1. Понятие тестирования ПО. Основные определения.

Тестирование программного обеспечения - проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, **тестирование** - это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

Верификация (Verification) - это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа. Т.е. выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.

Валидация (Validation) - это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе.

Mistake - ошибка, просчет человека

Fault - дефект, изъян в коде, образовавшийся в результате **Mistake**

Failure - неисправность, отказ, сбой. Внешнее проявление **Fault**. Когда программа натыкается на Fault и выдает неверный результат

Error - невозможность выполнить задачу вследствие отказа **Failure**. Отказ может быть вызван внешними силами

Тестовый случай - Input-Processing-Output. Должен быть повторяемым.

- **Входные значения** - данные или управляющие последовательности.
- **Выполнение** (предусловия, условия выполнения, постусловия)
- **Ожидаемый результат:**
    - выходные данные или изменения состояния, или другие последствия
    - должен быть определен до запуска

Тестовый сценарий - набор тестовых случаев

2. Цели и принципы тестирования (ISTQB).

Цели тестирования:

- Обнаружение дефектов
- Повышение уверенности в уровне качества
- Предоставление информации для принятия решений
- Предотвращение дефектов

Принципы тестирования:

1. Тестирование демонстрирует наличие дефектов (их отсутствие показать нельзя) Тестирование может показать наличие дефектов в программе, но не доказать их отсутствие. Тем не менее, важно составлять тест-кейсы, которые будут находить как можно больше багов. Таким образом, при должном тестовом покрытии, тестирование позволяет снизить вероятность наличия дефектов в программном обеспечении. В то же время, даже если дефекты не были найдены в процессе тестирования, нельзя утверждать, что их нет.

2. Исчерпывающее тестирование недостижимо (полное тестовое покрытия не достичь) Невозможно провести исчерпывающее тестирование, которое бы покрывало все комбинации пользовательского ввода и состояний системы, за исключениям совсем уж примитивных случаев. Вместо этого необходимо использовать анализ рисков и расстановку приоритетов, что позволит более эффективно распределять усилия по обеспечению качества ПО.

3. Раннее тестирование (чем раньше тем лучше) Тестирование должно начинаться как можно раньше в жизненном цикле разработки программного обеспечения, и его усилия должны быть сконцентрированы на определенных целях.

4. Скопление дефектов (несколько модулей содержат основную массу ошибок) Есть сложные куски программы. Дефекты в основном в них. Модули сами по себе сложные, поэтому все баги в них Разные модули системы могут содержать разное количество дефектов – то есть, плотность скопления дефектов в разных элементах программы может отличаться. Усилия по тестированию должны распределяться пропорционально фактической плотности дефектов. В основном, большую часть критических дефектов находят в ограниченном количестве модулей. Это проявление принципа Парето: 80% проблем содержатся в 20% модулей.

5. Парадокс пестицида (если часто проводить правки, то тесты со временем ломаются и их нужно чинить) (жуки привыкают к пестициду) Прогоняя одни и те же тесты вновь и вновь, Вы столкнетесь с тем, что они находят все меньше новых ошибок. Поскольку система эволюционирует, многие из ранее найденных дефектов исправляют и старые тест-кейсы больше не срабатывают. Чтобы преодолеть этот парадокс, необходимо периодически вносить изменения в используемые наборы тестов, рецензировать и корректировать их с тем, чтобы они отвечали новому состоянию системы и позволяли находить как можно большее количество дефектов.

6. Тестирование зависит от контекста (если речь о больничном софте, то нужно лучше тестировать. Также тесты не должны быть сложнее, чем реальные кейсы, то есть проверяем на реальных данных). Выбор методологии, техники и типа тестирования будет напрямую зависеть от природы самой программы. Например, программное обеспечение для медицинских нужд требует гораздо более строгой и тщательной проверки, чем, скажем, компьютерная игра. Из тех же соображений, сайт с большой посещаемостью должен пройти через серьезное тестирование производительности, чтобы показать возможность работы в условиях высокой нагрузки.

7. Заблуждение об отсутствии ошибок. Обнаружение и исправление дефектов не помогут, если созданная система не подходит пользователю и не удовлетворяет его ожиданиям и потребностям.

3. Основная цель тестирования. Уровень доверия, корректное поведение, реальное окружение.

Основная цель тестирования - увеличение приемлемого уровня пользовательского доверия в том, что программа функционирует корректно во всех необходимых обстоятельствах

- Уровень доверия
- Корректное поведение
- Необходимые обстоятельства - требование реального окружения

- **Уровень доверия**
    - Наглядность
    - Уровень остаточного обнаружения дефектов
        - Число дефектов обнаруженных тестом или набором тестов
        - Число дефектов обнаруженных в заданное время

      Пример:
      Меньше 10-ти критических дефектов найдено за последние 7 дней.

    - Требования к надежности
        - Сложно показать без испытаний, то есть работающего ПО

      Пример:
      Среднее время между отказами не должно быть меньше 5000 часов.

- Корректное поведение
  Для определения корректного поведения необходимо определение:
    - Из требований
    - Из спецификаций
    - Зависит от уровня тестирования

- Реальное окружение
  Реальное окружение включает в себя:
    - Реалистичное количество данных - таких же, как в целевой системе
      Например:
      В университете 5000 студентов, небольшой рост.
      Поэтому необходим тест на 5000, 6000, 7000 студентов, но не на 10000!

    - Реалистичный набор, комбинация входных данных

4. Тестирование и качество. Уровни восприятия тестирования в компании.

Уровень 0 - тестирование == отладка 1.

1. Не отличает некорректное поведение и ошибки в программе
2. Не учитывает требования надежности и безопасности

Уровень 1 - предназначение – показать корректность ПО

Как правило, на данном уровне нет формальных правил написания тестов

Уровень 2 - Демонстрация ошибок

На данном уровне может происходить конфликт разработчиков и тестировщиков

Уровень 3 - Тестирование может показать наличие ошибок

1. Используя ПО мы подвержены рискам
2. Риск – последствия незначительные
3. Риск – последствия катастрофические
4. Тестировщики и разработчики совместно снижают риски

Уровень 4 - Тестирование - это возможный способ оценки качества программного обеспечения в терминах найденных дефектов

Способы оценивания качества:

- Разработка стандартов
- Обучение
- Анализ дефектов

5. Участники тестирования, их роль, квалификация и обязанности.

- Проектирование тестов. Основываясь на сценарии, словесное описание тестов.
    - На основании формальных критериев
    - На основании знаний предметной области, опыта и экспертизы
- Автоматизация тестов - программист то бишь.
    - Знание средств, скриптов
- Исполнение тестов. Ручками исполняет, может делать любой.
    - Нет специальных требований квалификации
- Анализ результатов. Просто анализирует полученные результаты.
    - Знания предметной области

Роли:

- Test manager - определение объёмов тестирования, стратегии, списания
- Test designer - анализ системы, определение тестовых случаев, обзор тестового покрытия
- Test engineer - вся грязная работа. Создание тестов, прогон тестов, заполнение отчетов об ошибках.

6. Мониторинг прогресса и контроль тестирования (ISTQB)

Целью мониторинга тестирования - является обзор процесса тестирования и предоставление результата заинтересованным лицам. Информация отслеживается вручную или автоматически и может быть использована для измерения критериев выхода, таких как покрытие. Метрики также могут быть использованы для оценки прогресса тестирования по сравнению с запланированным расписанием и бюджетом. Метриками могут являться тестовое покрытие, количество пройденных тестов, количество найденных дефектов и т.д.

Контроль тестирования описывает любые направляющие или корректирующие действия, принятые как результат по полученной и собранной информации и значениям метрик в результате мониторинга. Контроль тестирования может затрагивать любые действия по тестированию, а также воздействовать на другие действия и задачи жизненного цикла ПО. Такими действиями могут быть правильная приоритизация усилий тестирования, привлечение большего количества ресурсов на тестирование, уменьшение объема предстоящего релиза и т.д.

7. Модульное тестирование. Понятие модуля. Драйверы и заглушки.

Модульное тестирован - это процесс в программировании, позволяющий проверить модули исходного кода программы.

Цель модульного тестирования - изолировать отдельные части программы и показать их работоспособность.

В ходе модульного тестирования решаются задачи:

- Поощрение изменений - позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно
- Упрощение интеграции - устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию “снизу вверх”
- Документирование кода - тесты можно рассматривать как “живой документ”
- Отделение интерфейса от реализации

Понятие модуль (модуль программы) - компонент минимального размера, который может быть независимо протестирован в ходе верификации программной системы

Модуль может быть одним из:

- Модуль - это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований
- Модуль - это программный модуль, т.е. Минимальный компилируемый элемент программной системы
- Модуль - это задача в списке задач проекта (с точки зрения его менеджера).
- Модуль - это один класс или их множество с единым интерфейсом.

Заглушка - часть программы, которая симулирует обмен данными с тестируемым компонентом, выполняет имитацию рабочей системы.

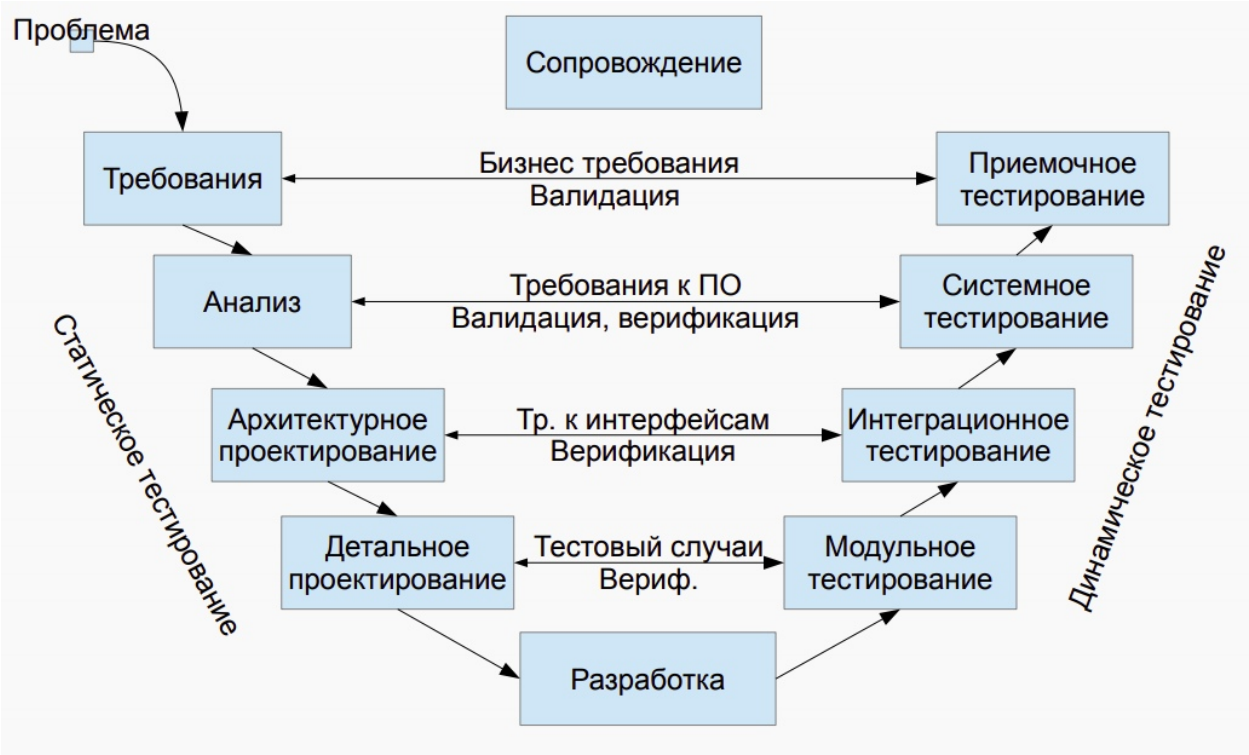
- Эмулирует поведение подчиненной программы
    - Подпрограмма, функция, процедура
    - Аппаратное прерывание, передача данных
- Интерфейс совпадает, внутренность нет
    - Предопределенные ответы для заданных аргументов, исключения
    - Постоянная генерация прерываний
- Используются вместо реальной программы
    - Компилируется и линкуется
- Простая
    - Нельзя тестировать заглушку!
- Обычно 1 строка кода
- Возможна дополнительная логика
    - 50 раз возвращать 42, на 51 - IndexOutOfBounds
- Может читать значения из текстового файла
- Возможна настройка драйвером перед выполнением

Драйвер - определенный модуль теста, который управляет тестируемым нами элементов.

- Эмулирует вызывающий модуль
- Обычно уже более сложная программа
    - Устанавливает окружение
    - Подготавливает входные данные
- Дополнительно может:
    - Запускать серию тестов
    - Настраивать заглушки
    - Формировать журнал результатов

8. V-образная модель. Статическое и динамическое тестирование.

V-образная модель - описывает подход к разработке приложений, при котором тестирование ведется параллельно с разработкой на каждом из её этапов.



Преимущества V-модели:

- Тестировщики начинают работу на ранних этапах разработки → некоторые дефекты будут обнаружены раньше
- Тестирование включено в каждый этап жизненного цикла
- Заранее пишутся тестовые планы и сценарии → все готово при старте динамического тестирования

Недостатки V-модели:

- К старту проекта все команды должны быть сформированы - большое стартовое вложение
- Очень большое количество документации и работы с ней (особенно, если требования начинают меняться)

Статическое тестирование:

- Не включает выполнения кода
- Ручное, автоматизированное
- Неформальное, сквозной контроль, инспекция

Динамическое тестирование:

- Запуск модулей, групп модулей, всей системы
- После появления первого кода (а иногда перед!)

9. Валидация и верификация. Тестирование методом "чёрного" и "белого" ящика.

Верификация - это проверка на соответствие правилам. Правила оформляются в виде документа. То есть, должен быть документ с требованиями к документации. Если документация соответствует требованиям этого документа, то она прошла верификацию.

Валидация - это проверка правильности выводов. То есть, должен быть свод знаний, в котором описано, как получить описание конструкции на основе данных об объекте. Проверка правильности применения этих выводов - есть валидация. В том числе это проверка описания на непротиворечивость, полноту и понятность.

Методы “черного ящика”
Метод тестирования при котором не используется знания о внутреннем устройстве тестируемого объекта

- Спецификации, требования, дизайн
- Запуск и сравнение результатов с эталоном

Методы “белого ящика”
Тестирование кода на предмет логики работы программы и корректности её работы. Тестирование позволяет проверить внутреннюю структуру программы. Исходя из этой стратегии тестировщик получает тестовые данные путем анализа логики работы программы.

- Переходы, утверждения, условия
- Анализ путей, структуры

Валидация → черный ящик
Верификация → белый ящик

10. Тестовый случай, тестовый сценарий и тестовое покрытие.

Тестовый случай (Input → Processing → Output)
Имеет:

- Входные значения. Данные или управляющие воздействия
- Предусловия, условия выполнения, постусловия
- Ожидаемый результат
    - Выходные данные и состояния, изменения в них, и другие последствия теста
    - Определен до запуска теста

Характеристики:

- Повторяемый, автоматизируемый
- Учитывает состояния (если есть)
    - Переходы между состояниями
        - Правильные: Корректный результат
        - Неправильный: Корректные сообщения об ошибках

Тестовый сценарий - это последовательность тестовых случаев. Показывает типичное использование системы. Также должен обрабатывать не только корректное поведение, но и вариант ошибки, вроде НЕВЕРНОГО pin кода и блокировать карточку после 3-х раз.

Свойством является определение корректного поведения в:

- Требованиях (системное, приемочное тестирование)
- Архитектуре (интеграционное тестирование)
- Проектных документах (модульное тестирование)

Тестовое покрытие - это одна из метрик оценки качества тестирования, представляющая собой плотность покрытия тестами требований либо исполняемого кода.
Полное покрытие к сожалению невозможно :(
Выбор может производиться с помощью следующих методов:

- Эквивалентное разбиение (партиции эквивалентности)
- Таблица решений (альтернатива)
- Таблица переходов
- Сценарии использования

11. Полное тестовое покрытие. Оценка объема и времени полного покрытия.

Полное тестовое покрытие недостижимо, поэтому тестовое покрытие выбирается с помощью определенных методик:

- Эквивалентное разбиение (партиции эквивалентности)
- Таблица решений (альтернатива)
- Таблица переходов
- Сценарии использования

Необходимо балансировать между количеством тестов, стоимостью и скоростью разработки. Чем больше тестов - тем меньше дефектов, но зависимость не линейная (с каждым новым тестом обнаруживается все меньше ошибок). Но возрастает стоимость и время разработки.

12. Повторяемость тестового сценария. Автоматизированное тестирование. Регрессионное тестирование.

Повторяемость - все написанные тесты всегда будут выполняться однообразно.

Автоматическое тестирование - тестирование с помощью программных средств, аналогично ручному тестированию.

- Регрессионное тестирование
- Повторение тестового сценария
- Приемочное тестирование
- Сокращение ручного труда
- Проверка одного приложения в разных окружения

Регрессионное тестирование - собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода. Такие ошибки - когда после внесения изменений в программу перестает работать то, что должно было продолжать работать - называют регрессионными ошибками. Регрессионные тесты удобно автоматизировать и запускать после каждого изменения/добавления участка кода, проверяя, что не сломалось ничего из ранее работающего.

Задачи регрессионного тестирования:

- Проверка и утверждение исправления ошибки
- Тестирование последствия исправлений, так как внесенные исправления могут привести к ошибкам
- Гарантировать функциональную преемственность и совместимость
- Уменьшение стоимости и сокращения времени выполнения тестов

13. Цели и задачи интеграционного тестирования. Алгоритм интеграционного тестирования. Стратегии интеграции.

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

- Проверяет интерфейсы и взаимодействие модулей (компонент) или систем
    - Вызовы API, сообщения между ОО компонентами
    - Баз Данных, пользовательских графических интерфейс
    - Интерфейсы взаимодействия (сетевые, аппаратные, локальные …)
    - Инфраструктурные
- Может проводиться когда два компонента разработаны (спроектированы)
    - Остальные добавляются по готовности

Стратегии интеграции

- Больше объем интеграции - больше сложность
    - Для каждого интерфейса должен быть разработан короткий тест план
- Выбор в зависимости от архитектуры ПО
- Последовательность имеет значение
- Можно тестировать нефункциональные характеристики

Алгоритмы интеграции:

- **Сверху вниз** - вначале тестируются все высокоуровневые модули и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня стимулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами.

- **Снизу вверх** - Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного 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

- **Большой взрыв** - Все или практически все разработанные модули собираются в виде законченной системы или ее основной части и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования. Перед запуском обязательно молиться!!!

- **Ядро** - при данном тестировании сначала создаются жизненно необходимые модули, а потом уже идет апгрейд их. Например, экран + клавиатура + мышь - работают с минимальным функционалом. Можно добавить цвет на экран, подсветку для клавиатуры и т.п.

- **Функциональная** - по одной функции. Сценарий проверяет совокупность действий. Например, пользователь заходит на почтовый сайт, листает письма, просматривает новые, пишет и отправляет письмо, выходит с сайта. Такие сценарии могут быть очень сложными и иметь множество вариаций, поэтому данный вид тестирования выполняется обычно в самом конце и считается очень дорогим.

14. Тестирование системы целиком - системное тестирование.

Системное тестирование ПО - это тестирование ПО, выполняемое на полной, интегрированной системе, с целью проверки соответствия системы исходным требованиям. Системное тестирование относится к методам тестирования черного ящика, и, тем самым, не требует знаний о внутреннем устройстве системы.

Задача: проверка как функциональность системы с точки зрения пользователя, так и нефункциональные характеристики.

При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи.

Включает несколько фаз:

- Системное тестирование - выполняется внутри организации-разработчика
- Альфа и Бета тестирование - выполняется пользователем под контролем разработчика
- Приемочное тестирование - выполняется пользователем. Платить за это продукт или нет.

15. Тестирование возможностей, стабильности, отказоустойчивости, совместимости. Тестирование производительности - CARAT.

Относится к нефункциональному тестированию.

Тестирование возможностей - минимальная нагрузка, состоящая из корректных и реальных данных, проверка возможностей и функционала системы

Тестирование стабильности - добавляем нагрузку, данные все ещё корректные. Проверяем как система работает в более-менее реальных условиях

Тестирование отказоустойчивости - пытаемся все сломать к чертям. Некорректные данные, большая нагрузка, сбои питания, восстановление после отказа и т.д.

Тестирование совместимости - запуск с различными версиями библиотек, на различном окружении. Смотрим как система со всем этим работает.

CARAT - подход к нагрузочному тестированию

- **Capacity (Нефункциональные возможности)** - максимальное количество (пользователей, записей в БД, файлов, КБ, ГГц), поддерживаемое системой одновременно, не нарушая других требований производительности.

- **Accuracy (Точность)** - корректность алгоритмов и результатов

- **Responce Time (Время ответа)** - время ответа сервиса при разных видах нагрузки

- **Availability (Готовность)** - способность сервиса обслуживать клиента
    - Коэф. Готовности = (MTBF - MTTR) / MTBF
        - MTBF - среднее время между сбоями
        - MTTR - среднее время восстановления

- **Throughput (Пропускная способность)** - количество операций в секунду, которое может поддерживать система.

16. Альфа и Бета тестирование. Приемочное тестирование.

Альфа и бета-тестирование являются частью системного тестирование. Проводятся пользователями.

Альфа-тестирование - имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком.

- Может проводиться до окончания системного тестирования
- Ранние отзывы пользователей об использовании системы в рабочем окружении

Бета-тестирование - в некоторых случаях выполняется распространение предварительной версии (в случае проприетарного программного обеспечения иногда с ограничениями по функциональности или времени работы) для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

- Могут быть ошибки
- Может быть не реализован весь функционал
- Ранние отзывы для разработчиков
- Превью для пользователей

Приемочное тестирование - формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью: определения удовлетворяет ли система приемочным критериям; вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

17. Статическое тестирование. Рецензия, технические анализ, сквозной контроль.

Статическое тестирование - анализ артефактов разработки программного обеспечения, таких как требования или программный код, проводимый без исполнения этих программных артефактов. В процессе статического тестирования, программные продукты рассматриваются оцениваться вручную, проводятся ревью или с помощью набора инструментов, сам код при это не запускается. Проводится перед динамическим тестированием и соответственно ошибки найденные на этом этапе обходятся нам дешевле.

Рецензирование (Review) - оценка состояния продукта или проекта с целью установления расхождений с запланированными результатами и для выдвижения предложений по совершенствованию.

- Основное предназначение - поиск дефектов
- Дополнительная цель - улучшение процесса
- Подготовка - формальная подготовка
- Ведущий - подготовленный модератор
- Рекомендованный размер группы: 3-6
- Формальная процедура - всегда
- Объем материалов - небольшой
- Сбор метрик - всегда
- Выходные данные - список дефектов, результаты метрик, формальный отчет

Технический анализ - проверка продукта на соответствие и практическую пригодность.

- Основное предназначение - поиск дефектов
- Дополнительная цель - принятие решений
- Подготовка - популяризация
- Ведущий - в зависимости от обстоятельств
- Рекомендованный размер группы: >3
- Формальная процедура - иногда
- Объем материалов - от среднего до большого
- Сбор метрик - иногда
- Выходные данные - формальный отчет

Сквозной контроль - представляет собой один из видов формального пересмотра артефактов методом “мозгового штурма”, который может проводиться на любом этапе разработки. Это дружественная встреча разработчиков, тщательно спланированная, с ясно определенными целями, повесткой дня, продолжительностью и составом участников.

- Основное предназначение - поиск дефектов
- Дополнительная цель - обмен опытом
- Подготовка - обычно нет
- Ведущий - автор

- Рекомендованный размер группы: 2-7
- Формальная процедура - обычно нет
- Объем материалов - небольшой
- Сбор метрик - обычно нет
- Выходные данные - неформальный отчет

18. Статическое тестирование. Инспекции.

Статическое тестирование - анализ артефактов разработки программного обеспечения, таких как требования или программный код, проводимый без исполнения этих программных артефактов. В процессе статического тестирования, программные продукты рассматриваются оцениваться вручную, проводятся ревью или с помощью набора инструментов, сам код при это не запускается. Проводится перед динамическим тестированием и соответственно ошибки найденные на этом этапе обходятся нам дешевле.

Инспекция ПО - это статическая проверка соответствия программы заданным спецификациями, проводится путем анализа различных представлений результатов проектирования (документации, требований, спецификаций, схем или исходного кода программы) на процессах ЖЦ.

Четко определенные шаги:

- **Вход** - постановка проблемы/цели
    - Входные критерии - определяются перед началом (проверяются ведущим)
- **Планирование** - определение сессий (проводимых тестов)
    - Определение объемов (зависит от артефакта, проводится с учетом check rate'a - сколько один человек может проверить за время инспекции)
    - Отбор инспекторов
    - Сбор метрик (цена изменений, время на проверку и т.д.)
    - Разрабатывание расписания
- **Обзор**
    - Обучение инспекторов
    - Назначение ролей
    - Распространение материалов

Шаги 4-7 являются одной сессией и могут повторяться

- **Подготовка**
    - Учет времени (отмечается начало, используется check rate)
    - Выделение проблем (с фокусом на основные), их классификация и подсчет
- **Обсуждение**
    - Сбор и протоколирование результатов подготовки
    - Решение по артефакту: принят/на переработку/отклонен и др.
- **Переработка** - может осуществляться автором/редактором/другим работниками
- **Выработка рекомендаций**
- **Выход**
    - Инспекция не может завершиться, пока не удовлетворены выходные критерии
    - Согласование документов всеми инспекторами

Роли инспекторов могут быть выбраны из следующих:

- Чек-лист — инспектор проверят по нему
- Документы — инспектор проверяет целостность между несколькими документами
- Фокус — поиск выделенных проблем
- Перспектива — представить роль и будущее пользователя
- Процедура — инспектор следует особой процедуре (.)
- Сценарий — следование заданному сценарию (более специфично, чем предыдущий)
- Стандарт — проверка на соответствие стандартам
- Точка зрения — инспектирует с точки зрения, например, пользователя

19. Статическое тестирование. Статический анализ кода.

Цель статического тестирования - нахождение дефектов в коде или моделях ПО. Фактически статический анализ - это исследование ПО с помощью специальных инструментов без его запуска, при динамическом тестировании ПО требуется запуск кода. Статический анализ выявляет дефекты, которые сложно найти при динамическом тестировании.

Преимущества статического анализа:

- Ранее обнаружение дефектов до исполнения тестов
- Ранее предупреждение о подозрительных аспектах в коде или дизайне с помощью вычисления метрик, таких как коэффициент сложности
- Определение дефектов, которые сложно обнаружить с помощью динамического тестирования
- Определение зависимостей и нарушений целостности в моделях ПО, например ссылок.
- Улучшение пригодности к сопровождению кода и дизайна
- Предотвращение дефектов путем усвоения уроков, полученных во время разработки

20. Выбор тестового покрытия с помощью анализа эквивалентности. Анализ граничных значений.

Тестовое покрытие - это одна из метрик оценки качества тестирования, представляющая собой плотность покрытия тестами требований либо исполняемого кода. Полное тестовое покрытие недостижимо из-за большого (возможно бесконечного) количество значений, которые могут быть поданы на вход.
Требуется выбирать для тестирования специфические значения / комбинации, которые определяет в итоге конечный набор тестов -> тестовое покрытие. Один из способов - эквивалентное разбиение + анализ граничных значений.

Эквивалентное разбиение - если вывод системы одинаковый, то входящие значения - в одной эквивалентной области. Определяет партиции эквивалентности: наборы значений, для которых поведение системы определено как одинаковое (с точки зрения требований например)
В каждой партиции проверяется только одно значение (может быть выбрано произвольно). Также применяется вместе с анализом граничных значений.

Недостаток: мы можем сказать, что два значения принадлежат к одной партиции, а на самом деле система возьмет и поведет себя по-разному, но мы это не проверим - упущенный баг. Так что тестировать одно значение - не самый хороший вариант. Если тесты автоматизированы, выполняются быстро и наборы значений конечен - лучше протестировать все.

Анализ граничных значений - определение партиций эквивалентности включает в себя определение граничных значений партиции. Это могут быть конкретные значения или бесконечные (обычно не учитываются).
Базовы подход (протестировать на границах, на приграничных значениях и на каком-нибудь значении из области)

Недостатки: не всегда легко определить границы (легко: int значения, нелегко: номер телефонов), спецификация не всегда описывает граничные значения, длину поля.

21. Выбор тестового покрытия с помощью таблицы решений.

Тестовое покрытие - это одна из метрик оценки качества тестирования, представляющая собой плотность покрытия тестами требований либо исполняемого кода. Полное тестовое покрытие недостижимо из-за большого (возможно бесконечного) количество значений, которые могут быть поданы на вход.

Одним из способов определения какие комбинации значений надо проверить - **таблица решений**. Можно сказать: один из вариантов составления тест-кейсов. Состоит из (input) Условия и (output) Действия - берутся из требований, по факту - реакция системы. Таблица помогает определить минимальное количество тестов, покрывающее все возможные варианты комбинаций исходных условий. Используется в системах со сложной логикой, представляет собой описание конечного автомата.

Пример проверки поля “Пароль”. T - true, F - False, - можно не проверять, V - ок, X - не ок

Условия					
Состоит из 12 и больше символов	T	F	T	F	T
Содержит буквы и цифры	T	T	F	F	T
Не совпадает с предыдущим	T	–	–	–	F
Действия					
Пароль действительный	V	X	X	X	X

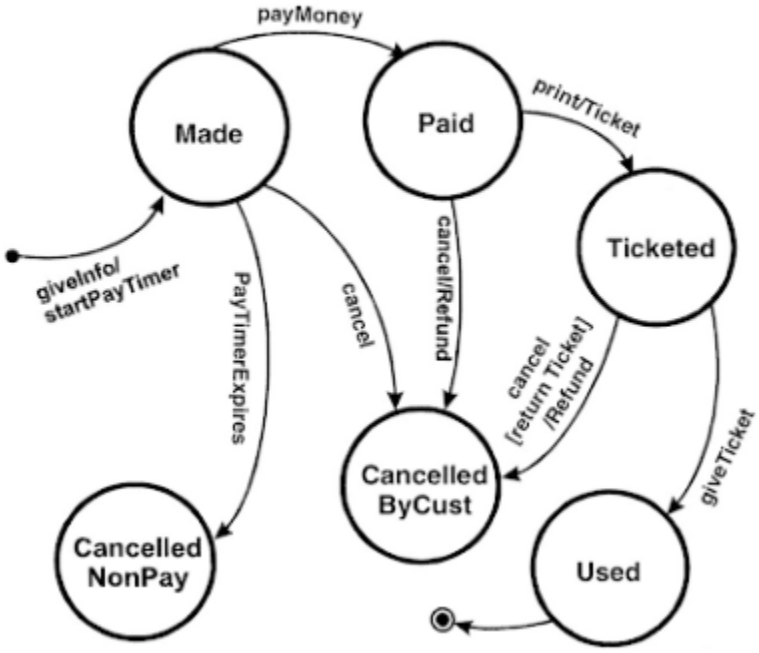
Упрощение (сокращение количества комбинаций) произошло за счет того, что условие “не совпадает с предыдущим” можно не проверять, если одно из предшествующих условий false.

22. Выбор тестового покрытия с помощью диаграммы состояний и таблицы переходов.

Тестовое покрытие - это одна из метрик оценки качества тестирования, представляющая собой плотность покрытия тестами требований либо исполняемого кода. Полное тестовое покрытие недостижимо из-за большого (возможно бесконечного) количество значений, которые могут быть поданы на вход.

Одним из способов определения какие комбинации значений надо проверить - **таблица переходов**. Описывает смену состояний системы. Определены все события, которые возникают во время работы приложения, и как приложение реагирует на эти события.

Пример диаграммы состояний



Ноды - состояния, над стрелками - события (то что заставляет систему сменить состояние) / действия (было инициировано сменой состояния). По идее диаграмма не закончена, так как не все состояния имеют пути до точки выхода.
По полученной диаграмме состояний можно составить таблицу переходов.

Таблица переходов состоит из четырех столбцов:

Текущее состояние	Событие	Действие	Следующее состояние
null	giveInfo	startPayTimer	Made
null	payMoney	--	null
null	print	--	null
null	giveTicket	--	null
null	cancel	--	null
null	PayTimerExpires	--	null

… Продолжение таблицы ….

Для заполнения таблицы переходов берутся все состояния и все события / действия, далее применяется декартово произведение (сочетание каждого с каждым). Каждая пара состояние + событие / действие - одна строка в таблицы. Если такое событие / действие для данного состояния невозможно, в следующее состояние вносится текущее состояние (в примере так, но вообще это не совсем корректно) или “Неопределенно” (так корректнее).

Можно ограничиться тестирование валидных комбинаций (переход в состояние определен), но если есть время - стоит покрыть тестами и невалидные (переход в “Неопределенно”) кейс - негативное тестирование.

23. Выбор тестового покрытия с помощью функционального тестирования.

Функциональное тестирование проводится на основе сценариев использования системы, соответственно проверяются функции системы, начиная с интерфейса пользователя. Какой-то устоявшийся функционал может автоматизироваться, при добавлении нового функционала он изначально проверяется вручную, далее может быть также автоматизирован.

Например:
У нас есть список, по которому можно делать поиск. И на это у нас есть автоматизированные тесты. Тут программист решил сделать ещё фильтрацию по тегам. Сначала он проверяет в ручную, что фильтрация происходит, а уже после пишет автоматизированные тесты, чтобы удостовериться в своей правоте.

Одной из программ для функционального тестирования web-сайтов является Selenium

24. Библиотека JUnit. Класс junit.framework.Assert. Основные аннотации для исполнения тестов.

JUnit - это open-source фреймворк (где-то пишут что библиотека) для модульного тестирования. Последняя версия является 5

Возможности JUnit:

- Тесты отмечаются с помощью аннотации @Test (есть расширения типа @ParametrizedTest)
- Набор методов для проверки утверждений - junit.framework.Assert
- Аннотации для маркировки действия до / после выполнения теста (@Before / @BeforeEach и @After @AfterEach)
- Вывод результатов, журнал тестов, UI
- Отключение конкретных тестов (@Disabled или @Ignore)
- Запуск тестов с разными типами параметров
- Тэги / фильтрация / группировка тестов для выполнения

Класс junit.framework.Assert - предоставляет методы для тестирования и сравнения объектов. Например, через assertEquals(1, getNum()) можно проверить что значение переданное совпадает с запланированным результатом.
assertTrue и assertFalse - проверят на boolean значения
assertNull - проверка значения на null и т.д.
fail() - фейлит тест. Например, если мы ещё не придумали тест, а нужно чтобы он как-то выполнился.

Пример шаблона для теста:

```
@Test
public void checkSomething(){
	assertAll(
		() -> assertEquals(1, getNum()),
		() -> assertTrue(true)
	);
}
```

assertAll - нужен для того, чтобы выполнились все ассерты, даже если один из них фейланется.

25. Библиотека JUnit. Дополнительные возможности, запуск с параметрами.

JUnit - это open-source фреймворк (где-то пишут что библиотека) для модульного тестирования. Последняя версия является 5

Возможности JUnit:

- Тесты отмечаются с помощью аннотации @Test (есть расширения типа @Parametrized)
- Набор методов для проверки утверждений - junit.framework.Assert
- Аннотации для маркировки действия до / после выполнения теста (@Before / @BeforeEach и @After @AfterEach)
- Вывод результатов, журнал тестов, UI
- Отключение конкретных тестов (@Disabled или @Ignore)
- Запуск тестов с разными типами параметров
- Тэги / фильтрация / группировка тестов для выполнения

Дополнительные возможности:
Например, аннотации @Test можно задать дополнительные параметры. Указать класс исключения @Test(expected = NullPointerException.class) и тестировать метод на исключения.
Указать таймаут: @Test(timeout = 1) значит тест будет завален, если выполняется больше 1 мс

Также можно группировать тесты, это делается с помощью классов. Каждый класс = 1 группа. Чтобы сделать подгруппу, создается класс внутри другого класса.

Параметризованные тесты
Значит тест будет у нас с определенными входными параметрами, они могут быть считаны например из CSV файла или перечислены в коде вручную.
@ParametrizedTest
@ValueSource(ints = {1, 2, 3, 4}) - где ints - указывается тип
public void test(int arg){

}

26. Анализ эквивалентности с использованием JUnit.

Анализ эквивалентности: вместо тестирования всех возможных значений определяются группы входных параметров, которые одинаково влияют на систему. В каждой группе выделяются граничные значения, для которых составляются тестовые сценарии. Поскольку для каждой группы граничных значений результат воздействия на систему будет схожий, можно использовать параметризованные тесты.

Но все равно JUnit не предоставляет возможности выполнить анализ эквивалентности на входных значениях, это задача уже разработчика тестов.

```
@ParametrezideTest
@ValuesSource(ints = {-1, 0, 1, Integer.MAX_VALUE})
void test (int num){
        assertTrue(Alg.checkIsNegative(num));
}
```

27. Тестирование алгоритмов с использованием JUnit.

JUnit - это open-source фреймворк (где-то пишут что библиотека) для модульного тестирования. Последняя версия является 5

Возможности JUnit:

- Тесты отмечаются с помощью аннотации @Test (есть расширения типа @Parametrized)
- Набор методов для проверки утверждений - junit.framework.Assert
- Аннотации для маркировки действия до / после выполнения теста (@Before / @BeforeEach и @After @AfterEach)
- Вывод результатов, журнал тестов, UI
- Отключение конкретных тестов (@Disabled или @Ignore)
- Запуск тестов с разными типами параметров
- Тэги / фильтрация / группировка тестов для выполнения

Для тестирования алгоритмов удобнее всего использовать @ParametrizedTest проверяя пары входных и выходных значений. Ещё в этом случае удобно считывать данные из файла, что не изменять код, а только подавать значения (методом черного ящика)
@CSVFileSource(files = “file.csv”)

Для тестирования алгоритма в изоляции от зависимостей, можно применять заглушки. Их можно реализовать средствами языка, однако удобнее использовать специальные библиотеки, облегчающие их написание, например Mockito.

```
@ParametrizedTest
@ValueSource(ints = 0)
public void test (int result){
        Student student = Mockito.mock(Student.class)
        Mockito.when(student.getDolgs()).thenReturn(result);
        assertEquals(“Обучается”, ISU.getStatus(student))
}
```

28. Модульное тестирование доменной модели с использованием JUnit.

Задача модульного тестирования является проверка работоспособности модуля, изолированного от других модулей. Так как в доменной модели модули обычно связаны во время исполнения, целесообразно использовать заглушки, для реализации которых служит библиотека Mockito

В остальном тестирование модуля доменной модели с точки зрения написания кода не очень отличается от тестирования алгоритма - если результаты работы модуля хорошо описываются таблицей решений или парами входных и выходных значений, целесообразно использовать параметризованные тесты.

Помимо этого, часто целесообразно не создавать инстанции тестируемого модуля в каждом отдельном тесте, а тестировать одну инстанцию, которая инициализируется в методе с аннотацией @BeforeAll и деинициализируется в методе с аннотацией @AfterAll

Часто доменная модель предполагает обработку пользовательских (или стандартных) исключений. Для этого в JUnit существует метод assertThrows, который принимает ссылку на класс исключения и анонимную функцию или инстанцию Executable. И возвращает полученное исключение, параметры которого можно проверять в дальнейшем.

```
@Test
void throwsOnInvalidPhoneNumber() {
  ContactBook contactBook = Mockito.mock(ContactBook.class);
  Mockito.when(contactBook.getNumberByName("Mom")).thenReturn("abcdefg");
  Phone phone = new Phone(contactBook);
  Exception exception = assertThrows(IllegalCharacterInNumberException.class, () -> {
    phone.call("Mom");
  });
  String expectedMessage = "Illegal character 'a', the phone number should contain only digits, '-' or '+' characters";
  String actualMessage = exception.getMessage();
  assertTrue(actualMessage.contains(expectedMessage));
}
```

29. Система Selenium. Архитектура, основные команды написания сценариев.

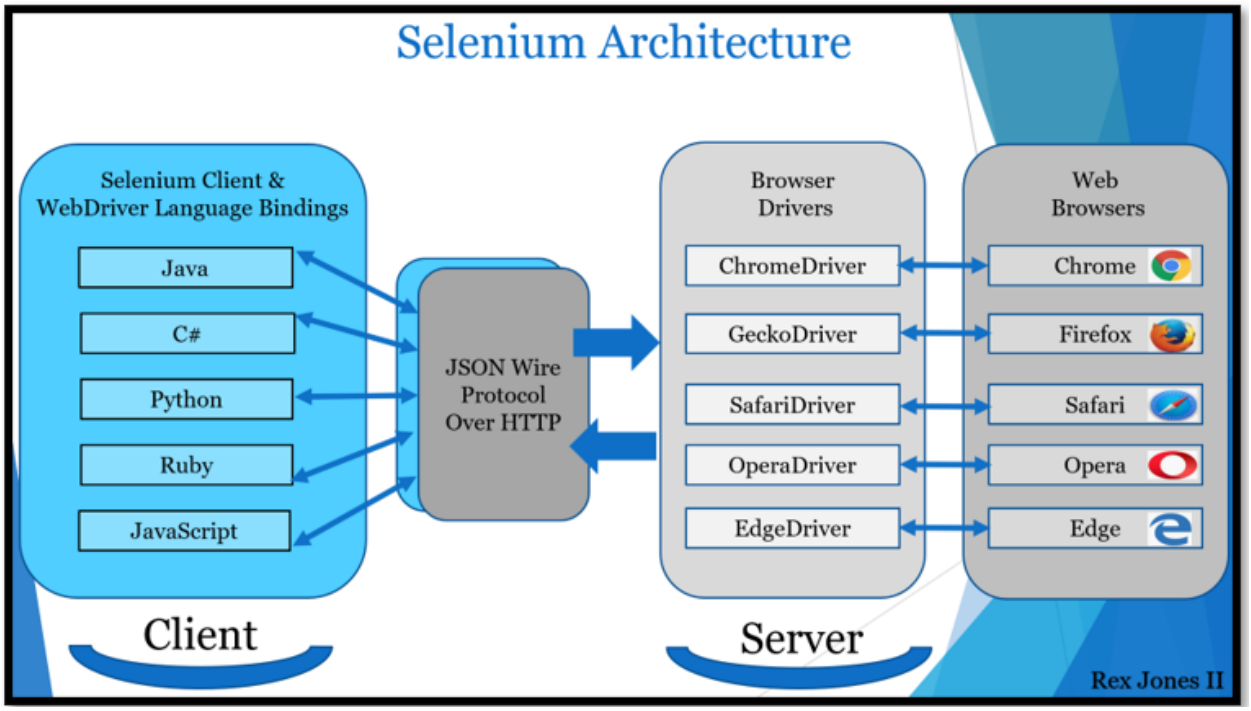
Система Selenium - набор средств автоматизации

- IDE
- WebDriver
- Grid

Предназначен для тестирования web-приложений. Является кроссбраузерным (использует драйвера браузеров, чтобы с ними взаимодействовать. Обычно для корректной работы ещё нужно, чтобы версия вашего браузера совпадала с версией драйвера иначе будет ошибка при попытке запуска).
Позволяет разрабатывать сценарии на многих языках:

- Python
- Java
- Php
- C#

Встроенные конструкции assert и механизм логирования ошибок.



Selenium Grid - позволяет запускать тесты на разных машинах в разных браузерах да и к тому же параллельно. В нем присутствует HUB и NODE.
HUB - это центральная точка, которая принимает запросы и направляет их к NODE. Так скажем командный пункт. В GRID может быть ТОЛЬКО ОДИН хаб.
NODE - selenium инстанс, который запускает команды, из HUB'а. Они могут запускаться на разных операционных системах с разными браузерами.

Selenium WebDriver - это программная библиотека для управления браузерами. WebDriver представляет собой семейство драйверов для различных браузеров плюс набор клиентских библиотек для этих драйверов на разных языках программирования. HTTP клиент шлет запрос к веб драйверу, он превращает их в веб сокетные и шлет браузеру. Браузер это дело обрабатывает

Selenium IDE - расширение для браузера. Позволяет тречить состояние тестов. Записывать действия.

Основные команды:

- click clickAndWait - для нажатия на кликабельные объекты (ссылки, кнопки)
- type - ввод значений
- select - выбор значений из списка
- open - открытия определенной страницы
- assert***
- wait*** - ожидание события
- verify*** - проверка

30. Система Selenium. Assertion & Verification. Команды. Команды wait**.

Система Selenium - набор средств автоматизации

- IDE
- WebDriver
- Grid

Предназначен для тестирования web-приложений. Является кроссбраузерным (использует драйвера браузеров, чтобы с ними взаимодействовать. Обычно для корректной работы ещё нужно, чтобы версия вашего браузера совпадала с версией драйвера иначе будет ошибка при попытке запуска).
Позволяет разрабатывать сценарии на многих языках:

- Python
- Java
- Php
- C#

Встроенные конструкции assert и механизм логирования ошибок.

Assertion and Verification:
Эти команды используются для проверки содержимого элемента интерфейса. Можно проверить текст страницы, присутствие элемента, наличие его атрибутов.

Если падает Verification - тест ПРОДОЛЖАЕТСЯ
Если падает Assertion - тест ФЕЙЛИТСЯ

Команды Verification:

- verifyTextPerent
- verifyTittle
- verifyElementPresent
- verifyValue

Команды Assertion (те же самые):

- assertTextPerent
- assertTittle
- assertElementPresent
- assertValue

Команды wait:

- waitForPageLoad(timeout) - загрузка страницы ошибка по таймауту
- waitForAlert
- waitForTable - полная загрузка таблицы
- waitForTittle - загрузка заголовка

31. Система Selenium. Selenium RC, WebDriver, Grid.

Selenium - это инструмент для автоматизированного управления браузерами. Наиболее популярной областью применения Selenium является автоматизация тестирования веб-приложений.

Selenium WebDriver – это программная библиотека для управления браузерами. Часто употребляется также более короткое название WebDriver. Иногда говорят, что это «драйвер браузера», но на самом деле это целое семейство драйверов для различных браузеров, а также набор клиентских библиотек на разных языках, позволяющих работать с этими драйверами.

Selenium RC – это предыдущая версия библиотеки для управления браузерами. Аббревиатура RC в названии этого продукта расшифровывается как Remote Control, то есть это средство для «удалённого» управления браузером. Эта версия с функциональной точки зрения значительно уступает WebDriver. Сейчас она находится в законсервированном состоянии, не развивается и баги не исправляются. Всем, кто сталкивается с ограничениями Selenium RC, предлагается переходить на использование WebDriver

Selenium Grid – это кластер, состоящий из нескольких Selenium-серверов. Он предназначен для организации распределённой сети, позволяющей параллельно запускать много браузеров на большом количестве машин. Одна из задач Selenium Grid заключается в том, чтобы «подбирать» подходящий узел, когда во время старта браузера указываются требования к нему – тип браузера, версия, операционная система, архитектура процессора и ряд других атрибутов. Ранее Selenium Grid был самостоятельным продуктом. Сейчас физически продукт один – Selenium Server, но у него есть несколько режимов запуска: он может работать как самостоятельный сервер, как коммутатор кластера, либо как узел кластера, это определяется параметрами запуска.

Selenium IDE – плагин к браузеру Firefox (и не только - прим. Автора т.к. есть для Chrome), который может записывать действия пользователя, воспроизводить их, а также генерировать код для WebDriver или Selenium RC. В котором выполняются те же самые действия. В общем, это «Selenium-рекордер».

32. Язык XPath. Основные конструкции, оси. Системные функции.

XPath - это язык запросов к элементам XML-документа.

Основные конструкции:

Выражение	Результат
div	Выбирает все узлы с именем "div"
/	Выбирает от корневого узла
//	Выбирает узлы от текущего узла, соответствующего выбору, независимо от их местонахождения
.	Выбирает текущий узел
..	Выбирает родителя текущего узла
@	Выбирает атрибуты

/ -- разделитель пути; первый элемент определяет путь от корня (/div -- множество div'ов на первом уровне от корня).
// -- определяет относительный путь (//div -- все div'ы в документе).
[] -- определяет предикат выборки, внутри можно использовать пути, функции и прочее.
| -- union множеств элементов, возвращённых Xpath (/div | //span)

ОСИ - это базы языка XPath. Для некоторых осей существуют сокращенные обозначения.

child:: - Содержит множество элементов-потомков (элементов, расположенных на один уровень ниже).
parent:: - содержит элемент-предок на один уровень назад. Это можно заменить на **..**
self:: - содержит текущий элемент. Это обращение можно заменить на **.**
following:: - содержит множество элементов, расположенных ниже текущего элемента по дереву

Системные функции:

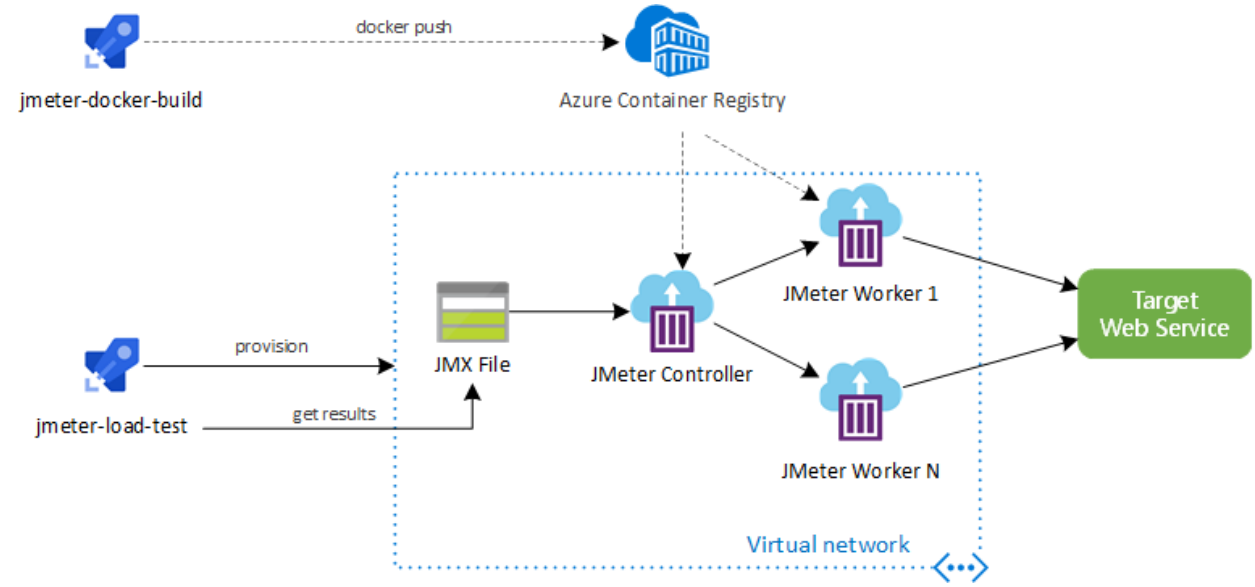
- **document(object, node-set?)** - возвращает документ, указанный в параметре object
- **format-number(number, string, string?)** - формирует число согласно образцу, указанному во втором параметре, третий параметр указывает именованный формат числа, который должен быть учтен
- **generate-id(node-set?)** - Возвращает строку, являющуюся уникальным идентификатором

33. Язык XPath. Функции с множествами. Строковые, логические и числовые функции.

- Функции с множествами
    - count()
    - position()
    - text()
    - last()
- Строковые
    - string-lenght()
    - matches()
    - contains()
    - start-with()
- Логические
    - true()
    - false()
    - not()
    - boolean()
- Числовые
    - sum()
    - round()
    - number()
    - ceiling()
    - floor()

34. Apache JMeter. Архитектура, Элементы тестового плана. Последовательность выполнения.

Архитектура



Тестовый план:

- **Thread Group**
  - Описывает пул пользователей для выполнения теста
  - Количество, возрастание и пр…
- **Семплеры**
  - Формируют запросы, генерируют результаты
  - Большой набор встроенных протоколов (TCP, HTTP, FTP …)
- **Логические контроллеры**
  - Определяют порядок вызова семплеров
  - Конструкции управления (if, loop, …)
  - Управление потоком
- **Слушатели**
  - Получают ответы
  - Осуществляют доп операции с результатами: просмотр, запись, чтение
  - Не обрабатывают данные (в командной строке)
- **Таймеры**
  - Задержки между запросами
  - Постоянные, в соответствии с законами
- **Assertion**
  - Проверяют результат
- **Элементы конфигурации**
  - Сохраняют предустановленные значения для семплеров
- **Препроцессоры**
  - Изменяют семплеры в их контексте (HTML Link Parser)

- **Постпроцессоры**
    - Применяются ко всем семплерам в одном контексте

Порядок выполнения

1. Конфигурационные элементы
2. Препроцессоры (Pre-Processors)
3. Таймеры (Timers)
4. Семплеры (Sampler)
5. Постпроцессоры (Post-Processors)
6. Assertions

35. Apache Jmeter. Дополнительные возможности. Распределенное тестирование.

Дополнительные возможности:

- Автоматическая генерация CSV-файла
- Ограничение полосы пропуска
    - Статическое (CPS), динамическое (error rate)
- IP Spoofing - замена src_ip, для разделения клиентов
    - Проверка балансировщиков нагрузок
- Поддержка теста TPC-C
    - Стандартный тест на OLTP нагрузку
- Генерация HTML документов с подробным анализом тестов.

Распределенное тестирование:

Один компьютер не всегда может эмулировать необходимую в тестовых целях нагрузку, в связи с чем возникает необходимость генерации тестовой нагрузки с нескольких узлов. Для этого необходимо:

- Запустить JMeter - сервер на всех машинах, которые будут генерировать тестовые запросы лййцццццццц
- Отредактировать файл jmeter.properties на JMeter клиенте, который будет запускать тесты. В свойстве remote_hosts необходимо перечислить все хосты Jmeter-серверами
- Запустить тест на клиенте

36. Область деятельности тестирования безопасности. Риски безопасности. Цифровые активы (digital assets). Методы доступа и обеспечения безопасности. Политики безопасности

Область деятельности:

- Фокус - безопасность функциональности, определенной:
  – Спецификациями и требованиями
  – Сценариями использования и моделями
  – Анализом рисков
- Дополнительно внимание уделяется
  – Политики и процедуры безопасности
  – Поведению взломщика/атакующего
  – Известным уязвимостями и дефектам безопасности

Риски безопасности – отдельная группа рисков в разработке и эксплуатации ПО:
– Опасность нарушения конфиденциальности, целостности или доступности информации или информационных систем.
– Экспозиция риска = функция от вероятности наступления и величины ущерба
– Стандартные методы работы с рисками: Оценка (assessment), контроль и управление риском.

Цифровые активы – информация и физические объекты
– Данные пользователей и бизнес-планы
– Разработанное ПО, документация, модели, диаграммы
– Документы, процессы, торговые секреты,
– Налоговая и финансовая информация
– Презентации и обучающие курсы
– Сообщения и Emails
– Информация о работниках
– Прототипы устройств
– Возможность оказывать услуги
– Репутация компании и доверие пользователей

Типичные методы доступа:
– Внутрикорпоративный LAN и WiFi
– VPN из публичных сетей
– Физическая передача объектов (DVD, USB,..)
– Почта и мессенджеры

Обеспечение безопасности:
– Шифрование (криптостойкость, алгоритмы)
– Аутентификация и токены (сертификаты и политики паролей)
– Авторизация и права доступа

Политики безопасности – устанавливаются административно.
Примеры:
– Приемлемое использование, минимальный уровень доступа, управление аккаунтами пользователей
– Классификация информации (публичная, ДСП, секретная, частная, …)
– Безопасность серверов и мобильных устройств, физический доступ
– Реакция на инциденты, мониторинг безопасности
– Защита от «закладок», тесты безопасности

37. Тестирование безопасности. Практически используемые методы. Безопасный код. Основные подходы. Common Weakness Enumeration

Как и тестирование в целом, не может найти все дефекты (уязвимости) безопасности

- Зависит от ЖЦ разработки и использования
- Практически используются несколько подходов:
  – Статические анализаторы безопасного кода
  – Фаззинг (Fuzz testing)
  – Тестирование на проникновение (Penetration testing)

Безопасный код – куча стандартов, зависят от языка программирования и базового API
– SEI CERT Coding Standards (C++, Java,...)
– OWASP Secure Coding Practices Quick Reference Guide
– Microsoft secure coding practices
Статические анализаторы и статическое тестирование позволяют найти основные дефекты
– OWASP toolset
– Find Security Bugs (java)
– …

CERT Top 10 Secure Coding Practices:
– Проверяйте все входные значения
– Следуйте всем предупреждениям компилятора
– Проектируйте и устанавливайте политики безопасности
– Keep it simple!
– Запрещайте доступ если явно не разрешено
– Реализуйте принцип минимальных доступных привилегий
– Очищайте данные отправленные к другим системам/модулям
– Практикуйте многоуровневую защиту
– Проверяйте качество при помощи тестирования
– Реализуйте стандарты безопасного кодирования

Common Weakness Enumeration – поддерживается сообществом разработчиков и содержит списки типичных уязвимостей с примерами и советами как их избежать
– CWE Top 25 Most Dangerous Software Errors
– OWASP Top 10 document
– CWE Weaknesses in Software Written in C++
– CWE Weaknesses in Software Written in Java
Общая база, уязвимости отсортированы по системам, языкам программирования

38. Fuzzy testing (Фаззинг). Типы фаззинга

Нечеткое тестирование.
Фаззинг - это метод автоматического тестирования программного обеспечения, который заключается в предоставлении некорректных, неожиданных или случайных данных в программу и ее функции. В процессе тестирования отслеживается поведение программы, например, чтобы она не выбрасывала исключений, не пыталась завершиться с ошибкой или не возникло утечки памяти.

- Одна из форм анализа уязвимостей
- Запускает программу на множестве аномальных (зачастую случайных) входных данных
- Цель — не дать атакующему воспользоваться эксплойтом в результате сбоя
- Автоматически сгенерировать случайные тестовые случаи
- Мониторинг приложение для поиска ошибок

Типы:

- **Dumb Fuzzing** — изменение (mutating) существующих тестов или данных для создания новых тестовых данных
    - Taof, GPF, ProxyFuzz, Peach Fuzzer

- **Smart Fuzzing** — определение новых тестов на основе моделей входных данных
    - SPIKE, Sulley, Mu 4000, Codenomicon, Peach Fuzzer

- **Evolutionary** — генерирование тестов на основе ответов от программ
    - Autodafe, EFS, AFL

39. Penetration Testing. Тестирование на проникновение. Dynamic Application Security Testing (DAST) Tools

Penetration testing (тестирование на проникновение) - метод оценки безопасности компьютерных систем или сетей средствами моделирования атаки злоумышленника.

- В общем случае ничем не отличается от хакинга, что незаконно
- Существуют турниры и разборы в сети
- Обычно этому не обучают, специалистами являются “фрилансеры”
- Для проведения такого тестирования требуется заключения полноценного договора с группой “тестеров”

Примером могут служить:
Auth, ddos, password steal и т.д.

DAST Tools - это средства для динамического тестирования безопасности приложения. Они выполняют автоматическое сканирование, которое имитирует вредоносные внешние атаки с попытками эксплуатации распространенных уязвимостей. Их задача - обнаружить незапланированные результаты раньше, чем это сделает злоумышленник. Чтобы обнаружить уязвимости, средства DAST проверяют все токи доступа по HTTP, а также моделируют случайные действия и работу пользователей.

Незаконно это использовать не для себя. Иначе только с письменного разрешения!

ПО:

- ManageEngine Vulnerability Manager Plus
- Paessler PRTG
- Rapid7 Nexpose

Примеры:
Межсайтовый скриптинг, SQL-инъекции, инъекции команд, обход пути и небезопасная конфигурация сервера.

40. Организация тестов безопасности в циклах и типах разработки. Тестирование общих механизмов безопасности.

Организация тестов безопасности в цикле разработки

- Планирование: цель - определение сферы тестирования в соответствии с рисками
- Анализ и проектирование: определение угроз и рисков на базе аудита требований и известных уязвимостей
- Реализация и выполнение тестов (с учетом перспективы внутренних и внешних пользователей, взломщиков и т.д.)
- Отчеты по результатам
- Интеграция тестов в процесс разработки

Особенности при последовательной разработке:

- Раннее определение рисков и требований к безопасности
- Требования безопасности могут меняться, но не отражаться в требованиях
- Обычно выполняются в конце проекта
- Сложность исправления найденных проблем

Особенности при итеративной или инкрементальной разработке:

- Потребности в тестах безопасности возникают во время проекта (например во время спринта)
- Возможно изменение (включая полное) подходов
- Тесты могут проводиться в течении всего проекта
  Стратегия на избегание риска может не сработать за один релизный цикл

Тестирование общих механизмов:

- **System Hardening (“закалка”)** - удаление всего лишнего, добавление ПО безопасности
  - Тесты могут проводить аудит системы и прикладного ПО (пароли по умолчанию, лишняя конфигурация, известные дыры в библиотеках…)
- **Аутентификация и Авторизация** - наиболее часто ломаемый модуль
  - Тестирование брутфорс паролей, фильтры на ввод (XSS, injection), доступность URL
- **Шифрование**
  - Тесты на дизайн, разработанный код, конфигурацию
- **Брандмауэры и Зоны**
  - Тесты могут сканировать порты, отсылать битые пакет, dos, ddos, фрагментация
- **Средства обнаружения взлома**
  - Различное инвазивное изменение данных и сообщений
- **Средства обнаружения вредоносного ПО**
  - Нет смысла в реальных вирусах - “Eicar” (anti-malware test fil)
- **Обфускация данных**
  - Реверс инженеринг, брутфорс (например unXOR)