**Тестирование программного обеспечения –** проверка соответствия между реальным и ожидаемым поведением программы, и на наличие дефектов.

Тестирование разделяется по видам и по уровням.

**Уровни тестирования**

**Модульное(Unit Testing) –** модульное тестирование направлено на тестирование отдельных модулей, которые изолированы от других модулей и компонентов. Обычно его выполняет сам программист.

Можно поделить на две группы:

* **Тесты состояния**, проверяющие что вызываемый метод объекта сработал корректно, проверяя состояние тестируемого объекта после вызова метода.
* **Тесты взаимодействия,** в которых тестируемый объект производит манипуляции с другими объектами. Применяются, когда требуется удостовериться, что тестируемый объект корректно взаимодействует с другими объектами.

**Модуль** – минимальный компонент, выполняющий какой-либо функционал функционал.

Цели:

1. Проверка компонента на соответствие требованиям.
2. Обнаружение ошибок в компоненте (особенно при внесении изменений, чтобы посмотреть что код не сломался, или не сломалось что-то в другом месте).
3. Предотвращение пропуска ошибок на более высокие уровни тестирования.

Плюсы:

1. Мы точно знаем место где произошла ошибка.
2. Документируют код. Можно посмотреть на тест и понять, что принимает функция и что возвращает.
3. Помогают проверить, не сломали ли мы ничего при рефакторинге.
4. Могут выступать индикаторами низкого качества кода. Когда части кода плохо изолированы друг от друга, это создает сложность с раздельным тестированием этих частей.

Junit, Mockito, Spring Test

**Интеграционное тестирование** необходимо чтобы протестировать работу модулей в связке друг с другом, как единого блока.

Берутся модули, над которыми было проведено модульное тестирование, группируются в более крупные множества и тестируются.

Преимущества:

1. Помогают протестировать модули в связке
2. Медленнее модульных и сложнее написать
3. Не указывают в каком месте конкретно проблема  
   Spring Test

**Системное тестирование –** задача состоит в том, чтобы убедится в корректности работы всей системы в целом. Наше внимание должно быть сосредоточено на общем поведении системы с точки зрения конечных пользователей.

Цели:

1. Проверка системы на соответствие требованиям.
2. Обнаружение ошибок в системе.

**End2End (Сквозное тестирование) -** процесс тестирования, при котором происходит подробная эмуляция пользовательской среды. Самый трудозатратный и дорогой. копирует поведение пользователя при работе с ПО в контексте всего приложения.

Проводится тестирование от начала до конца, а также его интеграции с внешними сервисами

Суть этого тестирования — посмотреть, так ли работает программа для конечного клиента, как рассчитывалось изначально? При этом нужно учитывать, что пользователю все равно, функционирует ли программа «как надо», ему главное, чтобы программа функционировала и оправдывала ожидания, поэтому основной упор делается на корректное функционирование.

Проводится:

* Тестирование UI
* **Тестирование локализации и интернационализации**
* Тяжелый отлов найденной проблемы, и установление конкретного места ошибки.

**Приемочные тесты —** это формальные тесты, которые проверяют, отвечает ли система требованиям бизнеса. При этом во время тестирования должно быть запущено само приложение, и основное внимание уделяется воспроизведению поведения пользователей. В ходе этого тестирования возможен даже замер производительности системы, и в случае несоответствия установленным требованиям внесенные изменения могут быть отклонены.

**Виды тестирования по объекту тестирования:**

**Функциональное тестирование –** тестирование ПО в целях проверки реализуемых функциональных требований, то есть что ПО обладает всем функционалом, требуемым заказчиком. Проверка того, какие функции ПО реализованы и того, насколько верно они реализованы.

Может проводится на всех уровнях тестирования (компонентном, интеграционном, системном, приемочном).

Функциональное тестирование может проводится на основе функциональных требований. При этом для тестирования создаются тестовые случаи (testcases), составление которых учитывает приоритетность функций ПО, которые необходимо покрыть тестами. Таким образом мы можем убедиться в том, что все функции разрабатываемого продукта работают корректно при различных типах входных данных, их комбинаций, количества и т.д.

• На основе бизнес-процессов, которые должно обеспечить приложение. В этом случае, нас интересует не так работоспособность отдельных функций ПО, как корректность выполняемых операций, с точки зрения сценариев использования системы. Таким образом, тестирование в данном случае будет основываться на вариантах использования системы (usecases).

Преимущества функционального тестирования:

• имитирует фактическое использование системы;

Недостатки функционального тестирования:

• возможность упущения логических ошибок в программном обеспечении;

• вероятность избыточного тестирования.

**Тестирование безопасности (Safety testing**) — Тестирование программного продукта с целью с целью определить его безопасность.

**Тестирование взаимодействия -**  это функциональное тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в **себя тестирование совместимости** (compatibility testing) и **интеграционное тестирование** (integration testing).

**Интеграционное тестирование** предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами).

**Уровни** интеграционного тестирования:

• Компонентный интеграционный уровень (Component Integration testing)

Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

• Системный интеграционный уровень (System Integration Testing)

Проверяется взаимодействие между разными системами после проведения системного тестирования.

Подходы к интеграционному тестированию:

• **Снизу вверх (Bottom Up Integration)**

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 (см. также Integration testing - Bottom Up)

• **Сверху вниз (Top Down Integration)**

Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами. Таким образом мы проводим тестирование сверху вниз. (см. также Top Down Integration)

• **Большой взрыв ("Big Bang" Integration)**

Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если 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 (см. также Integration testing - Big Bang)

**НЕФУНКЦИОНАЛЬНЫЕ**

**Регрессионное тестирование** — это проверка ранее протестированной программы, позволяющая убедиться, что внесенные изменения не повлекли за собой появления дефектов в той части программы, которая не менялась.

**Тестирование производительности**

тестирование ПО, позволяющее осуществлять оценку быстродействия программного продукта при определённой нагрузке. Тест производительности выполняется до и после проведения оптимизации с целью выявить изменения в производительности.

Если оптимизация не удается, и производительность снижается, то программист может отказаться от неудачной оптимизации. В случае повышения производительности величину этого повышения можно сравнить с ожидаемыми результатами, чтобы убедиться в успешности оптимизации.

Задачей теста производительности является выявление фактов повышения и понижения производительности, чтобы можно было избежать неудачных модернизаций.

К тестированию производительности можно отнести следующие виды тестирования:

**Нагрузочное тестирование -** тестирование ПО, позволяющее осуществлять оценку быстродействия программного продукта при плановых, повышенных и пиковых нагрузках. Осуществление нагрузочного тестирования перед вводом системы в промышленную эксплуатацию позволяет избегать неожиданных потерь в производительности через полгода - год, когда система будет заполнена данными. (Gatling)

**Стресс-тестирование (stress testing)** – тестирование ПО, которое оценивает надёжность и устойчивость системы в условиях превышения пределов нормального функционирования и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса.

Это проверка программы в таких стрессовых ситуациях как наличие большого объёма входных параметров, нехватка дискового пространства или маломощный процессор.

Стресс тестирование предназначено для проверки настроенного решения и серверной группы на одновременное обслуживание большого количества пользователей. При таком тестировании проверяется не только серверная группа, но и влияние, оказываемое настройками на производительность системы в целом и ее отказоустойчивость. Для проведения такого тестирования необходимо иметь набор компьютеров, эмулирующих работу групп пользователей.

Зачастую, вручную создать необходимые условия нагрузки не представляется возможным, поэтому прибегают к:

• Фокусировании на транзакциях определенных типов, что более сильно влияет на возникновение граничных ситуаций, нежели при нагрузочном тестировании.

• Использование скриптов, виртуальных пользователей и прочих автоматизированных симуляторов интенсивного рабочего процесса, что позволяет исследовать поведение программного обеспечения при пиковой нагрузке.

**Тестирование стабильности (stability/endurance/soak testing)** – тестирование ПО, при котором проверяется работоспособность ПО при длительном тестировании со среднем уровнем нагрузки.

При тестировании, длительность его проведения не имеет первостепенного значения, основная задача — наблюдая за потреблением ресурсов, выявить утечки памяти и проследить чтобы скорость обработки данных и/или время отклика приложения в начале теста и с течением времени не уменьшалась. В противном случае вероятны сбои в работе продукта и перезагрузки системы.

**Объемное тестирование (VOLUME TESTING)** - это тип тестирования программного обеспечения, когда программное обеспечение подвергается огромному объему данных. Это также упоминается как **тестирование наводнения.** Объемное тестирование проводится для анализа производительности системы путем увеличения объема данных в базе данных.

С помощью объемного тестирования влияние на время отклика и поведение системы можно изучить при воздействии большого объема данных.

**Smoke** тестирование – короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода приложение стартует и выполняет основные функции.

Выполняется поверхностное тестирование наиболее важных модулей приложения на предмет наличия быстронаходимых критических и блокирующих дефектов. В случае отсутствия таковых дефектов приложение передается для полного цикла тестирования, в противном случае приложение уходит на доработку.

**Тестирование графического интерфейса (GUI testing)** — Анализ соответствия графического пользовательского интерфейса программы спецификациям, макетам, прототипам, стандартам. ведет ли себя программное обеспечение в соответствии со спецификацией, когда пользователь взаимодействует с ним с помощью клавиатуры и мыши и т.д.

**Тестирование удобства использования (Usability testing)** — Исследование, выполняемое с целью определения, удобна ли программа для ее предполагаемого применения и основанное на стандартах, лучших практиках и привлечении пользователей в качестве тестировщиков и суммировании и анализе полученных от них выводов.

**Конфигурационное тестирование** (Configuration Testing) — специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.)

**Тестирование документации** (требований)— это начальная стадия процесса тестирования, которая выступает как система раннего оповещения об ошибках. Процесс тестирования так или иначе начинается с документации и требований. Тестирование документации предполагает начало тестирования еще до разработки продукта. Тестировщик может указать на логические ошибки в постановке задачи, несоответствия в требованиях, а также составить чек-лист, список проверок по предоставленному требованию.

**Тестовые двойники –** объекты, помогающее изолироваться и подменять другие объекты и функции.

**Пустышка (Dummy) –** такой двойник не содержит поведения и используется в качестве заполнителя параметров. Никогда реально не вызывается.

**Подделка (Fake) –** это реальная написанная реализация, которая имеет более простую логику, чем реальный аналог.

**Заглушка (Stub) –** имеет заранее подготовленные ответы на вызовы методов. Практически не имеет логики. Используется чтобы заменить взаимодействие, не имеющее побочных эффектов.

Стабы помогают имитировать **входящие (incoming)** взаимодействия. То есть вызовы, совершаемые SUT к ее зависимостям для получения входных данных.

**Шпион (Spy)** – более сложная система. Гибрид реального объекта и мока. Он имеет поведение реального объекта, но может записывать определенную информацию о вызове его методов. Также можно переопределить поведение некоторых его методов.

**Мок (Mock)**. Самый сложный из тестовых двойников, но дающий наибольший контроль. Может иметь сложную логику ответов, зависящих от параметров вызовов, их количества, генерировать исключения при вызове методов с неопределенным поведением и имеет другой, полезный для тестирования функционал.

Моки помогают имитировать и изучать исходящие (outcoming) взаимодействия. То есть вызовы, совершаемые тестируемой системой (SUT) к ее зависимостям для изменения их состояния.

Используется чтобы заменить вызов, совершающий побочное действие.

**Мок vs стаб -** моки помогают эмулировать и изучать взаимодействие, а стабы просто эмулировать. Stub используется как заглушка, с заранее запрограммированным ответом на вызовы.

Mock использует подмену результатов вызова, проверяя сам факт взаимодействия, протоколирует и контролирует его.

**Позитивные и негативные тесты**

**Позитивное –** тест кейс использует только корректные входные данные и проверяет, что приложение правильно выполнило вызываемую функцию.

**Негативное –** тест кейс оперирует как корректными так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций. При таком тестировании часто выполняются некорректные операции.

**Классификация по знанию системы**

Тестирование **Белого ящика –** метод тестирования ПО, который предполагает полный доступ к коду проекта, внутренняя структура и реализация системы известны тестировщику.

Тестирование **Серого ящика –** предполагает частичный доступ к коду проекта.

Тестирование **Черного ящика –** тестирование, основанное на спецификации или тестирование поведения. Не предполагает доступа к системе, работа с внешним интерфейсом системы.

**Ручное тестирование** – это прямое взаимодействие QA-инженера и приложения. В его процессе можно получить обратную связь о продукте, что невозможно, если использовать автоматизированное тестирование. Тестирование от лица конечного пользователя, проверка UI.

Автоматизация наиболее применима в сложных приложениях с большой функциональной частью.

**Нагрузка на приложение**. Когда используется автоматизированное тестирование, становится возможным моделирование большой нагрузки, которая приближена к реальной ситуации.

**Временной фактор.** Ручное тестирование – это долгий и ресурсоемкий процесс, в то время как код для сценария пишется один раз.

**Повторяемость.** Код автотестов может быть использован неоднократно, особенно при внедрении новой функциональности.

**Подходы к написанию тестов**

**Test-Driven Development (TDD) —** разработка через тестирование; подход к разработке и тестированию, при котором сначала создаются тесты, которым должен удовлетворять код, затем его реализация.

Сперва пишется тест на новый, ещё не реализованный функционал, а затем пишется минимальное количество кода (ничего лишнего) для его реализации. При успешном прохождении теста, можно задуматься о качестве кода и сделать его рефакторинг.

* Полное покрытие кода тестами.
* Заранее задумываемся об использовании кода.
* Хорошие тесты являются неплохим примером использования кода (что-то вроде документации).