ХУЙ Тестироване ПО (2022):

1. Понятие тестирования ПО. Основные определения.

Тестирование программного обеспечения (Software Testing)
- проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечно-определенном наборе тестов.
- это одна из техник контроля качества, включающая в себя планирование работ, проектирование тестов , выполнение тестирования и анализ полученных результатов.

Валидация - проверка на соответствие ожиданиям и потребностям пользователя (Have we done the right thing?)

Верификация - проверка на качество и выполнение требований спецификации (Have we done the thing right?)

План Тестирования - это документ, описывающий весь объем работ по тестированию, начиная со стратегии, критериев тестирования до необходимого в процессе работы оборудования, специальных знаний.
Тест дизайн - это этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи, в соответствии с определёнными ранее критериями качеств.
Тестовый случай - это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функциональности.
Баг-репорт - это документ, описывающий ситуацию или последовательность действий приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Mistake - ошибки, просчет человека

Fault - дефект, изъян ПО в результате ошибки

Failure - неисправность, отказ, сбой, внешнее проявление дефекта

Error - невозможность выполнить задачу вследствие отказа

Тестовое Покрытие - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

2. Цели и принципы тестирования (ISTQB).

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

Принципы тестирования:

● Тестирование демонстрирует наличие дефектов (их отсутствие показать нельзя)
● Исчерпывающее тестирование недостижимо (полное тестовое покрытия не достичь)
● Раннее тестирование (чем раньше тем лучше)
● Скопление дефектов (80 % дефектов содержат 20% программы)
● Есть сложные куски программы. Дефекты в основном в них. Модули сами по себе сложные, поэтому все баги в них
● Парадокс пестицида (если часто проводить правки, то тесты со временем ломаются и их нужно чинить)
● 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)
● Заблуждение об отсутствии ошибок.

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”. Т.е:
● Уровень доверия
- Тестирование наглядно демонстрирует, что ошибок нет
- Уровень остаточного обнаружения дефектов
    ● Число дефектов обнаруженных тестом или набором тестов
    ● Число дефектов обнаруженных в заданное время
- С помощью испытаний можно показать надежность системы, описанную в требованиях к по
    ● Сложно показать без испытаний, т. е. работающего ПО
● Корректное поведение
Корректное поведение системы определяется из требований, спецификаций и зависит от уровня проводимого тестирования
● Необходимые обстоятельства - это требования реального окружения
Оно включает в себя:
- Реалистичное количество данных - таких же как в целевой системе
- Реалистичный набор, комбинация входных данных

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

Способы оценки качества:

- тестирование
- разработка стандартов: стандартизация интерфейса взаимодействия с ПО, что снижает количество дефектов
- обучение сотрудников используемым технологиям
- Технический анализ дефектов: поиск истинных причин возникновения ошибки

Уровни восприятия тестирования в компании:

- Уровень 0: отладка
    - невозможно определить некорректное поведение и найти ошибки в программе
    - Не учитывает требования надежности и безопасности
- Уровень 1: показать корректность ПО
    - невозможно доказать отсутствие ошибок
- Уровень 2: поиск ошибок разработчика тестировщиками
    - возможен конфликт разработчиков и тестировщиков
- Уровень 3: тестирование показывает наличие ошибок
    - контроль и минимизация рисков
- Уровень 4: тестирование, как возможность оценки качества программного обеспечения в терминах найденных дефектов
    - проведение функциональных и нефункциональных тестов

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

● Проектирование тестов: словесное описание тестов.
– На основании формальных критериев
– На основании знаний предметной области, опыта и экспертизы
● Автоматизация тестов: написание кода для тестов
– Знание средств, скриптов
● Исполнение тестов: выполнение написанных тестов
– Нет специальных требований к квалификации
● Анализ результатов
– Знания предметной области

6. Мониторинг прогресса и контроль тестирования (ISTQB)

Целью *мониторинга* тестирования является предоставление результата и обзора процесса тестирования. Информация отслеживается вручную или автоматически и может быть использована для измерения критериев тестирования, таких как покрытие. Метрики в мониторинге используются для оценки прогресса тестирования по сравнению с запланированным расписанием и бюджетом. Например, количество выполненных\невыполненных тестовых сценариев или стоимость тестирования

Контроль тестирования описывает любые направляющие или корректирующие действия, принятые на основе результатов по полученной и собранной информации и значениям метрик. Контроль тестирования может затрагивать любые действия по тестированию, а также воздействовать на другие действия и задачи жизненного цикла ПО

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

Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по отдельности (модули программ, объекты, классы, функции и т.д.). Т.Е. Тестирование отдельных компонентов ПО

модуль – это компонент минимального размера, который может быть независимо протестирован в ходе верификации программной системы. Например: метод / класс / программный модуль

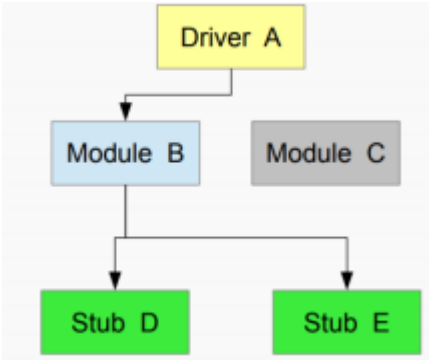
Характеристики модульных тестов (FIRST):

- Fast - должны исполняться сравнительно быстро
- Independent - результат каждого теста не зависит от состояния остальных тестов
- Repeatable - результат не меняется в зависимости от количества прогонов и окружения
- Self-verifying - для каждого теста однозначно определено прошел он или нет.
- Timely - TDD, тесты должны быть написаны до написания тестируемого кода (может не применяться)

Изолирование модуля за счет замещения того, что его вызывает и того, что вызывает он сам:

- Драйвер: эмулирует вызываемый компонент
  - Может быть использован для еще не разработанных главных модулей
  - Используются для тестирования низкоуровневых модулей
  - Необходимы настройка окружения и подготовка исходных данных
- Заглушка (stub): эмулирует поведение подчиненной программы
  - Может быть использована для еще не разработанных модулей
  - Используются для тестирования верхнеуровневых модулей
  - Интерфейсы заглушки и модуля, которого она замещает, должны совпадать
  - Сама заглушка не тестируется

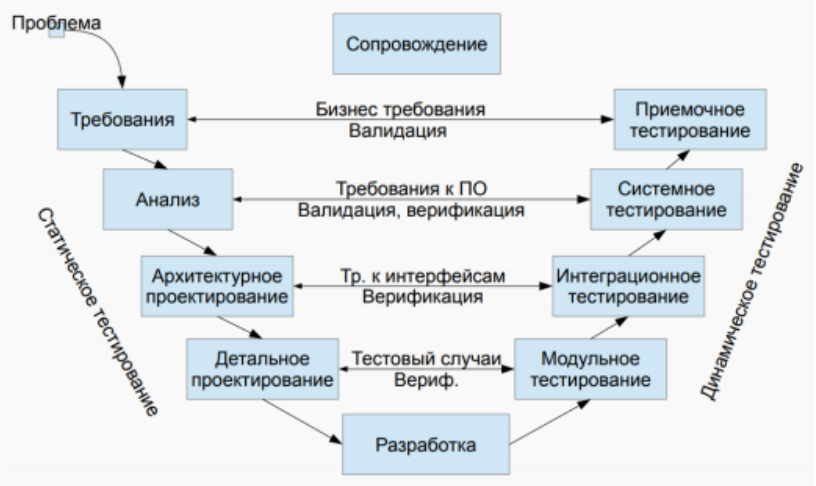
Изолируем модуль B



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

V-модель – это улучшенная версия классической каскадной модели. Здесь на каждом этапе происходит контроль текущего процесса, для того чтобы убедится в возможности перехода на следующий уровень. В этой модели тестирование начинается еще со стадии написания требований, причем для каждого последующего этапа предусмотрен свой уровень тестового покрытия.

В V-модели каждому этапу проектирования и разработки системы соответствует отдельный уровень тестирования.



Преимущества V-model

- 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-model

- К старту проекта все команды должны быть сформированы - большое стартовое вложение
- Очень большое количество документации и работы с ней

Статическое тестирование – тип тестирования, который предполагает, что программный код во время тестирования не будет выполняться

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации.

– вычитка исходного кода программы;

– проверка требований.

Динамическое тестирование – тип тестирования, который предполагает запуск программного кода. Таким образом, анализируется *поведение* программы во время ее работы.

Для выполнения динамического тестирования необходимо чтобы тестируемый программный код был *написан*, *скомпилирован* и *запущен*.

Динамическое тестирование является частью процесса валидации программного обеспечения.

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

Валидация - проверка на соответствие ожиданиями (Have we done the right thing?)

Верификация - проверка на качество и выполнение требований спецификации (Have we done the thing right?)

Метод «черного ящика» - метод тестирования при котором не используется знание о внутреннем устройстве тестируемого объекта.

– Спецификации, требования, дизайн.

– Запуск и сравнение результатов с эталоном

Метод «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

10. Тестовый случай, тестовый сценарий и тестовое покрытие.

Тестовый случай (Test Case) - это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Под тест кейсом понимается структура вида: Action > Expected Result > Test Result

Тестовый сценарий - последовательность тестовых случаев; состоит из набора входных значений, предусловий выполнения, ожидаемых результатов и постусловий, определяемых для покрытия определенных тестовых условий (или тестового условия) или целей (цели) тестирования.

Тестовое Покрытие - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

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

Тестовое Покрытие - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

Полное тестовое покрытие - количество тестов, которые покрывают абсолютно все пути или комбинации данных, которые могут существовать внутри программы

Полное покрытие невозможно.

public long multiply (int A, int B)

- Как протестировать?
- Сколько потребуется памяти?
- Сколько времени будет выполняться на 3 Ггц ЦПУ?

$$\frac{2^{32}\cdot 2^{32}}{3\cdot 10^{9}}=\frac{2^{10}\cdot 2^{10}\cdot 2^{10}\cdot 2^{34}}{3\cdot 10^{3}\cdot 10^{3}\cdot 10^{3}}\approx\frac{2^{34}}{3}\approx\frac{5723784000[c]}{365\cdot 24\cdot 60\cdot 60}=181,5[лет]$$

Для обеспечения более менее высокого уровня используются:

- Эквивалентные разбиения: это техника, при которой мы разделяем функционал (часто диапазон возможных вводимых значений) на группы эквивалентных по своему влиянию на систему значений.

- Таблицы решений: техника, помогающая наглядно изобразить комбинаторику условий из ТЗ.

- Таблицы переходов: Техника для визуализации ТЗ

- Сценарии использования(Use-Case диаграммы)

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

Повторяемость – все написанные тесты всегда будут выполняться однообразно

Регрессионное тестирование- это вид тестирования уже протестированной ранее программы, проводящееся после внесенных изменений для уверенности в том , что процесс модификации не внес или не активизировал ошибки в областях , не подвергавшихся изменениям. Проводится после изменений в коде программного продукта или его окружения.

- для регрессионного тестирования используются тест кейсы, написанные на ранних стадиях разработки и тестирования.
- Рекомендуется делать автоматизацию регрессионных тестов, для ускорения последующего процесса тестирования и обнаружения дефектов на ранних стадиях разработки программного обеспечения.

Автоматизированное тестирование: Выполнение тестов , реализуемое при помощи заранее записанной последовательности тестов.

Автоматизация применяется для:

- Регрессионного, приемочного тестирования

- Для повторяющихся тестовых сценариев

- При проверке приложения на разных окружениях

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

Интеграционное тестирование – это тип тестирования, при котором программные модули объединяются логически и тестируются как группа.

Целью тестирования является выявление багов при взаимодействии между интегрированными компонентами или системами и обнаружения дефектов в интерфейсах.

Стратегии интеграции:

- Top-down - вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками, затем по мере готовности они заменяются реальными активными компонентами.
- Bottom-up - все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования.
- End-to-end - сценарий проверяет совокупность действий. Например, пользователь заходит на почтовый сайт, листает письма, просматривает новые, пишет и отправляет письмо, выходит с сайта.
- Big-bang - все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование.
- Ядро (backbone) - у нас есть работающий функционал, который является ядром, и мы наращиваем новый функционал.

14. Тестирование системы целиком - системное тестирование.

Системное тестирование - процесс тестирования системы в целом с целью проверки того, что она соответствует установленным требованиям .

Системное тестирование относится к методам тестирования черного ящика, и, тем самым, не требует знаний о внутреннем устройстве системы.

Начинается после окончания интеграции и включает несколько фаз:

- Системное тестирование — выполняется внутри организации-разработчика
- Альфа- и Бета-тестирование — выполняется пользователем под контролем разработчика
- Приемочное тестирование — выполняется пользователем.

Результат — принять ПО и заплатить или отправить и не платить.

- Методики практически одинаковые, различная строгость интерпретации результатов.

Основной задачей системного тестирования является проверка функциональных и нефункциональных требований в системе в целом.

Можно выделить два подхода к системному тестированию:

- **на базе требований** (*requirements based*)

  Для каждого требования пишутся **тестовые случаи**

- **на базе случаев использования** (*use case based*)

  На основе представления о способах использования продукта создаются случаи использования системы

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

Нефункциональное тестирование.

- Возможности -- работают ли основные функции, юзать только валидные входные данные.
- Стаблильность -- проверка работоспособности приложения при длительном тестировании со средним уровнем нагрузки.
- Отказоустойчивость -- проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи
- Совместимость -- проверяется функционирование ПО в разных средах.

CARAT -- подход к нагрузочному тестированию.

- Capacity — Нефункциональные характеристики: максимальное количество (пользователей, записей в БД, файлов, Кб, ГГц),
  поддерживаемое системой
- Accuracy — Точность: корректность и точность работы алгоритмов и различных математических функций
- Response Time — Время ответа - время ответа сервиса от отправки запроса от пользователя до получения ответа под минимальной, расчетной, пиковой нагрузкой
- Availability — Готовность - доля времени, в течение которого система способна обслуживать клиента и обрабатывать запросы.

- Throughput — Пропускная способность - количество операций в секунду, которое может поддерживать система.

16. Альфа и Бета тестирование. Приемочное тестирование.

Приёмочное тестирование -- формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:определения удовлетворяет ли система приемочным критериям, следовательно, результат: вынесение решения заказчиком или другим уполномоченным лицом принимается приложение или нет

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

Бета-тестирование — в некоторых случаях выполняется распространение предварительной версии для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок или, чтобы получить обратную связь о продукте от его будущих пользователей.

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

Статическое тестирование – тип тестирования, который предполагает, что программный код во время тестирования не будет выполняться. Предполагает анализ артефактов разработки программного обеспечения, таких как требования или программный код

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации.

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

Технический анализ - проверка продукта на соответствие и практическую пригодность.

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

18. Статическое тестирование. Инспекции.

Статическое тестирование – тип тестирования, который предполагает, что программный код во время тестирования не будет выполняться. Предполагает анализ артефактов разработки программного обеспечения, таких как требования или программный код

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации.

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

1) Вход - постановка проблемы/цели
2) Планирование - определение сессий (проводимых тестов)
3) Обзор
4) Подготовка
5) Обсуждение
6) Переработка - может осуществляться автором/редактором/другими работниками
7) Выработка рекомендаций (follow up)
8) Выход

19. Статическое тестирование. Статический анализ кода.

Статическое тестирование – тип тестирования, который предполагает, что программный код во время тестирования не будет выполняться. Предполагает анализ артефактов разработки программного обеспечения, таких как требования или программный код

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации.

Статический анализ кода — анализ программного обеспечения, производимый без реального выполнения исследуемых программ. В большинстве случаев анализ производится над какой-либо версией исходного кода.

- Большое количество программ-анализаторов кода(C - Lint, Java — FindBugs)
- Позволяет обнаружить:
  - Неопределенное поведение
  - Нарушение алгоритмов использования библиотеки
  - Сценарии некорректного поведения
  - Переполнение буфера

- Утечки памяти и других ресурсов.

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

Тестовое покрытие (Test Coverage) - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода. Полное тестовое покрытие недостижимо. Требуется выбирать для тестирования специфические значения / комбинации, которые определят в итоге конечный набор тестов -> тестовое покрытие. Один из способов - эквивалентное разбиение + анализ граничных значений.

это техника, при которой мы разделяем функционал (часто диапазон возможных вводимых значений) на группы эквивалентных по своему влиянию на систему значений. Такое разделение помогает убедиться в правильном функционировании целой системы — одного класса эквивалентности, проверив только один элемент этой группы.

- Достоинства: стремится *не только сокращать количество тестов*, но и *сохранять приемлемое тестовое покрытие*.
- Недостатки: 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.

Класс эквивалентности (equivalence class) — одно или несколько значений ввода, к которым программное обеспечение применяет одинаковую логику.
Анализ граничных значений - определение классов эквивалентности, которое включает в себя поиск граничных значений классов. Это могут быть конкретные значения или бесконечные.

- Недостатки: не всегда легко определить границы

21. Выбор тестового покрытия с помощью таблицы решений.

Тестовое покрытие - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода. Полное тестовое покрытие недостижимо.
Decision Table (таблица решений) — техника, помогающая наглядно изобразить комбинаторику условий из ТЗ.
Один из способов определения какие комбинации значений надо проверить - таблица решений. Данный метод позволяет наглядно изобразить комбинаторику условий из ТЗ и определить минимальное количество тестов, покрывающих 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 из себя плотность покрытия тестами требований либо исполняемого кода. Полное тестовое покрытие недостижимо.
Один 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. Бывает в двух визуальных формах: таблица переходов / диаграмма состояний. В итоге полученные таблицу или диаграмму можно использовать для проведения функционального тестирования.
Пример диаграммы состояний:

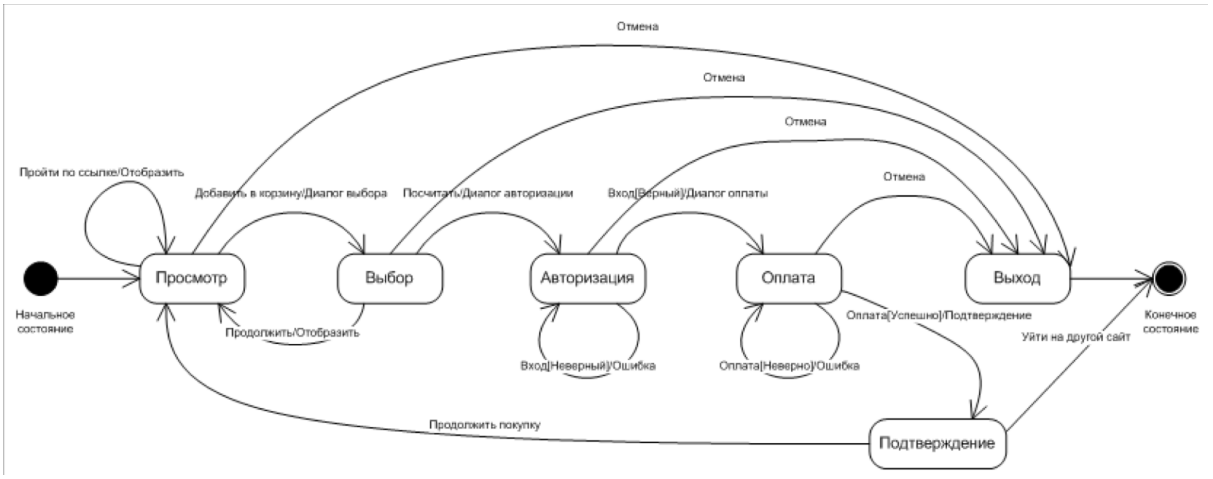


Таблица переходов состоит из четырех столбцов:

Таблица переходов:

Состояния		События/Условия		Текущее состояние	Событие/Условие	Действие	Новое состояние
				Просмотр	Пройти по ссылке	Отобразить	Просмотр
		Пройти по ссылке		Просмотр	Добавить в корзину	Диалог выбора	Выбор
		Добавить в корзину		**Просмотр**	**Продолжить**	**Не определено**	**Не определено**
		Продолжить		**Просмотр**	**Выписать**	**Не определено**	**Не определено**
Просмотр		Выписать		**Просмотр**	**Вход [неверный]**	**Не определено**	**Не определено**
Выбор		Вход [неверный]		**Просмотр**	**Вход [верный]**	**Не определено**	**Не определено**
Авторизация	X	Вход [верный]	=	**Просмотр**	**Оплата [неверно]**	**Не определено**	**Не определено**
Оплата		Оплата [неверно]		**Просмотр**	**Оплата [успешно]**	**Не определено**	**Не определено**
Подтверждение		Оплата [успешно]		**Просмотр**	**Отмена**	**Нет действия**	**Выйти**
Выйти		Отмена		**Просмотр**	**Продолжить покупку**	**Не определено**	**Не определено**
		Продолжить покупку		**Просмотр**	**Уйти на другой сайт**	**Не определено**	**Не определено**
		Уйти на другой сайт		**Выбор**	**Пройти по ссылке**	**Не определено**	**Не определено**
				{53 строки, сгенерированных по шаблону, не показаны}			
				Выйти	**Уйти на другой сайт**	**Не определено**	**Не определено**

23. Выбор тестового покрытия с помощью функционального тестирования.

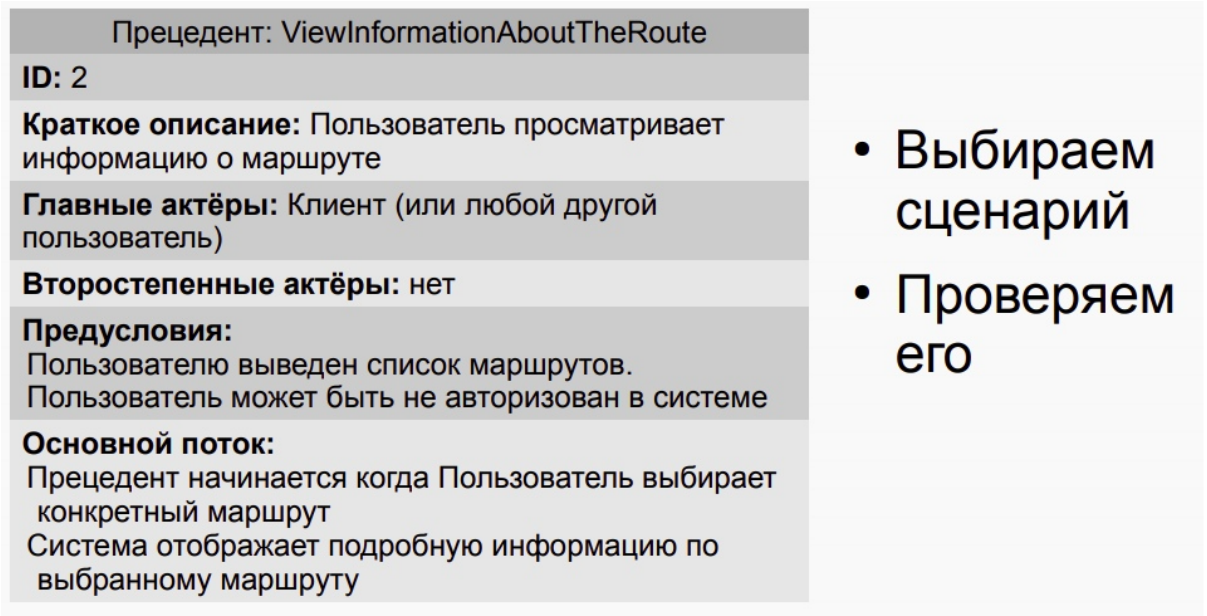
Тестовое покрытие - это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода. Полное тестовое покрытие недостижимо.

Функциональное тестирование рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.

Функциональное тестирование проводится на основе сценариев использования системы, соответственно проверяются функции системы, начиная с интерфейса пользователя.

- проверка функционала программы осуществляется на базе сценариев, где в явном виде описаны действия пользователя в системе
- основной элемент - графический интерфейс пользователя

Пример:



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

JUnit - open-source фреймворк для модульного тестирования. Последняя версия: JUnit 5, состоит из JUnit Platform + JUnit Jupiter + JUnit Vintage. Возможности JUnit 4 (JUnit 5):

- Тесты отмечаются как @Test
- Набор методов для проверки утверждений (assertEquals(), assertArrayEquals(),assertNull(),assertNotNull(),assertTrue(),assertFalse(),fail())
- org.junit.Assert - Аннотации для маркировки действий до / после выполнения теста (@Before (@BeforeEach), @After (@AfterEach), @BeforeClass (@BeforeAll), @AfterClass (@AfterAll))
- Отключение конкретных тестов - @Ignore (@Disabled)

- Запуск тестов с разными типами параметров - Тэгирование / фильтрация / группировка тестов для выполнения - @Category (@Tag)

Пример:

```
@Test (timeout=10)
public void longLoop() {
        assertEquals(PI.computeNNumber(1E10) ,3);
}

@Test (expected=IllegalArgumentException.class)
public void testSqrt() {
       Math.Sqrt(-5);
}
@Ignore("Помогите!")

@Test
public void add() {
       assertEquals(4, program.sum(2, 2));
}
```

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

JUnit - open-source фреймворк для модульного тестирования. Последняя версия: JUnit 5, состоит из JUnit Platform + JUnit Jupiter + JUnit Vintage. Возможности JUnit 4 (JUnit 5):

Дополнительные возможности аннотации @Test:

- Тестирование исключений
    - Параметр expected определяет класс исключения, которое выбросит метод.
    - В JUnit 5 появился assertThrows(...)
- Таймауты
    - Параметр timeout определяет максимальное количество времени, которое может выполняться тест (в миллисекундах)
    - В JUnit 5 появился assertTimeout(....)

Параметризованные тесты (JUnit 4):

Идея: тест работает с полями тестового класса, при каждом запуске теста в поля подставляются значения, определенные в специальном методе.

2 важные аннотации:

- @RunWith(Parameterized.class) - определение ранера
- @Parameters(value=Parameterized.class) - для задания параметров для тестов.

В JUnit 5 введена аннотацию @ParametrizedTest + есть возможность указать источник данных в виде @Value/Csv/CsvFile/Enum/Argument/Method + Source

Пример:

```
@ParameterizedTest(name = "{index} - {0} is a palindrome")
  @ValueSource(strings = { "12321", "pop" })
  void testPalindrome(String word) {
     assertTrue(isPalindrome(word));
  }
```

```
@RunWith(Parameterized.class)
public class FibonacciTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
                { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }
           });
    }

    @Parameter // first data value (0) is default
    public /* NOT private */ int fInput;

    @Parameter(1)
    public /* NOT private */ int fExpected;

    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
```

26. Анализ эквивалентности с использованием JUnit.

JUnit - open-source фреймворк для модульного тестирования. Последняя версия: JUnit 5, состоит из JUnit Platform + JUnit Jupiter + JUnit Vintage. Возможности JUnit 4 (JUnit 5):

Анализ эквивалентности - это техника, при которой мы разделяем функционал (часто диапазон возможных вводимых значений) на группы эквивалентных по своему влиянию на систему значений. Такое разделение помогает убедиться в правильном функционировании целой системы — одного класса эквивалентности, проверив только один элемент этой группы.
Так как воздействия на систему будет схожий, можно использовать параметризованные тесты.
Пример:

```
@ParameterizedTest
@ValueSource(ints = {-1, Integer.MIN_VALUE}) // boundary values
void testNegative(int number) {
    assertTrue(Algorithm.checkIsNegative(number));
}

@ParameterizedTest
@ValueSource(ints = {0, 1, Integer.MAX_VALUE}) // boundary values
void testNonNegative(int number) {
    assertFalse(Algorithm.checkIsNegative(number));
}
```

27. Тестирование алгоритмов с использованием JUnit.

JUnit - open-source фреймворк для модульного тестирования. Последняя версия: JUnit 5, состоит из JUnit Platform + JUnit Jupiter + JUnit Vintage. Возможности JUnit 4 (JUnit 5):

Если алгоритм удобно тестировать проверкой пар входных-выходных значений (метод черного ящика), в JUnit используются параметризованные тесты:

```
@ParameterizedTest
@CsvFileSource(files = "test_input.csv")
void testWithCsvFileSourceFromFile(String input1, String input2, int expected) {
    int actual = runAlgorithm(input1, input2);
    assertEquals(expected, actual);
}
```

Параметризованный тест подойдет и в том случае, когда логику алгоритма удобно представить таблицей решений.

Для того чтобы протестировать алгоритм в изоляции от зависимостей, применяются заглушки. Их можно реализовать средствами языка, однако удобнее использовать специальные библиотеки, облегчающие их написание (Mockito).

Для тестирования алгоритмов сортировки полезно использовать аналогию “хлебных крошек”, с помощью которого определяется последовательность перестановок во время сортировка(определяется последовательность самостоятельно)

28. Модульное тестирование доменной модели с использованием JUnit.

Задачей 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-выходных значений, целесообразно использовать параметризованные тесты.

Если входные параметры описываются структурами данных, которые неудобно представлять в формате .csv файла или объявлять на месте, можно определить для них интерфейс ArgumentsProvider, который можно использовать для инициализации входных данных для параметризованных тестов.

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. Пример:

```java
@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.

Selenium — это набор программ с открытым исходным кодом, которые применяют для тестирования веб-приложений и администрирования сайтов локально и в сети. Программы Selenium позволяют автоматизировать действия браузера:

- Selenium WebDriver - библиотека управления браузерами
- Selenium RC - предыдущее поколение библиотеки управления браузерами
- Selenium Server - сервер, позволяющий управлять браузером с удалённой машины
- Selenium Grid - кластер из нескольких Selenium Server. Позволяет параллельно запускать несколько браузеров на разных машинах.
- Selenium IDE - плагин к браузеру Firefox, который может записывать действия пользователя, воспроизводить их, а также генерировать код для WebDriver или Selenium RC, в котором выполняются те же самые действия.

Команды:

- click или clickAndWait — ссылки, переключатели, radio-кнопки
- open — открывает страницу
- assert***, verify *** - проверка содержимого элемента UI
- wait*** - ожидание события

Синхронизация:

- waitForPageLoad(timeout) загрузка страницы ошибка по таймауту
- waitForAlert
- waitForTable — полная загрузка таблицы
- waitForTitle — загрузка заголовка
- Другие команды синхронизации

Другие:

- store — сохранение значений в переменной
- echo — запись значения в лог selenium

Пример:

```
@Test
void sampleTest() {
  WebDriver driver = new FirefoxDriver(); // new ChromeDriver();

  String baseUrl = "https://www.google.com/";
  String expectedTitle = "Google";

  driver.get(baseUrl);
  assertEquals(expectedTitle, driver.getTitle());
}
```

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

Selenium — это набор программ с открытым исходным кодом, которые применяют для тестирования веб-приложений и администрирования сайтов локально и в сети.

Assert vs verify

- Предназначены для проверки содержимого элемента UI
- Если verification неуспешна — тест продолжается
- Если неуспешна assertion — тест останавливается

verifyTextPresent используется, чтобы проверить наличие определенного текста в любом месте страницы.

verifyElementPresent используется, когда необходимо проверить наличие определенного элемента интерфейса, а не его содержимое.

verifyText используется когда нужно проверить как текст, так и соответствующий ему элемент интерфейса пользователя.

Assertion - аналогично

Команды wait:

● waitForPageLoad(timeout) загрузка страницы ошибка по таймауту

● waitForAlert

● waitForTable — полная загрузка таблицы

● waitForTitle — загрузка заголовка

Пример:

import com.agility.focis.utilities.testObject.SeleniumUtils;
public void loginToApp() throws IOException, InterruptedException {
…

```
SeleniumUtils.waitForPageLoad();
…
}
```

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

Selenium RC – это предыдущая версия библиотеки для управления браузерами.

Selenium RC работает таким образом, что клиентские библиотеки могут связываться с сервером RC Selenium, передавая каждую команду Selenium для выполнения. Затем сервер передает команду Selenium в браузер с помощью команд Selenium-Core JavaScript.

RC взаимодействует с браузером с помощью Selenium Core — специального инструмента, который может работать с любым браузером через JavaScript.

Плюсы:

- Как правило работает одинакого для браузера

Минусы:

- Устарел
- Ограничевается возможностями JS и same-origin policy (веб-браузер разрешает сценариям, содержащимся на первой веб-странице, получать доступ к данным на второй веб-странице, но только если обе веб-страницы имеют одинаковое происхождение.)

WebDriver

Selenium WebDriver, или просто WebDriver – это драйвер браузера, то есть не имеющая пользовательского интерфейса программная библиотека, которая позволяет различным другим программам взаимодействовать с браузером, управлять его поведением, получать от браузера какие-то данные и заставлять браузер выполнять какие-то команды.

WebDriver фактически использует собственный и родной API каждого браузера для работы с ними

Плюсы:

- Быстрее
- Более гибкий

Минусы:

- Может работать по-разному в каждом браузере
- Могут возникать ошибки при обновлении браузеров

Selenium GRID

Selenium Grid – это кластер, состоящий из нескольких Selenium-серверов. Он предназначен для организации распределённой сети, позволяющей параллельно запускать много браузеров на большом количестве машин. Selenium Grid позволяет выполнять сценарии WebDriver на удаленных машинах (виртуальных или реальных) путем маршрутизации команд, отправляемых клиентом, в удаленные экземпляры браузера.

```
public class AuthPageTest {
    private static webDriver;
    @BeforeEach
    public void setUp() {
        WebDriverManager.chromedriver().setup();
        webDriver = new ChromeDriver();
        webDriver.manage().window().maximize();
    }
    @AfterEach
    public void setUp() {
        webDriver.quit();
    }
    @Test
    public void openPage() {
        driver.get("https://www.google.com/");
        String expectedTitle;

        assertEquals(expectedTitle, webDriver.getTitle());
    }
}
```

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

XPath — язык запросов к элементам XML документа.
Строка XPath описывает способ выбора нужных элементов из массива элементов, которые могут содержать вложенные элементы.
При выполнении запросов язык XPath оперирует такими сущностями как узлы. Узлы бывают нескольких видов: element (узел-элемент), attribute (узел-атрибут), text (узел-текст), namespace (узел-пространство имён), processing-instruction (узел-исполняемая инструкция), comment (узел-комментарий), document (узел-документ).

Для осуществления выборки узлов в основном используется 6 основных типов конструкций:

nodename	Выбрать все узлы с именем “nodename”
/	Выбрать всё от корневого узла
//	Выбрать все узлы, подходящие под параметры выборки, вне зависимости от того, где они находятся
.	Выбрать текущий узел
..	Выбрать предка текущего узла
@	Выбрать атрибуты

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

Обозначение	Описание
`*`	Обозначает *любое* имя или набор символов по указанной оси, например: `*` — любой дочерний узел; `@*` — любой атрибут
`$name`	Обращение к переменной. `name` — имя переменной или параметра
`[]`	Дополнительные условия выборки (или предикат шага адресации). Должен содержать логическое значение. Если содержит числовое, считается что это порядковый номер узла, что эквивалентно приписыванию перед этим числом выражения `position()=`
`{}`	Если применяется внутри тега другого языка (например HTML), то XSLT-процессор рассматривает содержимое фигурных скобок как XPath
`/`	Определяет уровень дерева, т. е. разделяет шаги адресации
`|`	Объединяет результат. Т. е., в рамках одного пути можно написать несколько путей разбора через знак `|`, и в результат такого выражения войдёт всё, что будет найдено любым из этих путей

Оси:
Оси — это база языка XPath.
Для некоторых осей существуют сокращённые обозначения.

* ancestor:: — Возвращает всех предков текущего узла.
* self:: Возвращает текущий узел
* child:: — Возвращает множество потомков на один уровень ниже.
* descendant:: — Возвращает полное множество потомков (то есть, как ближайших потомков, так и всех их потомков).
* parent:: — Возвращает родителя на один уровень выше.

Системные функции:
сказать document, format-number, key, element-available, lang

Функция	Описание
`node-set document(object, node-set?)`	Возвращает документ, указанный в параметре `object`
`string format-number(number, string, string?)`	Форматирует число согласно образцу, указанному во втором параметре. Третий параметр указывает именованный формат числа, который должен быть учтён
`string generate-id(node-set?)`	Возвращает строку, являющуюся уникальным идентификатором
`node-set key(string, object)`	Возвращает множество с указанным ключом (аналогично функции `id` для идентификаторов)
`string unparsed-entity-uri(string)`	Возвращает непроанализированный URI. Если такового нет, возвращает пустую строку
`boolean element-available(string)`	Проверяет, доступен ли элемент или множество, указанное в параметре. Параметр рассматривается как XPath
`boolean function-available(string)`	Проверяет, доступна ли функция, указанная в параметре. Параметр рассматривается как XPath
`object system-property(string)`	Параметры, возвращающие системные переменные. Могут быть: • `xsl: version` — возвращает версию XSLT процессора. • `xsl: vendor` — возвращает производителя XSLT процессора. • `xsl: vendor-url` — возвращает URL, идентифицирующий производителя. Если используется неизвестный параметр, функция возвращает пустую строку
`boolean lang(string)`	Возвращает `true`, если у текущего тега имеется атрибут `xml: lang`, либо родитель тега имеет атрибут `xml: lang` и в нём указан совпадающий строке символ

Пример:

//*[@class="input-group-btn"]	Выбрать все узлы в документе типа element с атрибутом «class», установленным со значением «input-group-btn»
//tr[td/text()="ИНН"]/td[@class="company-codes__value"]	Выбрать все узлы в документе с названием «tr» (выбрать все строки таблиц), в которых имеется узел «td» с текстом «ИНН» (ячейка с текстом «ИНН»), после чего выбрать прямого потомка выбранного узла «tr» с названием «td», у которого атрибут «class» установлен со значением «company-codes__value»

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

Здесь сказать про node, text, position, last, count, id

Функция	Описание
`node-set node()`	Возвращает сам узел. Вместо этой функции часто используют заменитель `*`, но, в отличие от звёздочки, функция `node()` возвращает и *текстовые* узлы
`string text()`	Возвращает узел, если он текстовый
`node-set current()`	Возвращает множество из одного элемента, который является текущим. Если мы делаем обработку множества с предикатами, то единственным способом дотянуться из этого предиката до текущего элемента будет данная функция
`number position()`	Возвращает позицию элемента в множестве элементов оси. Корректно работает только в цикле `<xsl:for-each/>`
`number last()`	Возвращает номер последнего элемента в множестве элементов оси. Корректно работает только в цикле `<xsl:for-each/>`
`number count(node-set)`	Возвращает количество элементов в `node-set`.
`string name(node-set?)`	Возвращает полное имя первого тега в множестве
`string namespace-url(node-set?)`	Возвращает ссылку на URL, определяющий пространство имён
`string local-name(node-set?)`	Возвращает имя первого тега в множестве, без пространства имён
`node-set id(object)`	Находит элемент с уникальным идентификатором

Логические функции

Функция	Описание
`boolean boolean(object)`	Приводит объект к логическому типу
`boolean true()`	Возвращает истину
`boolean false()`	Возвращает ложь
`boolean not(boolean)`	Отрицание, возвращает истину если аргумент ложь и наоборот

Логические операторы

Символ, оператор	Значение
`or`	логическое «или»
`and`	логическое «и»
`=`	логическое «равно»
`<`	логическое «меньше»
`>`	логическое «больше»
`<=`	логическое «меньше либо равно»
`>=`	логическое «больше либо равно»

Числовые функции

Функция	Описание
`number number(object?)`	Переводит объект в число
`number sum(node-set)`	Вернёт сумму множества. Каждый тег множества будет преобразован в строку и из него получено число
`number floor(number)`	Возвращает наибольшее целое число, не большее, чем аргумент (округление к меньшему)
`number ceiling(number)`	Возвращает наименьшее целое число, не меньшее, чем аргумент (округление к большему)
`number round(number)`	Округляет число по математическим правилам

Числовые операторы

Символ, оператор	Значение
`+`	сложение
`-`	вычитание
`*`	умножение
`div`	обычное деление (**не нацело!**)
`mod`	остаток от деления

Пример:
count(/file/row[count(./*) = 5]) = count(/file/row)

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

JMeter – это программное обеспечение, которое может выполнять нагрузочное тестирование, ориентированное на производительность системы. По различным протоколам или технологиям.

JMeter имитирует группу пользователей, отправляющих запросы на целевой сервер, и возвращает статистику, которая показывает производительность / функциональность целевого сервера / приложения в виде таблиц, графиков и т. Д.

Архитектура

Thread Group – Описывает пул пользователей для выполнения теста – Количество, возрастание и тд..

Семплеры – Формируют запросы, генерируют результаты – Большой набор встроенных протоколов (TCP, HTTP, FTP, JDBC, SOAP, JMS, SMTP, …..)

Логические контроллеры – Определяют порядок вызова семплеров – Конструкции управления (if, loop, …) – Управление потоком

Слушатели – Получают ответы – Осуществляют доп. операции с результатами: просмотр, запись, чтение и др. – Не обрабатывают данные! (в командной строке, нужен GUI)

Таймеры – Задержки между запросами – Постоянные, в соответствии с законами

Assertion – Проверяют результаты

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 элементов:

1. Конфигурационные элементы
2. Препроцессоры
3. Таймеры
4. Семплеры
5. Постпроцессоры
6. Assertions
7. Слушатели

Пример: ОЧЕНЬ ПРИЯТНО, ИДЕШЬ НАХУЙ

35. Apache Jmeter. Дополнительные возможности. Распределенное тестирование.

JMeter – это программное обеспечение, которое может выполнять нагрузочное тестирование, ориентированное на производительность системы. По различным протоколам или технологиям.

JMeter имитирует группу пользователей, отправляющих запросы на целевой сервер, и возвращает статистику, которая показывает производительность / функциональность целевого сервера / приложения в виде таблиц, графиков и т. Д.

Дополнительные возможности

- Автоматическая генерация CSV-файла
- Bandwidth Throttling - ограничение полосы пропускания
    - 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

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:

1. Запустить JMeter - сервер на всех машинах, которые будут генерировать тестовые запросы

2. Отредактировать файл jmeter.properties на Jmeter клиенте, который будет запускать тесты. В свойстве remote_hosts необходимо перечислить все хосты с Jmeter-серверами
3. Запустить тест на клиенте

Пример:

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

Область деятельности:

Основной фокус тестирования безопасности направлен на проверку безопасности функциональности

Определяется по:

– Спецификациям и требованиям
– Сценариям использования и моделям
-- Анализ возможных рисков

Дополнительно необходимо продумывать политики и процедуры безопасности, которые позволяют минимизировать возникновение рисков

Также особое внимание уделяется тому, какие цели и как будут вести себя взломщик, атакующий ваши ПО, поэтому для защиты ПО проводят анализ уже известных уязвимостей и дефектов безопасности и формируются подходы для обучения как разработчиков, так и тестировщиков для минимизации уязвимостей системы.

Риски безопасности:

1) Риски в разработке и эксплуатации ПО
2) Опасность нарушения конфиденциальности, целостности или доступности информации или информационных систем.

Для оценки возможных рисков безопасности при разработке и эксплуатации ПО используют один из популярных способов: экспозиция риска = функция от вероятности наступления и величины ущерба

Существуют стандартные методы работы с рисками:

1) Оценка: всестороннее изучение и определение параметров риска
2) Контроль: проверка определенных параметров риска
3) Управление риском: определение поведения при наступлении риска

Цифровые активы - информационный ресурс, обладающий правом на ценность и секретность

– Данные пользователей и бизнес-планы
– Разработанное ПО, документация, модели, диаграммы
– Документы, процессы, торговые секреты
-- Налоговая и финансовая информация, информация о работниках
– Возможность окаазывать услуги
– Репутация компании и доверие пользователей

Методы доступа и обеспечения безопасности:

Типичные методы конфиденциального доступа:

– Внутрикорпоративный LAN и WiFi
– VPN из публичных сетей
– Физическая передача объектов
– Почта и мессенжеры

Обеспечение безопасности:

– Шифрование
– Аутентификация и токены
– Авторизация и права доступа

Политика безопасности - совокупность документированных принципов, правил, процедур практических приемов, которые регулируют управление, защиту и распределение ценной информации. Формируется административными структурами

– Приемлемое использование, минимальный уровень доступа, правила управления аккаунтами пользователей
– Классификация информации(публичная, секретная)
– Безопасность серверов и мобильных устройств
– Реакция на инциденты, мониторинг безопасности
– Тесты безопасности

Пример: Code injections (SQL, PHP, ASP и т.д.)

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

Тестирование безопасности - это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

Особенности:

- 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)

Для написание более и менее безопасный кода следуют следующим “правилам”:

Использование различных стандартов языка программирования и базового API
– SEI CERT Coding Standards (C++, Java,...)
– OWASP Secure Coding Practices Quick Reference Guide
– Microsoft secure coding practices

Использование статических анализаторов и проведение статического тестирования
– OWASP toolset
– Find Security Bugs (java)

Основные подходы для обеспечения безопасного кода:

– 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 - Общая база уязвимостей систем и языков программирования. Поддерживается сообществом разработчиков

Содержит списки типичных уязвимостей с примерами и советами как их избежать

– CWE Top 25 Most Dangerous Software Errors

– CWE Weaknesses in Software Written in Java

Пример:

38. Fuzzy testing (Фаззинг). Типы фаззинга

Fuzzy testing - техника тестирования программного обеспечения, заключающаяся в передаче приложению на вход неправильных, неожиданных или случайных данных. Предметом интереса являются падения и зависания, нарушения внутренней логики и проверок в коде приложения, утечки памяти. Представляет из себя мониторинг приложения для поиска на случайно сгенерированных тестовых случаях

Цель — не дать атакующему воспользоваться эксплойтом в результате сбоя

Типы фаззинга:

Dumb Fuzzing — изменение (mutating) существующих тестов или данных для создание 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

Пример:

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

Penertration Testing - одна из методик выявления областей системы, уязвимых для вторжения и компрометации целостности и достоверности со стороны неавторизованных и злонамеренных пользователей или сущностей. Процесс тестирования проникновения включает в себя умышленные санкционированные атаки на систему, способные выявить как ее наиболее слабые области, так и пробелы в защите от посторонних проникновений, и тем самым улучшить атрибуты безопасности.

Основная цель: сделать систему достаточно компетентной для защиты от ожидаемых и даже неожиданных вредоносных угроз и атак.

Достоинства:

- выявления слабых и уязвимых областей системы еще до того, как их заметит хакер.
- оценка существующего механизма безопасности системы
  - Требует заключение полноценного договора с группой «тестеров»
  - Много типов: пробить систему аутентификации, завалить ddos-атаками, password steal

Dynamic Application Security Testing (DAST) Tools - это процедура, которая активно исследует работающие приложения с помощью тестов на проникновение для обнаружения возможных уязвимостей безопасности:

● Собраны все существующие уязвимости известных систем и движков

Пример: OpenVAS, OWASP

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
● Тесты обычно выполняются в конце проекта
● Сложность исправления найденных проблем

Особенности при итеративной или инкрементальной разработке:

● Потребности в тестах безопасности возникают во время проекта
● 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 безопасности:

● System Hardening («закалка») заключается в том, что берутся все инфраструктурные компоненты системы, удаляются все лишние системы, добавление конфигурации и ПО безопасности
● Аутентификация и Авторизация: необходимо тестировать брутфорс паролей, фильтры на ввод (XSS, injection), доступность URL
● Использование методов шифрования
● Использование брандмауэров и зонирование доступа: тесты сканируют порты, отсылают битые пакеты, ddos-атаки
● Использование средств обнаружения взлома
● Использование средств обнаружения вредоносного ПО
● Обфускация(маскировка) данных – это способ защиты конфиденциальной информации от несанкционированного доступа путём замены исходных данных фиктивными данными или произвольными символами.