Тестирование ПО – это процесс проверки соответствия между реальным и ожидаемым поведением программы. Появляется баг и мы заводим баг-репорт. Процесс исследование программы или её документации с какой-то целью. Цели могут быть разные.

Качество – это совокупность характеристик, которые определяют определённую готовность продукта. ISO – это международные стандарты качества.

Цель – подтвердить, что продукт готов.

**Модели разработки SDLC**

**Гибкая** – AJILE – SCRUM (принимать участие на всех этапах)

→ тестирование требований

→ писать тестовые сценарии, кейсы, готовить артефакты и чек-листы

→ даль резолюцию (отчёт)

**Каскадная** – чёткое следование процесса со сроками. Минус в том, что косяки на ранних этапах трудно исправить.

→ Проектирование (дизайн, модели, макеты, документация, требования)

→ Кодирование

→ Тестирование (тестируем только на этом этапе)

→ Эксплуатация

**Инкрементальная** (редкая) -

**V-образная** (редкая) – тестирование на каждом этапе разработки

**Спиральная** (редкая)

**Хаоса** – самая популярная

**Объекты тестирования (что тестируем)**

**Код** – область модульного тестирования (редко делают тестировщики). Обычно сам разработчик пишет модульные тесты (автоматизируется)

**Software**

→ WEB

→ мобильные приложения

→ десктоп

→ интернет вещей (умный дом, умные часы)

→ чат-боты

→ игры

→ другое (CRM, корпоративная шина)

**Hardware** – проверка работы программы на железе (не автоматизируется)

**Прототипы** – прототип продукта перед релизом нужно протестировать. Но он ещё сырой и может сильно измениться.

**Документация** – тестирование требований, что они нормальные и адекватные, понятные. Сюда относится также и описание продукта, как он работает (не автоматизируется)

**Тестирование требований**

**Требования** – это ожидаемое поведение приложения

**Прямые** – явно указаны и явно прописаны

**Косвенные - т**о, что при нажатии на ссылку она должна открываться не нужно, или при нажатии на крестик окно должно закрыться, это и так понятно.

**Типы требований**

1. **Нефункциональные** – производительность, интерфейсы работы. Передача данных может проходить по протоколу REST или SOAP.Или платформа – для айфонов или андройдов.

2. **Функциональные** (уровни требований)

→ **бизнес требования** – необходимо увеличить скорость обработки заявки

→ **требования пользователя –** при первом входе должно отобразиться согласие на использование cookies.

→ **функциональные требования** – работа с функционалом

**Источники требований**

Откуда брать требования, если их нет?

→ заказчик

→ мозговой штурм

→ документы

→ фокус группа

→ прототип

→ нормы

→ моделирование

→ наблюдение

→ описание

→ лучшие практики

→ анкетирование

→ конкуренты

**Характеристики требований**

Полнота

Непротиворечивость

Корректность

Недвусмысленность

Проверяемость

Атомарность

Выполнимость

Обязательность

**Виды тестирования**

**Функциональное**

→ **Функциональное тестирование** – это поведение продукта, что он умеет делать

→ **GUI тестирование элемента интерфейса** – если графический элемент имеет своё поведение (кнопка меняет свой цвет)

→ **Безопасность** - если функционал продукта отвечает за безопасность (антивирус)

**Нефункциональное**

→ **Безопасность** – защищённость данных, безопасность продукта для детей

→ **UI** – единый стиль интерфейса, соответствие требованиям дизайна

→ **Юзабилити** – удобство использования, как быстро пользователь купит продукт, то есть достигнет целей. Удобно ли попадать по кнопке. Большие ли списки (никто не любит большие списки).

→ **Локализация** – перевести тексты, региональные и правовые особенности разных стран. Где-то километры, где-то мили.

→ **Интернационализация** – возможность без проблем локализовать продукт. Заранее продумать так, чтобы можно было без проблем локализовать.

→ **Конфигурационное –** учитывание конфигураций разных браузеров

→ **Тестирование совместимости** – кроссбраузерность

→ **Инсталляционное** – установка, обновление, удаление

→ **Производительность**

→ **нагрузочное** – обеспечить 10 000 обращений в час

→ **стрессовое** – что будет если будет 100 000 в час

→ **стабильность** – работа при нагрузке за определённое время

→ **масштабируемость** – что нужно сделать для расширения продукта

→ **объёмы** – выдерживание тяжёлых объёмов информации (фильмы)

→ **Приоритетность** – что будет делать продукт в 1 очередь при перенагрузке.

→ **Помехоустройчивость** – продукт может упасть из-за потери связи, перенапряжения, как быстро он восстановится. Что будет если во время приложения поступит звонок.

→ **Документация**

**Функциональное тестирование**

Нужно проверить все функции программы и сравнить фактический результат с ожидаемым. Это те действия, которые могут потенциально выполнить пользователи.

**Требования**

**Входные данные** – что будет, если вводить в поле количества не цифры, а, например, буквы. Нужно пробовать негативные сценарии. Проверять поведение при ожидаемых и неожидаемых действиях.

**Выходные данные** – это ожидаемый результат выполнения программы, соответствует ли он фактическому.

**Прохождение сценариев** – делать то, что необходимо сделать

**Сравнение результатов** – ожидаемых с фактическими.

Результатом функциональных тестов является то какой функционал реализован и правильно ли он работает.

**Нефункциональное тестирование - локализация**

**Текст** – перевод (копиратеры)

**Функционал** – даты, время, валюты, часовой пояс, начало недели, формат бумаги, система мер, особенности имён, сортировка.

**Региональные особенности** – контроль символики и цветов, правовые особенности, толкование текста (нанять юристов)

**Контент** – аудио, текст, видео, документация

**Тестирование связанное с изменениями**

**Смок тест (дымовое тестирование)** – это вообще работает? Проверить критически важный функционал. Нужно знать какой функционал критически важный. Если это интернет магазин, то нужно пройти путь покупателя от начала и до конца. (самый высокоприоритетный тест)

**Регрессионное тестирование** – Проверить весь старый функционал после новых доработок.

**Тестирование сборки (версии)** – тестирование всего продукта

**Санитарное тестирование** – тестирование определённого участка. Проверка в каком-то важном критическом месте.

**Ретест** – протестировать исправленные баги.

**Уровни тестирования**

Определяем не порядок тестирование, а способы тестирования

**Модульное тестирование (unit test)** – 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. **Выполняется после Unit теста** тестировщиком. 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. Проверка в(на) соответствии с **test requirements**, **use case**

**Интеграционное тестирование -** несколько модулей программы тестируются вместе - какие-то модули на сервере, какие-то в базе данных и они все должны между собой взаимодействовать. Может быть на уровне взаимодействия разных программ, а не только модулей (такие тесты можно автоматизировать)

**Уровни интеграционного тестирования**:

**1. Компонентный интеграционный уровень** (CIT - [Component Integration testing](https://www.testing.guru/what-is-component-integration-testing/" \t "_blank)): Проверяется взаимодействие между компонентами одной системы после проведения компонентного тестирования.

**2. Системный интеграционный уровень** (SIT - [System Integration testing](https://www.softwaretestinghelp.com/system-integration-testing/" \t "_blank)): - это полное тестирование всей системы, состоящей из множества подсистем.

**Подходы к интеграционному тестированию**:

**1. 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.

**2. 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.

**→ Восходящий подход (Bottom-Up Approach)**: В восходящей стратегии каждый модуль на более низких уровнях последовательно тестируется с более высокоуровневыми модулями, пока не будут протестированы все модули. Для симуляции недостающих модулей используются драйвера для тестирования.

**3.** [**Гибридный/сэндвич-подход**](https://www.ques10.com/p/38806/describe-bi-directionalsandwitch-integration-testi/) **(Sandwich/Hybrid/Bi-Directional Approach)**: Представляет собой комбинацию восходящего и нисходящего подходов. Здесь целью является средний слой, в то время как драйверы заменяют верхний слой, а заглушки нижний пока компоненты этих слоев не будут разработаны;

**Системное тестирование** – вся программа тестируется полностью. 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;

**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.

**Приемочное тестирование (acceptance testing)**- уровень тестирования, целью которого является определение того, принимать ли систему

**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](https://en.wikipedia.org/wiki/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](https://www.softwaretestinghelp.com/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](https://www.softwaretestinghelp.com/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;

**Методы тестирования**

Определяет насколько глубоко погрузиться: можем ли мы получить доступ к базе данных, архитектуре, потоку хранения информации, доступ к коду.

**Чёрный ящик** – нет никакого доступа, только такое как у пользователя.

**Белый ящик** – доступ есть везде

**Серый ящик** – нет доступа к коду, но есть доступ в какие-то другие области, к которым нет у пользователя.

**Направления тестирования**

**Статическое** – не запускаем код – это работа над предотвращением дефектов.

→ Тестирование требований

→ Проверка архитектуры проекта

→ Проверка макета

→ Проверка базы данных на наличие полей

→ Стратегия тестирование

→ Подготовка тестовых данных

**Динамическое** – запускаем код – это поиск дефектов и устранение. Или просто использование программы и проверка функционала.

**Степень удалённости**

**Альфа тестирование** – продукт +- готов, его уже можно тестировать, находить дефекты и баги, пробовать использовать.

**Бета тестирование** – проверка продукта на реальной аудитории, выбирается ограниченный круг лиц, пробующих продукт. Продукт практически готов, проверяется нагрузка и некоторые незамеченные недочёты.

**По ожидаемому поведению**

**Позитивное** – проверка идеального предполагаемого поведение программы.

**Негативное** – проверка непредполагаемого поведения. Нужно оставить отметку, что тест пройден, если негативный сценарий предполагался и отработал как задумано.

**Деструктивное тестирование (destructive testing)**- одна из форм негативного тестирования с целью нарушить работоспособность приложения и обнаружить точку отказа.

*Пример: нагрузка приложения выше его предела, чтобы оно перестало работать*

**В реальном тестировании следует использовать несколько правил:**

1. Любое тестирование необходимо начинать с позитивных проверок. Наша первоочередная задача убедиться, что приложение работает в стандартных условиях и готово для использования.

2. Нельзя объединять позитивные и негативные тесты, так как это затрудняет локализацию дефектов.

**Этапы тестирования (Smoke, critical, extended)**

**Smoke test** направлен на проверку самой главной, самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения (или иного объекта, подвергаемого дымовому тестированию). Проводится на каждой новой промежуточной версии продукта (build). Иногда этот процесс называют **сертификацией билда**. Если **Smoke** не проходит, то остальное тестирование останавливается и ждет доработки кода.

**Сritical path test** направлен на исследование функциональности, используемой типичными пользователями в типичной повседневной деятельности. Например, можно написать всего 1 **test-case** и включить в него все шаги: регистрация → товар в корзину → оформление заказа. Обычно здесь тестируются новые или измененные функции, которые попали в промежуточную версию продукта (build) с точки зрения использования конечного пользователя.

**Extended test**направлено на исследование всей заявленной в требованиях функциональности — даже той, которая низко проранжирована по степени важности. Ещё одним направлением исследования в рамках данного тестирования являются нетипичные, маловероятные, экзотические случаи и сценарии использования функций и свойств приложения, затронутых на предыдущих уровнях.

**Sanity** – это то же самое, что и **Smoke**,только уже на стабильных билдах близких к релизу.

**Тестирование изменений (regression, retest)**

**Retest**

Подтверждает факт того, что дефект был исправлен. Он не требует оценку влияния изменений, хотя хороший тестировщик всегда потратит несколько минут на общую оценку работы приложения после такого фикса.

**Регрессионное тестирование**

Проводится после любого изменения (новая функциональность, исправление дефекта, удаление функциональности), чтобы убедиться в том, что приложение работает стабильно со всем остальным (своим окружением). Оно не включает в себя **ретест**, а проводится уже после него.

Нужно понимать, что на практике регресс не проводится каждый раз, когда в системе появляется изменение, а тестирует скорее "накапливающийся" эффект.

Например, две недели мы разрабатывали новые функциональности и тестировали только их. В конце этого периода (итерации) у нас запланирован релиз. И именно перед ним мы проведем одну общую регрессию для всего приложения

Регрессионное тестирование является одной из самых главных активностей тестировщика и проводится перед каждым релизом (выпуском продукта для конечного пользователя).

Если релиз запланирован на каждую неделю, то регрессия проводится с такой же периодичностью, поэтому **тесты для регрессии**, как и **дымные** **тесты** очень часто **автоматизируют** для экономии времени.

Обычно для проведения регрессии выделяют несколько дней, код проекта "замораживают"(code freeze) и в него нельзя вносить изменения, кроме исправлений критичных дефектов.

**В регрессионный набор тестов, из уже существующих тестов отбираются на основании следующих принципов (Какие тесты отбираются для регрессионного тестирования?):**

**1.** Тесты, проверяющие части приложения, в которые вносились изменения

Например, если дефект или новая функция локализованы в модуле регистрации, необходимо убедиться, что остальные компоненты этого модуля работоспособны

**2.** Тесты с высоким приоритетом

Мы не можем рисковать работоспособностью самых важных функций приложения, поэтому необходимо дополнительно проверять и их

**3.** Тесты, которые проверяют модули с наибольшей концентрацией дефектов

Согласно принципу скопления дефектов необходимо убедиться, что в уязвимой части приложения нет дополнительных проблем.

**Документация**

**Тест план (Test Plan)** — это документ, который описывает весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков.  
  
**Тест план должен отвечать на следующие вопросы:**

* Что необходимо протестировать?
* Как будет проводиться тестирование?
* Когда будет проводиться тестирование?
* Критерии начала тестирования.
* Критерии окончания тестирования.

**Основные пункты тест плана:**

1. Идентификатор тест плана (Test plan identifier);
2. Введение (Introduction);
3. Объект тестирования (Test items);
4. Функции, которые будут протестированы (Features to be tested;)
5. Функции, которые не будут протестированы (Features not to be tested);
6. Тестовые подходы (Approach);
7. Критерии прохождения тестирования (Item pass/fail criteria);
8. Критерии приостановления и возобновления тестирования (Suspension criteria and resumption requirements);
9. Результаты тестирования (Test deliverables);
10. Задачи тестирования (Testing tasks);
11. Ресурсы системы (Environmental needs);
12. Обязанности (Responsibilities);
13. Роли и ответственность (Staffing and training needs);
14. Расписание (Schedule);
15. Оценка рисков (Risks and contingencies);
16. Согласования (Approvals).

**Артефакты**

**Артефакт** – это любой документ, который вы заводите

**Чек-лист** – это список проверок без указания шагов – это перечень проверок. Он не подразумевает условий, шагов и результата. Это идея проверки

**Тест-кейс** – проверка, состоящая из каких-то шагов, условий и ожидаемого результата

**Чек-лист**

Мы лишь описываем идею, у каждого тестировщика это может быть по-своему.

**Атрибуты чек-листа**: (можно какие-то добавлять, какие-то убрать)

→ **номер**

→ **описание проверки**

→ **статус** (passed, failed, skipped, no run, blocked)

→ **комментарий**

**Примеры чек-листа для калькулятора:**

|  |  |  |
| --- | --- | --- |
| номер | Название | Статус |
|  | **Режимы** |  |
| 1 | Изменение режимов | passed |
| 2 | Есть 5 обычных режимов калькулятора | failed |
|  | **Режим** **обычный** |  |
| 3 | Сложение |  |
|  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер | Модуль | Проверка | Статус | Комментарий |
| 1 | Числовая панель | Сложить однозначные цифры |  |  |
| 2 | Числовая панель | Сложить двузначные цифры |  |  |
| 3 | Числовая панель | Сложить трёхзначные цифры |  |  |
| 4 | Числовая панель | Сложить десятичные цифры |  |  |
| 5 | Числовая панель |  |  |  |

**Плюсы –** гибкость, простота создания, простота визуализации, расширение текстового покрытия

**Минусы** – нет стандарта, неопределённых текстовых данных, неэффективен для джунов, высокая вероятность разночтения

**Тест-кейсы**

**Тест кейс** – это артефакт для тестирования, предназначенный для того, чтобы вести проверку идеи с подробным описанием необходимых шагов.

Сначала создаются чек-листы, а уже потом на его основе тест-кейсы с подробными шагами.

**Атрибуты тест-кейса:**

**→ номер кейса**

**→ название кейса**

**→ предусловия** – описать состояние системы перед проверкой

**→ шаги проверки** – должен быть простой и однозначный, не надо писать тестовые данные (не должны быть указаны логин и пароль в шаге авторизации)

**→ ожидаемый результат** – возле каждого шага проверки

→ статус

→ модуль

→ приоритет

→ связанное требование

**Правила**

**Независимость** – избегать ситуаций, когда для прохождения кейса 2, нужно сначала пройти кейс 1

**Однозначность** – избегать формулировок «ввести логин», лучше написать «ввести логин кириллицей»

**Полнота** – в тест-кейсе должно хватать данных, чтобы ни у кого не возникали вопросы

**Обезличенность** – нельзя писать «нажми на кнопку», нужно писать «зайти на сайт»

**Упрощай** – не стоит перебарщивать, не писать «нажмите на красную кнопку в правом верхнем углу экрана»

**1 тест-кейс = 1 цель** – один кейс проверяет одну какую-то идею

**Высокоуровневые тест-кейсы** – описывают поведение типо чек-листов, напоминают больше идею

**Низкоуровневые тест-кейсы** – каждый шаг подробно расписаны. Они лучше проработаны, но требуют много времени. Обычно используются они.

**Пример тест-кейса:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер | Название модуля | предусловия | проверка | шаги | Ожид. Рез-т |
| 1 | Числовая  панель | Приложение калькулятор запущено | Сложение однозначных цифр | 1. Нажать при помощи ЛКМ цифру 1 | В поле ввода появится ц. 1 |
| 2. Нажать ЛКМ на + |  |
| 3. Нажать ЛКМ на цифру 3 | В поле ввода появится ц. 3 |
| 4. Нажать ЛКМ на равно | В поле ввода появится р-т. 4 |
| 2 | Числовая  панель |  |  |  |  |
|  |  |  |  |  |  |

**Техники тест-дизайна**

Нужно делать не все проверки, а только самые необходимые для экономии времени. Они должны покрывать б**о**льшую часть приложения и основные места. Техники применяются в конкретных случаях и ситуация, поэтому не мы их выбираем.

**Классы эквивалентности** – не нужно проверять каждое число от 1 до 100 при проверке возраста, можно числа разбить на 2 класса – до 18 лет и после 18 лет и проверить 2 раза.

**Граничные значения** – берём для проверки границу 18, а также +-1 (16 и 17)

**Таблица принятия решений**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Условие** | **Значения 1** | **Значения 2** | **Значения 3** | **Значения 4** |
| Ввод корректных данных в поле Email | + | - | + | - |
| Ввод корректных данных в поле Password | + | - | - | + |
| Ввод некорректных данных в поле Email | - | + | - | + |
| Ввод некорректных данных в поле Password | - | + | + | - |
| **Действия** |  |  |  |  |
| Регистрация прошла успешно | + | - | - | - |
| Выдаётся ошибка «Введены неверные данные» | - | + | + | + |

**Сценарная техника** – разработка теста на основании действий пользователя

**ADHOC** – тестировать сценарий поведения пользователя рандомно как хочется, интуитивно. Пользоваться программой с целью какой-то, например совершить покупку, а не ч целью что-то поломать

**Предугадывание ошибок** – зная примерно, где могут быть ошибки проверить их сразу

**Попарное тестирование** – чтобы не проверять гигантское количество возможных вариантов развития событий, которые могут достигать 10 в 10 степени, применяется эта техника. Например:

|  |  |  |
| --- | --- | --- |
| Мужчина | До 25 лет | Дети есть |
| Женщина | 25-60 лет | Нет детей |
|  | После 60 лет |  |

Для проверки всех комбинаций потребуется 12 проверок. Чтобы это сократить нужно предположить, что мы будем проверять только часть комбинаций, а именно 1 и 2 столбцы 1 строки, затем 2 и 3 столбцы 2 строки. Исследования подтверждают, что таким образом находятся от 95% ошибок. Есть программа **Pairwise online** которая это делает (**Slothman**).

**Доменный анализ** –

**Причина / следствие** – причина клик – следствие вызов окна выбора

**Состояние и переходы** –

**Дефект и баг**

**Баг-репорт** – это документ, который описывает последовательность действий, которые привели к некорректной работе объекта тестирования, а также причины и ожидаемый результат. Другие члены команды могут делать баг-репорты.

**Баг-трекеры**

→ jira

→ Redmine

→ Mantis

→ Excel

**Локализация дефекта** – выявить причины возникновение дефекта, кто виноват. Нужно описать баг с технической стороны более подробно.

→ Проверить на разных версиях ПО

→ Проанализировать возможность влияния бага на другие области.

**Атрибуты баг-репортов**:

**→ Идентификатор**

**→ Тема**

**→ Шаги воспроизведения** – как дойти до проблемы

**→ Актуальный результат**

**→ Ожидаемый результат**

**→ Окружение** –браузер, ОС, устройство

**→ Критичность**

**→ Приоритет**

→ Комментарии

→ Скриншоты

→ Статус

**Заголовок** должен отвечать на 3 вопроса

1. что произошло?

2. Где произошло?

3. При каких условиях?

Нельзя писать «не работает поиск»

Надо писать «Нет логотипа на главной странице если пользователь с ролью user» или «невозможно открыть файл размером больше 200мб»

**Шаги воспроизведения** должны:

- Описывать каждое действие в отдельном шаге

- Описывайте безличные формулировки призывающие к действию

- Описывайте каждый шаг, пока не столкнётесь с дефектом

- Найдите кратчайший путь воспроизведения

- Найдите точный путь воспроизведения

- Пишите так, чтобы любой новичок мог его воспроизвести

Пример Баг репортика: Не отображается логотип на главной странице профиля

Шаги воспроизведения:

1. перейти по ссылке: http…

2. Ввести логин «login»

3. Ввести пароль «password»

4. Кликнуть sign in

Актуальный результат: В левом верхнем углу вместо логотипа – пусто (см. скриншот 1)

Ожидаемый результат: В левом верхнем углу находится логотип (согласно треб. №12)

**Критичность (severity**)

**01 blocker** – после бага нельзя работать с системой в целом или с ключевым функционалом, для продолжения тестирования необходимо исправить баг, может измениться на critical если найден обходной путь

**02 critical** – неправильно работает ключевая бизнес-логика, дыра в системе безопасности, проблема приводит к временному падению сервера или в нерабочее состояние часть системы.

**03 major** – часть бизнес-логики работает некорректно, можно дальше тестить используя другие тестовые данные, или если функционал работает при одних условиях, но падает при других.

**04 minor** – не нарушают бизнес-логику, очевидная проблема UI UX или грамматические ошибки

**05 Trivial** – не затрагивают функционал, приводит в ошибках в сторонних программах, но не нашей, опечатки и пропуски пробела, незначительные отклонения вёрстки

**Priority (приоритет)**

**01 high –** как можно быстрее

**02 normal –** решить обязательно

**03 low –** не срочно

**Советы**

Упрощайте шаги (укорачивайте)

1 отчёт на 1 баг

Проверьте на дубликаты

Прослеживаемость с другими артефактами

Убирать лишние шаги

Проверить на грамотность

**Ошибки**

Недостаточное предоставление данных

Название и ожидаемый результат не соответствуют

Занижение или завышение severity

Неверное употребление терминологии

Сложные речевые обороты

Отсутствует ожидаемый результат

Критика программиста

**Жизненный цикл бага**

После заведение баг-репорта, у бага есть определённый жизненный цикл

**Обнаружен** → **назначить** (баг репорт на кого-то) → **исправлен** → **проверка**

Но также после обнаружения бага баг-репорт может быть **отклонён** или **отложен.** Если программист «исправил» баг, но он не исправился, он также может быть **назначен**, **отклонён** или **отложен**

**Баг не исправят если**

Программист не смог воспроизвести баг

Нет актуального результата

Нет ожидаемого результата

Репорт написан непонятно / расплывчато

Нет необходимых скриншотов и логов

QA не смог объяснить важность проблемы

**Релиз бага** – это когда приложение даётся тестировщикам, зная, что там ошибки

**Утечка бага** – это когда баг обнаруживается пользователем или заказчиком, а не тестировщиком

**Существует шесть базовых типов задач**

**Эпик (epic)** — большая задача, на решение которой команде нужно несколько спринтов.

**Требование (requirement )** — задача, содержащая в себе описание реализации той или иной фичи.

**История (story)** — часть большой задачи (эпика), которую команда может решить за 1 спринт.

**Задача (task)** — техническая задача, которую делает один из членов команды.

**Под-задача (sub-task)** — часть истории / задачи, которая описывает минимальный объем работы члена команды.

**Баг (bug**) — задача, которая описывает ошибку в системе.

**Тестовые среды**

**Среда разработки (Development Env)** – за данную среду отвечают разработчики, в ней они пишут код, проводят отладку, исправляют ошибки

**Среда тестирования (Test Env)** – среда, в которой работают тестировщики (проверяют функционал, проводят smoke и регрессионные тесты, воспроизводят.

**Интеграционная среда (Integration Env)** – среда, в которой проводят тестирование взаимодействующих друг с другом модулей, систем, продуктов.

**Предпрод (Preprod Env)** – среда, которая максимально приближена к продакшену. Здесь проводится заключительное тестирование функционала.

**Продакшн среда (Production Env)** – среда, в которой работают пользователи.