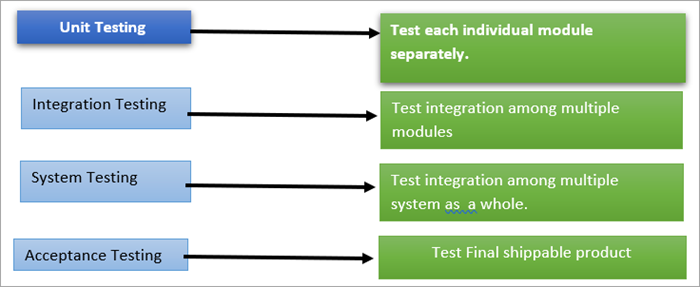


--> Testing is the process of checking the functionality of an application to ensure it runs as per requirements. Unit testing comes into picture at the developers’ level.

-->Test-driven development typically involves writing a test for a certain piece of functionality, running the test to see it fail and then writing the code to make the test pass. That way, developers can be confident that they have written code that does the job and other developers reusing components can run the test to be confident that their own code will properly function.

--> Behavior-driven development typically involves a developer, test engineer and a product manager (and potentially other stakeholders). The group meets to come up with concrete examples of acceptance criteria in a user story. These examples are described using a domain-specific language, like Gherkin, and put into a feature file. The feature file is converted into an executable specification where developers can then write an actual executable test.

**Levels of Testing:**

****

-->Unit testing involves the testing of each unit or an individual component of the software application. It is the first level of functional testing. The aim behind unit testing is to validate unit components with its performance.

-->The purpose of unit testing is to examine the individual components or pieces of methods/classes to verify functionality, ensuring the behaviour is as expected.

--> Unit testing plays a critical role in helping a software company deliver quality products to its customers.

-->Unit testing is of two types

* Manual
* Automated

-->Unit testing is commonly automated but may still be performed manually. Software Engineering does not favour one over the other but automation is preferred. A manual approach to unit testing may employ a step-by-step instructional document.

**Automated Testing is preferred over Manual Testing for the below reasons:**

| **Manual Testing** | **Automated testing** |
| --- | --- |
| When a test case is executed manually without an intervention of a tool is called manual testing. | When a test case is executed with the help of a tool without much manual intervention is called automated testing. |
| Repetitive manual efforts are included. | Repetitive manual efforts may be avoided. |
| Human efforts in manual testing could be erroneous and time consuming. | Automation tests are faster and error free compared to the manual efforts. |
| Testing resources required are more for running every test case manually thereby, adding to the investment in the resources. | Less testers are needed to execute automated tests using the designated automated tool(s) hence there is less investment in testing resources thus adding to the profitability. |
| Manual testing has to be limited to a small test coverage considering the timeline restrictions. Hence, there is a risk of skipping many test scenarios thus leading to risk of defect leakage as well. | Many different test scenarios can be automated and can be executed multiple times even under time and resource crisis hence leading to better test coverage and better quality of the deliverable. |

## Why perform Unit Testing?

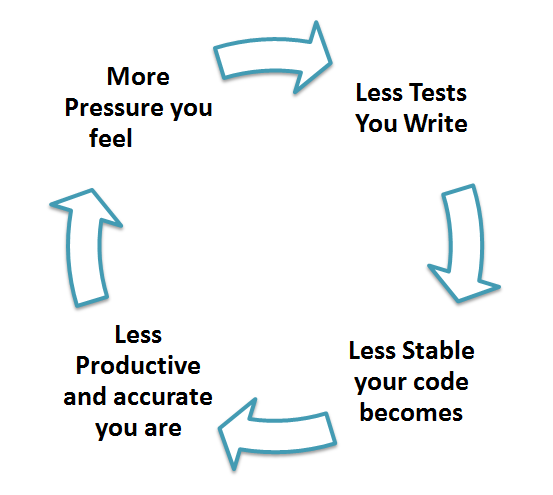
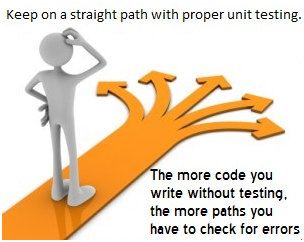
**-->Unit Testing** is important because software developers sometimes try saving time doing minimal unit testing and this is myth because inappropriate unit testing leads to high cost Defect fixing during System Testing, Integration Testing and even Beta Testing after application is built. If proper unit testing is done in early development, then it saves time and money in the end.

Here, are the key reasons to perform unit testing in software engineering:

1. Unit tests help to fix bugs early in the development cycle and save costs.
2. It helps the developers to understand the testing code base and enables them to make changes quickly
3. Good unit tests serve as project documentation
4. Unit tests help with code re-use. Migrate both your code and your tests to your new project. Tweak the code until the tests run again.

## Unit Testing Best Practices

* Unit Test cases should be independent. In case of any enhancements or change in requirements, unit test cases should not be affected.
* Test only one code at a time.
* Follow clear and consistent naming conventions for your unit tests
* In case of a change in code in any module, ensure there is a corresponding unit Test Case for the module, and the module passes the tests before changing the implementation
* Bugs identified during unit testing must be fixed before proceeding to the next phase in SDLC
* Adopt a “test as your code” approach. The more code you write without testing, the more paths you have to check for errors.



**JUNIT**

--> JUnit is an open-source framework that is used for writing and executing unit tests in Java programming language. Now, JUnit is used as a standard when there is a need to perform testing in Java.

## JUnit 5 Architecture

-->JUnit 5 works with Java 8 or above, hence, it supports lambda expression of Java code as well. It is the next generation of JUnit that comes with different versions named as following:

* JUnit Platform
* JUnit Jupiter
* JUnit Vintage

**#1) JUnit Platform:**

* JUnit Platform for JUnit 5 works with the usage of two APIs – one that enables users to create a testing framework and another that helps users to launch a test through the IDE and through the command line.

**#2) JUnit Jupiter:**

* This is a new version that comes with multiple and extensive lists of JUnit annotations.

**#3) JUnit Vintage**

* Sometimes, all that we need is that the older version of JUnit code works as it worked earlier despite the version up-gradation. This basically is used when there are situations to migrate from JUnit 3 or 4 to JUnit 5.

**Annotations:**

**@SpringBootTest:**

**-->**The @SpringBootTest annotation tells Spring Boot to look for a main configuration class (one with @SpringBootApplication, for instance) and use that to start a spring application context.

-->The @SpringBootTest annotation is useful when we need to bootstrap the entire container.

-->The annotation works by creating the ApplicationContext that will be utilized in our tests.

**JUnit lifecycle call back annotations for JUnit 4 vs JUnit 5:**

| **JUNIT 4 ANNOTATION** | **JUNIT 5 ANNOTATION** |
| --- | --- |
| @Before | @BeforeEach |
| @After | @AfterEach |
| @BeforeClass | @BeforeAll |
| @AfterClass | @AfterAll |
| @Test | @Test |

**@Before (JUnit 4) /@BeforeEach (JUnit 5):**

* The annotated method executes before the execution of each test method in the test class.
* This annotation can be used when you wish to have the resources or test data set up just before the initiation of each test.
* **For example,** if there are 5 Test cases in a JUnit test class then the method annotated with @Before/@BeforeEach executes 5 times prior to each of the test case’s execution.

**@After (JUnit 4) /@AfterEach (JUnit 5):**

* The annotated method executes after each test method in the test class executes.
* This annotation can be used when you wish to have to release used resources or test data after each test case runs.
* **For example,** if there are 5 Test cases in a JUnit test class then the method annotated with @After/@AfterEach executes 5 times after the test cases’ execution.

**@BeforeClass (JUnit 4) /@BeforeAll (JUnit 5):**

* The annotated method executes before all the test methods in a test class is executed.
* This annotation can be used when you wish to set up resources or test data at the class level.
* As this method is annotated with @BeforeClass/@BeforeAll is executed only once for a test class and the copy of the method gets shared across the class, and the method must be stated *static.*
* **For example,** if there are 5 Test cases in a JUnit test class then the method annotated with @BeforeClass/@BeforeAll executes once per test class before any test case initiates.

**@AfterClass (JUnit 4) /@AfterAll (JUnit 5):**

* The annotated method executes after all the test methods in a test class executes.
* This annotation can be used when you wish to release the used resources or test data at the class level.
* As this method is annotated with @AfterClass/@AfterAll is executed only once for a test class and the copy of the method gets shared across the class, the method must be stated *static.*
* **For example,** if there are 5 Test cases in a JUnit test class then the method annotated with @AfterClass/@AfterAll executes once per test class after all the test cases complete execution.

**@Test (JUnit 4 & JUnit 5):**

* The @Test annotation is common for JUnit 4 as well as JUnit 5. The annotated methods represent the test cases in the class.
* There could be multiple methods each annotated with @Test in a JUnit class. This implies that a class may have multiple test cases.
* There are different attributes or parameters to Test which one could be passed. You could add a forced time out for a test case or add an exception. This will be covered in detail in a separate tutorial.
* The annotated method cannot be private or static and cannot return any value.
* The @Test method has to be declared as public in JUnit 4 while Junit 5 allows a test case defined without the access modifier ‘public’ as it considers ‘no access modifier’ as ‘public’ by default.

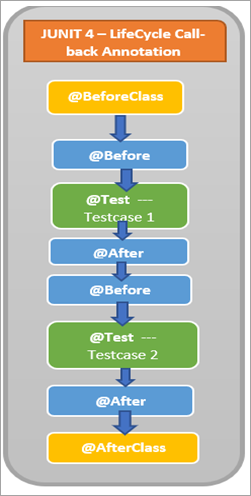
**@Ignore (junit4)/@Disabled (junit5) :**

This annotation can be used if you want to ignore some statements during test execution for e.g. disabling some test cases during test execution.

**@Test (expected=< IllegalArgumentException > .class):**

This annotation can be used if you want to handle some exception during test execution. For, e.g., if you want to check whether a particular method is throwing specified exception or not.

### **Sequential Workflow of the Lifecycle Annotations:**



**@Test (timeout=500) (junit4)/@Timeout (10) (junit5) :**

This annotation can be used if you want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time.

🡪**JUnit 5 test suites** are written with **@Suite** annotation. Suites help us run the tests spread into multiple classes and packages.

🡪We can use Include and Exclude annotations for filtering test packages, test classes or even test methods.

🡪Required dependency

**<dependency>**

**<groupId>**org.junit.platform**</groupId>**

**<artifactId>**junit-platform-suite-engine**</artifactId>**

**<version>**1.8.1**</version>**

**</dependency>**

**@Suite:**

When we want to run the suite, simply run it as a normal JUnit test class and it will execute all the included tests in the suite.

**@SuiteDisplayName:**

-->Use this annotation to give a display name for the annotated test class that is executed as a test suite on the JUnit Platform.

🡪@SelectPackages specifies the names of packages to select when running a test suite via @RunWith (JUnitPlatform.class).

🡪@SelectClasses specifies the classes to select when running a test suite via @RunWith (JUnitPlatform.class).

🡪If you want to exclude any specific package or include any package, then you may use **@IncludePackages** and **@ExcludePackages** annotations.

**🡪@IncludeTags (“production”)** will run the all tests with production inside selected package.

**🡪@ExcludeTags (“development”)** will exclude the all tests with development inside selected package

**Mockito**

**What is mocking?**

* Mocking is a way to test the functionality of a class in an isolation fashion.
* When we use mockito then we do not require a database connection or file server read or properties file to read to test functionality.
* The mocking object usually does the mocking of the real service. A mock object usually returns a dummy data, which corresponds, to some dummy input passed to it.
* Mockito uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but a proxy for actual implementations.

**What is Mockito?**

* The mockito is a very popular mocking framework and it is a java-based library that is used for unit testing of any java applications.
* Mockito is usually used to mock interfaces so that a dummy method can be used for a mock interface that can be used in unit testing.
* The main use of using the Mockito framework is to simplify the development of a test case by mocking external dependencies and then use them in the test code.

**Why need for mocking?**

* If you want to test a component that depends on the other component, but that component is under development. It generally happens when working in a team and different components are divided between several team members. In that case, mocking plays an important role in the testing. Without mocking, we need to wait for the completion of the required elements for testing.
* As you know, the real components perform slow operations while dealing with database connections or another complex read/ write operation. Sometimes the database queries can take a longer time to execute. In such cases, we require mock objects to perform testing and it can be done using mocking.
* Whenever there is any infrastructure concern or issue that makes the testing impossible. For example, when we create a connection to the database, some issues related to configurations occur. It requires mocking for creating mock components to provide unit testing.

**@Mock:**

**-->**Mock annotation is used to create a mock object.

-->The @Mock annotation is always used with @RunWith, a class-level annotation.

**@Spy:**

**-->**Spy annotation is used to create a real object and spy on that real object. This would help to call all the object methods while still tracking every interaction that is being mocked.

-->Below is the simple implementation of @Spy annotation:

@Spy

List<String> spyList = **new** ArrayList<String>();

@Test

**public** **void** whenUsingTheSpyAnnotation\_thenObjectIsSpied() {

spyList.add("one");

spyList.add("two");

Mockito.*verify*(spyList).add("one");

Mockito.*verify*(spyList).add("two");

*assertEquals*(2, spyList.size());

}

## Automatic Mock Injection

**@InjectMocks:**

**-->**InjectMocks annotation is used to mock a class with all its dependencies. This is quite useful to test the behaviour completely.

-->When MockitoAnnotations.openMocks() is called, Mockito will:

* Create mocks for fields annotated with the @Mock annotation
* Create an instance of the field annotated with @InjectMocks and try to inject the mocks into it

-->Using @InjectMocks is the same as we did when instantiating an instance manually, but now automatic.

-->Now we can apply the extension and get rid of the MockitoAnnotations.openMocks() method call.

**@ExtendWith (**MockitoExtension**.**class**)**

* The @ExtendWith annotation is used to load a JUnit 5 extension.
* MockitoExtension looks at the test class, finds member variables annotated with the @Mock annotation, and creates a mock implementation of those variables.
* It then finds member variables annotated with the @InjectMocks annotation and attempts to inject its mocks into those classes, using either construction injection or setter injection.

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>4.5.1</version>

<scope>test</scope>

</dependency>

**-->**If we do not want to share mock variables across all test cases, we can also inject mock objects to method parameters.

**@Test**

**void** **createOrderSetsTheCreationDate(@Mock** OrderRepository orderRepository**)** **{}**

**-->**If you do run into situations, that Mockito does not support, such as mocking static member variables or private constructors, then there is another powerful but complex tool called [**PowerMock**](https://github.com/powermock/powermock).

### **Test Coverage:**

-->The percentage of code that is tested by unit tests is called test coverage.

-->The objective is to have better and more test coverage of the code which in future continues to add up to the regression test suite and helps to increase automated test execution and verification, thereby, reducing the manual effort involved in regression testing.