tinder на своем ноутбуке или смартфоне, вы столкнулись с PWA.

Таким образом, PWA имеют гораздо более низкую стоимость кроссплатформенной разработки, чем скомпилированные приложения, которым требуется определенная кодовая база для каждой платформы.

Для того, чтобы веб-приложение можно было называть PWA, оно должно соответствовать определенным критериям:

- Обнаруживаемое (**Discoverable**): Приложение можно обнаружить в результатах веб-поиска и в поддерживаемых магазинах приложений;
- Устанавливаемое (**Installable**): можно закрепить и запустить приложение с главного экрана;
- Возможность повторного вовлечения (**Re-engagable**): приложение может получать push-уведомления, даже если неактивно;
- Независимо от сети (**Network-independent**): приложение работает в автономном режиме и в условиях слабого подключения к сети;

- Прогрессивное (**Progressive**): UX увеличивается или уменьшается в зависимости от возможностей устройства;
- Безопасное (**Safe**): приложение обеспечивает безопасный HTTPS endpoint и другие меры безопасности для пользователей;
- Отзывчивое (**Responsive**): приложение адаптируется к размеру и ориентации экрана, методу ввода;
- Линкованное (**Linkable**): делитесь и запускайте приложение по стандартной ссылке.

Итак, как можно ли узнать, является ли веб-сайт PWA? Ну, нет. По крайней мере, не совсем точно. Если вы не разработчик и не копаетесь в исходном коде сайта, у вас нет определенного способа точно сказать, построен ли сайт на технологии PWA. При этом есть несколько уловок, которые, хотя и не гарантируют точного результата, могут дать вам некоторые признаки того, что данный веб-сайт является PWA.

- **Одностраничный сайт.** Это самый простой способ узнать, может ли веб-сайт быть PWA. Он основан на природе PWA: Progressive Web Apps технически представляют собой одностраничный веб-сайт. Это не означает, что веб-сайт, построенный на основе PWA, имеет только одну страницу. Это означает, что событие просмотра страницы происходит только один раз, когда пользователь изначально загружает сайт. После этого все загрузки страниц обрабатываются Javascript. Это отличается от обычных веб-сайтов, где каждое изменение страницы вызывает перезагрузку страницы вместе со всем исходным кодом HTML. Так как это работает? Что ж, очень просто: взгляните на активную вкладку в вашем браузере. Если сайт является PWA, при смене страниц сайты не перезагружаются, что означает отсутствие анимации «загрузки» на вкладке браузера. Теперь давайте посмотрим на наш сайт [SimiCart](#) в качестве примера. При смене страниц сайты не перезагружаются! Технически вы просто все время остаетесь на одной «странице». Вот почему страницы PWA загружаются так быстро и плавно. Все страницы предварительно загружаются при первом посещении сайта и доставляются вам впоследствии. Они не зависят от скорости вашей сети и могут работать даже в автономном режиме!
- **Service Workers.** Service Workers - это название технологии, лежащей в основе прогрессивного веб-приложения, которая обеспечивает его автономные возможности, push-уведомления и кэширование ресурсов. Согласно Google, сервис-воркеры лежат в основе методов PWA. Итак, если мы сможем выяснить, использует ли веб-сайт технологию Service Workers, мы сможем сказать, может ли этот сайт быть PWA. Если вы используете браузеры на базе Chrome, вы можете легко это проверить с помощью Inspector Tool. Щелкните правой кнопкой мыши веб-сайт, который вы хотите проверить, выберите «Проверить элемент». Затем перейдите на вкладку «Приложение» - «Рабочие службы». Вы можете легко увидеть, есть ли на этом сайте Service Workers. Опять же, этот трюк только дает намек на то, что определенный веб-сайт является PWA. Несмотря на то, что Service Workers является основной частью PWA, они не являются эксклюзивной частью PWA. Веб-сайты, не относящиеся к PWA, также могут использовать Service Workers для улучшения своей функциональности. Если вы хотите узнать больше о PWA Service Worker, у нас есть [эксклюзивная статья](#) для вас, чтобы узнать все об этой удивительной технологии.
- **HTTPS** secure origin. PWA работают только по HTTPS.
- **Manifest.json** - Имеется файл настроек.

Accelerated Mobile Pages (AMP)

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

- AMP HTML — язык HTML, в котором часть тегов заменена на эквивалентные AMP-теги, а часть запрещена для использования;
- AMP JS — в работе используется собственная JS-библиотека, позволяющая элементам страницы загружаться асинхронно;
- Google AMP Cache — в процессе индексации AMP-страницы, поисковая система кэширует её данные и воспроизводит со своих серверов.

Многие воспринимают AMP как способ положить статический контент своего сайта (статьи, новости, заметки и т.д.) в кэш Google, чтобы при открытии из поиска этот контент загружался мгновенно (о высокой скорости

загрузки AMP страниц свидетельствует иконка молнии в результатах поиска :)). Естественно, если вам нужно добиться именно такого результата, то с AMP это сделать будет очень легко. Но AMP — это гораздо больше чем просто технология для работы со статическим контентом или кэшем Google. AMP уже давно используется как библиотека общего назначения, основанная на web компонентах, для создания быстрых динамических страниц и даже сайтов целиком, на которые пользователи попадают как из поиска, так и из других источников, включая прямые заходы. С этой точки зрения AMP можно поставить в один ряд с Polymer, React или Angular. Естественно с оглядкой на то, что AMP предназначена для простых (чтобы это не значило) сайтов, где основной упор делается на контент, а динамическая составляющая ограничена.

Отдельно хочется отметить, что несмотря на название — Accelerated Mobile Pages, AMP может использоваться для создания любых сайтов, как десктопных, так и мобильных. Сайт проекта — ampproject.org является замечательным примером того, что можно сделать с AMP для десктона.

Google Play Instant (Android Instant Apps (AIA))

Google Play Instant (прошлое название Android Instant Apps) — это функция, которая позволяет вам использовать приложение без необходимости полностью загружать его на свой телефон: просто найдите его в Play Store и нажмите «Открыть приложение». Более того, это позволяет вам перейти к определенному действию в приложении, которое вы не установили, просто нажав URL-адрес. Недавно Google добавил в Play Store кнопку «Попробовать» для некоторых приложений с мгновенным запуском Android.

Например, если вам прислали ссылку на видео в Buzzfeed, то можно сразу же открыть её в приложении этого сервиса, даже если оно у вас не установлено. Ранее выбор был лишь либо отдельно установить приложение, либо перейти по ссылке в браузере. Конечно, это несколько дольше, чем просто открыть веб-страницу, но всё же намного быстрее, чем устанавливать целое приложение ради одной ссылки. Обещают, что занимать весь процесс открытия чего-либо во «мгновенных приложениях» будет несколько секунд. Однако полноценную функциональность установленного приложения вы не получите, будет загружено лишь то, что необходимо для выполнения текущего действия, например, просмотра видео.

На странице часто задаваемых вопросов о приложениях Google с мгновенным запуском говорится, что эти приложения могут использовать следующие разрешения:

- ОПЛАТА
- ACCESS_COARSE_LOCATION
- ACCESS_FINE_LOCATION
- ACCESS_NETWORK_STATE
- КАМЕРЫ
- INSTANT_APP_FOREGROUND_SERVICE только в Android O.
- INTERNET
- READ_PHONE_NUMBERS только в Android O.
- ЗАПИСЬ АУДИО
- VIBRATE

Все, что отсутствует в этом списке, не поддерживается Instant Apps. Обратите внимание, что такие вещи, как Bluetooth, установка будильника, использование отпечатков пальцев и установка обоев, отсутствуют.

Другие ограничения включают отсутствие поддержки фоновых служб (приложений, которые могут запускаться без ведома пользователя), push-уведомлений, доступа к внешнему хранилищу или просмотра установленных приложений на устройстве. Приложения с мгновенным запуском также не смогут изменять настройки на устройстве пользователя, такие как обои.

Ограничения на размер сборки:

- Dynamic Feature Modules вообще не ограничены по размеру;
- Instant-Enabled Dynamic Feature Modules могут занимать до 10 Мб.

Если ваша фича:

- больше 10 Мб, то извините;
- от 4 до 10 Мб — доступна по кнопке «Попробовать» из Google Play и все;
- меньше 4 МБ — доступны все средства привлечения пользователей в Instant-Enabled модуль (запуск из рекламы, по ссылке, из сообщений и т.д.).

Примерами Instant Apps являются BuzzFeed, Skyscanner, Onefootball, Red Bull TV, приложение UNS ShareTheMeal, Sports.ru, Vimeo и многие другие.

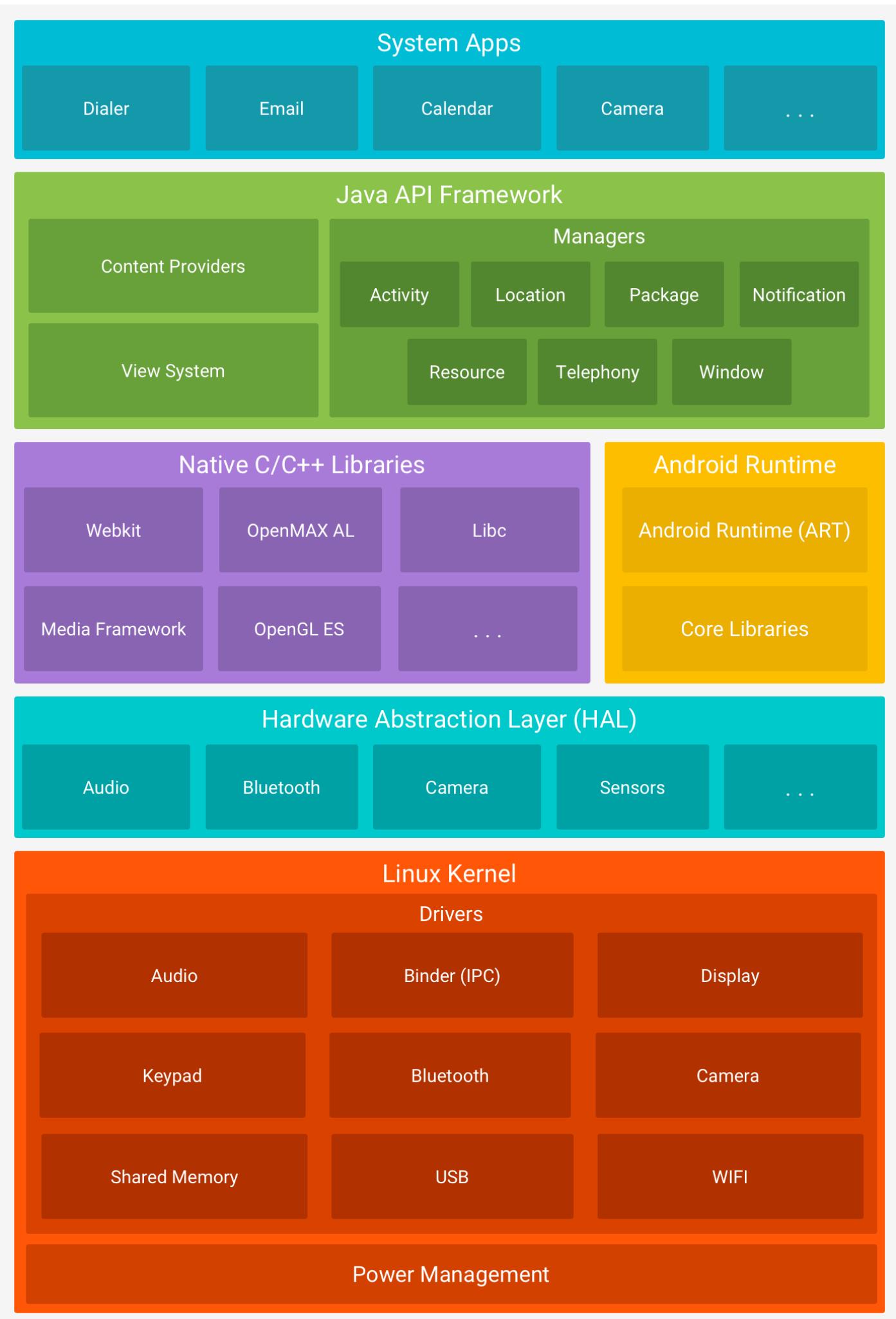
Источники:

- [What Is PWA? All You Need to Know About Progressive Web Apps](#)
- [Overview of Progressive Web Apps \(PWAs\)](#)
- [Используем AMP как библиотеку общего назначения для создания быстрых динамических сайтов](#)
- [Приложения с мгновенным запуском Android – что они значат для пользователей и разработчиков?](#)
- [Google Play Instant. Рефакторинг длиною в жизнь](#)

Доп. материал:

- [Сайт, где можно узнать, что может PWA](#)
- [12 Best Examples of Progressive Web Apps \(PWAs\) in 2021](#)
- [11 Best Progressive Web Apps \(PWAs\) Games in 2021](#)
- [Тестирование аналогов инсталлируемых приложений. Диана Пинчук. Comaga Spring 2019](#)
- [MDN Web Docs - Progressive web apps \(PWAs\)](#)
- [An Introduction to Progressive Web Apps](#)
- [All You Need to Know About Progressive Web App to Decide if Your Business Needs One](#)
- [Натив или гибрид? Специалисты Яндекса отвечают на главный вопрос мобильной разработки](#)
- [Офф. дока Google Play Instant](#)

Архитектура Android OS



Linux Kernel

Данный уровень является базовым в архитектуре Android, так как вся система Android построена на ядре Linux с некоторыми архитектурными изменениями.

Основные задачи, выполняемые ядром: Ядро содержит в себе драйверы, необходимые для взаимодействия с оборудованием. Например, возьмем Bluetooth. В наше время сложно найти устройство, которое бы не включало в себя эту функцию. Поэтому ядро должно включать драйвер для работы с ним. На схеме выше как раз таки перечислены все драйверы, входящие в ядро Linux, поэтому не будем перечислять их все по отдельности.

- Power Management - это своего рода система управления питанием. Она предоставляет различные средства, с помощью которых приложение может реагировать на режимы питания устройства, а также поддерживать необходимые компоненты устройства активными. Например, при чтении книги нам было бы удобно, если бы экран оставался постоянно активным. Или когда мы включаем музыку и она продолжает проигрываться в фоне при отключенном экране.
- Управление памятью. При запуске различных приложений ядро гарантирует, что пространство памяти, которое они используют, не конфликтует и не перезаписывает друг друга. Также оно проверяет, что все приложения получают достаточный объем памяти для своей работы, и в то же время следит, чтобы ни одно приложение не занимало слишком много места.
- Управление процессом. Каждое приложение в Android работает в отдельном процессе. Ядро же отвечает за управление этими процессами, а именно за создание, приостановку, остановку, или завершение процессов, за одновременное выполнение нескольких процессов, обмен данными между процессами, запуск процессов в фоновом режиме. Помимо этого ядро распределяет работу между процессорами устройства, что максимизирует производительность устройств с несколькими ядрами.

Hardware Abstraction Layer (HAL)

HAL обеспечивает связь между драйверами и библиотеками. Состоит он из нескольких библиотечных модулей, каждый из которых реализует интерфейс для определенного аппаратного компонента (Bluetooth, Камера итд.). И когда к оборудованию устройства обращаются через API-интерфейс, загружается необходимый для его работы модуль.

Если объяснять на пальцах, то когда от приложения поступает какое-либо сообщение, HAL его обрабатывает таким образом, чтобы оно стало понятным для драйверов. И наоборот.

Android Runtime (ART)

Основным языком Android был выбран Java, поскольку это один из самых популярных языков программирования. Для Java существует много наработок и специалистов, а написанные на нем программы переносимы между операционными системами.

Но для того, чтобы программа работала на Java необходима виртуальная машина – Java Virtual Machine. В Android используется виртуальная машина Android Runtime (ART). Эта машина специально оптимизирована для работы на мобильных устройствах: с нехваткой памяти, с постоянной выгрузкой и загрузкой приложений и т.д. В версиях Android ниже 5.0 Lollipop, использовалась виртуальная машина Dalvik - старая реализация виртуальной машины для Android.

В ART, как и в Dalvik, используется свой формат байт-кода – DEX (Dalvik executable). При сборке приложения исходные файлы сначала компилируются в файлы типа class обычным компилятором, а затем конвертируются специальной утилитой в DEX.

И Dalvik, и ART оптимизируют выполняемый код, используя механизм компиляции just-in-time (JIT) - компиляция происходит во время выполнения приложения, что позволяет оптимизировать код для выполнения на конкретном устройстве. При этом байт-код приложения можно переносить на другие устройства.

ART может компилировать байт-код заранее, а не во время выполнения, используя *ahead-of-time* (AOT). Система сама решает, когда и какие приложения необходимо скомпилировать. Например, когда устройство не загружено и подключено к зарядке. При этом ART учитывает информацию о приложении, собранную во время предыдущих запусков, что дает дополнительную оптимизацию.

Native C/C++ Libraries

Набор библиотек, написанных на языках C или C++ и используемых различными компонентами ОС.

Примеры библиотек:

- WebKit - представляет из себя движок веб-браузера и другие связанные с ним функции.
- Media Framework предоставляет медиа кодеки, позволяющие записывать и воспроизводить различные медиа-форматы.
- OpenGL - используется для отображения 2D и 3D графики.
- SQLite - движок базы данных, используемый в Android для хранения данных.

Java API Framework (Application Framework)

Набор API, написанный на языке Java и предоставляющий разработчикам доступ ко всем функциям ОС Android. Эти API-интерфейсы образуют строительные блоки, необходимые для создания приложений, упрощая повторное использование основных, модульных, системных компонентов и сервисов, таких как:

- Activity Manager - управляет жизненным циклом приложения и обеспечивает общий навигационный стек обратных вызовов.
- Window Manager - управляет окнами и является абстракцией библиотеки Surface Manager.
- Content Providers - позволяет приложению получать доступ к данным из других приложений или обмениваться собственными данными, т.е. предоставляет механизм для обмена данными между приложениями.
- View System - содержит строительные блоки для создания пользовательского интерфейса приложения (справки, тексты, кнопки итд.), а также управляет событиями элементов пользовательского интерфейса.
- Package Manager - управляет различными видами информации, связанными с пакетами приложений, которые в настоящее время установлены на устройстве.
- Telephony Manager - позволяет приложению использовать возможности телефонии.
- Resource Manager - обеспечивает доступ к таким ресурсам, как локализованные строки, растровые изображения, графика и макеты.
- Location Manager - возможность определения местоположения.
- Notification Manager - отображение уведомлений в строке состояния.

System Apps

Верхний уровень в архитектуре Android, который включает в себя ряд системных (предустановленных) приложений и тонну других приложений, которые можно скачать с Google Play.

Системные приложения на всех устройствах разные, но все они являются предустановленными производителями устройства (приложение для SMS-сообщений, календарь, карты, браузер, контакты итд.).

Этот уровень использует все уровни ниже (если смотреть на схему) для правильного функционирования приложений.

Источники:

- [Как работает Android. Архитектура платформы + оригинал из оффи доки](#)

Доп. материал:

- Более подробный цикл статей “Как работает Android”: часть [1](#), [2](#), [3](#), [4](#)
- [Android's HIDL: Treble in the HAL](#)

Архитектура Android Application

Приложения для Android можно писать на языках Kotlin, Java и C++. Инструменты Android SDK компилируют ваш код вместе с любыми файлами данных и ресурсов в **APK** или **Android App Bundle**.

Android package, который представляет собой архивный файл с расширением **.apk**, содержит содержимое приложения Android, которое требуется во время выполнения, и это файл, который устройства под управлением Android используют для установки приложения.

Пакет **Android App Bundle**, который представляет собой архивный файл с расширением **.aab**, содержит содержимое проекта приложения Android, включая некоторые дополнительные метаданные, которые не требуются во время выполнения. AAB - это формат для публикации в маркете, который нельзя установить на устройствах Android, он откладывает создание APK и подписание на более поздний этап. Например, при распространении вашего приложения через Google Play серверы Google Play генерируют оптимизированные APK-файлы, которые содержат только ресурсы и код, которые требуются конкретному устройству, запрашивающему установку приложения.

Каждое приложение Android находится в собственной изолированной программной среде безопасности или песочнице (**security sandbox**), защищенной следующими функциями безопасности Android:

- Операционная система Android - это многопользовательская система Linux, в которой каждое приложение является отдельным пользователем. По умолчанию система назначает каждому приложению уникальный идентификатор пользователя Linux (идентификатор используется только системой и неизвестен приложению);
- Система устанавливает разрешения для всех файлов в приложении, так что только идентификатор пользователя, назначенный этому приложению, может получить к ним доступ;
- У каждого процесса есть собственная виртуальная машина (VM), поэтому код приложения работает изолированно от других приложений;
- По умолчанию каждое приложение работает в собственном процессе Linux. Система Android запускает процесс, когда необходимо выполнить любой из компонентов приложения, а затем завершает процесс, когда он больше не нужен или когда системе необходимо восстановить память для других приложений.

В системе Android реализован принцип наименьших привилегий. То есть каждое приложение по умолчанию имеет доступ только к тем компонентам, которые необходимы ему для работы, и не более того. Это создает очень безопасную среду, в которой приложение не может получить доступ к частям системы, для которых у него нет разрешения. Однако есть способы для приложения обмениваться данными с другими приложениями и для приложения для доступа к системным службам:

- Можно организовать два приложения для использования одного и того же идентификатора пользователя Linux, и в этом случае они смогут получить доступ к файлам друг друга. Для экономии системных ресурсов приложения с одним и тем же идентификатором пользователя также могут работать в одном процессе Linux и совместно использовать одну и ту же виртуальную машину;
- Приложения также должны быть подписаны тем же сертификатом. Приложение может запрашивать разрешение на доступ кенным устройствам, таким как местоположение устройства, камера и соединение Bluetooth. Пользователь должен явно предоставить эти разрешения. Дополнительные сведения см. В разделе [«Работа с разрешениями системы»](#).

Компоненты приложения

Компоненты приложения являются основными строительными блоками Android приложения. Каждый компонент - это точка входа, через которую система или пользователь могут войти в ваше приложение. Некоторые компоненты зависят от других. Есть четыре разных типа компонентов приложения:

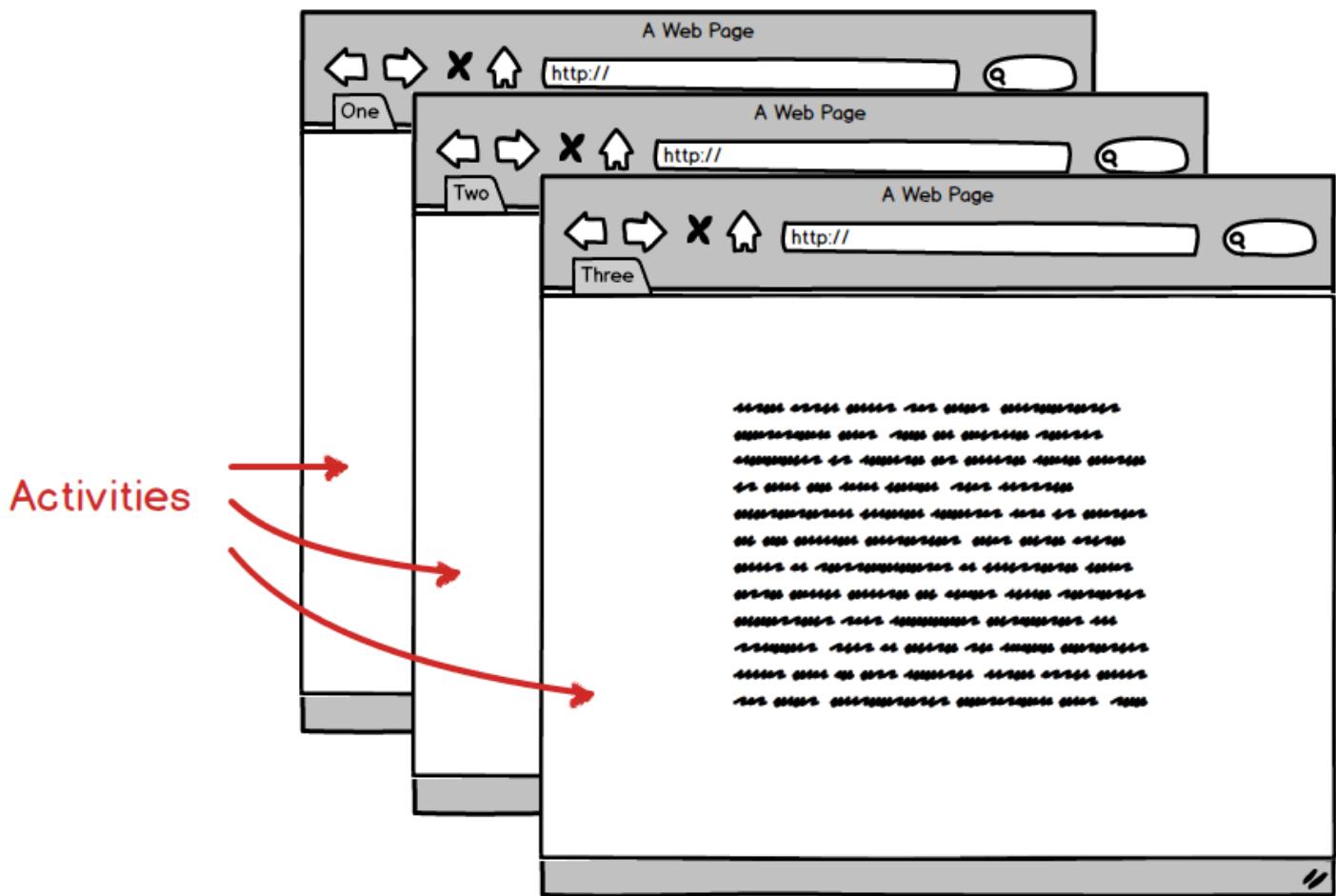
- Activities;
- Services;
- Broadcast receivers;
- Content providers.

Каждый тип служит определенной цели и имеет отдельный жизненный цикл, который определяет, как компонент создается и уничтожается.

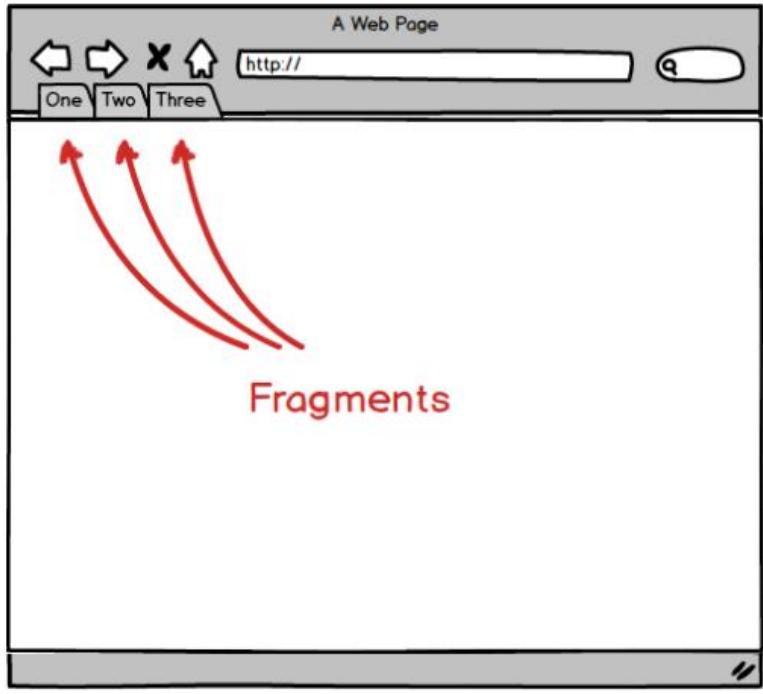
Начнем разбираться с таких компонентов, как активити и фрагменты, и рассмотрим кейсы, когда система может прекратить их работу.

Активити и фрагменты (activities and fragments)

С точки зрения тестировщика, активити можно воспринимать как экран, на котором отображаются элементы. Приложение состоит, как минимум, из одной активити. По сути, активити — это контейнер для UI-компонентов. Если активити — это контейнер, то фрагменты — это UI-компоненты активити. Фрагмент тоже, в свою очередь, является контейнером для UI-компонентов. Есть классная аналогия с браузером (спасибо разработчикам!) для более наглядного представления о том, как между собой связаны активити и фрагменты. Представим, что у нас открыто несколько окон одного браузера. Это несколько **активити** внутри одного приложения.



Внутри окна может быть открыто несколько вкладок. Это **фрагменты** внутри активити.



Фрагменты работают чуть быстрее, чем активити. Но на современных устройствах разница практически неощущима. В зависимости от того, каким образом решается задача, разработчик может написать приложение только на активити или на активити с использованием фрагментов.

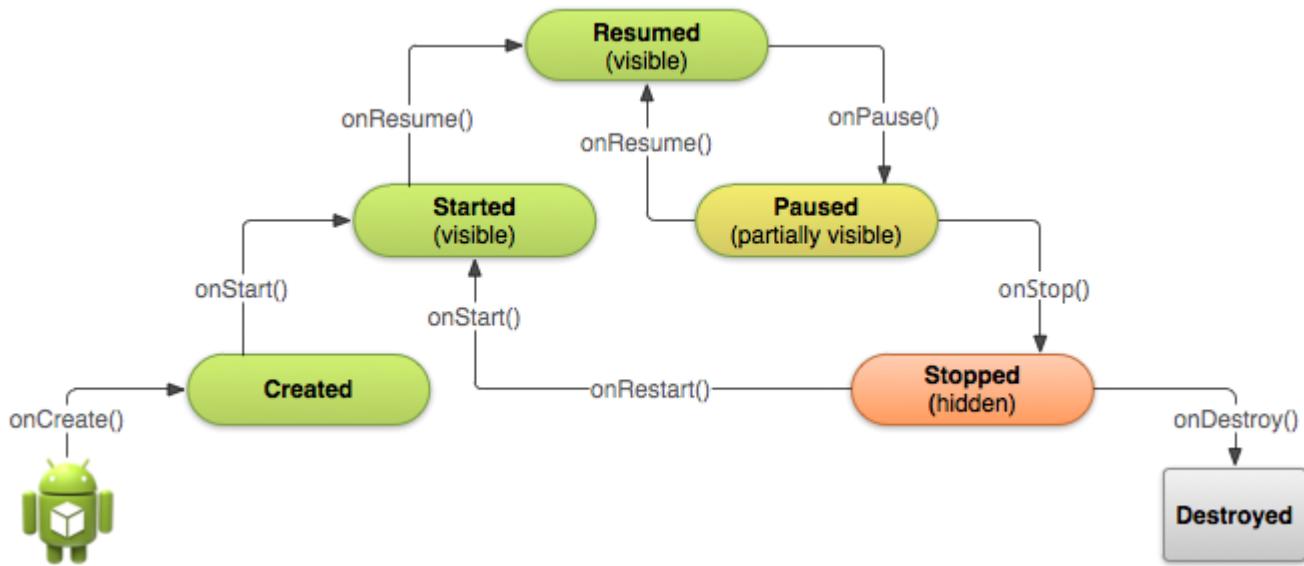
Операционная система при управлении жизненным циклом приложения работает именно с активити приложения. Поэтому пока нас больше всего интересует жизненный цикл активити.

Пользователи запускают большое количество приложений, а значит, создается много процессов с активити. Каждый запущенный процесс съедает оперативную память устройства, и ее становится все меньше. Но Андроид тщательно следит за процессами и в случае нехватки ресурсов для выполнения более приоритетной работы закрывает те, которые менее приоритетны. Давайте разберемся, в какой момент приложение «уязвимо» к таким решениям системы и как это может повлиять на его работоспособность.

Жизненный цикл активити

После запуска и до завершения активити всегда пребывает в одном из статусов. В тестировании нам это полезно для понимания потенциально проблемных кейсов и в контексте тестирования прерываний (Interrupt testing):

- Перед завершением Activity, будут вызваны соответствующие методы жизненного цикла;
- Метод onPause() должен остановить все "слушания" и обновления интерфейса. Метод onStop() должен сохранять данные приложения. И наконец, метод onDestroy() высвободит любые ресурсы, выделенные для Activity;
- Когда пользователь переключается обратно на приложение, которое было прервано системным событием, вызывается метод onResume(). На основе сохраненных данных, могут перерегистрироваться «слушатели» и переключиться обновления интерфейса.

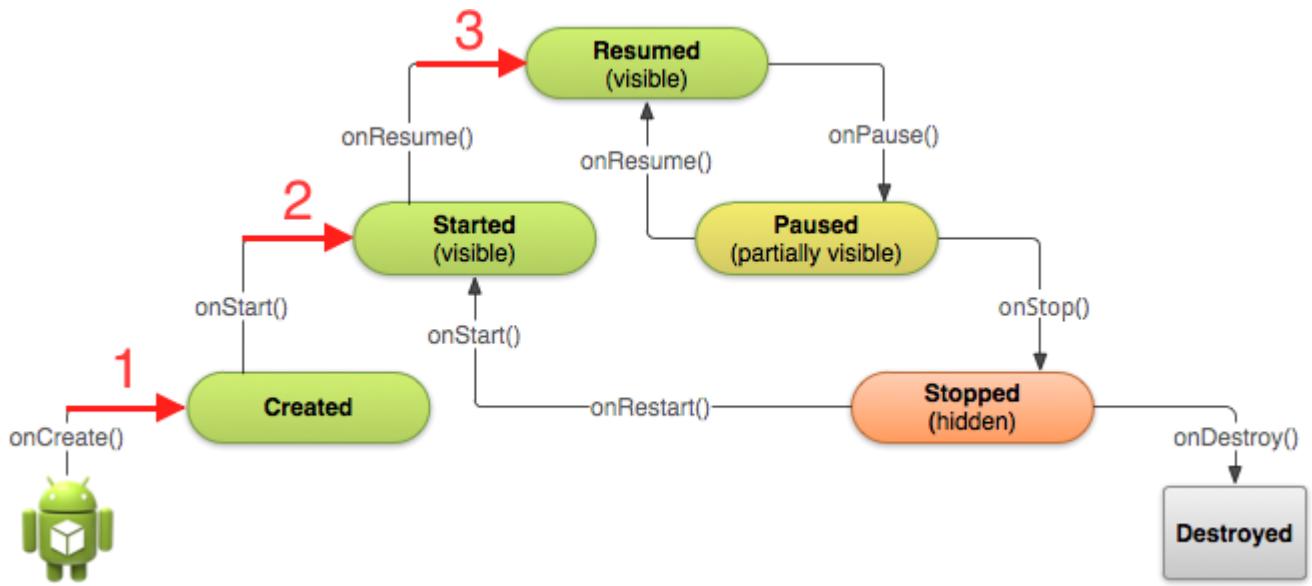


Жизненный цикл активити представлен следующими состояниями:

| Состояние | Пояснение |
|-----------|--|
| Created | Активити создается, пользователь еще не может с ней взаимодействовать |
| Started | Активити запускается, пользователь еще не может с ней взаимодействовать, но она уже видна |
| Resumed | Активити видна на экране и пользователь может с ней взаимодействовать |
| Paused | Активити частично видна на экране и пользователь не может с ней взаимодействовать (искл: в многооконном режиме видна полностью, но в данный момент пользователь взаимодействует с другой активити) |
| Stopped | Активити остановлена и не видна на экране, но еще хранится в памяти |
| Destroyed | Активити уничтожена и больше не хранится в памяти |

Теперь приведу примеры. Они покрывают основные кейсы, с которыми мы чаще всего сталкиваемся при тестировании.

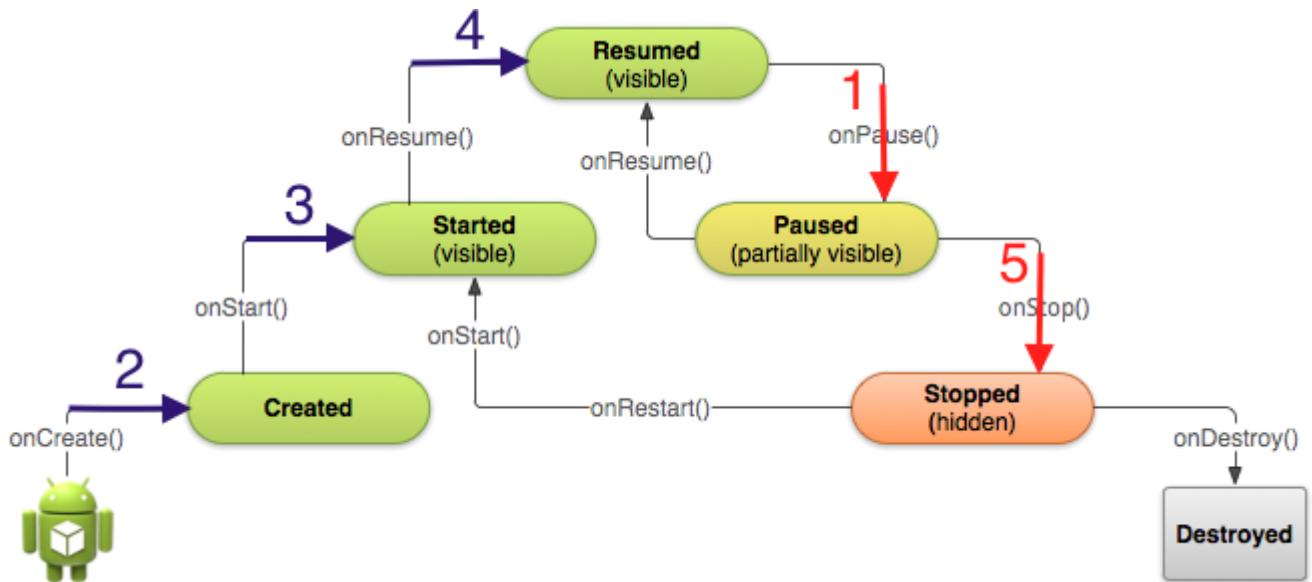
Первый запуск приложения



При первом запуске приложения создается и загружается главная активити в приложении.

При переходе в состояние «Resumed» активити доступна для взаимодействия с пользователем. Все замечательно, проблем нет.

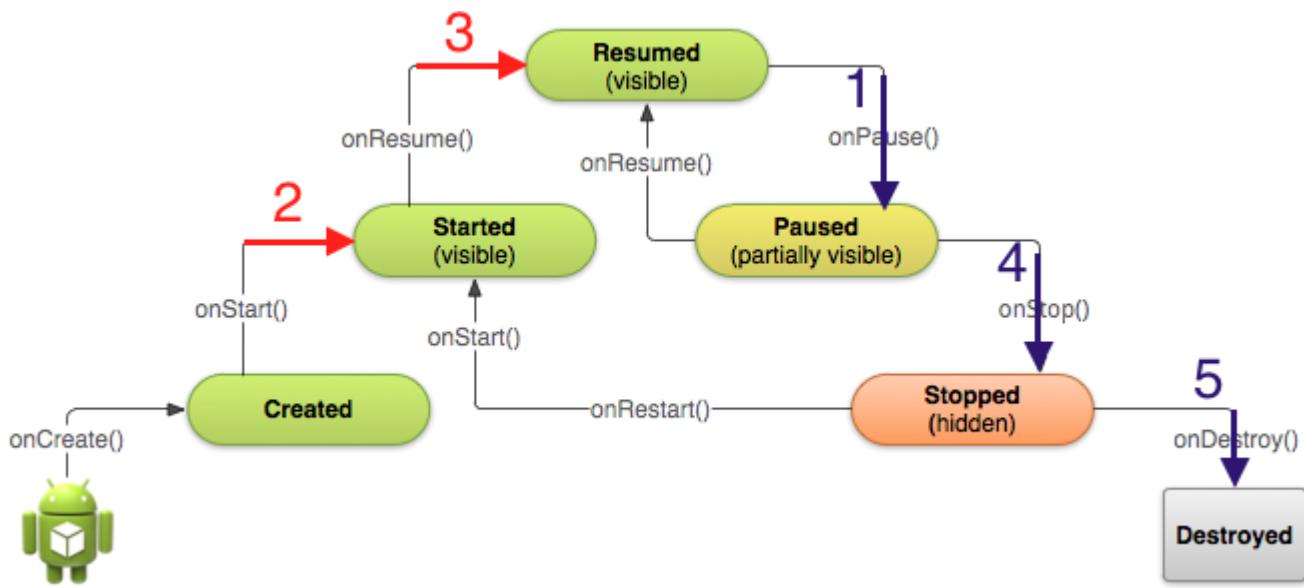
Переход с первого экрана на второй



- Шаг 1: При переходе с первого экрана на второй активити первого экрана сначала встает на паузу. В этот момент могут выполняться такие действия, как сохранение введенных данных и остановка ресурсоемких операций (например, проигрывание анимаций). Это происходит довольно быстро и неощутимо для пользователя.
- Шаг 2, 3, 4: Запускается активити второго экрана.
- Шаг 5: Останавливается активити первого экрана. Также в случае, если Андроид в этот момент пытается освободить память, активити первого экрана дополнительно может перейти в состояние «Destroyed» (то есть уничтожиться).

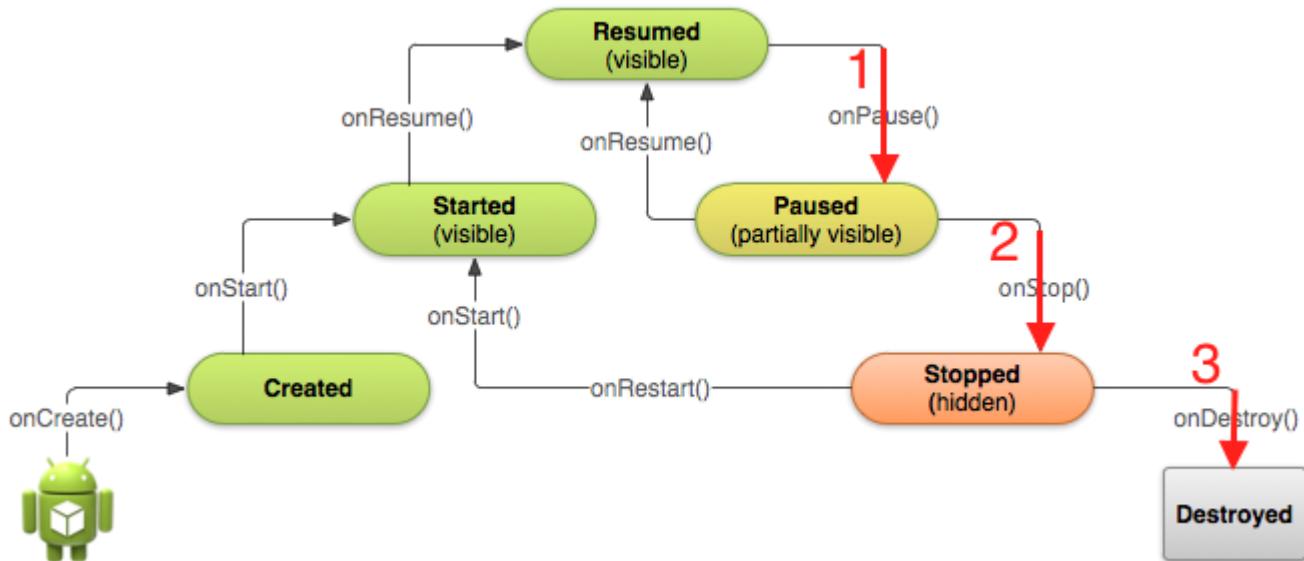
Чтобы лучше понять, что из себя представляет состояние «Paused» давайте представим такую ситуацию: экран, поверх которого появился алерт. Мы не можем с ним взаимодействовать, но в то же время мы его видим.

Возврат со второго экрана на первый



При возврате со второго экрана на первый происходит почти то же самое, только первая активити не создается заново. Она подгружается из памяти (если, конечно, не была уничтожена системой). Вторая активити уничтожается после того, как была остановлена, так как она убирается из стека переходов.

Сворачивание и выход из приложения



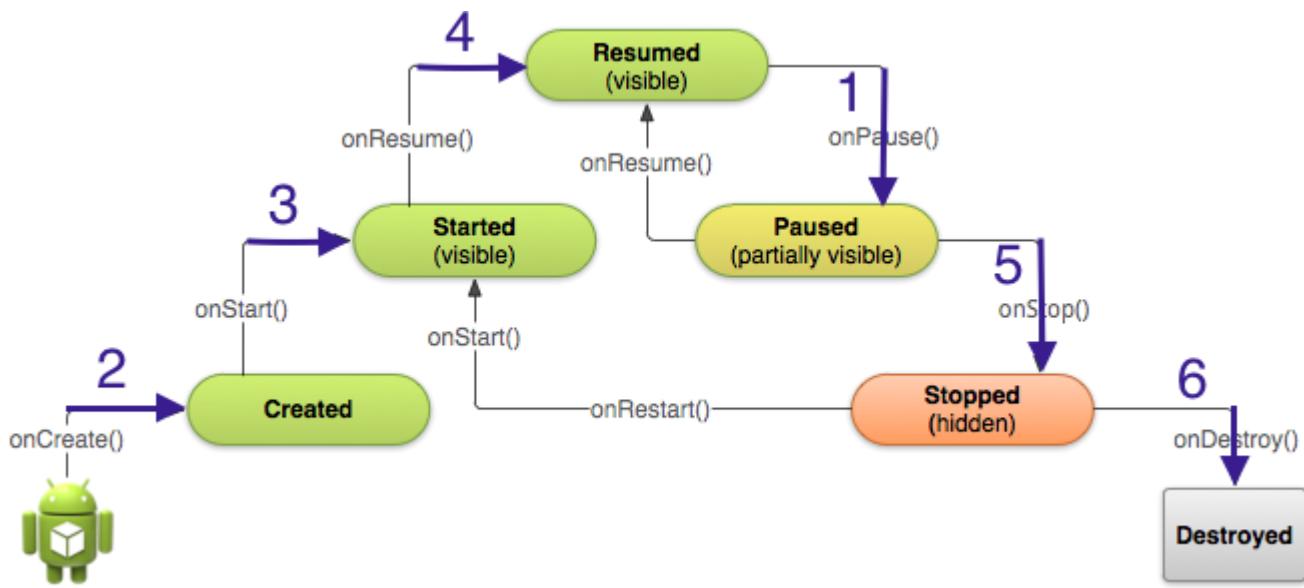
При сворачивании приложения (например, по кнопке Home) активити останавливается.

Важно, чтобы при разворачивании приложения активити корректно восстановилась и данные сохранились.

На всех экранах стараемся проверять сворачивание-разворачивание приложения. Этот кейс особенно актуален на формах ввода. Согласитесь, будет обидно, если текст письма, который вы так старательно набирали, сотрется при банальном сворачивании приложения. Или форма, состоящая из множества полей, снова станет пустой.

При выходе из приложения (по аппаратной кнопке назад) помимо шага 1 и шага 2, выполняется шаг 3. Активити уничтожается и при следующем запуске приложения создается заново.

Поворот экрана



Один из значимых случаев, который плодит баги — это поворот экрана. Оказывается, при повороте экрана активити проходит полный жизненный цикл. А именно уничтожается и снова создается. И так при каждом повороте. Поэтому, опять же, проверяем поворот на каждом экране.

Поддержка горизонтальной ориентации экрана, с одной стороны, позволяет проверить корректность работы всех этапов активити, с другой стороны, значительно увеличивает количество тест-кейсов.

Многооконный режим

Начиная с Андроида 7 (с Андроида 6 как экспериментальная фича) стал доступен многооконный режим. Пользователи получили возможность видеть сразу несколько приложений на экране одновременно. При этом только то приложение, с которым пользователь сейчас взаимодействует, находится в состоянии «**Resumed**». Остальные устанавливаются в состояние «**Paused**».

Don't keep activities (Не сохранять действия)

Надо ли проверять полный жизненный цикл активити, если приложение не поддерживает поворот экрана? Конечно, надо.

Если оперативная память давно не чистилась, ее не очень много, и в параллели с вашим приложением было запущено какое-нибудь ресурсоемкое приложение (например, Pokemon Go), то шанс, что Андроид решит «прибить» процесс с вашей активити при переключении на другое приложение, возрастает.

Как проверить корректность работы приложения вручную, если оно не поддерживает поворот экрана? Довольно просто: установить галку «**don't keep activities**» в настройках разработчика и перезапустить приложение.

В этом случае эмулируется ситуация, когда памяти не хватает и Андроид уничтожает все активити, с которыми пользователь сейчас не взаимодействует, оставляя только ту, что сейчас активна.

Это не значит, что галка должна всегда быть включенной при тестировании, но периодически смотреть приложение с опцией «**don't keep activities**» полезно, чтобы пользователи со слабыми устройствами не сильно страдали и не ругали нас.

Broadcast receiver (широковещательный приемник)

Для обработки внешних событий используется компонент **Broadcast receiver**. Приложение может подписываться на события системы и других приложений. Андроид доставляет событие приложению-подписчику, даже если оно не запущено, и таким образом, может «мотивировать» его запуск.

При тестировании нам важно понимать, какие ожидаются события и как они обрабатываются.

Например, в коде заранее было прописано, что приложение ждет сообщение от конкретного номера и имеет доступ к смс. Когда пользователю придет секретный код, то Broadcast receiver получит уведомление и в поле подтверждения операции будет введен смс-код.

Сервисы (Services)

Еще одна очень важная вещь в Андроиде — это сервисы. Они нужны для выполнения фоновых задач. При этом приложение не обязательно должно быть открыто пользователем в этот момент.

У сервисов есть несколько режимов работы. Пользователь может видеть, что сервис запущен, а может и совершенно не замечать его.

Если вы услышали волшебное слово «сервис» от разработчика, не поленитесь, выясните, какую логику работы заложили в него:

- Что делает сервис и в каком виде проявляет себя?
- Как ведет себя сервис, когда приложение не активно?
- Восстанавливается ли сервис после прерывания (входящего звонка, перезапуска телефона)?

Тут основной совет при проектировании тестовых сценариев — это обдумать пользовательские кейсы, когда работа сервиса может прерваться или начать конфликтовать с работой другого сервиса. Самые коварные ситуации: когда работа сервиса начинается, а пользователь этого не ждал. В этом случае полезно выяснить, что может спровоцировать запуск сервиса.

Самые простые и безобидные — это **сервисы без дополнительных параметров**. При ручном тестировании мы чаще всего не замечаем их работу. Например, если нужно отправить данные в систему аналитики, то для этой задачи нередко используют именно такие сервисы.

Еще один тип сервисов это **sticky-сервисы**. Если работа такого сервиса внезапно завершится, то спустя какое-то время sticky-сервис «возродится». Примечательно, что с каждым разом период до «возрождения» увеличивается, чтобы он меньше мешал работе системы. В некоторых приложениях примером sticky-сервиса может быть скачивание файлов на устройство. Возможно, вы замечали, если в «шторке» сбросить закачку, то спустя какое-то время она может восстановиться и продолжить скачивать файлы.

Самые важные сервисы — те, работу которых пользователь «ощущает» на себе и она для него важна. Они называются **foreground-сервисы**, и у них обязательно есть нотификация в «шторке», которую пользователь не может закрыть. Система их будет уничтожать в последнюю очередь, так как приоритет у таких сервисов самый высокий.

Например, музыкальный плеер. Если свернуть приложение и даже закрыть его, то плеер продолжает играть, пока пользователь не поставит его на паузу или не закроет. Или пока другое приложение или система не приостановят его работу. В частности, для музыкального плеера вариантов может быть много: входящий звонок, другое музыкальное приложение, звуковая нотификация.

Раз речь зашла о музыкальных плеерах, то стоит отметить такое понятие, как аудиофокус.

Аудиофокус

Представим, что при запуске нескольких аудиоплееров, они все будут играть одновременно. Вряд ли это кому-то понравится. Тут на помощь приходит аудиофокус, который запрашивается приложением у системы. В случае **обычного запроса аудиофокуса** система как бы оповещает все запущенные приложения: «Сейчас другое приложение будет говорить, помолчите, пожалуйста». Забавно, но это происходит именно в формате просьбы.

Если ваше приложение не очень вежливое, то оно спокойно может проигнорировать запрос. Наша задача — проверять эти ситуации.

Компромиссный режим запроса аудиофокуса называется «**временным перехватом аудиофокуса**». Это значит, что вашему приложению вернется аудиофокус, когда прервавшее его подаст системе сигнал, что аудиофокус освобожден.

Еще один вид запроса аудиофокуса — это «**duck**». Он просит остальные приложения не молчать, а уменьшить громкость наполовину пока воспроизводится звук запросившего фокус приложения. Например, такой запрос используется при проигрывании звука нотификации о новом сообщении в мессенджере.

Тестирование аудиофокуса очень важно, т.к. здесь все завязано на совести разработчиков и система не принимает особого участия в разрешении конфликтов. Ну а если баг все-таки вылезет к пользователям, то не сомневайтесь, они быстро вам об этом сообщат.

На этом, пожалуй, пока можно закончить. Конечно, есть еще много деталей и нет необходимости учитывать абсолютно все при тестировании. По моему опыту, тестировщику необходимо понимать из чего состоит приложение и как оно работает. Это дает возможность анализировать непростые баги, которые возникают на пересечении бизнес-логики приложения и особенностей работы операционной системы. Особенно если дело касается Андроида.

Content providers

Поставщик контента управляет общим набором данных приложения, которые вы можете хранить в файловой системе, в базе данных SQLite, в Интернете или в любом другом постоянном хранилище, к которому ваше приложение может получить доступ. Через поставщика содержимого другие приложения могут запрашивать или изменять данные, если поставщик содержимого позволяет это. Например, система Android предоставляет поставщика контента, который управляет контактной информацией пользователя. Таким образом, любое приложение с соответствующими разрешениями может запрашивать поставщика содержимого, например `ContactsContract.Data`, для чтения и записи информации о конкретном человеке. Заманчиво думать о провайдере контента как о абстракции базы данных, потому что для этого распространенного случая в них встроено множество API истроенная поддержка. Однако с точки зрения системного дизайна у них другое основное назначение. Для системы поставщик контента - это точка входа в приложение для публикации именованных элементов данных, идентифицированных схемой URI. Таким образом, приложение может решить, как оно хочет сопоставить данные, которые оно содержит, с пространством имен URI, передавая эти URI другим объектам, которые, в свою очередь, могут использовать их для доступа к данным.

Уникальный аспект системы Android заключается в том, что любое приложение может запускать компонент другого приложения. Например, если вы хотите, чтобы пользователь сделал снимок с помощью камеры устройства, вероятно, есть другое приложение, которое делает это, и ваше приложение может использовать его вместо разработки действия для самостоятельной съемки фотографии. Вам не нужно включать или даже ссылаться на код из приложения камеры. Вместо этого вы можете просто запустить действие в приложении камеры, которое делает снимок. По завершении фотография даже возвращается в ваше приложение, чтобы вы могли ее использовать. Пользователю кажется, что камера на самом деле является частью вашего приложения.

Когда система запускает компонент, она запускает процесс для этого приложения, если оно еще не запущено, и создает экземпляры классов, необходимых для компонента. Например, если ваше приложение запускает действие в приложении камеры, которое делает снимок, это действие выполняется в процессе, принадлежащем приложению камеры, а не в процессе вашего приложения. Следовательно, в отличие от приложений в большинстве других систем, приложения Android не имеют единой точки входа (нет функции `main ()`). Поскольку система запускает каждое приложение в отдельном процессе с правами доступа к файлам, которые ограничивают доступ к другим приложениям, ваше приложение не может напрямую активировать компонент из другого приложения. Однако система Android может. Чтобы активировать компонент в другом приложении, доставьте системе сообщение, в котором указывается ваше намерение (**intent**) запустить конкретный компонент. Затем система активирует компонент за вас.

Активация компонентов (Activating components)

Три из четырех типов компонентов - activities, services, and broadcast receivers - активируются асинхронным сообщением, называемым **intent**. Намерения связывают отдельные компоненты друг с другом во время выполнения. Вы можете думать о них как о мессенджерах, которые запрашивают действие у других компонентов, независимо от того, принадлежит ли компонент вашему приложению или другому. Намерение создается с помощью объекта Intent, который определяет сообщение для активации либо определенного компонента (explicit intent), либо определенного типа компонента (implicit intent).

Файл манифеста (The manifest file)

Прежде чем система Android сможет запустить компонент приложения, система должна знать, что компонент существует, прочитав файл манифеста приложения AndroidManifest.xml. Ваше приложение должно объявить все свои компоненты в этом файле, который должен находиться в корне каталога проекта приложения.

Манифест выполняет ряд функций в дополнение к объявлению компонентов приложения, например:

- Определяет любые разрешения (user permissions), которые требуются приложению, такие как доступ в Интернет или доступ для чтения к контактам пользователя;
- Объявляет минимальный уровень API, необходимый для приложения, в зависимости от того, какие API использует приложение;
- Объявляет аппаратные и программные функции, используемые или требуемые приложением, такие как камера, службы Bluetooth или сенсорный экран;
- Объявляет библиотеки API, с которыми необходимо связать приложение (кроме API-интерфейсов Android framework), например библиотеку Google Maps.

Ресурсы приложения (App resources)

Приложение Android состоит из большего, чем просто кода - для него требуются ресурсы, отдельные от исходного кода, такие как изображения, аудиофайлы и все, что связано с визуальным представлением приложения. Например, вы можете определять анимацию, меню, стили, цвета и макет пользовательских интерфейсов действий с помощью файлов XML. Использование ресурсов приложения позволяет легко обновлять различные характеристики вашего приложения без изменения кода. Предоставление наборов альтернативных ресурсов позволяет оптимизировать приложение для различных конфигураций устройств, например для разных языков и размеров экрана.

Для каждого ресурса, который вы включаете в свой проект Android, инструменты сборки SDK определяют уникальный целочисленный идентификатор, который вы можете использовать для ссылки на ресурс из кода вашего приложения или из других ресурсов, определенных в XML. Например, если ваше приложение содержит файл изображения с именем logo.png (сохраненный в каталоге res / drawable /), инструменты SDK генерируют идентификатор ресурса с именем R.drawable.logo. Этот идентификатор сопоставляется с целым числом для конкретного приложения, которое вы можете использовать для ссылки на изображение и вставки его в свой пользовательский интерфейс.

Одним из наиболее важных аспектов предоставления ресурсов отдельно от исходного кода является возможность предоставления альтернативных ресурсов для различных конфигураций устройств. Например, определяя строки пользовательского интерфейса в XML, вы можете перевести строки на другие языки и сохранить эти строки в отдельных файлах. Затем Android применяет соответствующие языковые строки к вашему пользовательскому интерфейсу на основе квалификатора языка, который вы добавляете к имени каталога ресурсов (например, res / values-fr / для французских строковых значений) и языковых настроек пользователя.

Android поддерживает множество различных квалификаторов для ваших альтернативных ресурсов. Квалификатор - это короткая строка, которую вы включаете в имя своих каталогов ресурсов, чтобы определить конфигурацию устройства, для которой следует использовать эти ресурсы. Например, вы должны создать разные макеты для своих занятий в зависимости от ориентации и размера экрана устройства. Когда экран устройства находится в портретной ориентации (высокий), вы можете захотеть, чтобы макет с кнопками был вертикальным, но когда экран находится в альбомной ориентации (широкий), кнопки можно выровнять по горизонтали. Чтобы изменить макет в зависимости от ориентации, вы можете определить два разных

макета и применить соответствующий квалификатор к имени каталога каждого макета. Затем система автоматически применяет соответствующий макет в зависимости от текущей ориентации устройства.

Дополнительные сведения о различных типах ресурсов, которые вы можете включить в свое приложение, и о том, как создавать альтернативные ресурсы для различных конфигураций устройств, см. В разделе [«Предоставление ресурсов»](#). Чтобы узнать больше о передовых методах разработки и разработке надежных приложений производственного качества, см. [Руководство по архитектуре приложений](#).

Источники:

- [Android Developers - Docs - Guides - Application Fundamentals](#)
- [Системный подход к тестированию Android-приложений, или О чём молчали разработчики](#)

Доп. материал:

- YaTalks 2021. Mobile: [Современная архитектура android приложений](#)
- [Видео: Урок 23. Жизненный цикл активити \(Activity Lifecycle\)](#)

Архитектура 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.

Mac OS X

Операционная система, установленная сегодня на все маки и (в измененном виде) на айдевайсы, ведет свою историю аж с 1988 года, который в мире IT известен также тем, что стал годом выпуска первой бета-версии операционной системы NeXTSTEP. Сама NeXTSTEP была детищем команды разработчиков Стива Джобса, который к тому времени уже покинул Apple и основал компанию NeXT, которая занялась разработкой компьютеров для образовательных нужд.

В момент своего появления на свет NeXTSTEP была поистине передовой операционной системой, которая включала в себя множество технологических новаций. В основе ОС лежало модифицированное микроядро Mach, дополненное компонентами ядра FreeBSD, включая эталонную реализацию сетевого стека. Более высокоразвитые компоненты NeXTSTEP были написаны с использованием языка Objective-C и предоставляли разработчикам приложений богатый объектно-ориентированный API. Система была снабжена развитым и весьма удобным графическим интерфейсом (ключевые компоненты которого сохранились в OS X и даже iOS) и мощной средой разработки, включавшей в себя в том числе известный всем современным разработчикам визуальный дизайнер интерфейса.

После провала NeXT и возвращения Стива Джобса в компанию Apple в 1997 году NeXTSTEP легла в основу проекта Rhapsody, в рамках которого началась разработка системы-наследника Mac OS 9. В 2000 году из Rhapsody был выделен открытый проект Darwin, исходники которого опубликованы под лицензией APSL, а уже в 2001 году появилась на свет OS X 10.0, построенная на его основе. Спустя несколько лет Darwin лег в основу операционной системы для готовящегося к выпуску смартфона, о котором до 2007-го, кроме слухов, не было известно почти ничего.

XNU и Darwin

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.

Фреймворки

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.

Далеко не все фреймворки доступны в обеих ОС. Многие из них специфичны только для iOS. В качестве примеров можно привести AssetsLibrary, который отвечает за работу с фотографиями и видео, CoreBluetooth, позволяющий получить доступ к синезубу, или iAd, предназначенный для вывода рекламных объявлений в приложениях. Другие фреймворки существуют только в настольной версии системы, однако время от времени Apple переносит те или иные части iOS в Mac OS или обратно, как, например, случилось с фреймворком CoreMedia, который изначально был доступен только в iOS.

Все стандартные системные фреймворки можно найти в системном каталоге /System/Library/Frameworks/. Каждый из них находится в своем собственном каталоге, называемом бандлом (bundle), который включает в себя ресурсы (изображения и описание элементов интерфейса), хидеры языка си, описывающие API, а также динамически загружаемую библиотеку (в формате dylib) с реализацией фреймворка.

Одна из интересных особенностей фреймворков — их версионность. Один фреймворк может иметь сразу несколько разных версий, поэтому приложение, разработанное для устаревших версий системы, будет продолжать работать, даже несмотря на изменения, внесенные в новые версии ОС. Именно так реализован механизм запуска старых iOS-приложений в iOS 7 и выше. Приложение, разработанное для iOS 6, будет выглядеть и работать именно так, как если бы оно было запущено в iOS 6.

SpringBoard

Уровнем выше находятся приложения, системные и устанавливаемые из магазина приложений. Центральное место среди них занимает, конечно же, SpringBoard (только в iOS), реализующее домашний экран (рабочий стол). Именно оно запускается первым после старта системных демонов, загрузки в память фреймворков и старта дисплейного сервера (он же менеджер композитинга, он же Quartz Compositor), отвечающего за вывод изображения на экран.

SpringBoard — это связующее звено между операционной системой и ее пользователем, графический интерфейс, позволяющий запускать приложения, переключаться между ними, просматривать уведомления и управлять некоторыми настройками системы (начиная с iOS 7). Но также это и обработчик событий, таких как касание экрана или переворот устройства. В отличие от Mac OS X, которая использует различные приложения и демоны-агенты для реализации компонентов интерфейса (Finder, Dashboard, LaunchPad и другие), в iOS почти все базовые возможности интерфейса пользователя, в том числе экран блокировки и «шторка», заключены в одном SpringBoard.

В отличие от других стоковых приложений iOS, которые располагаются в каталоге /Applications, SpringBoard наравне с дисплейным сервером считается частью фреймворков и располагается в каталоге /System/Library/CoreServices/. Для выполнения многих задач он использует плагины, которые лежат в /System/Library/SpringBoardPlugins/. Кроме всего прочего, там можно найти, например, NowPlayingArtLockScreen.lockboundle, отвечающий за отображение информации о проигрываемой композиции на экране блокировки, или IncomingCall.serviceboundle, ответственный за обработку входящего звонка.

Начиная с iOS 6 SpringBoard разделен на две части: сам рабочий стол и сервис BackBoard, ответственный за коммуникации с низкоуровневой частью ОС, работающей с оборудованием (уровень HAL). BackBoard отвечает за обработку таких событий, как касания экрана, нажатия клавиш, получение показания акселерометра, датчика положения и датчика освещенности, а также управляет запуском, приостановкой и завершением приложений.

SpringBoard и BackBoard имеют настолько большое значение для iOS, что, если каким-либо образом их остановить, вся система застынет на месте и даже запущенное в данный момент приложение не будет реагировать на касания экрана. Это отличает их от домашнего экрана Android, который является всего лишь стандартным приложением, которое можно остановить, заменить или вообще удалить из системы (в этом случае на экране останутся вполне рабочие кнопки навигации и строка состояния со «шторкой»).

Приложения

На самой вершине этой пирамиды находятся приложения. iOS различает встроенные (стоковые) высоко привилегированные приложения и сторонние, устанавливаемые из iTunes. И те и другие хранятся в системе в виде бандлов, во многом похожих на те, что используются для фреймворков. Разница заключается лишь в том, что бандл приложения включает в себя несколько иную метаинформацию, а место динамической библиотеки занимает исполняемый файл в формате Mach-O.

Стандартный каталог хранения стоковых приложений — /Applications/. В iOS он абсолютно статичный и изменяется только во время обновлений системы; пользователь получить к нему доступ не может. Сторонние приложения, устанавливаемые из iTunes, напротив, хранятся в домашнем каталоге пользователя /var/mobile/Applications/ внутри подкаталогов, имеющих вид 4-2-2-2-4, где два и четыре — это шестнадцатеричные числа. Это так называемый GUID — уникальный идентификатор, который однозначно идентифицирует приложение в системе и нужен в том числе для создания изолированной песочницы (sandbox).

Sandbox

В iOS песочницы используются для изолирования сервисов и приложений от системы и друг от друга. Каждое стороннее приложение и большинство системных работают в песочнице. С технической точки зрения песочница представляет собой классический для мира UNIX chroot, усиленный системой принудительного контроля доступа TrustedBSD MAC (модуль ядра sandbox.kext), которая отрезает приложениям не только доступ к файлам за пределами домашнего каталога, но и прямой доступ к железу и многим системным функциям ОС.

В целом заключенное в sandbox приложение ограничено в следующих возможностях:

- Доступ к файловой системе за исключением своего собственного каталога и домашнего каталога пользователя.
- Доступ к каталогам Media и Library внутри домашнего каталога за исключением Media/DCIM/, Media/Photos/, Library/AddressBook/, Library/Keyboard/ и Library/Preferences/.
- Доступ к информации о других процессах (приложение «считает» себя единственным в системе).
- Прямой доступ к железу (разрешено использовать только Cocoa API и другие фреймворки).
- Ограничение на использование оперативной памяти (контролируется механизмом Jatsam).

Все эти ограничения соответствуют sandbox-профилю (набору ограничивающих правил) container и применяются к любому стороннему приложению. Для стоковых приложений, в свою очередь, могут применяться другие ограничения, более мягкие или жесткие. В качестве примера можно привести почтовый клиент (профиль MobileMail), который в целом имеет такие же серьезные ограничения, как и сторонние приложения, но может получить доступ ко всему содержимому каталога Library/. Обратная ситуация — SpringBoard, вообще не имеющий ограничений.

Внутри песочниц работают многие системные демоны, включая, например, AFC, предназначенный для работы с файловой системой устройства с ПК, но ограничивающий «область видимости» только домашним каталогом пользователя. Все доступные системные sandbox-профили располагаются в каталоге /System/Library/Sandbox/Profiles/* и представляют собой наборы правил, написанных на языке Scheme. Кроме этого, приложения также могут включать в себя дополнительные наборы правил, называемых entitlement. По сути, это все те же профили, но вшитые прямо в бинарный файл приложения (своего рода самоограничение).

Смысл существования всех этих ограничений двойной. Первая (и главная) задача, которую решает sandbox, — это защита от вредоносных приложений. Вкупе с тщательной проверкой опубликованных в iTunes приложений и запретом на запуск не подписанных цифровым ключом приложений (читай: любых, полученных не из iTunes) такой подход дает прекрасный результат и позволяет iOS находиться на вершине в списке самых защищенных от вирусов ОС.

Вторая проблема — это защита системы от самой себя и пользователя. Баги могут существовать как в стоковом софте от Apple, так и в головах юзеров. Sandbox защищает от обоих. Даже если злоумышленник найдет дыру в Safari и попытается ее эксплуатировать, он все равно останется в песочнице и не сможет навредить системе. А юзер не сможет «сломать свой любимый телефончик» и не напишет гневных отзывов в адрес Apple. К счастью, знающие люди всегда могут сделать jailbreak и обойти защиту sandbox (собственно, в этом и есть смысл джейлбрейка).

Многозадачность

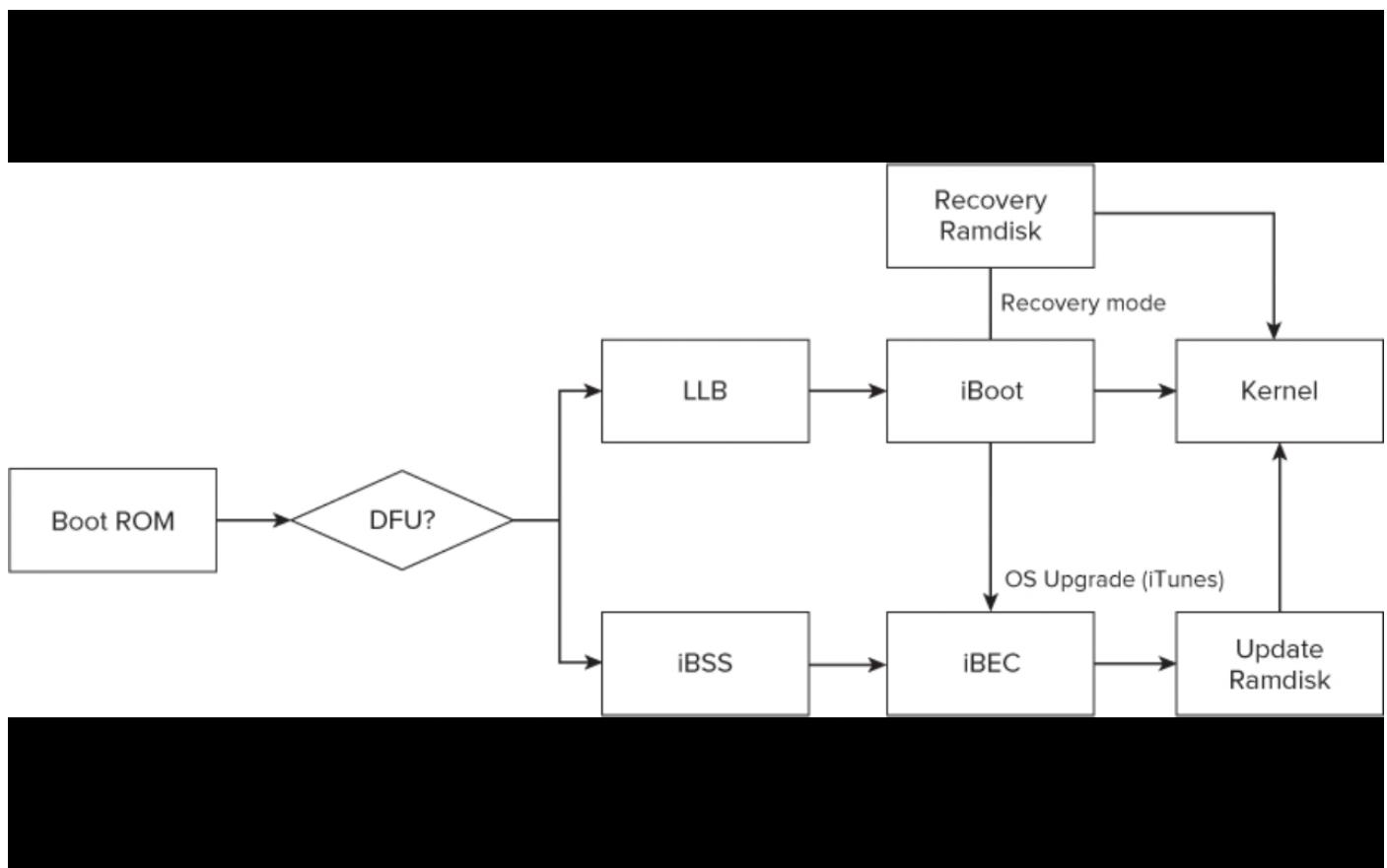
Одна из самых спорных особенностей iOS — это реализация многозадачности. Она вроде бы и есть, а с другой стороны, ее нет. В сравнении с традиционными настольными ОС и пресловутым Android iOS не является многозадачной операционной системой в привычном смысле этого слова и не позволяет приложениям свободно работать в фоне. Вместо этого ОС реализует API, который приложение может использовать для выполнения отдельных задач, пока оно находится в фоновом режиме.

Впервые такой API появился в iOS 4 (до этого фоновые задачи могли выполнять только стоковые приложения) и наращивался по мере развития операционной системы. Сегодня (речь идет об iOS 7) так называемый Background API позволяет делать следующее:

- проигрывать аудио;
- совершать VoIP-звонки;
- получать информацию о смене местоположения;
- получать push-уведомления;
- планировать отложенный вывод уведомлений;
- запрашивать дополнительное время для завершения работы после перехода в фоновый режим;
- обмениваться данными с подключенными к девайсу аксессуарами (в том числе Bluetooth);
- получать и отправлять данные по сети (начиная с iOS 7).

Такие ограничения на работу в фоне необходимы в первую очередь для того, чтобы сохранить заряд батареи и избежать лагов интерфейса, так знакомых пользователям Android, где приложения могут делать в фоне все что захотят. На самом деле Apple настолько сильно заботится о сохранении батареи, что даже реализовала специальный механизм для группировки фоновых действий приложений и их запуска в нужные моменты, например тогда, когда смартфон активно используется, подключен к Wi-Fi-сети или к зарядному устройству.

Шесть стадий загрузки iOS



- Boot ROM. После включения устройства первым запускается минималистичный загрузчик, прошитый в постоянную память устройства. Его задача — произвести начальную инициализацию железа и передать управление первичному загрузчику LLB. Boot ROM всегда имеет заводскую прошивку и не может быть обновлен.
- Low Level Bootloader (LLB). Далее управление получает LLB. Это первый загрузчик, задача которого — найти в памяти устройства iBoot, проверить его целостность и передать ему управление либо переключить девайс в режим восстановления, если это не удалось. Код LLB хранится в NAND-памяти устройства и обновляется вместе с установкой новой версии прошивки. Кроме всего прочего, он выводит на экран загрузочный логотип.
- iBoot. Это вторичный и основной загрузчик айдевайсов. Он включает в себя драйвер файловой системы, с помощью которого получает доступ к содержимому NAND-памяти, находит ядро и передает ему управление. В iBoot также встроен драйвер UART, с помощью которого можно производить отладку ядра и ОС, подключив девайс к COM-порту или USB-порту компа (с помощью кабеля USB — UART).

- Ядро. Здесь все как обычно. Ядро производит инициализацию оборудования, после чего передает управление демону launchd.
- Launchd. Это первичный процесс iOS и Mac OS X, он подключает файловые системы, запускает демоны/службы (например, backupd, configd, locationd), дисплейный сервер, фреймворки, а на последнем этапе загрузки отдает управление SpringBoard. В iOS и Mac OS X launchd используется как замена стандартного /bin/init в UNIX, однако его функциональность гораздо шире.
- SpringBoard. Вот и экран блокировки!

Первые четыре этапа в этой цепи образуют *chain of trust*, реализованный с помощью сверки цифровой подписи загружаемого компонента. Цифровую подпись имеют LLB, iBoot и ядро, что позволяет исключить внедрение в цепочку хакнутого загрузчика или ядра, которые могут быть использованы для загрузки сторонней операционной системы или джейлбрейка. Единственный способ обойти этот механизм — найти дыру в одном из загрузчиков и воспользоваться ею для обхода проверки. В свое время было найдено несколько таких дыр в Boot ROM (наиболее известен экспloit limera1n от geohot, актуальный для iPhone 1–4), а в начале 2014 года и в iBoot (хакер iH8sn0w, экспloit так и не был опубликован).

Удерживая кнопку «Домой» при включении iPhone, можно заставить iBoot загрузиться в так называемый режим восстановления (Recovery), который позволяет восстановить прошивку iOS или обновить ее, используя iTunes. Однако механизм автоматического OTA-обновления использует другой режим, именуемый DFU (Device Firmware Upgrade), который активируется на раннем этапе загрузки сразу после Boot ROM и реализован в двух компонентах: iBSS и iBEC. По сути, это аналоги LLB и iBoot, конечная цель которых — не загрузить ОС, а перевести смартфон в режим обновления.

Источники:

- [Как устроена iOS](#)

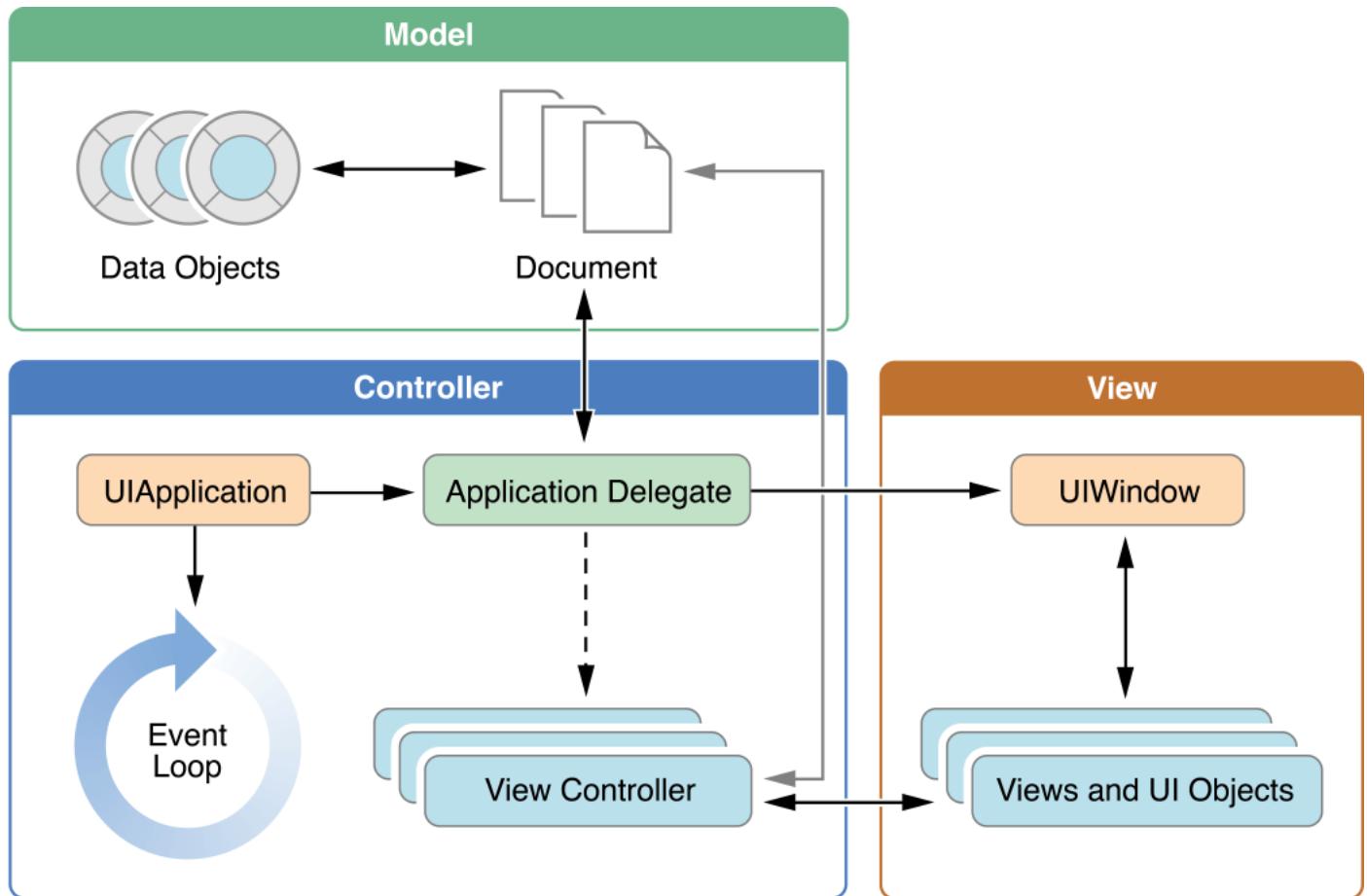
Архитектура iOS Application

Приложения представляют очень сложное взаимодействие между вашим кодом и системными фреймворками. Системный фреймворк предоставляет базовую инфраструктуру, с которой все приложения должны работать, а вы пишете код для настройки этой инфраструктуры.

Фреймворки iOS основаны на таких шаблонах проектирования, как MVC и делегирование.

Жизненный цикл iOS приложения

Точки входа для каждого C-приложения является функция main. Во время запуска UIApplicationMain функция устанавливает несколько ключевых объектов и запускает приложение. Сердцем каждого iOS приложения является объект UIApplication, задача которого — облегчить взаимодействие между системой и другими объектами приложения. Рисунок ниже показывает объекты, обычно встречающиеся в iOS приложениях. А еще ниже представлено описание роли каждого объекта в приложении. Первое что стоит отметить — iOS приложения используют архитектуру Model-View-Controller. Эта архитектура имеет решающее значение для создания приложений, которые могут работать на различных устройствах с различными размерами экрана.



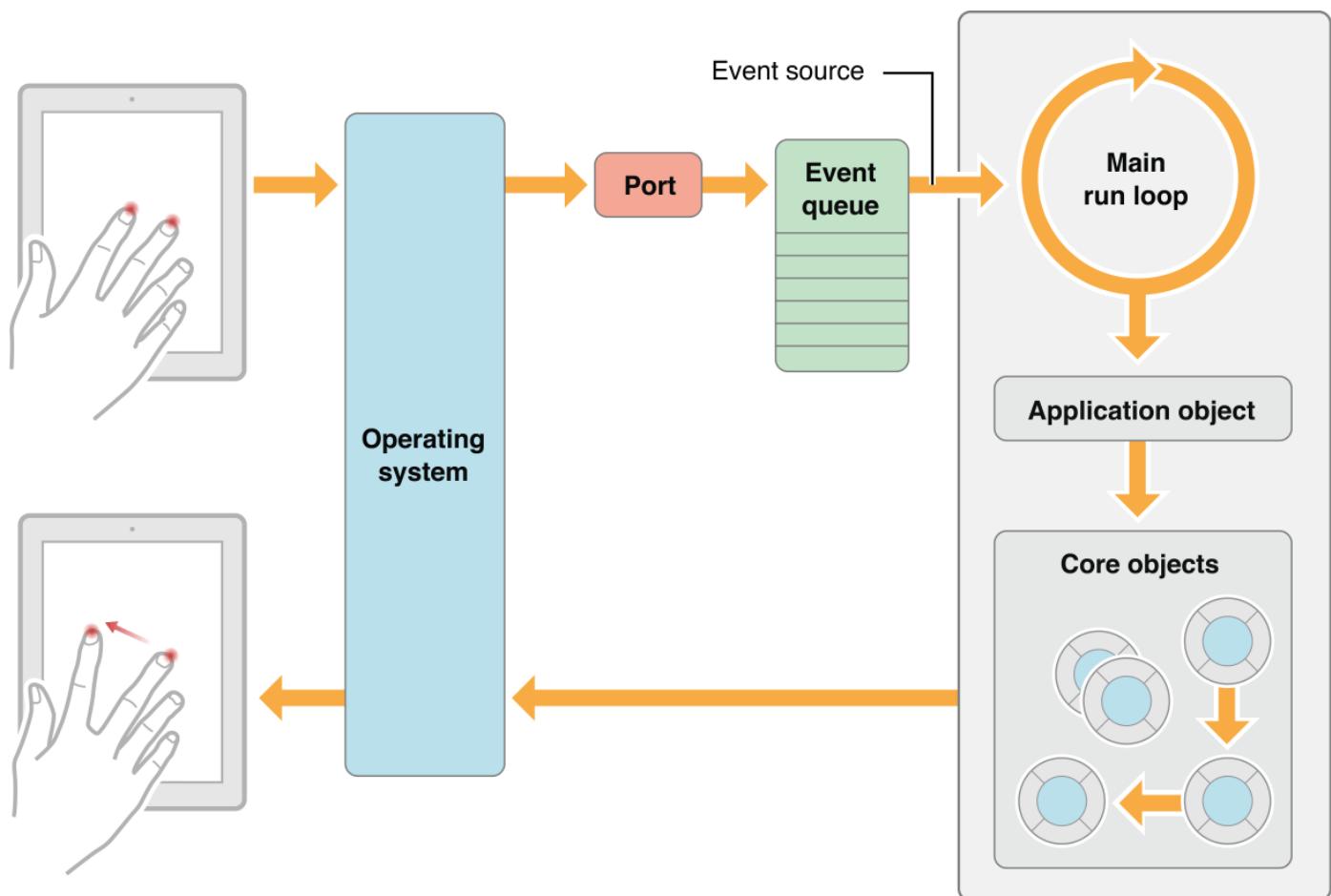
| |
|---------------------------------|
| Custom Objects |
| System Objects |
| Either system or custom objects |

- Объект UIApplication: UIApplication управляет циклом обработки событий и разными типами поведения приложений высокого уровня. Он также сообщает о ключевые переходах приложения и некоторых специальных событиях (например, входящих уведомлений Push) своему делегату, который представляет собой пользовательский объект. Используйте UIApplication «как есть», без подклассов.
- Объект UIDelegate: App Delegate является сердцем вашего кода. Этот объект работает в связке с объектом UIApplication и служит для обработки инициализации приложения, перехода между состояниями, а также обеспечивает множество событий в приложениях высокого уровня.
- Documents и Data Model Objects: Модель данных может быть специфической для вашего приложения. Пример: банковская база данных и приложения со списком фильмов будут иметь разную модель данных. Приложения могут также использовать объекты Documents (подклассы UIDocuments) для управления моделями данных. Объекты Documents необязательны, но предлагают удобный способ группировать данные в пакетах файлов.
- Объект ViewController: Объект ViewController служит для управления отображением контента вашего приложения на экране. Класс UIViewController является базовым классом для всех объектов View Controller. Он предоставляет функциональные возможности по умолчанию для загрузки представления, показывая их, врача их в ответ на поворот устройства, а также несколько других стандартных систем поведения.
- Объект UIWindow: Выводит и координирует один или несколько View на экране. Большинство приложений имеют только один Window, в котором представлен весь контент приложения на

главном экране. Но могут потребоваться дополнительный Window. Например, для отображения вспомогательных элементов на внешнем экране.

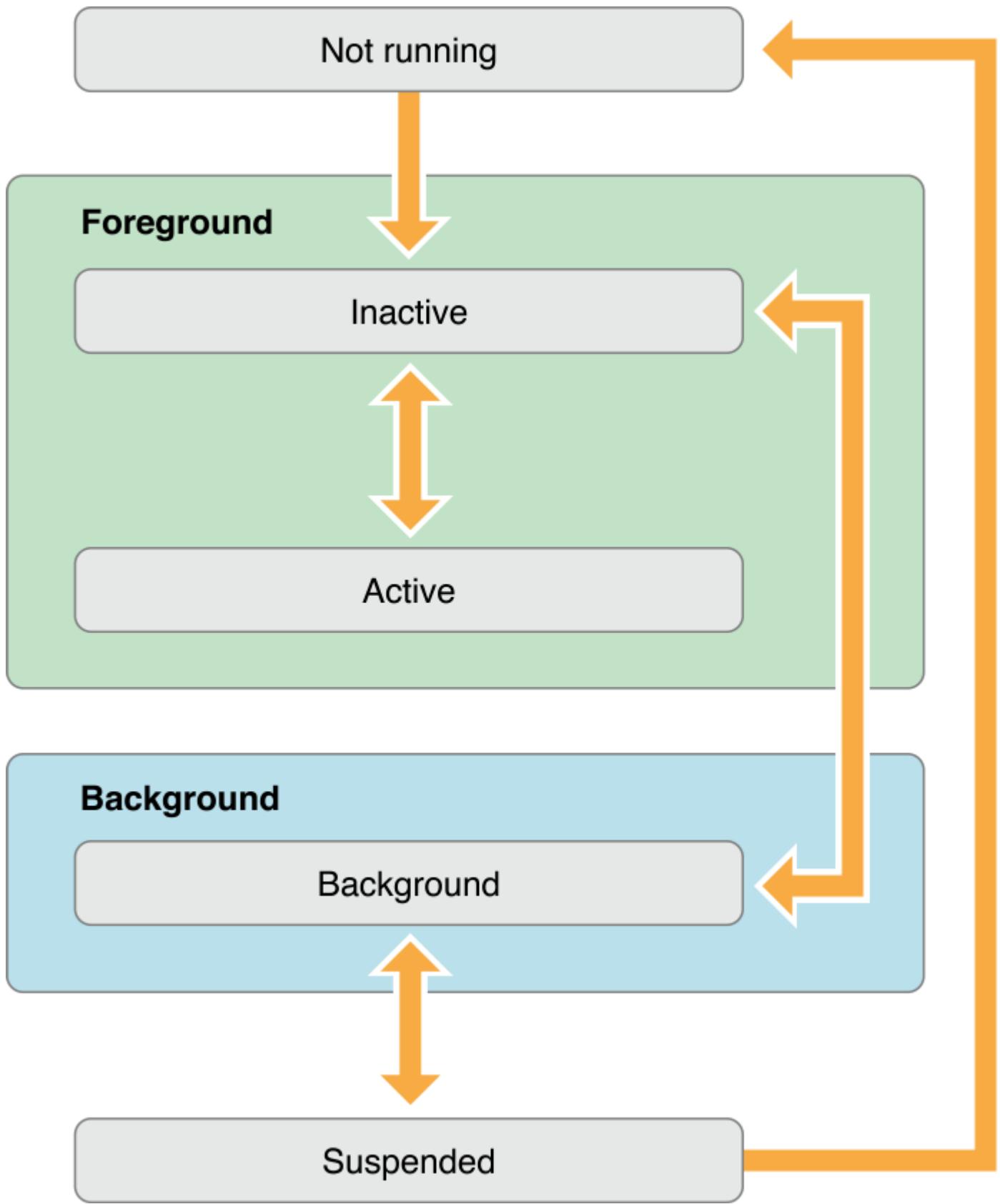
- Объекты View, объекты Controls, объекты Layer: Объекты View обеспечивают визуальное представление содержимого вашего приложения. View является объектом, который рисует содержимое в назначеннной прямоугольной области и реагирует на события в пределах этой области. Объекты Controls представляют из себя специальные типы View, отвечающие за реализацию привычных объектов интерфейса: кнопки, текстовые поля, переключатели и т. д. Объекты Layer - это объекты данных, которые представляют из себя визуальный контент.

Приложение в главном цикле обрабатывает все пользовательские события. Главный цикл работает в главном потоке приложения. На изображении ниже показана схема работы главного цикла.



В любой момент времени ваши приложения 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 запущено 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.



Большинство переходов между состояниями обеспечивается соответствующими методами в AppDelegate.

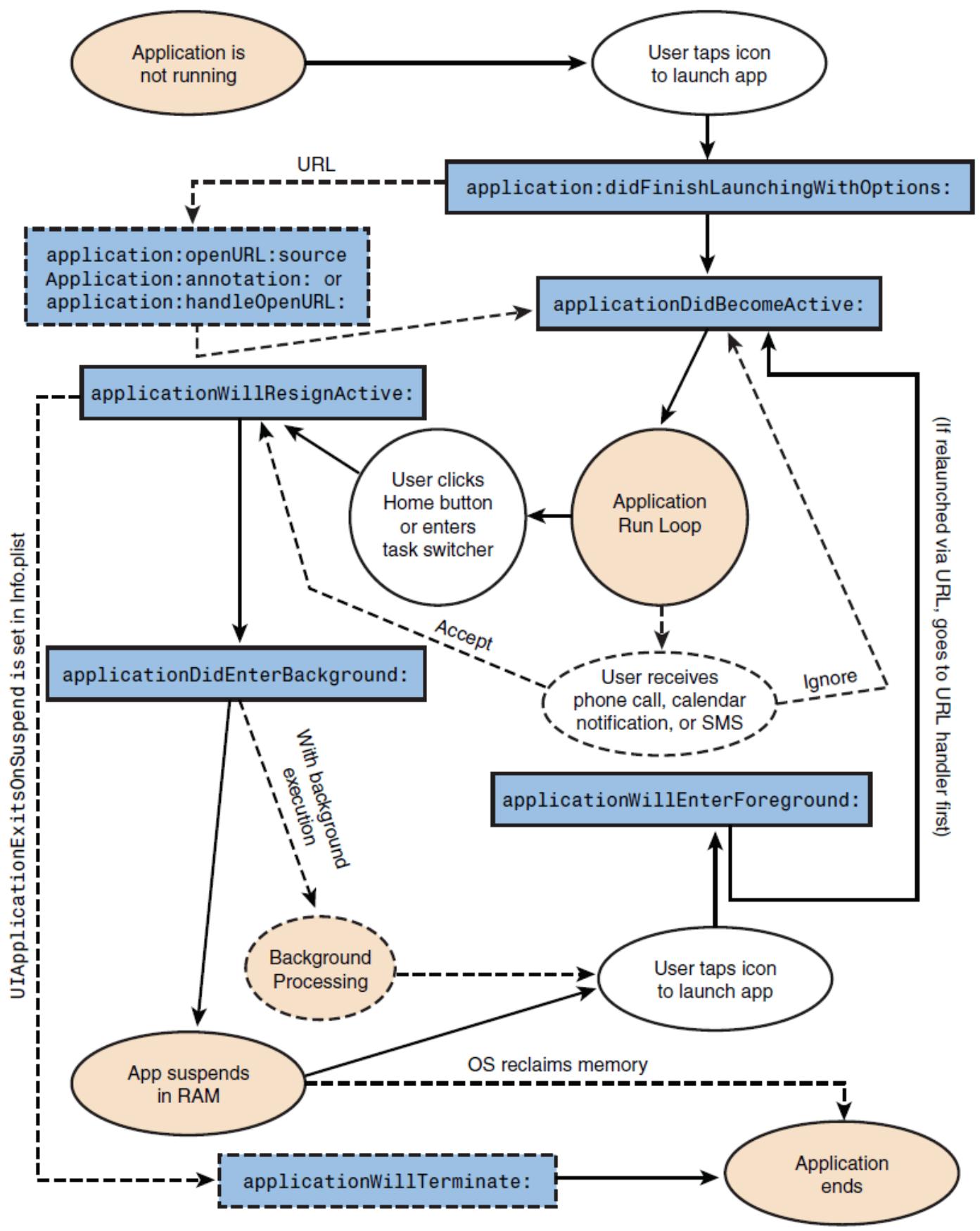
Приложение должно быть готово к завершению в любое время. Завершение — это нормальная часть жизненного цикла. Система обычно выключает приложения, для очищения памяти и подготовки к запуску других приложений, которые запущены пользователем, но система также может выключить приложения, которые некорректно 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.

Система создает приложение в основном потоке и вы можете создавать отдельные потоки, если вам это необходимо, для решения каких либо задач. Для приложений iOS, предпочтительным методом является использование Grand Central Dispatch (GCD), оперирующим с объектами, и другими интерфейсами асинхронного программирования не создавая и управляя потоками собственноручно. Такие технологии как GCD позволяют определить работу, которую вы хотите сделать и в каком порядке вы хотите ее сделать, но пусть система решает как лучше выполнить эту работу для CPU. Когда система управляет вашими потоками вам становится легче писать код, обеспечивается большая корректность кода, а так же увеличивает общую производительность.

The iOS application life cycle.



Источники:

- [Preworking 4: жизненный цикл iOS приложения](#)

Доп. материал:

- [Официальная документация](#)
- [Общее представление об архитектуре Clean Swift](#)

iOS/Android Developer Settings

В настройках обеих платформ скрыто меню разработчика с полезными возможностями.

Информации по меню в iOS почти никакой, нашел лишь упоминание таких пунктов:

- Energy Diagnostic: измерение энергопотребления;
- Network Link conditioner: изменение скорости сети;
- iAd Developer App Testing: отображение и обновление iAds;
- PassKit Testing: дополнительное логирование.

Android developer options:

- **Общие (General):**
 - Память (**Memory**): отображение статистики памяти, такой как среднее использование памяти, производительность памяти, общий объем доступной памяти, средний объем используемой памяти, объем свободной памяти и объем памяти, используемый приложениями;
 - Сделать отчет об ошибке (**Take bug report**): получите копию текущих файлов журнала устройства, чтобы поделиться с кем-нибудь. Когда вы получите уведомление о том, что отчет об ошибке готов, коснитесь его, чтобы поделиться им;
 - Демонстрационный режим UI (**System UI demo mode**): упрощает создание чистых скриншотов за счет отображения стандартной предустановленной панели уведомлений, на которой не отображаются уведомления или предупреждения о низком заряде батареи. “Включить демонстрационный режим” (Enable Demo Mode) позволяет вам изменить внешний вид строки состояния с помощью команд демо-режима adb. Или вы можете использовать “Показать демонстрационный режим” (Show Demo Mode), чтобы скрыть уведомления и отобразить предварительно заданную строку состояния;
 - Пароль резервного копирования рабочего стола (**Desktop backup password**): устанавливает пароль резервного копирования, чтобы вы могли использовать команды adb для резервного копирования и восстановления приложений и данных устройства под защитой паролем;
 - Бодрствовать (**Stay awake**): экран остается включенным каждый раз, когда устройство подключено;
 - Включить журнал отслеживания интерфейса хост-контроллера Bluetooth (HCI) (**Enable Bluetooth Host Controller Interface (HCI) snoop log**): фиксирует все пакеты HCI Bluetooth в файле, хранящемся в /sdcard/btsnoop_hci.log. Вы можете получить пакеты, а затем использовать такую программу, как Wireshark, для анализа и устранения неполадок с информацией.
- **Отладка (Debugging):** Параметры отладки предоставляют способы настройки отладки на устройстве и установления связи между устройством и вашим компьютером для разработки. Включите отладку по USB, чтобы ваше устройство Android могло связываться с вашей машиной разработки через Android Debug Bridge (adb). Параметр “Подождать отладчик” (Wait for Debugger) недоступен, пока вы не используете команду “Выбрать приложение для отладки” (Select debug app). Если вы включите “Ожидание отладчика” (Wait for Debugger), выбранное приложение будет ожидать подключения отладчика перед своим выполнением. Другие опции отладки включают следующее:
 - Постоянно хранить данные журнала на устройстве (**Store logger data persistently on device**): выберите тип сообщений журнала, которые вы хотите постоянно хранить на устройстве. Параметры отключены, все, все, кроме радио, или только ядро;
 - Выберите приложение фиктивного местоположения (**Select mock location app**): используйте этот параметр, чтобы подделать местоположение устройства по GPS. Чтобы использовать эту опцию, загрузите и установите приложение фиктивного местоположения GPS;
 - Включить проверку атрибутов представления (**Enable view attribute inspection**): сохраняет информацию об атрибутах представления в переменной-члене mAttributes экземпляра View, чтобы ее можно было использовать для отладки. Вы можете получить доступ к информации об

- атрибутах через пользовательский интерфейс Layout Inspector (без его включения элемент «Атрибуты» недоступен);
- Включить уровни отладки графического процессора (**Enable GPU debug layers**): включите этот параметр, чтобы разрешить загрузку уровней проверки Vulkan из локального хранилища устройства;
 - **Сети (Networking):** Параметры сети позволяют настраивать параметры Wi-Fi и DHCP. Нажмите “Выбрать конфигурацию USB” (Select USB Configuration), чтобы указать, как компьютер будет идентифицировать устройство. Вы можете настроить устройства только для зарядки, для передачи файлов (MTP), для передачи изображений (PTP), для использования вашего мобильного Интернета на ПК (RNDIS) или для передачи аудио или MIDI файлов. Нажмите “Версия Bluetooth AVRCP” (Bluetooth AVRCP version) и выберите версию профиля, которую вы хотите использовать для управления всем аудио / видео оборудованием Bluetooth, к которому у вашего устройства есть доступ. Чтобы точно настроить воспроизведение звука на устройстве есть следующие параметры: Bluetooth Audio Codec, Bluetooth Audio Sample Range, Bluetooth Audio Bits Per sample, Bluetooth Audio Channel Mode, Bluetooth Audio LDAC Codec. В следующем списке описаны другие способы настройки Wi-Fi и DHCP:
 - Сертификация беспроводного дисплея (**Wireless display certification**): включает расширенные средства управления конфигурацией и настройки для сертификации беспроводного дисплея в соответствии со спецификациями, изложенными в Спецификации Wi-Fi дисплея Wi-Fi Alliance. Сертификация распространяется на Android 4.4 (уровень API 19) и выше;
 - Включить подробное ведение журнала Wi-Fi (**Enable Wi-Fi verbose logging**): увеличивает уровень ведения журнала Wi-Fi для каждой беспроводной сети (SSID), к которой вы подключаетесь, в соответствии с относительной мощностью принимаемого сигнала (RSSI). Дополнительные сведения о журналах см. В разделе [Запись и просмотр журналов с помощью Logcat](#).
 - Агрессивный переход от Wi-Fi к сотовой сети (**Aggressive Wi-Fi to cellular handover**): при низком уровне сигнала Wi-Fi более эффективно переключает соединение для передачи данных в сотовую сеть;
 - **Ввод (Input):**
 - Показывать касания (Show taps), нужно для отображения мест касаний экрана. Под вашим пальцем или стилусом появляется круг, который следует за вами при перемещении по экрану. Касание работает как указатель, когда вы записываете видео на свое устройство;
 - Местоположение указателя (Pointer Location), нужно для отображения местоположения указателя (касания) на устройстве с помощью перекрестия. В верхней части экрана появится полоса для отслеживания координат перекрестия. Когда вы перемещаете указатель, координаты на полосе отслеживают положение перекрестия, а путь указателя отображается на экране;
 - **Рисование (Drawing):** Параметры рисования предоставляют визуальные подсказки о пользовательском интерфейсе приложения и о том, как он работает. Включите параметр “Показать границы макета” (Show Layout Bounds), чтобы отображать границы, поля и другие конструкции пользовательского интерфейса вашего приложения на устройстве. Другие опции рисования включают следующее:
 - Принудительное направление макета RTL (**Force RTL layout direction**): Принудительное направление макета экрана справа налево (RTL) или слева направо (по умолчанию);
 - Масштаб анимации окна (**Window animation scale**): устанавливает скорость воспроизведения анимации окна, чтобы вы могли проверить ее производительность на разных скоростях. Чем меньше масштаб, тем выше скорость;
 - Масштаб анимации перехода (**Transition animation scale**): устанавливает скорость воспроизведения анимации перехода, чтобы вы могли проверить ее производительность на разных скоростях. Чем меньше масштаб, тем выше скорость;
 - Имитация вторичных дисплеев (**Simulate secondary displays**): создание вторичного дисплея в качестве наложения на устройстве. Это полезно при поддержке дополнительных дисплеев с помощью Presentation API. См. [Дополнительные дисплеи](#);

- **Аппаратное ускорение рендеринга (Hardware accelerated rendering):** Параметры аппаратного ускорения рендеринга позволяют оптимизировать ваше приложение для целевых аппаратных платформ за счет использования аппаратных опций, таких как графический процессор, аппаратные уровни и сглаживание мультисэмплов (MSAA). Нажмите “Имитировать цветовое пространство” (Simulate color space), чтобы изменить цветовую схему всего пользовательского интерфейса устройства. Варианты относятся к типам дальтонизма. Возможны следующие варианты: Disabled (без моделируемой цветовой схемы), Monochromacy (черный, белый и серый), Deuteranomaly (красно-зеленый), Protanomaly (красно-зеленый) и Tritanomaly (сине-желтый). Протаномалия относится к красно-зеленой цветовой слепоте со слабостью в красных тонах, а дейтераномалия к красно-зеленой дальтонии со слабостью в зеленых тонах. Если вы делаете снимки экрана в смоделированном цветовом пространстве, они выглядят нормально, как если бы вы не меняли цветовую схему. Вот некоторые другие опции:
 - Настроить рендеринг GPU (**Set GPU renderer**): измените графический движок OpenGL по умолчанию на графический движок OpenGL Skia;
 - Принудительный рендеринг GPU (**Force GPU rendering**): заставляет приложения использовать графический процессор для 2D-рисования, если они были написаны без графического рендеринга по умолчанию;
 - Показать обновления GPU view (**Show GPU view updates**): отображает любой экранный элемент, нарисованный графическим процессором;
 - Отладка перерисовки GPU (**Debug GPU overdraw**): отображает цветовую кодировку на вашем устройстве, чтобы вы могли визуализировать, сколько раз один и тот же пиксель был отрисован в одном кадре. Визуализация показывает, где ваше приложение может выполнять больше рендеринга, чем необходимо. Дополнительные сведения см. В разделе [Визуализация перерисовки графического процессора](#);
 - Отладка непрямоугольных операций обрезки (**Debug non-rectangular clip operations**): отключает область обрезки на холсте для создания необычных (непрямоугольных) областей холста. Обычно область обрезки предотвращает рисование чего-либо за пределами круглой области обрезки;
 - Принудительное сглаживание (**Force 4x MSAA**): включает мультисэмпловое сглаживание (MSAA) в приложениях OpenGL ES 2.0;
 - Отключить HW-оверлеи (**Disable HW overlays**): использование аппаратного оверлея позволяет каждому приложению, отображающему что-либо на экране, использовать меньше вычислительной мощности. Без наложения приложение разделяет видеопамять и должно постоянно проверять наличие коллизий и отсечения, чтобы отобразить правильное изображение. Проверка использует большую вычислительную мощность;
- **Медиа (Media):** Включите параметр “Отключить маршрутизацию звука USB” (Disable USB audio routing), чтобы отключить автоматическую маршрутизацию на внешние аудиоустройства, подключенные к компьютеру через порт USB. Автоматическая маршрутизация может мешать работе приложений, поддерживающих USB. В Android 11 и более поздних версиях, когда приложение без разрешения RECORD_AUDIO использует UsbManager для запроса прямого доступа к USB-аудиоустройству с возможностью захвата звука (например, USB-гарнитуре), появляется предупреждающее сообщение, предлагающее пользователю подтвердить разрешение на использование устройства. Система игнорирует любой параметр «всегда использовать», поэтому пользователь должен подтверждать предупреждение и предоставлять разрешение каждый раз, когда приложение запрашивает доступ. Чтобы избежать такого поведения, ваше приложение должно запросить разрешение RECORD_AUDIO;
- **Мониторинг (Monitoring):** Параметры мониторинга предоставляют визуальную информацию о производительности приложения, например о длинном потоке (long thread) и операциях с графическим процессором. Нажмите “Профилировать рендеринг графического процессора” (Profile GPU Rendering), а затем “На экране в виде полос” (On screen as bars), чтобы отобразить визуализацию профилирования графического процессора в виде полос. Для получения дополнительной информации см. [Profile GPU rendering](#);

- **Приложения (Apps):** Параметры приложения помогают понять, как ваше приложение работает на целевом устройстве.
 - Ограничение фоновых процессов (**Background process limit**), нужно чтобы установить количество процессов, которые могут работать в фоновом режиме одновременно.
 - Сбросить ограничение в ShortcutManager (**Reset ShortcutManager rate-limiting**), нужно во время тестирования, чтобы фоновые приложения могли продолжать вызывать shortcut APIs, пока не будет достигнут предел;
 - Не сохранять активити (**Don't keep activities**) нужно чтобы увеличить время автономной работы, уничтожая все активити, как только пользователь покидает главное окно активити, либо для тестирования некоторых кейсов.

Источники:

- [Android Developers - Android Studio - User guide - Configure on-device developer options](#)

Основные различия iOS и Android

Последние годы обе системы заимствовали что-то друг у друга и пользовательский опыт теперь у них мало чем отличается, тем не менее, различий всё ещё хватает в другом:

- iOS - закрытая система, а Android - open-source;
- Языки разработки: Android - Kotlin/Java, iOS – objective-C или Swift;
- Гайдлайны: [Human Interface Guidelines](#) (HIG) у iOS и [Material Design](#) у Android;
- Простота получения root-прав на многих устройствах Android;
- Различная целевая аудитория, в т.ч. разный ее размер (у Android сегодня ориентировано 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);
- Различный market share (например, в США в лидерах айфоны);
- Отличия в модерации приложений для публикации в магазине приложений - у Android процедура значительно быстрее и проще;
- Функция «Назад» (виртуальная кнопка или жест). Во всех Android-смартфонах клавишу, возвращающую на шаг назад, можно закрепить в нижней части экрана (вместе с виртуальными кнопками возврата на главный экран и вызова меню запущенных программ). С жестами ситуация не отличается: на каком экране (или в каком приложении) ни находился бы юзер, проведя пальцем вправо или влево от соответствующей стороны дисплея, он сможет вернуться на шаг назад. В iOS все работает немного иначе. Вернее, универсальности на уровне системы, а не отдельной программы не предусмотрено. В некоторых приложениях жест «Назад» присутствует (например, в популярном Instagram) и действует едва ли не с середины дисплея — удобно. Где-то такое работает только с левого края экрана (скажем, в приложении «Настройки»). Еще где-то его просто нет. «Передвигаться» по меню такие программы предлагают нажатием на виртуальную клавишу, предусмотренную разработчиками конкретной программы;
- В сети в целом часто жалуются на файловую систему iOS. Вернее, претензии предъявляются к ее серьезной ограниченности. Если к Android-устройству можно подключить накопитель и переписать данные с него или на него, или оперировать файлами через тот же telegram, то с iPhone такое уже не пройдет (не считая условных фотографий, и то не без «костылей», если речь не про синхронизацию с Mac);
- Отсутствие возможности набора текста свайпами на русской раскладке фирменной клавиатуры Apple (в сторонних можно);
- Скачать какой-нибудь .ipa и установить в обход App Store не получится;
- Возможность установки свежей версии «операционки» на смартфоны от Apple, вышедшие более пяти лет назад и уже не доступные в продаже.

Источники:

- [Как я перешел на iPhone после многих лет с Android-смартфонами. Плюсы и минусы экосистемы Apple](#)

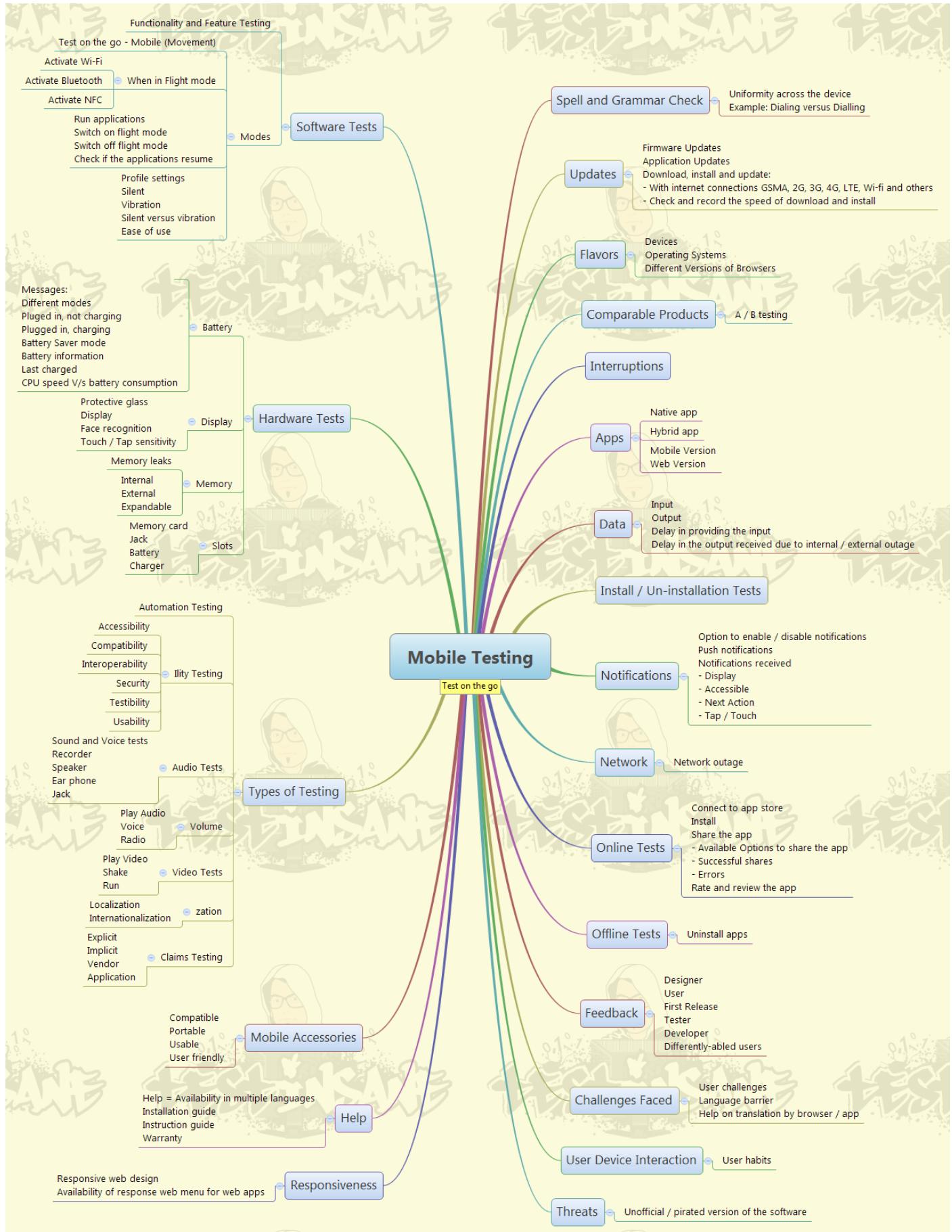
Доп. материал:

- [32 отличия дизайна мобильного приложения под iOS и Android](#)
- [Гайдлайны Google Material и Apple Human Interface. Android, iOS и Material You](#)

Последнее обновление Android/iOS, что нового?

Актуализируется перед собеседованием, т.к. написанное здесь будет слишком быстро устаревать.

Основные проверки при тестировании мобильного приложения



ISTQB® – FOUNDATION LEVEL SPECIALIST SYLLABUS - MOBILE APPLICATION TESTING

| Business and Technology Drivers | Mobile Applications Test Types | Common Test Types and Test Process for Mobile Applications | Mobile Application Platforms, Tools and Environment | Automating the Test Execution |
|--|---|---|---|---|
| Mobile Analytics Data | Testing for Compatibility with Device Hardware | Common Test Types Applicable for Mobile Testing | Development Platforms for Mobile Applications | Automation Approaches and Frameworks |
| Business Models for Mobile Apps | Testing for Device Features | Installability Testing | Common Development Platform Tools | Automation Solutions for Mobile Application Testing |
| Mobile Device Types | Testing for Different Displays | Security Testing | Emulators and Simulators Overview | Evaluation of Test Automation Tools for Mobile Applications |
| Types of Mobile Applications | Testing for Device Temperature | Performance Testing | Using Emulators and Simulators | Approaches for Test Lab Creation |
| Mobile Application Architecture | Testing for Device Input Sensors | Stress Testing | Setting up a Test Lab | |
| Test Strategy for Mobile Apps | Testing Various Input Methods | Usability Tests | | |
| Challenges of Mobile Application Testing | Testing for Screen Orientation Change | Database Testing | | |
| Risks in Mobile Application Testing | Testing for Typical Interrupts | Globalization and Localization Testing | | |
| | Testing for Access Permission to Device Features | Accessibility Testing | | |
| | Testing for Power Consumption and State | Additional Test Levels Applicable for Mobile Applications | | |
| | Testing for App Interaction with Device Software | Field Testing | | |
| | Testing for Notifications | Testing for Application Store Approval and Post-Release Testing | | |
| | Testing for Quick-Access Links | Experience-Based Testing Techniques | | |
| | Testing for OS-provided User Preferences | Personas and Mnemonics | | |
| | Testing for Different Types of Apps | Tours and Heuristics | | |
| | Testing for Inter-operability with Multiple Platforms and OS Versions | Mobile App Specific Tours | | |
| | Testing for Co-existence with other Apps on the | Session-Based Test | | |

В описанный ниже чек-лист вошли только общие характеристики. Естественно, в тестируемом приложении может быть функциональность, для которой нужно применять отдельный подход и создать отдельные сценарии. То же самое верно для производительности, удобства использования, безопасности и прочего тестирования, которое необходимо вашему приложению. Чек-лист состоит из восьми разделов:

- Функциональное тестирование;
- Тестирование совместимости;
- Тестирование безопасности;
- Тестирование локализации и глобализации;
- Тестирование удобства использования;
- Стressовое тестирование;
- Кросс-платформенное тестирование;
- Тестирование производительности.

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

- Установка/удаление/накатка версий;
- Запуск приложения (отображение Splash Screen);
- Работоспособность основного функционала приложения;
 - Авторизация (по номеру телефона/через соц. сети/e-mail);
 - Регистрация (по номеру телефона/через соц. сети/e-mail);
 - Онбординг новых пользователей;
 - Валидация обязательных полей;
 - Навигация между разделами приложения;
 - Редактирование данных в профиле пользователя;
 - Проверка оплаты;
 - Тестирование фильтров;
 - Бонусы;
- Корректное отображение ошибок;
- Работа с файлами (отправка/получение/просмотр);
- Тестирование тайм-аутов;
- Тестирование заглушек (не соединения с интернетом/нет, например, товаров и т.д.);
- Тестирование pop-up, алертов;
- Тестирование WebView;
- Скролл/свайп элементов;
- Тестирование PUSH уведомлений;
- Сворачивание/разворачивание приложения;
- Разные типы подключений (сотовая связь/Wi-Fi);
- Ориентация экрана (альбомная/портретная);
- Темная/светлая темы;
- Реклама в приложении;
- Шаринг контента в соц. сети;
- Работа приложения в фоне;
- Пагинация страниц;
- Политики конфиденциальности и прочие ссылки на документы.

Тестирование совместимости: Тестирование совместимости используется, чтобы убедиться, что ваше приложение совместимо с другими версиями ОС, различными оболочками и сторонними сервисами, а также аппаратным обеспечением устройства.

- Корректное отображение гео;
- Информации об операциях (чеки и т.д.);
- Различные способы оплаты (Google Pay, Apple Pay);
- Тестирование датчиков (освещенности, температуры устройства, гироскоп и т.д.);

- Тестирование прерываний (входящий звонок/cmc/push/будильник/режим «Не беспокоить» и т.д.);
- Подключение внешних устройств (карта памяти/наушники и т.д.).

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

- Тестирование разрешений (доступ к камере/микрофону/галерее/и т.д.);
- Данные пользователя (пароли) не передаются в открытом виде;
- В полях, с вводом пароля и подтверждением пароля, данные скрываются астерисками.

Тестирование локализации и глобализации: Тестирование интернационализации/глобализации приложения включает тестирование приложения для различных местоположений, форматов дат, чисел и валют, а также замену фактических строк псевдострочками. Тестирование локализации включает тестирование приложения с локализованными строками, изображениями и рабочими процессами для определенного региона.

- Все элементы в приложении переведены на соответствующий язык;
- Тексты защиты внутри приложения и пользователь в настройках приложения может выставить необходимый язык;
- Тексты зависят от языка в системных настройках;
- Тексты приходят с сервера;
- Корректное отображение форматов дат (ГОД — МЕСЯЦ — ДЕНЬ или ДЕНЬ — МЕСЯЦ — ГОД.);
- Корректное отображение времени в зависимости от часового пояса.

Тестирование удобства использования

Тестирование удобства использования помогает удостовериться в простоте и эффективности использования продукта пользователем, с целью достижения поставленных целей. Иными словами, это не что иное, как тестирование дружелюбности приложения для пользователя.

- Корректное отображение элементов на устройствах с различными разрешениями экранов;
- Все шрифты соответствуют требованиям;
- Все тексты правильно выровнены;
- Все сообщения об ошибках верные, без орфографических и грамматических ошибок;
- Корректные заголовки экранов;
- В поисковых строках присутствуют плейсхолдеры;
- Неактивные элементы отображаются серым;
- Ссылки на документы ведут на соответствующий раздел на сайте;
- Анимация между переходами;
- Корректный возврат на предыдущий экран;
- Поддерживаются основные жесты при работе с сенсорными экранами (swipe back и т.д.);
- Пиксель-перфект.

Стрессовое тестирование: Стрессовое тестирование направлено на определение эффективности производительности приложения в условиях повышенной нагрузки. Стress-тест в этом контексте ориентирован только на мобильные устройства.

- Высокая загрузка центрального процессора;
- Нехватка памяти;
- Загрузка батареи;
- Отказы;
- Низкая пропускная способность сети;
- Большое количество взаимодействий пользователя с приложением (для этого может понадобиться имитация реальных условий состояния сети).

Кросс-платформенное тестирование: Важный вид тестирования, который необходимо проводить для понимания того, будет ли должным образом отображаться тестируемый продукт на различных платформах, используемых целевой аудиторией.

- Работоспособность приложения на различных устройствах разных производителей

Тестирование производительности: Если пользователь устанавливает приложение, и оно не отображается достаточно быстро (например, в течение трех секунд), оно может быть удалено в пользу другого приложения. Аспекты потребления времени и ресурсов являются важными факторами успеха для приложения, и для измерения этих аспектов проводится тестирование производительности.

- Время загрузки приложения;
- Обработка запросов;
- Кэширование данных;
- Потребление ресурсов приложением (например расход заряда батареи).

Помимо прочего, можно использовать **эвристики и мнемоники:** I SLICED UP FUN, COP FLUNG GUN, SFDPOT, LONG FUN CUP.

Источники:

- [Чек-лист тестирования мобильных приложений](#)

Доп. материал:

- [ISTQB Mobile Application Testing](#)
- [Жизнь без AppStore и Google Play: работаем с Huawei Mobile Services и AppGallery](#)
- [Особенности тестирования Android без Google-сервисов](#)
- Больше чек-листов и идей можно найти в разделе полезных ресурсов
- YaTalks 2021. Mobile: [Моделирование угроз для мобильных приложений](#)
- [Как тестировать мобильные игры](#)
- [Free Mobile App Testing Tutorial](#)
- [Как тестировать мобильное приложение](#)

Каким образом тестировщик получает приложение на тест?

Android:

- Разработчик скинет .apk :) или .aab, который нужно разархивировать;
- Из CI-агента. Тот же Jenkins/TeamCity может присыпать ссылку на билд в tg-канал или можно забрать его вручную;
- Сбилдить в Android Studio самому из нужной ветки;
- [Открытые и закрытые бета-тестирования приложений в Google Play](#);

iOS:

- внутреннее и внешнее тестирование в [TestFlight](#);
- сервис [tiny.app.link](#)

Симуляторы и эмуляторы

Реальное устройство: позволяет запускать мобильные приложения и проверять его функциональность.

Тестирование реального устройства гарантирует, что ваше приложение будет работать без проблем на клиентских телефонах. Когда устройств становится слишком много, их иногда собирают в так называемые фермы устройств. Реальными устройствами также не обязательно обладать, сейчас широко распространены облачные решения.

Эмулятор: пытается дублировать устройство - это полноценная виртуалка (контейнер) со своей сетевой картой и диском, то есть представляет собой полную повторную реализацию конкретного устройства или платформы изолированно внутри нашей хост-системы. Одним из недостатков такого подхода является скорость работы. Примером служит эмулятор в Android Studio, хотя можно найти и неофициальные образы Android-устройств.

Симулятор: пытается дублировать только 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 с настоящим устройством, но при этом под капотом оно является лишь ПО-имитацией, не работающей изолированно от нашей системы и использующей общий диск и сеть. Примером служит симулятор в XCode.

Push-уведомления: принципы работы и способы тестирования

- + как отправить пуш через постман или андроид студио

Push-уведомления — это сообщения, отправляемые приложением на мобильное устройство клиента. Они обычно используются для доставки обновлений продуктов, напоминаний, персонализированных предложений, последних новостей и любой информации, которая является неотъемлемой частью функциональности приложения и требует особого внимания или быстрых действий.

Принцип работы push-уведомлений

- пользователь устанавливает приложение на устройство;
- выдается запрос прав на отправку уведомлений, и в случае успеха — ОС получает токен (идентификатор устройства) у службы push-уведомлений;
- ОС передает токен на сервер для подключения к уведомлениям;
- сервер шлет уведомления при наступлении определенного события.

В случае iOS уведомления работают через облачную платформу APNS (Apple Push Notification Service).

Если говорить о решении push-уведомлений от Android, то есть несколько вариантов. Самый простой способ действовать — использовать Firebase Cloud Messaging (для устройств Android с Google Apps). Если у ваших пользователей есть устройства Huawei (а именно, без Google Apps), вам следует прибегнуть к Huawei Push Kit.

Конечно, вы также можете создать собственного провайдера push-уведомлений или использовать готовые проекты, поскольку платформа имеет открытый исходный код.

Разница между push-уведомлениями в iOS и Android

Функции push-уведомлений в iOS и Android довольно сильно различаются.

iOS основана на модели push Opt-In, которая не позволяет брендам отправлять мобильные push-уведомления пользователям своих приложений до тех пор, пока эти пользователи не согласятся их получать. Android, с другой стороны, автоматически разрешает пользователям получать push-уведомления с возможностью отказаться от них вручную.

Подход Android по сравнению с iOS по умолчанию дает более широкую аудиторию пользователей с поддержкой push. Однако, когда у пользователей нет возможности легко отказаться от их получения, нерелевантные или слишком частые уведомления могут подтолкнуть клиентов отключить сообщения или удалить приложение.

Тестирование push-уведомлений

Не приходят push-уведомления: Чтобы разобраться в причине, для начала проверьте, чтобы в меню устройства была активирована соответствующая функция (разрешены уведомления для конкретного приложения). Затем убедитесь, что не включен режим «Не беспокоить». Если всё настроено правильно, но уведомления не приходят, попробуйте перезагрузить устройство и заново авторизоваться в приложении. Бывает так, что необходимо заново отправить push-токен на серверную часть сервиса. Проверьте также, какой стиль уведомления используется (необходим «Баннер» либо «Предупреждение»). Если не помогло всё перечисленное, попробуйте перезайти в свою учетную запись магазина приложений, либо откройте саму программу, в том случае, если на другие приложения тоже не приходят push-уведомления (стоит также проверить наличие интернета на устройстве).

Переходы по push-уведомлению: При тестировании необходимо проверить такие сценарии (с учётом того, что пользователь может быть авторизован или неавторизован):

- переход по push-уведомлению с заблокированного экрана;
- переход по push-уведомлению из «шторки»;
- пользователь находится в приложении;
- переход по push-уведомлению при свёрнутом приложении;
- пользователь разлогинился после получения push;
- переход по push-уведомлению с включенным «Don't keep Activities» (характерно для Android-приложений).

Существуют push-уведомления, которые ведут на определенный экран с выбором определенных фильтров. В таком случае необходимо проверить, что переход осуществляется на правильный экран. Если это был поисковой запрос, то проверьте, что текст поискового запроса отображается в строке поиска и выдача товаров соответствует поиску. Также могут передаваться определенные фильтры, в таком случае необходимо проверить, что выбраны все «зашитые» фильтры.

Если push-уведомление ведет на WebView, то проверьте, что WebView открывается корректно на обеих платформах. И что в push зашифрован корректный URL.

Устаревший push-токен: У устройства изменился push-токен, когда восстановили приложение из резервной копии системы и не передался новый push-токен.

Очередь со стороны Apple: В Apple большая очередь на отправку push-уведомлений, они приходят с задержкой (Apple не гарантирует доставку push).

Проверка максимального и минимального количества отображаемых символов: В iOS и Android имеется лимит отображаемых символов. Он разный. Максимальное значение количества символов для платформы iOS – ограничение в 4 строки (178 символов), а для Android – не более 13 строк (663 символа). Не забудьте также проверить push-уведомление, содержащее минимальное количество символов, для обоих платформ можно задать 1 символ.

Кастомный звук для push-уведомления: При тестировании push-уведомлений важно учитывать тот факт, что звук push-уведомления может быть задан кастомный. В таком случае необходимо проверять и звуковое сопровождение нотификации.

Изображения в push-уведомлениях: Push-уведомление может содержать изображение, при отправке пуша – клиент получает ссылку на изображение и перед показом загружает его, далее происходит процесс обогащения пуша картинкой – она устанавливается. Уведомление отображается после загрузки картинки. Если push-уведомление содержит картинку, необходимо проверить, что она отображается.

Локальные push-уведомления: Локальные уведомления планируются самим приложением и служат для своевременного и актуального информирования пользователей, пока приложение не работает на переднем плане. Чтобы уведомление отобразилось, его необходимо запланировать самому пользователю. В таких случаях проверяем кейсы, связанные с таймингом отправки сообщения.

Проблемы на серверной стороне: В другие приложения приходят push-уведомления, но не приходит на наше, хотя push-токен отправлен на сервер. Стоит проверить корректность отправки push на другие аккаунты сервиса и другие устройства. При отсутствии push-уведомлений сообщите команде серверной разработки.

Источники:

- [Тестирование push-уведомлений в мобильных приложениях](#)
- [Механизм пуш-уведомлений для iOS и Android](#)

Покрытие девайсов

Большинство багов обнаруживается на покрытии около 30% девайсов - разрешение, мощность, версия о^стнижняя граница для данной апкы. 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](#)
- [Облачные платформы для мобильного тестирования](#)
- [Android phone and tablet market shares](#)
- [The most popular iPhones - 2020](#)
- [Mobile Vendor Market Share Worldwide](#)
- [Яндекс.Радар Мобильные ОС](#)
- <https://mobile-review.com/all/articles/android/genocid-android-flagmanov/>

Middleware



Связующее программное обеспечение (англ. middleware; также переводится как промежуточное программное обеспечение, программное обеспечение среднего слоя, подпрограммное обеспечение, межплатформенное 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 разрвана 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 нашине 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 простоев, 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](#)

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).

Как успешно зарелизить продукт в App Store и Google Play

Специфика работы с платформой Apple

- Review Guidelines. Любые реджекты, бани и блокировки всегда исходят из нарушений этих гайдов, ну или потому что Apple посчитали, что вы их нарушаете. Гайдлайны регулярно обновляются, особенно много обновлений бывает перед и после конференции WWDC;
- Следующий по важности документ — [гайдлайны интерфейсов](#). Здесь представлены рекомендации и лучшие практики по построению красивых, удобных и понятных интерфейсов для всех устройств

Apple. Его необходимо прочитать хотя бы раз любому дизайнеру интерфейсов, а также полезно будет QA-специалистам и разработчикам;

- [App Store Connect](#) - навряд ли часто открывают. В то же время это исчерпывающая инструкция по использованию консоли для управления вашим приложением. Всякий раз, когда у вас возникает вопрос по работе с Connect'ом, пожалуйста, сверяйтесь с документацией;
- Если у вас уже было опубликовано приложение с поддержкой iPad, то выпилить поддержку уже не получится.

Топ-8 причин для реджектов от Apple (в Google Play примерно также):

- App Completeness - не работающие приложения с крашами и багами;
- Inaccurate Metadata - подробно и точно описывайте функционал приложений, прикрепляйте правильные скриншоты;
- Incomplete Info - важно рассказать о вашем продукте все, что стоит знать ревьюерам. Проверяйте актуальность тестовых учеток перед сабмитом. Если вы что-то экспериментируете в билде, расскажите об этом команде ревью;
- Unusual Interface - Apple не зря сделали целый портал про свои интерфейсы. Они хотят чтобы все приложения работали так, как пользователи от них этого ждут. Хороший пример Албания, в которой привычное для нас качание головой вверх-вниз означает нет, а из стороны в сторону - да. Apple это весь мир, не будьте в этом мире Албанией;
- Web Content Aggregators - сайты обернутые в iOS приложение обычно не принимаются;
- Similar App Submission - Apple хотят видеть уникальные приложения и если вы вдруг решили запустить сразу пять одинаковых игр дабы увеличить шансы на взрывной рост, то рискуете остаться без взрывного роста и без приложений;
- Misleading Offers - мы все прошли через рентген для камеры нокии. Не обещать того, чего нет;
- Not Enough Value - Apple хотят видеть уникальные приложения. Это касается не только конкретного разработчика, но и всех вместе. Не нужно делать десятитысячный калькулятор и миллионный фонарик, их достаточно. Также не нужно делать приложение, которым будут пользоваться три человека. Ваше приложение должно нести в себе нужный широкому пользователю, уникальный функционал.

Специфика работы с платформой Google Play Developer:

- Необходимо следить за [предстоящими обновлениями гайдлайнов](#) перед и после конференции Google I/O;
- Все политики вы найдете в [Google Play Developer Policy Center](#), они написаны на русском языке, что заметно удобнее. Во многом политики Google и Apple схожи, однако различия есть и иногда они весьма значительны - обратите на это свое внимание, прочитать и использовать политики только одной платформы - плохая идея;
- Самый полезный ресурс для понимания работы консоли Google Play - это [Google Play Academy](#), там есть ответы на 90% вопросов и все возможные сценарии ее использования как с точки зрения пользователя (разработчика) так и для маркетологов и ASO-специалистов;
- Если у вас есть вопросы по дизайну своего продукта, вы сомневаетесь о проценте прозрачности или даже не знаете правильно ли используете жесты, вы найдете ответ на свой вопрос и/или сможете открыть для себя что-то новое и полезное для ваших приложений в гайдлайнах [Material Design](#).

Соблюдайте правила сторов

В Apple и Google сидят весьма смывленые ребята, основная задача которых не допустить того, чтобы приложения противоречили их политикам:

- не прячьте части функционала приложения от ревьюеров;
- не пытайтесь обойти комиссию платформ и прятать 3rd-party эквайринг;
- не вводите пользователя в заблуждение;
- не запрашивайте лишние доступы, объясняйте зачем нужны доступы, которые запрашиваете.

Источники:

- Как успешно зарелизить продукт в App Store и Google Play

Доп. материал:

- [Policies & publishing on Google Play](#)
 - [Соответствие правилам для страниц приложений в Google Play](#)
 - [Запуск приложения или игры](#)
 - [Pre-launch testing for mobile games: tools and best practices on Google Play](#)
 - [App Store Review Guidelines](#)

Android Debug Bridge (ADB)

<https://developer.android.com/studio/command-line/adb.html>

Тестирование требований к мобильным приложениям

<https://habr.com/ru/company/mobileup/blog/336992/>

Разработка и тестирование мобильных дип линков (mobile deep links)

<https://software-testing.ru/library/testing/mobile-testing/2837-mobile-deep-links>

Тестирование сохраненных поисков

<https://telegra.ph/Testirovaniye-sohranennyh-poiskov-05-20>

Тестирование покупок в приложениях

<https://developer.apple.com/apple-pay/sandbox-testing/>

<https://yandex.ru/search/?text=%D1%82%D0%B5%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B5+%D0%BF%D0%BE%D0%BA%D1%83%D0%BF%D0%BA%D0%B8+%D1%83+apple&lr=38&redircnt=1626946142.1>

для андроид приложений тестовые покупки доступны как на девелоп окружениях, так и на проде

(Не обновлялось) Сети и около них

Некоторая база в одной статье: [Сети для начинающего IT-специалиста. Обязательная база](#)

<https://www.softwaretestinghelp.com/computer-networking-basics/>

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:

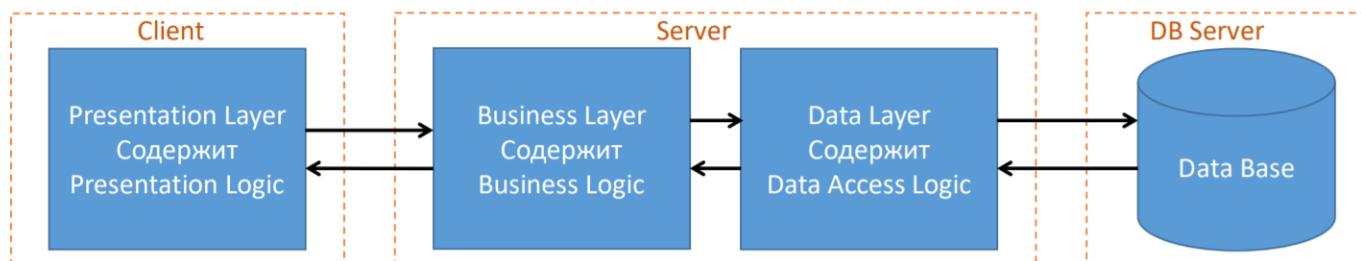
- «Толстый клиент»: на сервере реализованы главным образом функции доступа к базам данных, а основные прикладные вычисления выполняются на стороне клиента.
- «Тонкий клиент»: на сервере выполняется основная часть прикладной обработки данных, а на клиентские рабочие станции передаются уже результаты обработки 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.

Многоуровневая архитектура «клиент-сервер»

3-уровневая архитектура



Уровень представления

Статический или динамически генерированный контент отображаемый через браузер (front-end)

Уровень логики

Уровень подготовки данных для динамически генерируемого контента, уровень сервера приложений (application server). Middleware платформы: JavaEE, ASP.NET, PHP и т. д.

Уровень данных

База данных включающая в себя данный и систему управления над ними или же готовая RDBMS система, предоставляющая доступ к данным и методы управления (back-end)

Доп. материал:

- [О модели взаимодействия клиент-сервер простыми словами. Архитектура «клиент-сервер» с примерами](#)
- [Клиент-серверная архитектура в картинках](#)
- [Тестировщик с нуля / Урок 11. Клиент-серверная архитектура. Веб-сайт, веб-приложение и веб-сервис](#)
- [Клиент-серверная архитектура](#)

Уровни/модель 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. Представление (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.

Доп. материал:

[Модель OSI - 7 уровней за 7 минут](#)

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 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.

HTTP/2 стал первым бинарным протоколом. Если сравнивать его с прошлой версией протокола, то здесь разработчики поменяли методы распределения данных на фрагменты и их отправку от сервера к пользователю и наоборот. Новая версия протокола позволяет серверам доставлять информацию, которую

клиент пока что не запросил. Это было внедрено с той целью, чтобы сервер сразу же отправлял браузеру для отображения документов дополнительные файлы и избавлял его от необходимости анализировать страницу и самостоятельно запрашивать недостающие файлы.

Еще одно отличие http 2.0 от версии 1.1 – мультиплексирование запросов и ответов для решения проблемы блокировки начала строки, присущей HTTP 1.1. Еще в новом протоколе можно сжимать HTTP заголовки и вводить приоритеты для запросов.

Доп. материал:

- [HTTP для тестировщиков](#)
- [Официальная документация](#)
- [Обзор протокола HTTP](#)
- [Отличия http 1.1 и http/2](#)
- [В чем разница между http/1.1 и http/2?](#)
- [Чем опасна ошибка смешанного контента на сайте](#)
- [Что такое смешанное содержимое \(mixed content\) и как его исправить](#)

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:

- [Протокол HTTP](#)
- [Заголовки HTTP](#)

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

Абсолютно любой веб-сервер должен работать, по крайней мере с двумя методами GET и HEAD. Если сервер не смог определить метод, указанный в заголовке запроса клиента, он должен вернуть код статуса 501, если же метод серверу известен, но неприменим к данному ресурсу, будет возвращен код статуса 405. Как в первом, так и во втором случае, сервер должен включить в свой ответ, заголовок Allow со списком методов, которые он поддерживает.

Метод OPTIONS

Данный метод используется для выяснения поддерживаемых веб-сервером возможностей или параметров соединения с конкретным ресурсом. Сервер включает в ответный запрос заголовок Allow, со списком поддерживаемых методов и возможно информацию о поддерживаемых расширениях. Тело запроса клиента, содержит информацию об интересующих его данных, но на данном этапе формат тела и порядок работы с ним, не определен, пока, сервер должен его игнорировать. С ответным запросом сервера, происходит аналогичная ситуация.

Чтобы выяснить возможности сервера, клиент должен указать в запросе URI, символ - "*", то есть данный запрос к серверу выглядит как: OPTIONS * HTTP/1.1. Кроме прочего, данный запрос может быть использован для проверки работоспособности сервера и поддержки им протокола HTTP, версии 1.1. Результаты данного запроса не кэшируются.

Метод GET

Метод GET, применяется для запроса конкретного ресурса. Так-же с помощью GET, может быть инициирован некий процесс, при этом, в теле ответа, включается информация о ходе выполнения инициированного запросом действия.

Параметры для выполнения запроса, передаются в URI запрашиваемого ресурса, после символа "?". Запрос в таком случае выглядит примерно так: GET /some/resource?param1=val1¶m2=val2 HTTP/1.1.

Как установлено в стандарте HTTP, запросы методом GET, являются идемпотентными, то есть, повторная отправка одного и того-же запроса, методом GET, должна приводить к одному и тому-же результату, в случае, если сам ресурс, в промежутках между запросами, изменен не был, что позволяет кэшировать результаты, выдаваемые на запрос методом GET.

Кроме вышесказанного, существуют еще два вида метода GET, это:

условный GET, содержащий заголовки If-Modified-Since, If-Match, If-Range и им подобные,

частичный GET, содержащий заголовок Range с указанием байтового диапазона данных, которые сервер должен отдать. Данный вид запроса используется для докачки и организации многопоточных закачек.

Порядок работы с этими подвидами запроса GET, стандартами определен отдельно.

Метод HEAD

Данный метод, аналогичен методу GET, с той лишь разницей, что сервер не отправляет тело ответа. Метод HEAD, как правило используется для получения метаданных ресурса, проверки URL (есть-ли указанный ресурс на самом деле) и для выяснения факта изменения ресурса с момента последнего обращения к нему.

Заголовки ответа могут быть закэшированы, при несоответствии метаданных и информации в кэше, копия ресурса помечается как устаревшая.

Метод POST

Метод POST, используется для передачи пользовательских данных на сервер, указанному ресурсу. Примером может послужить HTML форма с указанным атрибутом Method="POST", для отправки комментария к статье. После заполнения необходимых полей формы, пользователь жмет кнопку "Отправить" и данные, методом POST, передаются серверному сценарию, который в свою очередь выводит их на странице комментариев. Таким-же образом, с помощью метода POST, можно передавать файлы.

В отличии от GET, метод POST, не является идемпотентным, то есть неоднократное повторение запроса POST, может выдавать разные результаты. В нашем случае, будет появляться новая копия комментария при каждом запросе.

Если в результате запроса методом POST, возвращается код 200 (Ok) или 204 (No Content), в тело ответа сервера, добавляется сообщение о результате выполнения запроса. Например, если был создан ресурс, сервер вернет 201 (Created), указав при этом URI созданного ресурса в заголовке Location.

Ответы сервера, на выполнение метода POST, не кэшируются.

Метод PUT

Используется для загрузки данных запроса на указанный URI. В случае отсутствия ресурса по указанному в заголовке URI, сервер создает его и возвращает код статуса 201 (Created), если ресурс присутствовал и был изменен в результате запроса PUT, выдается код статуса 200 (Ok) или 204 (No Content). Если какой-то из переданных серверу заголовков Content-*, не опознан или не может быть использован в данной ситуации, сервер возвращает статус ошибки 501 (Not Implemented).

Главное различие методов PUT и POST в том, что при методе POST, предполагается, что по указанному URI, будет производиться обработка, передаваемых клиентом данных, а при методе PUT, клиент подразумевает, что загружаемые данные уже соответствуют ресурсу, расположенному по данному URI.

Ответы сервера при методе PUT не кэшируются.

Метод PATCH

Работает аналогично методу PUT, но применяется только к определенному фрагменту ресурса.

Метод DELETE

Удаляет ресурс, расположенный по заданному URI.

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.

Источник: [Методы HTTP](#)

Доп. материал:

- [Parameter Binding](#)
- [HTTP POST with URL query parameters — good idea or not?](#)
- [Идемпотентный метод](#)

Различия методов GET и POST

Get:

запрос инфо от сервера

ограничение кол-ва символов: длина url 2048

передача неважных неконфиденц.данных (напр.язык,фильтры)

нет body

данные передаются в url

Get можно использовать как Post, например передавать параметры в строке
(?параметр1&параметр2&параметр3..)

Но принято использовать конкретные методы запроса под конкретные задачи

Post:

добавление инфо на сервере

ограничение кол-ва символов (разработчик ставит ограничение, так как можно все поломать, в некоторых случаях), ограничено только срывом соединения клиент-сервер

можно передавать конфиденц.данные

есть body

данные передаются в body

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 методом 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.

Коды ответов/состояния сервера (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)

Почему ошибка 404 относится к 4 - клиентской, если по идеи 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.

На какой метод не может вернуться ошибка 501?

The HTTP 501 Not Implemented серверный код ответа на ошибку указывает, что метод запроса не поддерживается сервером и не может быть обработан. Единственными методами, которые необходимы серверам для поддержки (и, следовательно, не должны возвращать этот код), являются GET и HEAD.

TCP/IP

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:

- Канальный уровень (Network Access Layer) или уровень связи, содержащий методы связи для данных, которые остаются в пределах одного сегмента сети;
- Межсетевой уровень или интернет-уровень (Internet Layer), обеспечивающий межсетевое взаимодействие между независимыми сетями;
- Транспортный уровень (Transport Layer), обрабатывающий связь между хостами;
- Прикладной уровень (Application Layer), который обеспечивает обмен данными между процессами для приложений.

Доп. материал:

- [TCP/IP: что это и зачем это тестировщику](#)
- [Модель и стек протоколов TCP/IP - Курс "Компьютерные сети"](#)
- [New IP — следующий этап развития Интернета или ужесточение контроля над пользователями](#)

Endpoint, ресурс, URI, URL, URN

Смысл в том, что сайт, написанный на любом языке, поддерживающем HTTP запросы, не посыпает на сервер никаких PHP/C/Python команд, а общается с 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 эндпоинтов (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=qw3eqwe12e1w&q=URL, где q = URL - это строка запроса, введенная пользователем.

Синтаксис:

[protocol]://www.[domain_name]:[port 80]/[path or exact resource location]?[query]#[fragment]

Источник: [url и uri - в чем различие?](#)

Веб-сервис (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 предоставается 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.

Сокет/веб-сокет (socket/web-socket)

<https://yandex.ru/search/?text=http+vs+web->

socket%D1%81%D0%BE%D0%BA%D0%B5%D1%82+%D0%B4%D0%B2%D1%83%D0%BD%D0%B0%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9%2C+%D0%B0%D1%85%D1%82%D1%82%D0%BF+%D0%BE%D0%B4%D0%BD%D0%BE%D0%BD%D0%BD%D0%BD%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9&lr=38&redircnt=1626867396.1

<https://habr.com/ru/post/498996/>

<https://ru.stackoverflow.com/questions/507746/%D0%92-%D1%87%D0%B5%D0%BC-%D1%80%D0%B0%D0%B7%D0%BD%D0%B8%D1%86%D0%B0-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-socket%D0%BE%D0%BC-%D0%B8websocket%D0%BE%D0%BC>

[https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D0%B5%D1%82_\(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81\)](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D0%B5%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81))

REST и SOAP

- 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 стандартизованные 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.

Как понять используется rest или soap? Одного факта, что запросы в XML не достаточно, ведь в REST не всегда используется JSON. В SOAP [свой XML](#) и если есть последовательность специфичных нод (<soap:Envelope ...> <soap:Header>), то почти наверняка это SOAP. В дополнение к этому SOAP всегда использует метод POST.

Доп. материал:

- [REST v SOAP - A few perspectives](#)
- [Тестировщик с нуля / Урок 17. Тестирование веб-сервисов. SOAP и XML, REST и JSON для тестировщика](#)
- [Базовые знания REST API](#)

gRPC

<https://habr.com/ru/company/otus/blog/545688/>

JSON и 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 & Validation Tool](#)

Источник: [Веб-технологии для разработчиков - HTTP - Коды ответа HTTP - 501 Not Implemented](#)

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.
- [POP3](#) (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.

Доп. материал:

[TCP против UDP или будущее сетевых протоколов](#)

Куки (cookies) и их тестирование

<https://www.softwaretestingmaterial.com/website-cookie-testing/>

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 следящими 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 первым файлом 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](#)
- [Каким будет мир без cookie-файлов?](#)
- [«Осторожно, печеньки!»: советы начинающим тестировщикам в сфере безопасности](#)
- [Cookies уходят. Да здравствует FLoC?](#)

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](#)
- [HTML5 Web Storage - обзор веб-хранилища](#)

Статические и динамические веб-сайты

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.

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 `/etc/asound.conf`, для которого требуются права root, вы можете использовать команду - `sudo nano /etc/asound.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](#)
- Command Line с нуля (Bash, Unix): [часть 1](#), [часть 2](#)
- [How To Run / Execute Command Using SSH](#)

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.

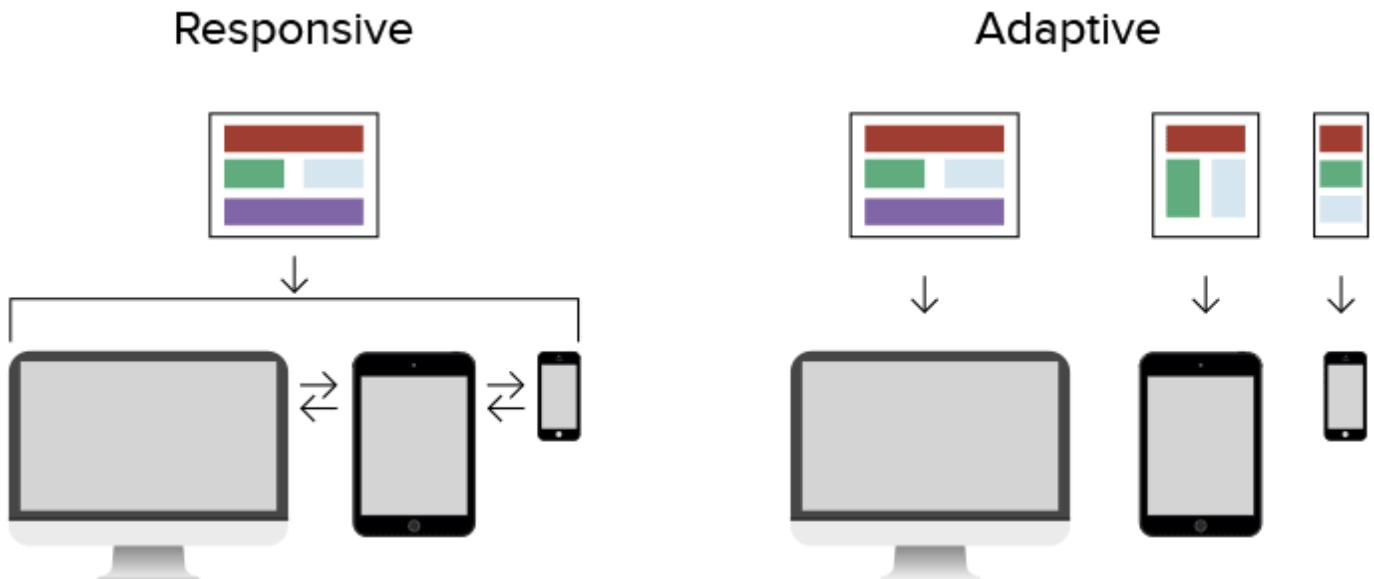
Почему возникают кросс-браузерные ошибки:

- 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.
- Приступить 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.

Доп. материал:

<https://www.mindfulqa.com/cross-browser-testing/>

Адаптивный и отзывчивый веб-дизайн (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.

Какие заголовки (headers, хедеры) важны тестировщику?

Очевидно, смотря что мы тестируем. В основном это заголовки, касающиеся авторизации, кук, кэша и юзер-агент, хотя для того же security тестера они будут иные.

Доп. материал:

- [Нужные HTTP-заголовки](#)
- [Такие разные заголовки! Изучаем HTTP-взаимодействие](#)

Авторизация и аутентификация

| 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?

Идентификация — процедура, в результате выполнения которой для субъекта идентификации выявляется его идентификатор, однозначно определяющий этого субъекта в информационной системе.

Аутентификация — процедура проверки подлинности, например проверка подлинности пользователя путем сравнения введенного им пароля с паролем, сохраненным в базе данных.

Авторизация — предоставление определенному лицу или группе лиц прав на выполнение определенных действий.

<https://www.kaspersky.ru/blog/identification-authentication-authorization-difference/29123/>

<https://itsecforu.ru/2020/09/29/%F0%9F%91%A5-%D1%87%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%aaa-%D0%B0%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F-%D0%B0%D0%B2%D1%82%D0%BE%D1%80%D0%B8%D0%B7/>

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 односторонних 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 пересыпаться 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 разлогинят. 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.

Что у 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](#)
-

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 целенаправленное тестирование самого кэша или требуется наличие кэша по каким-либо причинам). Дело в том, что кэш очевидно искачет 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.

Брокер сообщений (Message broker)
https://en.wikipedia.org/wiki/Message_broker

<https://habr.com/ru/post/466385/>

AJAX

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:

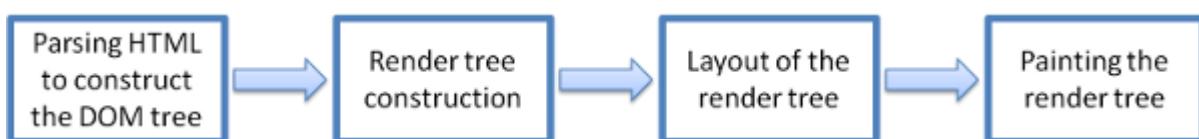
- Пользователь заходит на веб-страницу и нажимает на какой-нибудь ее элемент.
- Браузер формирует и отправляет запрос серверу.
- В ответ сервер генерирует совершенно новую веб-страницу и отправляет ее браузеру и т. д., после чего браузер полностью перезагружает всю страницу.

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 фанты -> 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 котором 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 обслуживаются 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 посыпает 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 рассыпается 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 широковещательный 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) рассыпается 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 посыпает 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.

(Не обновлялось) 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](#)
- [Круглый стол: "Базы данных и где они обитают \(On-premise vs Cloud\)"](#)
- [Тестирование баз данных](#)
- [Что такое База Данных \(БД\)](#)

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](#)
- [Курс Тестирование ПО. Занятие 34. NoSQL база данных. Сравнение SQL и NoSQL - QA START UP](#)

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: $\{X\} \rightarrow \{Y\}$.
- 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, содержащие 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 $R.A \rightarrow\!\!\!> R.B$ 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 в 5НФ. Идея «декомпозиции до конца» выдвигалась до начала исследований в области хронологических данных, но не нашла поддержки. Однако для хронологических баз данных максимальная 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?

Триггер (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

Что вы знаете о требованиях ACID?

Требования ACID на простом языке

Что такое “федеративные таблицы” в Mysql?

<https://ru.stackoverflow.com/questions/573097/%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D1%84%D0%B5%D0%B4%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B5-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D1%8B-%D0%B2-mysql>

Тип памяти BLACKHOLE

http://www.plam.ru/compinet/mysql_rukovodstvo_professional/p45.php

Так как тестировать базы данных?

<https://www.softwaretestinghelp.com/database-testing-process/>

<https://stackoverflow.com/questions/260342/what-best-practices-do-you-use-for-testing-database-queries>

<https://engineering.helpscout.com/testing-code-that-talks-to-the-database-7d15a5391fb9>

<https://towardsdatascience.com/testing-your-database-a0eee0d44115>

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 (вертикальная черта) 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_name> 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_name> 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.

LEFT JOIN

RIGHT JOIN



INNER JOIN

FULL OUTER JOIN



imgflip.com

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 совсем 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](#)

[SQL на котиках: Джоины \(Joins\)](#)

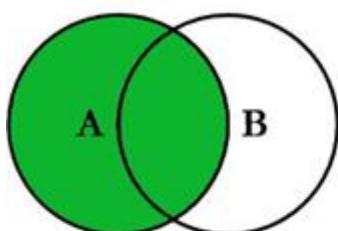
Типы данных в SQL?

- 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

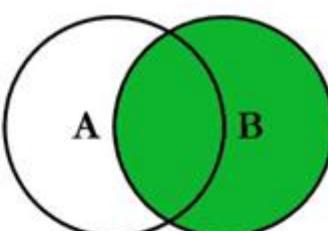
Шпаргалки SQL

| | | | | | | |
|--|---|---|---|---|---|--|
| ТАБЛИЦЫ | ПОЛЯ | ОГРАНИЧЕНИЯ | ВЫБОРКА ДАННЫХ | АГРЕГАТНЫЕ ФУНКЦИИ | СОЕДИНЕНИЯ ТАБЛИЦ | ПРЕДСТАВЛЕНИЯ |
| <pre>CREATE TABLE table1 id INT PRIMARY KEY, tname VARCHAR NOT NULL, tprice INT DEFAULT 0</pre> <p>Создать новую таблицу с тремя колонками</p> <pre>ALTER TABLE table1 RENAME TO table2;</pre> <p>Переименовать таблицу</p> <pre>DROP TABLE table;</pre> <p>Удалить таблицу из базы данных</p> <pre>TRUNCATE TABLE table;</pre> <p>Удалить из таблицы все данные</p> | <pre>ALTER TABLE table ADD column_name;</pre> <p>Добавить новое поле в таблицу</p> <pre>ALTER TABLE table RENAME column1 TO column2;</pre> <p>Перенестивать поле</p> <pre>ALTER TABLE table DROP COLUMN column_name;</pre> <p>Удалить поле из таблицы</p> | <pre>ALTER TABLE table ADD constraint;</pre> <p>Добавить ограничение целостности</p> <pre>ALTER TABLE table DROP constraint;</pre> <p>Удалить ограничение целостности</p> <pre>CREATE TABLE table (column1 INT, column2 INT, column3 VARCHAR, PRIMARY KEY (column1, column2));</pre> <p>Первичный ключ из комбинации полей</p> <pre>CREATE TABLE table (column1 INT PRIMARY KEY, column2 INT, FOREIGN KEY(column2) REFERENCES table2(column1));</pre> <p>Внешний ключ</p> | <pre>SELECT column1, column2 FROM table;</pre> <p>Общее количество записей</p> <pre>SELECT MAX(column1) FROM table;</pre> <p>Максимальное значение поля column1</p> <pre>SELECT MIN(column1) FROM table;</pre> <p>Минимальное значение поля column1</p> <pre>SELECT AVG(column1) FROM table;</pre> <p>Сортировка по возрастанию/убыванию</p> <pre>SELECT * FROM table ORDER BY column1 ASC (DESC);</pre> <p>Сортировка по возрастанию/убыванию</p> <pre>SELECT * FROM table ORDER BY column1 LIMIT n OFFSET offset;</pre> <p>Пропустить offset записей и вывести записи</p> | <pre>SELECT column1, aggregate(column2) FROM table GROUP BY column1;</pre> <p>Группировка с использованием агрегатных функций</p> | <pre>SELECT t1.column1, t2.column2 FROM table1 t1 INNER JOIN table2 t2 ON condition</pre> <p>Внешнее соединение</p> <pre>SELECT t1.column1, t2.column2 FROM table1 t1 LEFT JOIN table2 t2 ON condition</pre> <p>Внешнее левое соединение</p> <pre>SELECT t1.column1, t2.column2 FROM table1 t1 RIGHT JOIN table2 t2 ON condition</pre> <p>Внешнее правое соединение</p> <pre>SELECT t1.column1, t2.column2 FROM table1 t1 FULL OUTER JOIN table2 t2 ON condition</pre> <p>Полное соединение</p> | <pre>CREATE VIEW view_name(column1, column2) AS SELECT column1, column2 FROM table;</pre> <p>Представление, содержащее 2 поля таблицы</p> <pre>CREATE RECURSIVE VIEW view_name AS select statement -- anchor part UNION (ALL) select statement -- recursive part</pre> <p>Рекурсивное представление</p> <pre>CREATE TEMPORARY VIEW view_name AS SELECT column1, column2 FROM table;</pre> <p>Временное представление</p> <pre>DROP VIEW view_name;</pre> <p>Удаление представлений</p> |

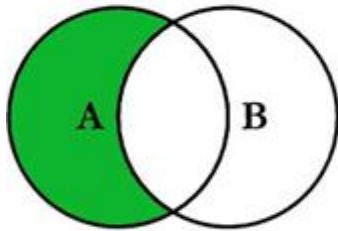
SQL JOINS



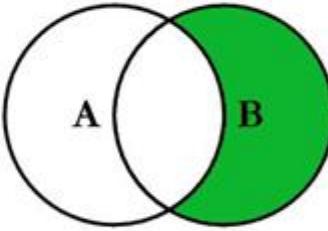
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



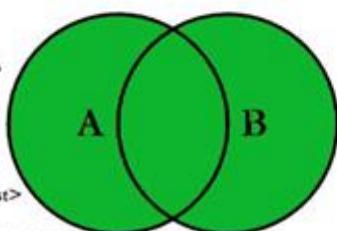
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL.
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

© C.L. Moffett, 2008



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Microsoft® SQL Server™

| String Functions | Date Functions | Create a Stored Procedure |
|--|---|---|
| Exact Numerics bit decimal tinyint money smallint numeric bigint | DATEADD (datepart, number, date) DATEDIFF (datepart, start, end) DATENAME (datepart, date) DATEPART (datepart, date) DAY (date) GETDATE() GETUTCDATE() MONTH (date) YEAR (date) | CREATE PROCEDURE name @variable AS datatype = value AS -- Comments SELECT * FROM table GO |
| Approximate Numerics float real | | |
| Date and Time smalldatetime timestamp datetime | | |
| Strings char text varchar | Dateparts Year yy, yyyy Quarter qq, q Month mm, m Day of Year dy, y Day dd, d Week wk, ww Hour hh Minute mi, n Second ss, s Millisecond ms | Create a Trigger CREATE TRIGGER name ON table FOR DELETE, INSERT, UPDATE AS -- Comments SELECT * FROM table GO |
| Unicode Strings nchar ntext nvarchar | | |
| Binary Strings binary image varbinary | | |
| Miscellaneous cursor table sql_variant xml | | |
| Type Conversion | Mathematical Functions ABS LOG10 ACOS PI ASIN POWER ATAN RADIANS ATN2 RAND CEILING ROUND COS SIGN COT SIN DEGREES SQUARE EXP SQRT FLOOR TAN LOG | Create a View CREATE VIEW name AS -- Comments SELECT * FROM table GO |
| CAST (expression AS datatype) CONVERT (datatype, expression) | | |
| Ranking Functions | | Create an Index CREATE UNIQUE INDEX name ON table (columns) |
| RANK NTILE DENSE_RANK ROW_NUMBER | | |
| Grouping (Aggregate) Functions | | Create a Function CREATE FUNCTION name (@variable datatype(length)) RETURNS datatype(length) AS BEGIN DECLARE @return datatype(length) SELECT @return = CASE @variable WHEN 'a' THEN 'return a' WHEN 'b' THEN 'return b' ELSE 'return c' RETURN @return END |
| AVG MAX BINARY_CHECKSUM MIN CHECKSUM SUM CHECKSUM_AVG STDEV COUNT STDEVP COUNT_BIG VAR GROUPING VARP | String Functions ASCII REPLICATE CHAR REVERSE CHARINDEX RIGHT DIFFERENCE RTRIM LEFT SOUNDEX LEN SPACE LOWER STR LTRIM STUFF NCHAR SUBSTRING PATINDEX UNICODE REPLACE UPPER QUOTENAME | |
| Table Functions | | |
| ALTER DROP CREATE TRUNCATE | | |

Available free from
AddedBytes.com

----- (Не обновлялось) 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
 - Сенсорные 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):

① Порядок перевода денег

From

A hand-drawn diagram of a card payment form. It consists of several fields: a large top field labeled ①, a middle row with two fields labeled ② and ③, a small field labeled ④ below them, and a bottom row with two fields labeled ⑤ and ⑥.

To

A hand-drawn diagram of a card payment form, similar in structure to the 'From' form, but with a single large top field labeled ①.

① Номер карты

Amount to

② Cardholder

Число

③ Valid Thru

Send

④ CVC

Кнопка

Или вот еще с просторов, реальное тестовое задание. Можно их много найти, если поискать.

| | |
|---------------------------------------|---|
| Account | <input type="text" value="9116289 (USD)"/> |
| Current Balance | <input type="text" value="3000"/> USD |
| Amount | <input type="text" value="250"/> USD |
| Operation: | <input checked="" type="radio"/> Deposit <input type="radio"/> Withdraw |
| Comment | <input type="text"/> |
| Attachment | <input type="text" value="C:\Storage\bank_account_info.pdf"/> <input type="button" value="BROWSE"/> |
| <input type="button" value="SUBMIT"/> | |

On a website there is a form which allows user to deposit or withdraw money. Current balance and shows available money for the selected Account. User must specify amount in the edit field, select operation, attach file with bank account information and submit the request. Minimal allowed amount for Deposit is 250 USD. User also may leave a comment to this request. As soon as request is submitted, the Current Balance must change to the specified amount. If something went wrong an error message must be displayed to the user and Current Balance must not change.

Доп. материал:

- [Пароли, их тестирование и использование](#)
- [Принципы и тестовые сценарии для тестирования паролей](#)
- [Как Тестируать? Форма Входа](#)
- [Acceptable email address syntax according to RFC](#)
- учитывать копипаст + валидация мейла отдельно! по факту З разных поля

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.

Доп. материал:

Классы эквивалентности пример

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. Сматрит 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-кашки.

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».

Доп. материал:

[Мы нашли все самые крутые логические задачи](#)

[Еще примеры](#)

- Дан веб-сайт, на котором есть каталог и реализована регистрация. На каких уровнях и что будете тестировать, конкретно по пунктам?
- Данна 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
- Написать тест-кейсы/тест-план для тестирования будильника/лифта/весов/светофора/кофейного автомата/...
- Как изменится тест-план для кофейного автомата, если оплата происходит только со смартфона через оператора сотовой связи (SMS)?
- Протестировать поиск адресов
- Протестировать установку приложения при недостаточном количестве места на телефоне
- Протестировать требование: приложение не должно быть доступно для скачивания пользователям некоторых стран
- You plug the charger into your cell phone and see that the battery isn't charging. Please list out where are all the potential breaking points and what would you check to ensure that you found the breaking point
- You are the QA Engineer at Uber and you just heard that passengers are no longer receiving text messages. What are your next steps in troubleshooting?
- Есть проект к которому вас подключают. Срок его сдачи - через 2 недели. Есть команда которая его разрабатывала и РМ проекта. Есть коммуникация с клиентом. Как вы построите процесс работы по этому проекту чтобы сдать проект в срок и на чём вы будете основывать идею что проект "Готов"?
- 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 создаешь свою ветку;
- [Тут можно запросить платное тестовое с проверкой](#) (просто наткнулся, так что хз что там)
- [Сайт для тренировки тестирования](#)
- [Еще один](#)
- [И ешё](#)
- [И ещё один](#)
- [И ешё](#)
- [Несколько примеров задач с решениями](#)

- [Scandiweb Entry Level Test](#)
- [Тестовое задание на позицию специалист по тестированию \(QA специалист\) в СПб ИАЦ](#)
- [Как тестировать веб сайт? Практика в примерах](#)
- [Как испортить приложение и оттолкнуть пользователей. Разбираем и исправляем ошибки в приложении Роскомнадзора](#)
- [Learn new technologies using real environments right in your browser](#)
- [Записи прямых эфиров с разбором тестовых задания для тестировщика](#)
- [Тестовое задание для специалиста по тестированию + \[Инструкция\] Заказ визового приглашения](#)
- [Вопросы и задания на собеседовании для ручного тестировщика](#)
- [Shop — это бесплатное приложения для тестирования](#)
- [Тестирование программы, которая определяет тип треугольника по трем его сторонам](#)
- [Еще один тренировочный сайт](#)
- [Hacking challenge site](#)
- Тестовое задание: написать кейсы для [нового метода API](#)
- [Тестовые площадки для тренировок настоящих ниндзя](#)
- <https://cantunsee.space/>

Vladislav Eremeev, [02.04.21 10:01]

[Forwarded from Pavel Panov]

На собеседовании интересный вопрос задали.

Делюсь.

Есть некий обучающий портал с видео.

Видео можно смотреть бесплатно до некоторой величины.

При просмотре видео на 80% считается, что просмотрщик согласен заплатить (необходимо пометить видео как просмотренное, добавить в некий список, не суть)

Vladislav Eremeev, [02.04.21 10:01]

[Forwarded from Pavel Panov]

Необходимо накидать тестов, как проверить просмотр 80% контента.

Vladislav Eremeev, [02.04.21 10:01]

[Forwarded from Pavel Panov]

На собеседовании дали ютуб, какое видео (от фонаря), я еще открыл ДЕв тулс

Vladislav Eremeev, [02.04.21 10:01]

[Forwarded from Pavel Panov]

подождем еще вариантов.

(но для вас - встречный вариант - а если просмотреть 10% (20... 30...) - и начать сначала?

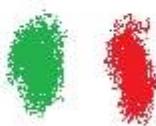
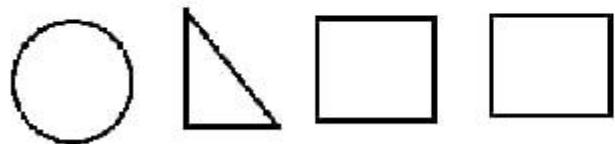
Спрашивали написать конкретный тест кейс проверок. Вот прям дали ютуб и сидят ждут ответ, в zoom. Причем не ограничивают во времени в разумных пределах)

ЗАДАНИЕ 3

Есть фигуры: 1 треугольник, 2 одинаковых квадрата и 1 круг.

Есть краски: красная, зеленая.

Перечислите все варианты разной раскраски данной группы фигур.



UTM Tag Builder

Generated URL

[Copy URL](#)

URL Address

https://

UTM Source

The referrer: (e.g. google, newsletter)

UTM Medium

Marketing medium: (e.g. cpc, banner, email)

UTM Campaign

Product, promo code, or slogan: (e.g. spring_sale)

UTM Term

Identify the paid keywords

Campaign content

Use to differentiate ads

Маркетологи используют специальные параметры в URL, чтобы лучше отслеживать кампании. Эти параметры называются параметрами UTM (модуль отслеживания Urchin). У нас есть страница, которая принимает URL-адрес, индивидуальные параметры UTM и компилирует конечный URL-адрес, которым люди могут поделиться. Сотни людей приходят сюда каждую неделю, поэтому необходимо, чтобы этот сайт работал без проблем - компиляция URL, а также поддержка различных браузеров и устройств. Ваша задача создать «тест-кейсы» для этого сайта

Описание задачи

В мобильном Android/iOS приложении на стартовом экране есть две кнопки управляющие рекламой:

"Load" - стартует загрузку полноэкранной рекламы.

"Show" - показывает ранее загруженную рекламу на новом экране.

Процесс загрузки рекламы:

Приложение делает GET запрос на сервер и в ответе ожидает JSON с полем "adm" в котором лежит HTML разметка рекламного креатива.

Загрузка бывает успешной, если ответ валидный и не успешной.

После успешной загрузки рекламу можно показать нажав кнопку "Show". После старта показа рекламы ее можно закрыть или кликнуть на нее. В Android должен быть осуществлен переход в браузер, в iOS может открыться SKStoreProductViewController либо приложение может открыть Safari. После закрытия браузера (возвращение в приложение) пользователь должен вернуться на экран рекламы.

Чтобы вернуться на стартовый экран рекламу нужно закрыть, для этого надо нажать на "крестик". В Android рекламу можно закрыть по нажатию на системную кнопку "назад".

При каждом событии (окончание загрузки рекламы/старт показа/клик по рекламе/закрытие рекламы) должен отправиться GET запрос на сервер (трекинг события) и отобразиться всплывающее окно в приложении (callback в приложении).

Задание

1. Описать все возможные тестовые сценарии для данного приложения с учетом специфики Android и iOS.
2. Какое тестовое окружение потребуется?
3. Какие тестовые сценарии критические (требуют незамедлительного исправления), а какие нет (можно будет исправить в следующей версии)
4. Оцените время требуемое для проверки всех тестов.

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](#)

Тестирование чашки для кофе

Не уверен, что это еще популярно спрашивать, но на всякий случай пара примеров с просторов.

Сначала – позитивное тестирование.

Функции чашки:

- вмешать напитки,
- переносить напитки,
- возможность пить из нее,
- возможность греть напитки в микроволновке.

Проверим сначала «вмешать»:

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 идеи, 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.

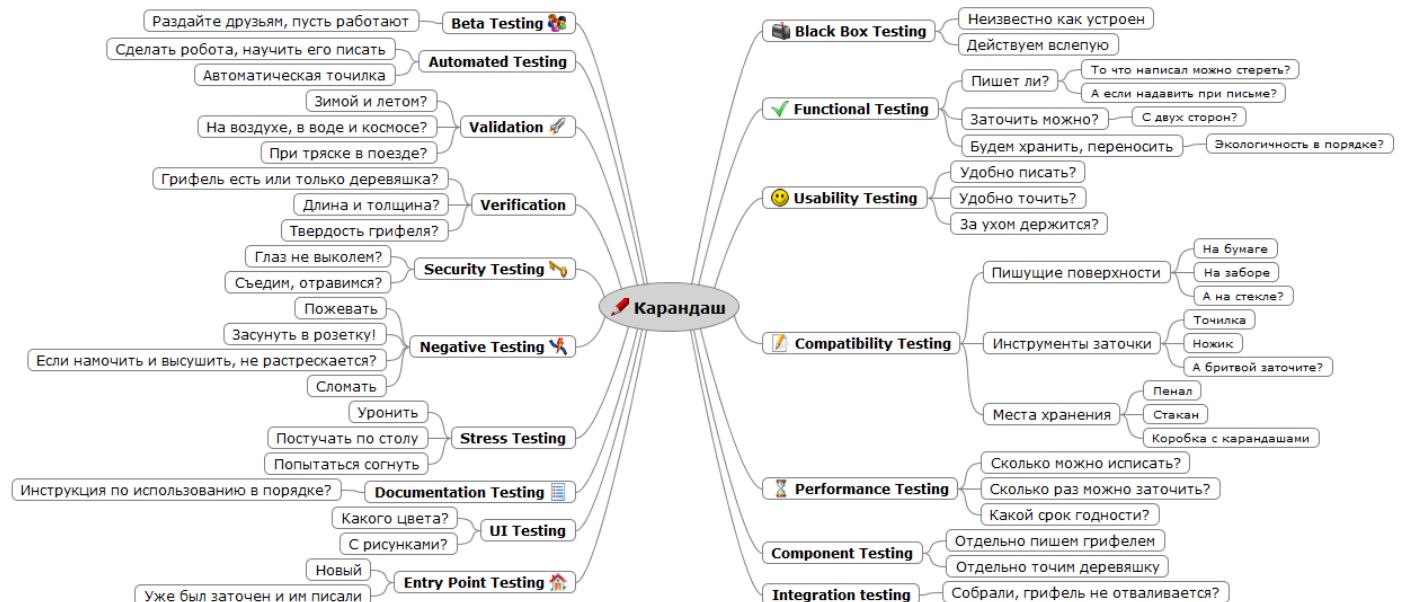
Еще – «тестирование безопасности»:

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 сервисов.

Тестирование карандаша



[Вот тебе комп и работающий сайт. Сделай мне 401-ю ошибку](#)

Задача на умение пользоваться инструментами, позволяющими подменять трафик. По обстоятельствам.

[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; Кросбраузерное 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 стоятся 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 Р, М и О 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.

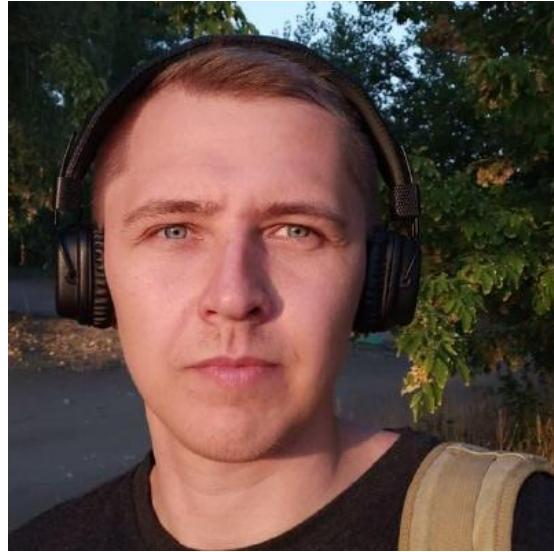
Далее полученные значения Р, М и О подставляются в формулу: $(O + 4 M + P) / 6$

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")

----- Контакты -----



Автор: Еремеев Владислав

[LinkedIn](#) - [Telegram](#)

Поскольку проект open-source, каждый может внести посильный вклад в его (и всех читающих) развитие. Смело можете писать, особенно если:

- нашли ошибку;
- прочитали тему, ничего не поняли и нужно переписать понятнее;
- не нашли то, чего искали;
- в другом месте нашли что-то полезное, чего тут нет и хотите, чтобы это увидели и другие;

Контрибьютинг горячо приветствуется, особенно если:

- имеете опыт тестирования определенного домена или направления и есть что рассказать;
- любые экспертные правки и что-то полезное, что можно один раз написать и потом этим кидаться в своих джунов.

Также приветствуются аргументированная критика и предложения.

Любое использование материалов проекта, в т.ч. коммерческое, допускается при указании авторства/первоисточника согласно лицензии GPL-3.0 (пламенный привет компаниям, которые вырезали контакты и закинули материал в корпоративную базу знаний как свой).

Если ресурс чем-то помог, поддержите проект материально скромным донатом:

в РФ +79044121375 на баланс;

в СНГ https://boosty.to/qa_bible

Удачи в карьере!