Препод – Евгений Александрович

Введение в тестирование ПО

Тестирование ПО – процесс анализа ПО и её документации с целью выявления дефектов и повышения качества продукта.

Тестирование ПО является процессом в силу того факта, что оно выполняется большим кол-вом людей на протяжении длительного периода времени.

Тестировать нужно:

1. Программы
2. Код программ
3. Прототип ПО
4. Документация
   1. Требования к ПО
   2. Функциональные спецификации
   3. Архитектуру
   4. План проекта
   5. Тестовые случаи сценария
5. Сопроводительную документацию
6. Интерактивную помощь
7. Рук-ва по установке и использованию

Дефект (баг, фича) – Не соответствие фактического и ожидаемого результата

Ожидаемый результат – поведение ПО в ответ на наши действия

Тест-кейс – набор входных данных, условий выполнения и ожидаемых рез-тов, разработанный с целью проверки того и иного свойства поведения ПО. (сценарии для теста)

Тест-план – Часть проектной документации, описывающая пи регламентирующая процесс тестирования.

Билд – промежуточная версия ПО; Релиз – итоговая версия ПО;

Статическое тестирование – Процесс анализа самой разработки ПО, т.е. без запуска ПО

Динамическое тестирование – тестовая деятельность, предусматривающая эксплуатацию продукта.

Методы тестирования:

Метод белого ящика – тест без запуска. Тестировщик имеет доступ и исходному коду, тесты основаны на его знании кода.

Метод белого ящика используется тогда, когда ПО не собрано воедино и его нельзя пускануть.

Метод чёрного ящика – он заключается в том, что таксировщик имеет доступ к ПО только через интерфейсы, которые будут иметь заказчик или пользователь.

Тестирования ведётся с исп. спецификаций или иных док-тов, описывающих требования к системе на основе применения пользовательского интерфейса для ввода входных и получении выходных данных

Цель метода – проверить работу всех функция приложения на соответствие функциональным требованиям.

Метод серого ящика – совокупность подходов белого и чёрного ящика.

Этот метод используется при тестировании веб-приложения, когда татуировщик знает принципе функционирования технологий, на которых оно построены, но не может видеть код самого приложения

Уровни тестирования:

Компонентное тестирование – тестирование отдельного модуля (в т.ч. класс и метод)

Интеграционное тестирование – проверка взаимодействия модулей

Системное тестирование - полная проверка ПО

Функциональное тестирование - Процесс проверки ПО, сконцентрированный на анализе соответствия ПО требования и спецификацию. Функциональное тестирование, бывает ручным и автоматизированным

Его цели:

* + 1. Обнаружить дефекты
    2. Определить степень соответствия ПО ожиданиям заказчика
    3. Принять решение о передачи ПО заказчику

Приёмочный тест - самый первый и быстрый уровень. Если тест не проходит – тест прекращается.

Тест критического пути – тестер имитирует пользователя

Расширенный тест – Тест всего функционала продукта

Качество – показатель степени соответствия продукта предъявляемым к нему требованиям. Качество продукта определяется качество процесса его разработки.

Регрессионное тестирования – Проверка на то, что новый код не сломал старый

Тестирование нового функционала – проверка новых фич.

Конфигурационное тестирование – проверка того, как приложение работает с различными настройками и оборудованием.

Тестирование совместимости - Проверка того, как приложения взаимодействуют с ОС и другими приложениями.

Тестирование удобства использования – Проверка того, насколько юзеру удобно юзать ПО

Тестирования интернационализации – проверка возможности и качества перевода

Тестирование локализации –проверка качества перевода продукта на конкретный язык

Исследовательское тестирование – Тест, основанный на профессиональной интуиции тестировщика, которая может предсказать опасные и малоисследованные способы.

Позитивное тестирование – проверка в тепличных условиях

Негативное тестирование – проверка с ПрЫкОлАмИ (в плохих условиях)

Основные типы требований:

1. Функциональные (что система должна делать);
2. Нефункциональные (с соблюдением каких требований она должна это делать (Время, скорость отклика)).

Уровни требований:

1. Бизнес требования;
2. Пользовательские требования;
3. Функциональные требования.

Хорошо проработанные требования позволяют:

1. Выработать общее понимание между разрабом и заказчиком;
2. Определить рамки проекта;
3. Более точно определить финансовые и временные хар-ки проекта;
4. Обезопасить заказчика от риска получить продукт, который ему не нужен;
5. Обезопасить заказчика от потери денег посредством «неконтролируемого; размытия границ».

Пути выявления требований:

1. Интервью;
2. Наблюдение;
3. Самостоятельное описание;
4. Семинары;
5. Прототипирование.

Стандарт IEEE 830-1998 – описывает развёрнутое описание требований, которые может быть оптимизировано для нужд конкретной организации

В описание требований рекомендуется включать:

1. Введение
   1. Назначение док-та;
   2. Поддерживаемые соглашение;
   3. Предполагаемая аудитория и рекомендации по последовательности работы с док-том для каждого класса читателей;
   4. Границы проекта и ссылка на «концепцию»;
   5. Ссылки;
2. Общее описание
   1. Общий взгляд на продукт (что есть продукт);
   2. Особенности продукта;
   3. Класса и хар-ки пользователей;
   4. Операционная среда (где работает);
   5. Ограничение проектирования и реализации (На каких языках можно и нельзя это писать, что не так с ОС, минимальные системные требования, цвета интерфейса и т.д.);
   6. Документация для пользователей;
   7. Предложения и зависимости;
3. Функции системы
   1. Наименование i-й функции системы;
   2. Описание и приоритеты этой функции;
   3. Последовательности «воздействия-реакции» для данной функции;
   4. Функциональные требования этой функции;
4. Требования к интерфейсу
   1. Ссылки на стандарты UI;
   2. Стандарты шрифтов, значков, кнопок и т.д.;
   3. Конфигурация экрана или ограничение разрешения;
   4. Стандартные кнопки, функции или ссылки одинаковые для всех экранов;
   5. Хоткеи;
   6. Стандарты отображения сообщений;
   7. Стандарты конфигурации для упрощения локализации ПО;
   8. Спец. Возможности для людей с проблемами со зрением;
   9. Интерфейсы оборудования;
   10. Интерфейсы ПО;
   11. Интерфейсы передачи информации;
5. Другие нефункциональные требования
   1. Требования к производительности;
6. Приложения
   1. Приложение А. Словарь терминов (глоссарий);
   2. Приложение Б. Модели анализа (Все модели, построенные в процессе анализа требований);
   3. Приложение В. Список вопросов (не решённые проблемы).

Каждое требование должно быть:

1. Завершённым;
2. Непротиворечивым;
3. Корректным;
4. Недвусмысленным;
5. Проверяемым;
6. Модифицируемым;
7. Прослеживаемым;
8. Проранжированным по важности, стабильности и срочности.

Проблемы незавершённости:

Проблема незавершённости – когда мы знаем, что должны получить, но хз как и с какими характеристиками мы должны это делать.

Для того чтобы избежать таких приколов, нужно спрашивать общие вопросы, типо «как должно работать, на чём, где, зачем». Их преимущество:

1. Они универсальные;
2. Они не навязывают решение;
3. Они не загоняют заказчика в ситуации, когда ему нужно выбивать одну ситуацию из двух.

Ещё одна проблема незавершённости – TBD (To be Defined, или «Будет определено»)

Проблемы противоречивости:

Проблемы противоречивости – это тогда, когда есть противоречие между двумя требованиями, или внутри одного требования.

Для их устранения, стоит просто прочитать требования хотя бы раз, а также нужно определить, что требование во всех частях называется одинаково.

А также, нужно проверить что требование «А» не противоречило:

1. Требованиям про «А»;
2. Требованиям более общим покрывающими «А»;
3. Требованиям, которые связаны с «А»;
4. Требования, про что-то другое, но аналогичные требованию «А»;
5. Как следуя требованию «А», мы можем сломать другие требование.

Проблемы некорректности:

Ошибки некорректности могут быть вызваны:

1. Опечатками;
2. Остатками устаревших требований;
3. Наличием «озолочением» - догорим, но useless функционалом;
4. Наличием технически невыполнимых требований;
5. Наличием неаргументированных требований.

Проблемы двусмысленности;

«Если что-то можно понять двумя способами, это обязательно произойдёт»

За сим, нужно писать требования так, чтобы исключить неверность их понимания.

Типо «Приложение должно уметь принимать большие файлы». Тут нужно уточнить насколько большие, ибо для одного 1мб это много, а для другого 1гб эта мало.

Проблема непроверяемости:

Проблема непроверяемости – это тогда, когда нельзя проверить требование.

Например: «В приложении должно быть 0 ошибок» или «Приложение должно поддерживать все версии всех ОС».  
Даже чисто теоретически, проверить на это нельзя.

Такое ваще не должно быть, ибо тестерам нужно чтобы было что тестировать.

Проблемы непроранжированности:

Проблема непроранжированности – это тогда, когда нету ранжирования требований, и хз чему уделять внимание.

Хорошие требования делятся по:

1. Важности;
2. Приоритетности выполнения;
3. Стабильности (шанс на изменение требования).

**Важно про требования:**

1. Набор требований немодифицируем (из-за того, что в случае малейших изменений, уходит оч много времени на их склейку и вылавливание всех противоречий);
2. Набор должен быть проранжирован (для того, чтобы запилить для начала важное, а фичи допиливать позже);
3. Набор должен быть прослеживаемый (должны быть ссылки на все прыколы от заказчика, чтобы не забывать, что уже запилено или нужно будет допилить

Одна из техник работы с требованиями – **Взаимный пересмотр**

Взаимный пересмотр делится на:

1. Неформальный перепросмотр (тестеры просто свапаются листочками с требованиями, и ищут ошибки);
2. Технический перепросмотр (то же самое, только официально. Что требует подготовки, специалистов и прочее);
3. Формальная инспекция (херня для оч крупных проектов, и в случае больших проектов. Описывается специальными стандартами, требует соблюдения широкого спектра правил и протоколирования результатов).

Способы работы с требованиями:

1. Закидывать вопросами заказчика, коллег. Тем самым, уточняя как представление как и что должно работать;
2. Создавать тест-кейсы – когда мы видим требование, нужно спросить себя «Как это тестировать?». Если хз как – с требованиями проблемы;
3. Рисовать рисунки, схемы и диаграммы.

**Планирование тестовых испытаний**

ЖЦ ПО в контексте тестирования удобнее всего:

1. Начало (Сбор пожеланий заказчика и его требований);
2. Уточнение (Проводятся юниттесты и интеграционные тесты. Прогеры работают);
3. Разработка (Разрабы потеют и допиливают программу);
4. Передача заказчику (Приёмочное тестирование и разворачивание проги у заказчика, проводится пиар-компания);

Чтобы достигнуть успеха, нужно сконцентрироваться на 3-х задачах:

1. Разработка тестов
   1. Разработка методологии и плана тестирования;
   2. Участие в установлении стандарта качества;
   3. Разработка спецификаций тестов;
2. Разработка и выполнение тестов
   1. Разработка и поддержка ручных и автоматизированных тестов;
   2. Выполнение тестов;
   3. Управление билдами (оценка состояния проекта);
3. Отчётность о тестах
   1. Доведение до сведения проектной группы информации о качестве продукта;
   2. Мониторинг ошибок с целью их устранения;

Планирование тестов:

Планирование тестов – команда тестровщиков разрабатывает планы и методики тестирования и таким образом формирует тестовую стратегию, используемую в проекте. План тестирования включает: описание типов тестов, тестируемые составляющие и информацию о требуемых ресурсов.

Планирование - е процесс организации и установления процесса тестирования его описания и согласование его со всеми ключевыми людьми проекта.

Тестовый план – документ, являющийся частью проектной документации, и описывающий что, когда, кем, и как будет тестироваться. Он описывает процесс тестирования конкретного продукта в конкретном проекте.

Ответственным разработку тест-плана – тимлид.

Что нужно сделать на стадии тестирования:

* Понять, что за продукт будут тестировать, как он работает, для чего он нужен;
* Понять, как продукт будет использоваться;
* Разобраться с требованиями;
* Разобраться что будет тестироваться и как;
* Решить какие методы и техники тестирования будут юзаться;
* Определить критерии качества продукта;
* Определить риски (что может сломаться во время разработки. Напр. Болезнь тимлид) и предупредить их возникновение;
* Выводы по всему вышеперечисленному записать в тест-план;
* Озаботиться на тестовом оборудовании.

Артефакты, создаваемые на стадии планирования:

1. Тестовый план;
2. Матрица конфигураций (Все версии билдов и чё там поменялось);
3. Запрос на выделение тестового оборудования.

Риск - сочетание вероятности наступления события и последствие события;

Типо, шанс падения с высотки и последствия падения – риск.

Перечень работ – Перечень фич, которые будут подвергаться тестированию.  
Здесь же, есть перечень фич, которые не будут подвергаться тестированию, в связи с тем, что они реализуются не нами, а другими разрабами, и мы надеемся на них.

Критерии качества и оценка качества процесса – здесь отображается перечень критериев качества, на основании которых будет приниматься заключение об уровне качества продукта и возможности передачи продукта заказчику. Критерии качества относятся к качеству продукта. В текстовых планах могут быть записаны и критерии качества самого процесса тестирования, с целью последующий оценки качества процесс.  
Это необходимо чтобы в случае необходимости существовала возможность оценить, насколько грамотно был построен процесс тестирования, были ли какие-либо проблемы, и , если были, разобраться почему и как их предотвратить их в будущем.

Оценка рисков – Описывается риск, его вероятность и степень его влияния на проект.

Документация и письма – тут размещается перечень артефактов (тест план, тестовые сценарии, тестовые автоматизированные скрипты, дефект-репорты, отчёты о результатах тестирования).

Тестовая стратегия – Описывается стратегия тестирования, методы и типы тестов, каким образом будет выполняться тестирования. Содержание зависит от разраба и заказчика и конкретного проекта.

Артефакты - то что по факту остаётся после тестирования (итоги тестов, ошибки, которые были после них выявлены и т.д.)

Ресурсы - Перечень ключевых людей на проекте. Список тестировщиков с их ролями на проекте и их зоны ответственности.  
Помимо человеческих, перечисляются hardware + software resources.

Расписание и ключевые точки – тут описывается график тестирования в согласовании с графиком выпуска билдов и проектным планом, который разрабатывается менеджером проекта. Сюда же включается основные даты (напр. Даты окончания промежуточных фаз работы над проектом)

Критерии хорошего тест-плана:

1. Полным, корректным и недвусмысленным;
2. Должны быть определены цели тестирования;
3. Должны быть определены – тестовый подход, стратегия, методы, виды и типы тестирования;
4. Запланированный подходи должен быть выполним;
5. Должно быть определено тестовое оборудование, окружение, ПО;
6. Должны быть установлены адекватные критерии качества;
7. Должны быть определены критерии прохождения приёмочного теста и условия прекращения тестирования;
8. Должны быть определены все артефакты, подлежащие сдаче;
9. Должны быть перечислены тестовые рес-сы;
10. Должен быть определён график тестирования;
11. Тест план должные быть принятому в компании шаблону (если в проекте не решено иначе).

Преимущество хорошего тест-плана:

1. Позволяет грамотно организовать действия;
2. Позволяет управлять процессом более эффективно;
3. Позволяет узнать минимальный уровень тестов;
4. Позволяет согласовать график проведения тестов с заказчиком.

**Тестировщик**

**(Я забыл написать тему)**

Важные правила описания ошибок:

1. Как можно подробнее описать что и в какой последовательности было сделано;
2. Описать симптомы ошибок;
3. Прикладывать логи, файлы, конфигурации (хар-ки пк) и скриншоты;

Описание дефекта должно включать:

1. Предусловия;
2. Навигация к проблемной функции;
3. Перечень шагов по воспроизведению;
4. Описание рез-та (дефекта);
5. При необходимости, указать правильное поведение системы, привести пример из документации.

*При описании дефекта, важно докопаться до сути дефекта*

Необходимость документирования:

1. Зафиксированная проблема будет исправлена программистом с большей вероятностью;
2. В электронном описании содержатся и подробности воспроизведения проблемы;
3. Отчёт нужен для проверки;
4. Другой таксировщик на проекте может посмотреть предыдущие отчёты;
5. Программист сопровождающий прогу, может узнать является ли тот костыль на который он смотрит является вынужденной мерой для исправления ошибки;
6. Запись сохраняется для дальнейшего рассмотрения руководством.

Уровень ошибок:

1. Тривиальные
2. Мажорные
3. Критические
4. Блокер

Неграмотное написание отчёта об err может привести:

1. Дефект может быть исправлен неверно;
2. По дефекту могут возникнуть доп. Вопросы, что приведёт к трате времени;
3. При регрессивном тестировании возможно нельзя будет воспроизвести эффект;
4. Разраб возможно не сможет понять в чём заключается дефект;
5. Разраб возможно не сможет воспроизвести дефект.

**Информационная безопасность**

*Зачем беспокоиться по поводу защищенности и безопасности своих программ:*

1.Существует ряд категорий информации и сведений, защита которых требуется национальными нормативно-правовыми актами страны.

2.Многие пользователи достаточно образованы и имеют высокие требования при выборе продуктов, которые они используют в своей работе.

3.Защищённый продукт защищает интересы не только пользователя, но и разработчика.

Определения тестирования безопасности.

1.Тестировнаие программного обеспечения с целью обнаружения уязвимостей в области безопасности.

2.Процесс определения того факта, что особенности системы, отвечающие за её безопасность, реализованы согласно тому факта, что особенности системы, отвечающие за её в безопасность, реализованы согласно тому, как они были спроектированы, и что они являются адекватным в контексте окружения, в котором приложение выполняется.

3.Аттестация объектов информатизации – комплекс организационно-технических мероприятий, в результате которых посредством специального документа подтверждается, что объект соответствует требованиям стандартов или иных нормативно-технических документов по безопасности информации, утвержденных Гостехкомиссией России.

Терминология

Программный продукт – множество компьютерных программ, процедур вместе с соответствующей документацией и данными.

Программа – данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определенного алгоритма.

Защищённость программного продукта – способность противодействовать несанкционированному вмешательству в нормальный процесс его функционирования, а также попыткам хищения, незаконной модификации, использования, копирования или разрушения самого продукта, его составляющих, данных и информации, входящих в состав продукта, доступных ему в процессе выполнения или заложенных в него во время разработки.

Безопасность программного продукта – способность продукта достигать приемлемого уровня риска для здоровья и наследственности людей, их бизнеса, компьютерных систем.

Информация – это сведения о лицах, предметах, фактах, событиях, явлениях и процессах независимо от формы их представления.

Источники проблем:

* + - 1. Конфликт интересов между разработчиком продукта и его пользователями
      2. Конфликт интересов между пользователями продукта, а также между пользователями продукта и пользователями компьютерной системы, в которой эксплуатируется продукт.
      3. Форс-мажорные обстоятельства (перебои с электропитанием, ошибки в работе аппаратного обеспечения, ошибки в реализации используемых технологий.)

Принципы безопасности и защищенности программных продуктов:

1.Конфидециальность.

2.Целостность

3.Проверка подлинности.

4.Проверка полномочий.

5.Доступность.

6.Подотчестность.

Основные категории объектов защиты программного продукта:

1. Данные
2. Информация.
3. Функции ПП.

Данные – информация, представленная в виде файлов программы, информация, полученная от пользователей, из переменных операционной системы, по сети или в результате взаимодействия между потоками и процессами операционных систем.

Информация (в качестве объекта защиты) – сведения, доступные продукту и имеющие определенную ценность, факт нарушения конфиденциальности, целостности которых может вести к какому-либо виду ущерба.

Доступ к функциям программного продукта также должен быть ограничен таким образом, чтобы исключить возможные варианты ущербов, что решается путём реализации функций авторизации и предоставления прав доступа.

**Уровни представления**

* Набор данных (рассмотрение продукта, как набора файлов, из которых он состоит)
* Множество потоков данных, когда продукт выполняется (обычно такие потоки описываются в виде data-flow диаграмм на этапе разработки ПП)
* Набор функций (сервисов)
* Предмет договорных отношений между разработчиком и пользователем (разработчик может внести в продукт некоторые функции, которые отслеживают выполнение условий договора)
* Часть компьютерной системы, в которой выполняется продукт (продукт должен быть безопасным для самой системы и её пользователей)

Типы тестов уязвимости

Цель состоит в обеспечении надежной и безопасной работы программы в разумных и даже не слишком разумных рабочих сценариях.

Среда состоит из файлов, приложений, ресурсов системы и других локальных, либо сетевых ресурсов, используемых приложением.

Ввод пользователя – данные, которые анализируются и используются программной и происходят от внешних сущностей.

Внутренние данные и логика – это внутренне хранящиеся переменные и логические пути, имеющие неограниченное число потенциальных вариантов использования.

Атаки через среду

Ряд важных вопросов, которые необходимо задать о степени доверия приложения этим взаимодействиям:

1. Насколько приложение доверяет своей локальной среде и удаленным ресурсам?
2. Помещает ли придожение конфиденциальную информацию в ресурс, могут читать другие приложения?
3. Доверяет ли оно каждому загружаемому файлу или библиотеке, не проверяя их содержимое?
4. Может ли злоумышленник воспользоваться этим доверием, чтобы заставить приложение выполнить свой приказ.

Программное обеспечение

Перечень категорий программного обеспечения, используемого специалистами в области безопасности для тестирования компьютерных систем:

1. Сетевые утилиты SSH клиенты.
2. Системы обнаружения вторжения. (защита windows систем, защита linux систем, сканеры уязвимостей, уклонение от атак)
3. Управление доступом. (сетевые экраны)

Принцип современной защиты информации можно выразить так – поиск оптимального соотношения между доступностью и безопасностью.

Защита информации включает в себя кроме технических мер ещё и обучение или правильный подбор обслуживающего персонала.

Компьютерная защита – это постоянная борьба с глупостью пользователей и интеллектом хакеров.

«Тестирование безопасности»

***Тестирование безопасности*** - тестирование программного обеспечения с целью обнаружения уязвимостей в области безопасности.

***Программный продукт*** – множество компьютерных программ, процедур вместе с соответствующей документацией и данными

***Программа*** – данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определенного алгоритма

***Защищенность*** ***программного*** ***продукта*** – способность противодействовать несанкционированному вмешательству в нормальный процесс его функционирования, а также попытками хищения, незаконной модификации, использования, копирования или разрушения самого продукта, его составляющих данных и информации, входящих в состав продукта, доступных ему в процесса выполнения или заложенных в него во время разработки.

***Безопасность*** ***программного*** ***продукта*** – способность продукта достигать приемлемого уровня риска для здоровья и наследственности людей, их бизнеса, компьютерных систем, имущества или окружающей среды в отсутствие нарушений законов и норм права при данном способе применения.

***Информация*** – сведения о лицах, предметах, фактах, событиях, явлениях, и процессах независимо от формы их представления.

***Источники проблем***

* Конфликт интересов между пользователями продукта, а также между пользователями продукта и пользователями компьютерной системы, в которой эксплуатируется продукт
* Конфликт интересов между разработчиком продукта и его пользователями
* Форс-мажорные обстоятельства (перебои с электропитанием, ошибки в работе программного обеспечения, ошибки в реализации используемых технологий)

***Принципы безопасности и защищенности***

* Конфиденциальность
* Целостность
* Проверка подлинности (аутентификация)
* Проверка полномочий (авторизация)
* Доступность
* Подотчетность

***Категории объектов защиты***

В качестве основных категорий объектов защиты программного продукта выделяют

- **данные**; (информация, представленная в виде файлов программы, а также обрабатываемая информация, полученная от пользователей, из переменных операционной системы, по сети или в результате взаимодействия между потоками и процессами ОС).

- **информация**; (различного рода сведения, личная переписка).

-**функции** **ПП**; (должен быть ограничен таким образом, чтобы исключить возможные варианты ущербов, что решается путем реализации функций авторизации и предоставления прав доступа).

***Уровни представления программных продуктов с позиции информационной безопасности***

* **Набор данных** (рассмотрение продукта как набора файлов, из которых он состоит)
* **Множество потоков данных,** когда продукт выполняется, нужно следить чтобы ни что не препятствовало нормальному ходу потоков, а также =, чтобы неавторизованные субъекты или процессы не смогли читать или удалять данные из потоков
* **Набор функций (сервисов)** необходимо обеспечить разграничение доступа к функциям продукта, а также обеспечит меры по сохранению доступности, правильности и надежности работы этих функций
* **Предмет договорных отношений между разработчиком и пользователем** (разработчик может внести в продукт некоторые функции, которые отслеживают выполнение условий договора со стороны пользователей, а пользователи могут пытаться обойти или отключить эти функции)
* **Часть компьютерной системы,** в которой выполняется продукт, Продукт должен быть безопасным для самой системы и ее пользователей, и в то же время, продукт и его данные должны быть защищены от неавторизованных процессов и пользователей в рамках данной системы.

***Типы тестов уязвимости***

Среда состоит из файлов, приложений, ресурсов системы и других локальных, либо сетевых ресурсов, используемых приложением

***Ввод пользователя*** – это данные, которые анализируются и используются программой и происходят от внешних сущностей.

***Внутренние данные и логика*** – это внутреннее хранящиеся переменные и логические пути, имеющие неограниченное число потенциальных вариантов использования.

**Атаки через среду**

Ряд важных вопросов, которые необходимо задать о степени доверия приложения этим взаимодействиям:

- Насколько приложение доверяет своей локальной среде и удаленным ресурсам?

- Помещает ли приложение конфиденциальную информацию в ресурс (реестр), который могут читать другие приложения?

- Доверяет ли оно каждому загружаемому файлу или библиотеке, не проверяя их содержимое?

- Может ли злоумышленник воспользоваться этим доверием, чтобы заставить приложение выполнить свой приказ?

**Дефекты, жизненный цикл дефектов, отчёты о дефектах**

Программная ошибка – изъян в разработке, который вызывает несоответствие результатов с ожиданиями

Баг – отклонение фактического результата от ожидаемого.

Дефект – ошибка, которая может быть, а может и не быть последствием сбоя.

Дефект – поведение проги, затрудняющее или делающее невозможным достижение целей пользователя или удовлетворения интересов участников. Если нельзя пофиксить – ограничение технологии.

Жизненный цикл дефекта:

<ВСТАВИТЬ ПИКЧУ>

1. Обнаружен – дефект отправлен юдишкам которым нужно знать о дефекте;
2. Назначен – юдишки начинают фиксить баг;
3. Исправлен – юдишка который исправлял, сказал что он исправлен. Горк одобряет;
4. Проверен – юдишка который контролирует работу юдишек которые исправляли ошибку, сказал что бага нет;
5. Открыт заново – глупые юдишки не смогли пофиксить баг с первого трая, и он всплыл снова
6. Отклонён – главный юдишка, сказал что «не баг а фича»
7. Отложен – юдишки не знают, стоит ли заниматься этой фичей

Если баг не «закрыт» или «отклонён», то он существует.

Основные атрибуты об ошибке:

1. Id (уникальный Id. Чаще всего формируется системой отслеживания ошибок);
2. Краткое описание (шо за рофл произошёл и где);
3. Подробное описание (По сути, та же инфа что и во втором пункте, но желательно ваще всё что знаешь);
4. Шаги воспроизведения (чё ты сделал, чтобы рофл произошёл);
5. Воспроизводимость (Показывает, насколько часто рофл происходит);
6. Важность (критическая/critical - Прога не работает; – высокая/major - Прога работает, но с рофлами; средняя/medium - То, что не крашит прогу, почти не мешает работать, но вызывает кеки у заказчика; низкая/minor - То, что де-факто не мешает работе);
7. Срочность (asap – нужно править сейчас, высокая – Нужно исправить быстро, обычная – в очередь, низкая – если останется время нужно бы фиксануть);
8. Симптом – (косметический – опечатки, кривые картинки, кривое меню и т.д.; Потеря/повреждение данных – когда БД троллит тебя и теряет инфу; Проблема в документации – ошибка не в проге, а в документах к ней; Некорректная операция – 2+2=5; Проблема инсталляции – ошибка прям во время установки; Ошибка инсталляции – установка не устанавливает; Нереализованная функциональность – прогеры чё-то не сделали; Низкая производительность; Крах системы – всё ломается; Неожиданное поведение – когда рофельные чувачечи сказали что ctrl+o вызывает открытие нового окна, а оно его закрывает; Недружественное поведение – **я сам хз. Звучит так, будто юзера просят зарегистрироваться, а он против**; Расхождение с требованиями – когда оно работает, но не так как должно; Предложения по улучшению).

Дополнительные атрибуты об ошибке:

1. Возможность обойти баг – когда если чуток изменить сценарий, всё заработает;
2. Дополнительная инфа – если хотите написать что-то;
3. Приложения (аттачи) – разные приколы которые файлы.

Рекомендации по репортам:

* Тщательно объяснить как воспроизвести;
* Описать всё максимально подробно;
* Писать всё понятно;
* Дать ссылку на соответствующее требование.
* Если есть обход – нужно сказать;
* Окружение;
* Без эмоций;
* Один отчёт – один эррор;
* Писать сразу же когда обнаружили ошибку;
* Найти наиболее серьёзные последствия ошибки;
* Возможно вам придётся по этому же репорту фиксить.

**Тестирование юзабилити**

Юзабилити – мера реакции пользователя, когда он сталкивается с продуктом или системой.

Юзабилити – это мера качества пользовательского опыта, приобретенного при взаимодействии с продуктом или системой.

Главные требования к юзабилити приложения:

1. Быть простым в изучении и эффективным в использовании.
2. Легко вспоминаться при последующем использовании.
3. Удовлетворять требования пользователя с минимальным количеством ошибок.

[Главные тезисы тестирования юзабилити](https://habr.com/ru/post/20459/)

Предмет тестирования юзабилити – впечатление пользователя о приложении.

Карточная сортировка – это классификационный метод, при котором пользователи сортируют различные элементы разрабатываемого продукта по нескольким категориям.

Контекстное исследование – метод структурированного интервью, которое отличается от обычного:

1. Учет контекста, в котором используется изучаемый продукт.
2. Совместная оценка продукта пользователем и разработчиком.
3. В фокусе оценки продукта находится его удобство для пользователя.
4. Оценочные листы помогают удостовериться в том, что продукт выполнен с учетом принципов функциональности дизайна.

Прототипирование – это создание или использование модели конечного продукта, позволяющее протестировать его составляющие на любых стадиях разработки. (примерно на пре-альфа версии)

Различают вертикальное и горизонтальное прототипирование:

1. Горизонтальное – широко, но без конкретики;
2. Вертикальное – отдельный функционал.

Обзоры – интервью с пользователями со специальными вопросами. Вопросы могут варьироваться в зав-ти от целей исследования:

* Демографическая инфа;
* Оценка функционального наполнения продукта;
* Оценка дизайна.

Используется для проведения маркетинговых исследованиях, идентификации потенциальны з пользователей, установления из нужд, компетентности, так и для оценки реакции пользователей.

Анкеты - то же что и обзоры, но это список вопросов дающиеся на листиках. Существуют платные варианты

Плюралистическая проработка – большая группа в которую входят: пользователи, разрабы, специалисты по здоровью и охране труда, эксперты в области оценки продуктов. Группа шаг за шагом прорабатывает каждый эл-т взаимодействия пользователя с продуктом.

Протоколы самоотчёта - Бланка, в которых пользователи сами фиксируют всё что да. Этот метод является экономным, но нельзя зарегистрировать эмоции от взаимодействия и реакции пользователей.

Фиксация «мыслей в слух» - пользователя просят произносить всё что он думает во время работы в слух, а его слова либо записывают в потом анализируют, либо сразу пишут на листик и анализируют.

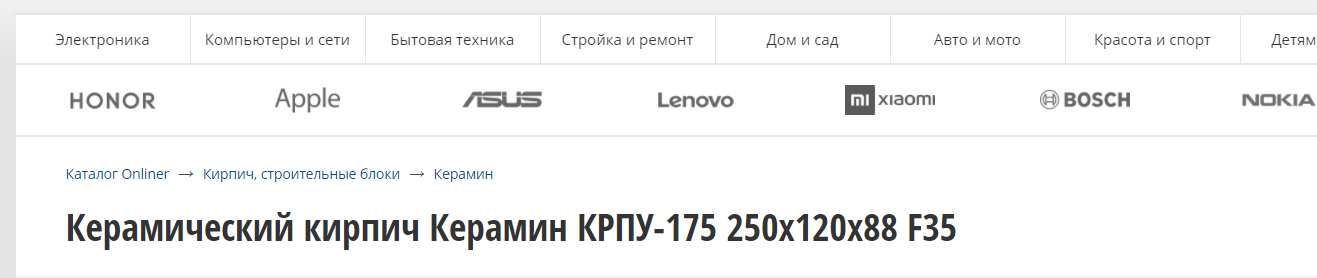
Метод фокусных групп – опрос специально собранной группы пользователей. Основное преимущество – позволяет выявить спонтанные реакции и оценить отношение к этой идеи в целом. Главный недостаток – низкая точность.

Фокус группа - группа людей, которые обсуждают как им удобнее будет удобнее взаимодействовать с прогой, и желательно, чтобы она была близка с целевой аудиторией. Методом дискуссии, они выясняют как им будет удобнее будет взаимодействовать с прогой. Группе необходим лидер.

Эвристическое исследование – группа профессионалов проверяет отношение человека и машины (отдельно). Если кратко, это когда профи чекают прогу, и смотрят как она взаимодействует с человеком.

Экспертиза компонентов – анализ конкретный набор продуктов, с которыми взаимодействует пользователь. Суть в том, что чекается каждый шаг для определённой задачи, и определяется их: очевидность, логичность переходов и их оформление, удобство.

«хлебные крошки»:



Хлебные крошки позволяют узнать путь на страницу (где ты находишься) и позволяет вернуться.

**Техническое тестирование**

Тестирование – процесс проверки соответствия заявленных к продукту требований и реально реализованной функциональности.

Функциональное тестирование – проверяет на то, реализовано ли то что необходимо.

Техническое тестирование – тестирование ПО, при котором в фокусе лежит не функционирование, а способность функционировать в различных условиях.

Виды технического тестирования:

1. Оценочное – проверка на производительность;
2. Нагрузочное – проверка возможностей софта на кол-во юзеров которое может одновременно пользоваться софтом (чаще всего тестируются API и серваки);
3. Объёмное – проверка возможностей софта, на V данных которое может обработать система и скорость этой обработки;
4. Тестирование надёжности – тестирование системы в течении длительного времени, при реальной нагрузке (будто пользователи юзают);
5. Стресс тест – проверка системы под максимальной нагрузкой;
6. Аварийное тестирование – как ведёт себя система после падения;
7. Тестирование масштабирования – анализ тестирования на расширяемость системы.

VU – виртуальный пользователь. Исполняемый процесс ОС, который с помощью специальных инструментов имитирует действия пользователя (заскриптован, не случайный, работает не через GUI).

RU – реальный пользователь. (полностью случайные действия, использует случайные наборы данных, работает через GUI)

Скрипт – программный код.

VU­ – Скрипт, который имитирует человека.

Группа – набор VU, выполняющих действие по заранее определённому алгоритму.

Сценарий – набор групп. Имитирует нагрузку, близкую к реальной.

Объект тестирования – та часть системы, которая подвергается нагрузке.

Тестируемые функции – функции, над которыми проводятся тесты.

Модель нагрузки – характеристика всех пользователей, которые работают с системой. Параметры: кол-во пользователей, выполняемые функции, частота выполнения функций.

Проведение нагрузочного тестирования – точное воспроизведение нагрузки с контролем окружения и мониторингом выполнения.

Требования к приложению при техническом тестировании:

1. Время отклика;
2. Утилизация ресурсов;
3. Модель нагрузки;
4. Стенд;
5. Время восстанавливаемости;
6. Время наработки на отказ (какое время система может проработать под избыточной нагрузкой до отказа);
7. Прочее.

**Ошибки тестов**

Классические ошибки тестировщика

1. Недостаточное внимание к тестированию требованиям – приводит к срывам сроков;
2. Недостаточное внимание тест-кейсам – тест кейсы нужно писать перед разработкой, а не позже. Тест кейсы пишутся под требования, а не под готовый продукт;
3. Непонимание того факта, что тест-кейсы должны проверять требования, а не дублировать их;
4. Написание некачественных приёмочных тестов – тест кейсы пишутся не так как понятно тестировщику, а так, как понятно всем людям. Так же, нужно учитывать: 1 тест-кейс = 1 проверка;
5. Несвоевременность отметок о прохождениях или сбоях;
6. Неграмотное написание отчётов в системе отслеживания ошибок – нужно использовать BTS (Bug Tracking System), нельзя экономить время на создание отчёта, обязательно использовать темплейты.

Портрет тестировщика – целью является определить, что же определяет из себя IT-шник. Эти правила поставила под собой группа кампаний HeadHunter.

Функциональные диаграммы и тестирование с помощью них

Метод функциональных диаграмм или диаграмм причинно-следственных связей помогает систематически выбирать высоко результативные тесты. Кроме этого, метод функциональных диаграмм дает полезный побочный эффект, так как позволяет обнаруживать неполноту и неоднозначность исходных спецификаций.

Функциональная диаграмма – это формальный язык, на который транслируется спецификация, написанная на естественном языке.

Методика использования функциональных диаграмм:

1. Спецификация разбивается на рабочие участки, так как для больших спецификаций функциональные диаграммы становятся слишком громоздкими.
2. В спецификации *определяются причины и следствия*. *Причина* – это отдельное входное условие или класс эквивалентных входных условий. *Следствие* – это выходное условие (результат выполнения программы).

Причины и следствия определяются путем последовательного чтения спецификации. Каждым причине и следствию приписывается уникальный номер.

1. Анализируется семантическое содержание спецификации, которая преобразуется в булевский граф (функциональную диаграмму), связывающий причины и следствия.
2. Диаграмма дополняется примечаниями, задающими ограничения и описывающими комбинации причин и следствий, которые являются невозможными из-за синтаксических или внешних ограничений.
3. Путем методического прослеживания состояний условий диаграммы она преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.
4. Столбцы решений преобразуются в тесты.

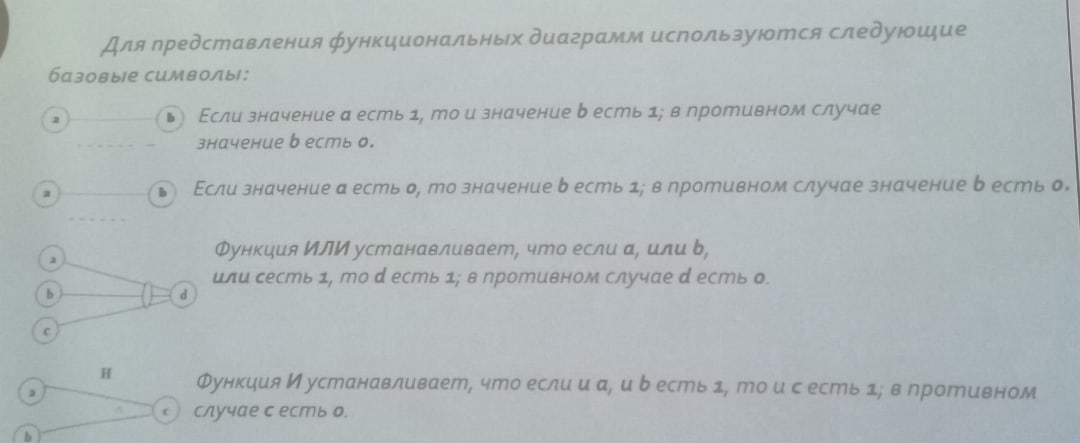
Процедура генерации таблицы решений заключается в следующем:

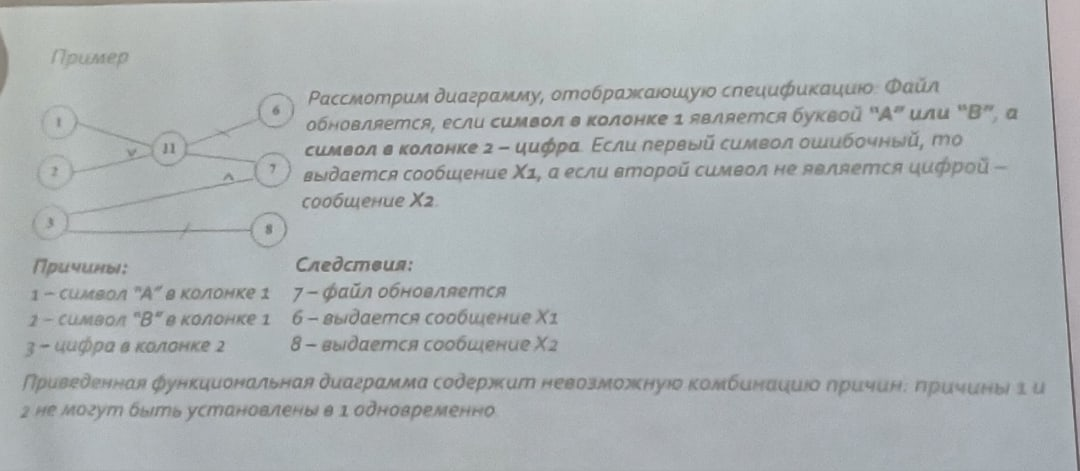
1. Выбрать некоторое следствие, которое должно быть в состоянии 1.
2. Найти все комбинации причин (с учетом ограничений), которые установят это следствие в 1, прокладывая из этого следствия обратную трассу через диаграмму.
3. Построить столбец в таблице решений для каждой комбинации причин.
4. Для каждой комбинации причин определить состояния всех других следствий и поместить их соответствующий столбец таблицы решений.

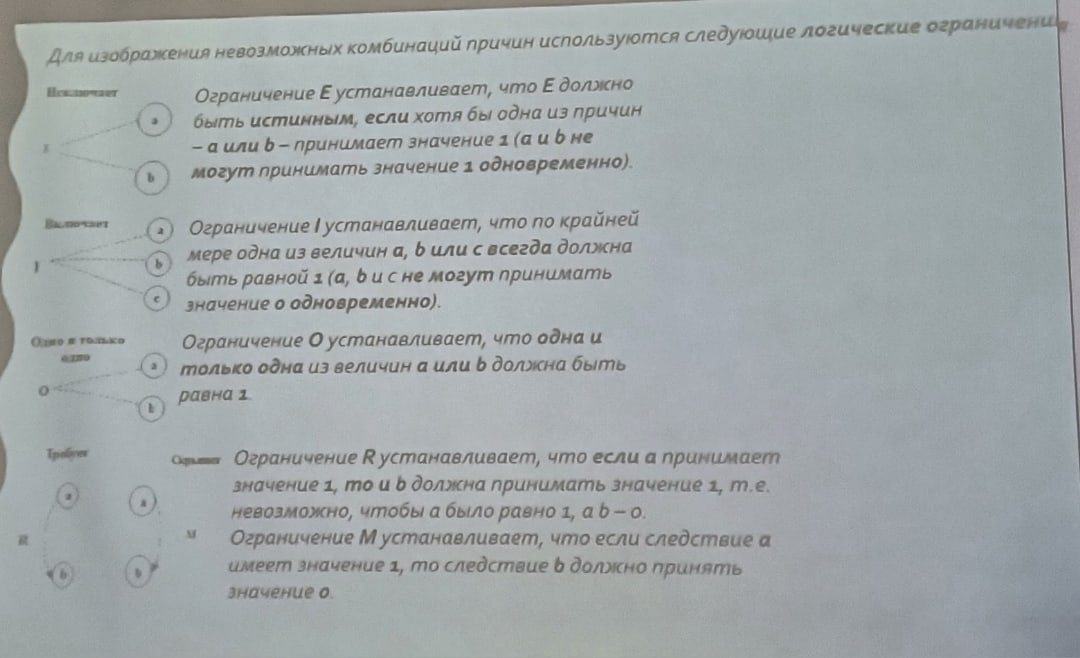
При выполнении этого шага необходимо руководствоваться следующими положениями;

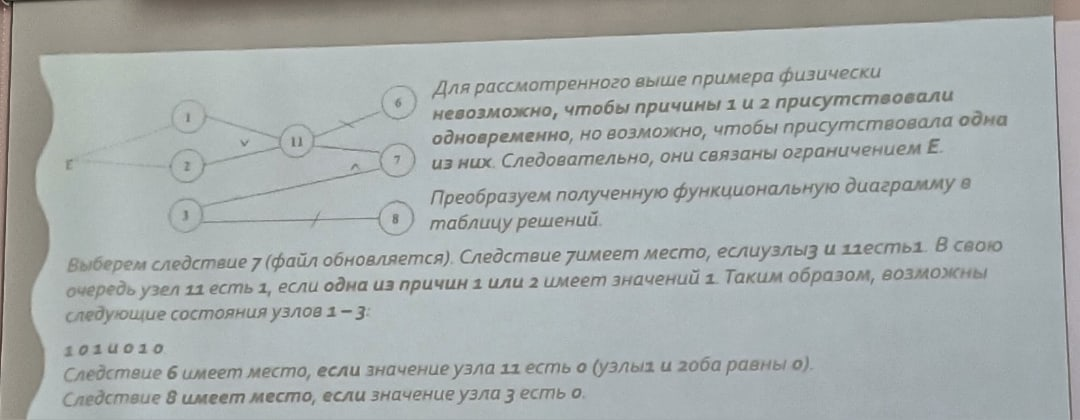
* Если обратная трасса прокладывается через узел ИЛИ, выход которого должен принимать значение 1, то одновременно не следует устанавливать в 1 более одного входа в этот узел. Цель данного правила – избежать пропуска определенных ошибок из-за того, что одна причина маскируется другой.
* Если обратная трасса прокладывается через узел ИЛИ, выход которого должен принимать значение о, то все комбинации входов, приводящие выход в 0, должны быть в конечном счете перечислены.
* Если обратная трасса прокладывается через узел и, выход которого должен принимать значение 0, то необходимо указать лишь одно условие, согласно которому все входы являются нулями.
* Каждый узел диаграммы диаграммы может находиться в двух состояниях – 0 или 1, обозначает состояние “отсутствует”, а 1 – “присутствует”.

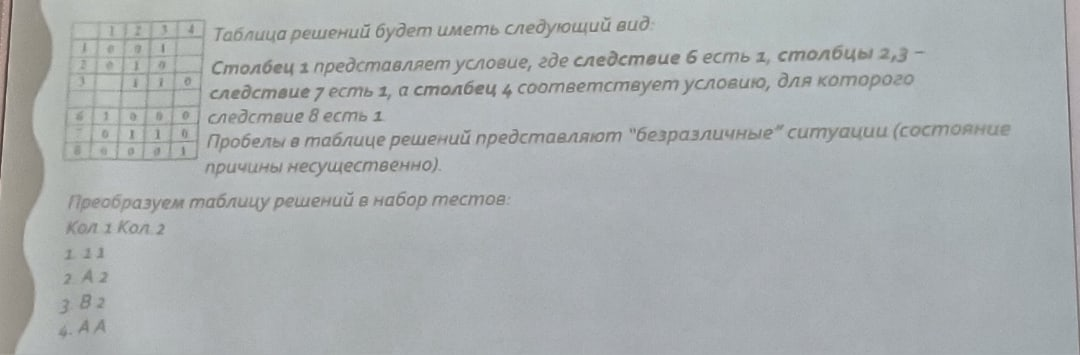
Для представления функциональных диаграмм используюстя следующие базовые символы:











**Модульное тестирование**

Модульное тестирование – процесс в программировании позволяющий проверить на корректность отдельные модули исходного кода программы. Идея состоит в том, чтобы писать тесты для каждой функции или метода. Это позволяет быстро проверить не сломалась ли программа после модификации кода без теста всей программы. Цель модульного тестирования – изолировать отдельные части программы и показать что по отдельности это части работоспособны.

1. Поощрение изменений – модульное тестирование позволяет программистам проводить рефакторинг будучи уверенным что модуль по прежнему работает корректно. Это поощряет к изменениям кода, поскольку достаточно легко проверить что код работает после изменений.
2. Упрощение интеграции – помогает устранить сомнение по поводу отдельных модулей
3. Документирование – модульные тесты можно рассматривать как документацию.
4. Отделение интерфейса от реализации – поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса распространяется часто распространяется на связанные с ним. Т.Е. оно поощряет создание автономного кода
5. Сложный код – На 1 строку исходника, требуется 3-5 итоговых. Как и любой тест, он не может отловить всё.
6. Результат известен приблизительно – В случае если новый тест показывает новое значение, стоит проверить вручную что точнее.
7. Ошибки интеграции и производительности – бесполезно при тестах на производительность. Так же это обозначает что ошибки интеграции , системного уровня и функций исполняемых в нескольких модулях не будут определены.w
8. При общей низкой культуре программирования - Нужно хранить все старые версии прог и пройдённых тестов, чтобы чуть что откатится.

Техника модульного тестирования – Для большинства ЯП существуют библиотеки для тестирования.

Веб приложение – клиент-серверное приложение, в котором клиентом выступает браузер, а в серверном – веб-сервер. Основная часть находится на стороне веб-сервера, который обрабатывает полученные запросы в соответствии с бизнес логикой продукта и формирует ответ обратно пользователю.

**Тестирование клиент-серверных приложений**

Клиент – как правильно, клиент это браузер, но встречаются и исключения (бывает, что клиент взывает к серваку, который взывает к другому серваку). В классическом случает, чтобы получить, например, графический интерфейс, клиент вынужден получить запрос, а после обработать полученный ответ сервера.

Сервер – сервер, принимающий HTTP запросы от клиентов и выдающий им ответы. Серваком называют как железо, так и ПО, благодаря которому сервак реализован. Наиболее популярные: Apache, ISS, NGINX.

БД – Фактически не является частью веб-сервера, но в ней хранится вся динамическая информация приложения

Варианты тестирования со стороны клиента:

1. Кроссбраузерность – проверка на правильность отображения в разных браузерах;
   1. Функциональные возможность продукта на стороне клиента;
   2. Правильность отображения эл-тов графики;
   3. Шрифты и размеры текстовых символов;
   4. Доступность и функциональность разнообразных форм, включая их интерактивность
2. Веб-формы на стороне клиента – формы которые заполняет клиент
3. Опечатки

Варианты тестирования со стороны сервера:

1. Нагрузочное тестирование – имитирует работу с прогой множества пользователей;
2. Тестирование БД – тесты проводятся на:
   1. В БД инфа сохраняется в неизменном виде;
   2. Сохранённая инфа должна отображаться в любой части приложения одинаково;
   3. Названия и структура БД должна соответствовать документации;
   4. Быстрота;
   5. Правильно работа удаления и псевдоудаления;

Запросы – Происходят через HTTP. Без HTTP – протокол передачи данных. Взаимодействуют через сообщения,

Тестовое окружение

Основной объем тестирования практически любой сложной системы обычно выполняется в автоматическом режиме. Кроме того, тестируемая система обычно разбивается на отдельные модули, каждый из которых тестируется вначале отдельно от других, затем в комплексе.

Это означает, что для выполнения тестирования необходимо создать некоторую среду, которая обеспечит запуск и выполнение тестируемого модуля, передаст ему входные данные, соберет реальные выходные данные, полученные в результате работы системы на заданных входных данных. После этого среда должна сравнить реальные выходные данные с ожидаемыми и на основании данного сравнения сделать вывод о соответствии поведения модуля заданному.

Тестовое окружение также может использоваться для отчуждения отдельных модулей системы от всей системы. Разделение модулей системы на ранних этапах тестирования позволяет более точно локализовать проблемы возникающие в их программном коде. Для поддержки работы модуля в отрыве от системы тестовое окружение должно моделировать поведение всех модулей, к функуциям или данным которых обращается тестируемый модуль.

Поскольку тестовое окружение само является программой (причем зачастую реализованной не на том языке программирования, на котором написана система), оно само должно быть протестировано. Целью тестирования тестового окружения является докозательство того, что тестовое окружение никаким образом не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

**Драйверы и заглушки**

Тестовое окружение для программного кода на структурных языках программирования состоит их двух компонентов – драйвера, который обеспечивает запуск и выполнение тестируемого модуля, и заглушек, которые моделируют функции, вызываемые из данного модуля. Драйвер и заглушки могут иметь различные уровни сложности, требуемый уровень сложности выбирается в зависимости от сложности тестируемого модуля и уровня тестирования.

Драйвер может выполнять следующие функции:

* Вызов тестируемого модуля
* 1 + передача в тестируемый модуль входных значений и прием результатов.
* 2 + вывод выходных значений.
* 3 + протоколирование процесса тестирования и ключевых точек программ.

Заглушки могут выполнять следующие функции:

* Не производить никаких действий (такие заглушки нужны для корректной сборки тестируемого модуля).
* Выводить сообщения о том, что заглушка была вызвана
* 1 + выводить сообщения со значениями параметров, переданных в функцию.
* 2 + возвращать значение, заранее заданное во входных параметрах теста.
* 3 + выводить значение, заранее заданное во входных параметрах теста.
* 3 + принимать от тестируемого ПО значения и передавать их в драйвер.

**Генераторы сигналов (событийно-управляемый код)**

Значительная часть сложных программ в настоящее время использует ту или иную форму межпроцессного взаимодействия. Это обусловлено естественной эволюцией подходов к проектированию программных систем, которая последовательно прошла следующие этапы:

* Монолитные программы, содержащие в своем коде все необходимые для своей работы инструкции. Обмен данными внутри таких программ производится при помощи передачи параметров функций и использования глобальных переменных. При запуске таких программ образуется один процесс, который выполняет всю необходимую работу.
* Модульные программы, которые состоят из отдельных программных модулей с четко определенными интерфейсами вызовов. Объединение модулей в программу может происходить либо на этапе сборки исполняемого файла (статическая сборка или static linking), либо на этапе выполнения программы (динамическая сборка или dynamic linking). Преимущество модульных программ заключается в достижении некоторого уровня универсальности – один модуль может быть заменен другим. Однако, модульная программа все равно представляет собой один процесс, а данные, необходимые для решения задачи, передаются внутри процесса как параметры функций.
* Программы, использующие межпроцессное взаимодействие. Такие программы образуют программный комплекс, предназначенный для решения общей задачи. Каждая запущенная программа образует один или более процессов. Каждый из процессов использует для решения задачи либо свои собственные данные и обменивается с другими процессами только результатом своей работы, либо работает с общей областью данных, разделяемых между разными процессами. Для решения особо сложных задач процессы могут быть запущены на разных физических компьютерах и взаимодействовать через сеть. Преимущество использования межпроцессного взаимодействия заключается в еще большей универсальности – взаимодействующие процессы могут быть заменены независимо друг от друга при сохранении интерфейса взаимодействия. Другое преимущество состоит в том, что вычислительная нагрузка распределяется между процессами. Это позволяет операционной системе управлять приоритетами выполнения отдельных частей программного комплекса, выделяя большее или меньшее количество ресурсов ресурсоемким процессам.

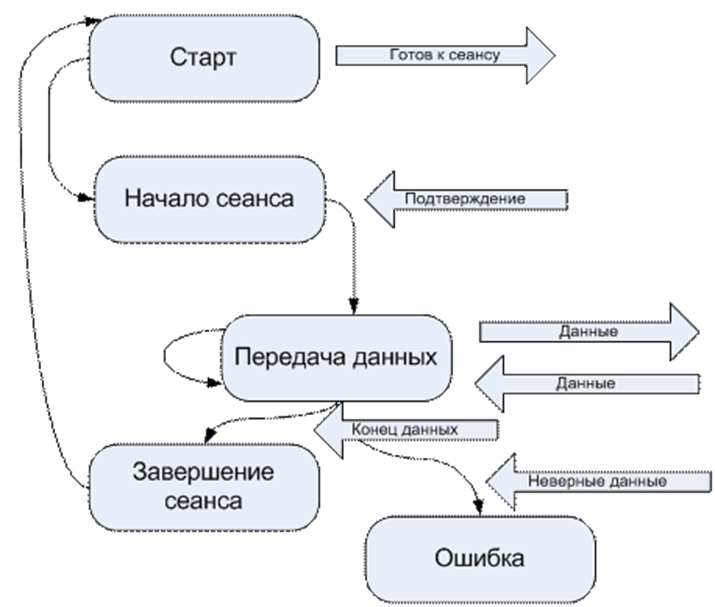
При выполнении многих процессов, решающих общую задачу, используются несколько типичных механизмов взаимодействия между ними, направленных на решение следующих задач:

* передача данных от одного процесса к другому;
* совместное использование одних и тех же данных несколькими процессами;
* извещения об изменении состояния процессов.

Во всех этих случаях типичная структура каждого процесса представляет собой конечный автомат с набором состояний и переходов между ними. Находясь в определенном состоянии, процесс выполняет обработку данных, при переходе между состояниями – пересылает данные другим процессам или принимает данные от них.

Для моделирования конечных автоматов используются stateflow или SDL-диаграммы, акцент в которых делается соответственно на условиях перехода между состояниями и пересылаемыми данными.

Так, на Рис. 2 показана схема процесса приема/передачи данных. Закругленными прямоугольниками указаны состояния процесса, тонкими стрелками – переходы между состояниями, большими стрелками – пересылаемые данные. Находясь в состоянии «Старт» процесс посылает во внешний мир (или процессу, с которым он обменивается данными) сообщение о своей готовности к началу сеанса передачи данных. После получения от второго процесса подтверждения о готовности начинается сеанс обмена данными. В случае поступления сообщения о конце данных происходит завершение сеанса и переход в стартовое состояние. В случае поступления неверных данных (например, неправильного формата или с неверной контрольной суммой), процесс переходит в состояние «Ошибка», выйти из которого возможно только завершением и перезапуском процесса.



Тестовое окружение для такого процесса также должно иметь структуру конечного автомата и пересылать данные в том же формате, что и тестируемый процесс. Целью тестирования в данном случае будет показать, что процесс обрабатывает принимаемые данные в соответствии с требованиями, форматы передаваемых данных корректны, а также, что процесс во время своей работы действительно проходит все состояния конечного автомата, моделирующего его поведение.