1. **Что такое процесс тестирования? Для чего?**

Модульное тестирование (unit testing, юнит-тестирование) – вид тестирования, направленный на оценку корректности исходного кода программы.

покрывают атомарные участки кода, что позволяет удостовериться в их работоспособности (в т.ч. после внесения изменений).

Назначение

►Улучшают качество архитектуры приложения

►Стимулируют написание простых методов.

►Упрощают интеграцию кода.

►Помогают документированию кода.

►Минимизируют зависимости в системе.

►Отладка и рефакторинг (скорость, повторный запуск)

1. **Что такое TestNG и Junit?**

**TestNG** – это фреймворк для тестирования, написанный на Java, он взял много чего с JUnit и NUnit, но он не только унаследовался от существующей функциональности Junit, а также внедрил новые инновационные функции, которые делают его мощным, простым в использовании.

**JUnit — библиотека для модульного тестирования программ Java. Созданный Кентом Беком и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit для разных языков программирования, берущей начало в SUnit Кента Бека для Smalltalk. JUnit породил экосистему расширений — JMock, EasyMock, DbUnit, HttpUnit и т. д.**

**Библиотека JUnit была портирована на другие языки, включая PHP (PHPUnit), C# (NUnit), Python (PyUnit), Fortran (fUnit), Delphi (DUnit), Free Pascal (FPCUnit), Perl (Test::Unit), C++ (CPPUnit), Flex (FlexUnit), JavaScript (JSUnit).**

**JUnit – это Java фреймворк для тестирования, т. е. тестирования отдельных участков кода, например, методов или классов. Опыт, полученный при работе с JUnit, важен в разработке концепций тестирования программного обеспечения.**

1. **Что такое юнит тестирование и какие его основные признаки?**

**Модульное тестирование (unit testing) — тесты, задача которых проверить каждый модуль системы по отдельности. Желательно, чтобы это были минимально делимые кусочки системы, например, модули.**

**Unit — модульные тесты, применяемые в различных слоях приложения, тестирующие наименьшую делимую логику приложения: например, класс, но чаще всего — метод. Эти тесты обычно стараются по максимуму изолировать от внешней логики, то есть создать иллюзию того, что остальная часть приложения работает в стандартном режиме.**

**Данных тестов всегда должно быть много (больше, чем остальных видов), так как они тестируют маленькие кусочки и весьма легковесные, не кушающие много ресурсов (под ресурсами я имею виду оперативную память и время).**

**Основная идея модульного тестирования заключается в том, чтобы написать тесты, в которых проверена наименьшая «единица» кода. Модульные тесты обычно написаны на том же языке программирования, что и исходный код приложения. Они создаются непосредственно для проверки этого кода. То есть модульные тесты — это код, который проверяет корректность другого кода.**

1. **Что такое тест-кейс?**

**Тест-кейс — это чёткое описание действий, которые необходимо выполнить, для того чтобы проверить работу программы (поля для ввода, кнопки и т.д.). Данное описание содержит: действия, которые надо выполнить до начала проверки — предусловия; действия, которые надо выполнить для проверки — шаги; описание того, что должно произойти, после выполнения действий для проверки — ожидаемый результат.**

1. **Назначение @Test аннотации?**

**Это самая важная аннотация в TestNG, в которой находится основная логика теста. Все автоматизируемые функции находятся в методе с аннотацией @Test. Она имеет различные атрибуты, с помощью которых может быть настроен запуск метода.**

1. **Что такое test suit? Как его создать?**

**Его коллекция тестов. Он позволяет вам запускать такую ​​коллекцию, как группу.**

1. **Что обеспечивает Assert class?**

**Assert - это специальная конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы. Эта конструкция может автоматически сигнализировать при обнаружении некорректных данных, что обычно приводит к аварийному завершению программы с указанием места обнаружения некорректных данных.**

**8. Поясните назначение @BeforeSuite / @AfterSuite @BeforeTest / AfterTest**

**@BeforeClass / @AfterClass @BeforeMethod / @AfterMethod?**

Значит существует 10 управляемых аннотаций TestNG:

1. @BeforeSuite – указывает, что данный метод будет запускаться перед любым методом тестового класса.

2. @BeforeGroups – аннотирует методы, которые будут выполняться перед первым методом в любой из указанных групп.

3. @BeforeClass – указывает, что метод будет выполнен до всех тестовых методов тестового класса.

4. @BeforeTest – аннотированный метод будет запускаться до всех тестовых методов.

5. @AfterTest – аннотированный метод будет запущен после всех тестовых методов, принадлежащих классам внутри тега <test>.

6. @BeforeMethod – аннотированный метод будет выполняться перед каждым тестовым методом.

7. @AfterMethod – аннотированный метод будет запускаться после каждого тестового метода.

8. @AfterClass – аннотированный метод будет запущен после всех тестовых методов в текущем классе.

9. @AfterGroups – аннотируется методы, которые будут выполняться после всех методом в любом из указанных групп.

10. @AfterSuite – указывает, что данный метод, будет запускаться после всех методов тестового класса.

1. **Как и для чего используют параметризацию в тестировании?**

**При написании unit тестов часто возникает необходимость проводить абсолютно идентичные испытания кода с разными данными. Примером такого кода может быть вычисление квадратного корня — для входных данных выше нуля мы ожидаем получить значения корня, а при передаче числа меньше 0, код должен возвращать ошибку (или, в редакции для эстетов, комплексное число). Очевидно, что для проверки этого поведения придётся написать два абсолютно идентичных теста, отличающихся только данными. Это настолько некрасиво, что хочется сразу с этим что-нибудь сделать.**

**В**[**JUnit**](http://easyjava.ru/category/testing/junit/)**есть механизм, позволяющий отделить код теста от данных теста — параметризованные тесты.**

**10.Что такое TDD и DDT?**

**Подход TDD(Test Driven Development) заключается в том, что прежде, чем написать какой-то код, вы сначала пишете тест, который будет служить спецификацией, то есть определять, что должен делать этот код.**

**Обычно применение концепции «разработка через тестирование» означает использование модульных тестов для управления созданием кода приложения. Но на самом деле этот подход можно применять на любом уровне.**

**Подход TDD переворачивает всё с ног на голову, и вместо того, чтобы сначала писать код, а затем писать модульные тесты для проверки этого кода, вы сначала напишите модульный тест, а затем напишите код, чтобы этот тест стал зелёным. Таким образом, модульное тестирование «управляет» разработкой кода.**

**Этот процесс повторяется снова и снова. Вы пишете еще один тест, который определяет больше функциональности того, что должен делать код. Затем вы пишете и модифицируете код, добиваясь успешного завершения теста.**

**После того, как вы получили «зелёный» результат, вы приступаете к рефакторингу кода, то есть реорганизуете или очищаете его, чтобы сделать более кратким.**

**Часто эту цепочку процессов называют «Красный-Зелёный-Рефакторинг» потому что сначала модульный тест не проходит (красный), затем пишется код, подстраиваясь под тест, добиваясь, чтобы он успешно завершился (зелёный), и, наконец, код оптимизируется (рефакторинг).**

**Data Driven Testing (DDT) – подход к созданию/архитектуре автоматизированных тестов (юнит, интеграционных, чаще всего применимо к backend тестированию), при котором тест умеет принимать набор входных параметров, и эталонный результат или эталонное состояние, с которым он должен сравнить результат, полученный в ходе прогонки входных параметров. Такое сравнение и есть assert такого теста. Притом как часть входных параметров, могут передаваться опции выполнения теста, или флаги, которые влияют на его логику.**