1. **Testing**:
2. In general Testing is largely divided into the following types:
3. **Unit Tests:**
4. **Integration Tests:**
5. **UI Tests:**
6. **Instrumentation Testing:**
7. **System Test:**
8. Following are some of the testing frameworks used in Android
9. **JUnit4/JUnit**
10. **Espresso**
11. **[Mockito](https://www.journaldev.com/21816/mockito-tutorial)**
12. **Powermock**
13. **Robolectric**
14. **Hamcrest**

**Testing:** What is Alfa Testing, Beta Testing, WhiteBox Testing, and Black Testing, Unit Testing, Integral Testing, System Testing, JUnit Testing, Android Espresso testing, IOS XC Testing. TestNG, Appium, Selenium

**Unit Testing Or Local Unit Test**

1. It focuses on the smallest unit of software design or code. In this, we test an individual unit or group or functionality of interrelated units. It is often done by the programmer by using sample input and observing its corresponding outputs.
2. A *local* test runs directly on your own workstation, rather than an Android device or emulator. As such, it uses your local Java Virtual Machine (JVM), rather than an Android device to run tests.  Local tests enable you to evaluate your app's logic more quickly. A *unit* test verifies the behavior of a small section of code, the *unit under test*. It does so by executing that code and checking the result. This is written in - src/test/java package
3. This testing methodology is done during the development process by the software developers and sometimes QA staff.
4. To perform unit testing, a developer writes a piece of code (unit tests) to verify the code to be tested (unit) is correct.
5. We generally use Unit Testing to test business logic of application whether thay are working as expected or not. It is faster & less expensive test and it’s run on local JVM & not required Emulator/physical device to run/ the unit test cases
6. Unit Testing is used to verify a small chunk of code by creating a path, function or a method.
7. **Unit testing** is a software testing technique in which individual components/parts of the software is tested, i.e., a group of computer programs, usage procedure, etc. Unit testing of an object is done during the development of an application or project. The aim of unit testing is to isolate a segment of code (unit) and verifies its correctness. A unit is referred to as an individual function or procedure (program). The developers usually perform it during testing.
8. Tools/Framework for Unit Testing:
9. **JUnit**: JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit that originated with SUnit.
10. **Mockito:** Mockito is a Java-based mocking framework used for unit testing of Java application. Mockito plays a crucial role in developing testable applications. Mockito was released as an open-source testing framework under the MIT Licence. We can also use Mockito with other testing frameworks ike JUnit and TestNG.
11. **Truth library:** Truth is a library for performing assertions/check the condition in tests

6. Unit Testing and its importance can be understood by below-mentioned points:

* Unit Testing is used to identify defects early in software development cycle.
* Unit Testing will compel to read our own code. i.e. a developer starts spending more time in reading than writing.
* Defects in the design of code affect the development system. A successful code breeds the confidence of developer.

Example:

a) In a program we are checking if the loop, method, or function is working fine or not

b) Misunderstood or incorrect, arithmetic precedence.

c) Incorrect initialization

d) This is written in - src/test/java package

**Instrumentation Tests**

1. Instrumented tests run on Android devices, whether physical or emulated. As such, they can take advantage of the Android framework APIs. Instrumented tests therefore provide more fidelity than local tests, though they run much more slowly.
2. To run instrumented UI tests using Android Studio, you implement your test code in a separate Android test f older - src/androidTest/java
3. Jetpack includes various frameworks that provide APIs for writing UI tests:

* The [Espresso testing framework](https://developer.android.com/training/testing/espresso) (Android 4.0.1, API level 14 or higher) provides APIs for writing UI tests to simulate user interactions with *Views* within a single target app. A key benefit of using Espresso is that it provides automatic synchronization of test actions with the UI of the app you are testing. Espresso detects when the main thread is idle, so it is able to run your test commands at the appropriate time, improving the reliability of your tests.
* [Jetpack Compose](https://developer.android.com/jetpack/compose) (Android 5.0, API level 21 or higher) provides a set of [testing APIs](https://developer.android.com/jetpack/compose/testing) to launch and interact with Compose screens and components. Interactions with Compose elements are synchronized with tests and have complete control over time, animations and recompositions.
* [UI Automator](https://developer.android.com/training/testing/ui-automator) (Android 4.3, API level 18 or higher) is a UI testing framework suitable for cross-app functional UI testing across system and installed apps. The UI Automator APIs allows you to perform operations such as opening the Settings menu or the app launcher on a test device.
* [Robolectric](http://robolectric.org/) (Android 4.1, API level 16 or higher) lets you create local tests that run on your workstation or continuous integration environment in a regular JVM, instead of on an emulator or device. It can use Espresso or Compose testing APIs to interact with UI components.

**Integration Testing**

1. The objective is to take unit-tested components and build a program structure that has been dictated by design. Integration testing is testing in which a group of components is combined to produce output.
2. Integration test can be when other app component required to test particular unit/functionality/code like(Context, Activity, etc)
3. it is a type of software testing in which the different units, modules or components of a software application are tested as a combined entity.
4. testing two or more separate functions, or component groups to ensure they work as expected when integrated.

Integration testing is of four types: (i) Top-down (ii) Bottom-up (iii) Sandwich (iv) Big-Bang

Example

(a) **Black Box testing:**- It is used for validation. In this, we ignore internal working mechanisms and focus on what is the output. **[Black Box Testing](https://www.geeksforgeeks.org/software-engineering-black-box-testing/)** is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. Only the external design and structure are tested.

(b) White box testing:- It is used for verification. In this, we focus on internal mechanisms i.e. how the output is achieved?. [White Box Testing](https://www.geeksforgeeks.org/software-engineering-white-box-testing/) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. Implementation and impact of the code are tested.

Image

**Differences between Black Box Testing vs White Box Testing:**

|  |  |
| --- | --- |
| Black Box Testing | White Box Testing |
| It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it. | It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software. |
| Implementation of code is not needed for black box testing. | Code implementation is necessary for white box testing. |
| It is mostly done by software testers. | It is mostly done by software developers. |
| No knowledge of implementation is needed. | Knowledge of implementation is required. |
| It can be referred as outer or external software testing. | It is the inner or the internal software testing. |
| It is functional test of the software. | It is structural test of the software. |
| This testing can be initiated on the basis of requirement specifications document. | This type of testing of software is started after detail design document. |
| No knowledge of programming is required. | It is mandatory to have knowledge of programming. |
| It is the behavior testing of the software. | It is the logic testing of the software. |
| It is applicable to the higher levels of testing of software. | It is generally applicable to the lower levels of software testing. |
| It is also called closed testing. | It is also called as clear box testing. |
| It is least time consuming. | It is most time consuming. |
| It is not suitable or preferred for algorithm testing. | It is suitable for algorithm testing. |
| Can be done by trial and error ways and methods. | Data domains along with inner or internal boundaries can be better tested. |
| **Example:** search something on google by using keywords | **Example:** by input to check and verify loops |
| **Types of Black Box Testing:**    * A. Functional Testing * B. Non-functional testing * C. Regression Testing | **Types of White Box Testing:**    * A. Path Testing * B. Loop Testing * C. Condition testing |

**Regression Testing**

1. Every time a new module is added leads to changes in the program. This type of testing makes sure that the whole component works properly even after adding components to the complete program.
2. Regression testing is testing existing software applications to make sure that a change or addition hasn't broken any existing functionality.

**Smoke Testing**

1. This test is done to make sure that the software under testing is ready or stable for further testing. It is called a smoke test as the testing of an initial pass is done to check if it did not catch the fire or smoke in the initial switch on.
2. After releasing the build to QA environment, Smoke Testing is performed by QA engineers/QA lead. Whenever there is a new build, QA team determines the major functionality in the application to perform smoke testing
3. Smoke Testing is done whenever the new functionalities of software are developed and integrated with existing build that is deployed in QA/staging environment. It ensures that all critical functionalities are working correctly or not.

Example

If the project has 2 modules so before going to the module make sure that module 1 works properly

**Alpha Testing**

1. This is a type of validation & acceptance testing which is done before the product is released to customers. It is typically done by QA people or developer of internal organisation .
2. d

**Beta Testing**

The beta test is conducted at one or more customer sites by the end-user of the software. This version is released for a limited number of users for testing in a real-time environment for real users.

**Android Google Play Store Testing**

1. **internal testing :** Create an internal testing release to quickly distribute your app to up to 100 testers for initial quality assurance checks. We recommend running an internal test before releasing your app to the closed or open tracks.
2. Tester: Play Store -> DashBoard -> Testing -> Internal Testing -> Tester -> Create tester / email list(list name, add multiple email by separating , between them)
3. Or you can select Closed testing group
4. Add your feedback url or emailid, where tester will provide feedback for your app
5. testers can join to test your app on the web. Copy the test app link & share/distribute them to all tester. It can done only by distributing the test link(Testers can join your test on the web)
6. If you want to run internal tests parallelly with closed and open tests then create different versions of your app.
7. **closed Testing - Alpha Testing:** Create a closed testing release to test pre-release versions of your app with a wider set of testers to gather more targeted feedback. Once you've tested with a smaller group of colleagues or trusted users, you can expand your test to an open release.  On your Closed testing page, a Closed testing track will be available as your initial closed test. If needed, you can also create and name additional closed tracks. If you're testing an existing app that you've published before, only users in your test group will receive an update for your closed version.
8. Tester: Play Store -> DashBoard -> Testing -> Closed Testing -> Tester -> Create tester / email list(list name, add multiple email by separating , between them)
9. Or you can select Internal testing group
10. Add your feedback url or emailid, where tester will provide feedback for your app
11. How testers join your test
    1. Join on Android: Testers can join your test using Google Play on Android
    2. Join on the web: Testers can join your test on the web
12. **open:** Create an open testing release to run a test with a large group and surface your app's test version on Google Play. If you run an open test, anyone can join your testing program and submit private feedback to you. Before choosing this option, make sure your app and store listing is ready to be visible on Google Play. Create and manage open testing releases to make your app available to testers. Anyone can join your tests on Google Play.
13. Tester: Play Store -> DashBoard -> Testing -> Open Testing -> Tester ->
14. You can manage No of tester by electing Limited Or Unlimited tester
15. Add your feedback url or emailid, where tester will provide feedback for your app
16. How testers join your test
    1. Join on Android: Testers can join your test using Google Play on Android
    2. Join on the web: Testers can join your test on the web

**WhiteBox Testing:**

**BlackBox Testing:**

**Alfa Testing:**

**Beta Testing:**

**Internal Testing:**

**Espresso**

Espresso comes under the **Instrumentation**  type of testing . It is used to do automated UI testing by writing short and concise Android UI tests. Espresso tests are written inside src | **androidTest** folder files. ***JUnit*** is the popular unit-testing framework developed by the Java community upon which the **espresso testing** framework is build.

**Most Freequest Annotation for Espresso Testing**

1. **@Test:** @Test is used to differentiate a normal method from the test case method. @Test annotation, then that particular method is considered as a Test case and will be run by JUnit Runner.
2. **@Before:** @Before annotation is used to refer a method, which needs to be invoked before running any test method available in a particular test class
3. **@After:** @After will be called or executed after each test case is run.
4. **@BeforeClass:** @BeforeClass will be called or executed only once before running all test cases in a particular class. It is useful to create resource intensive object like database connection object.
5. **@AfterClass:** @AfterClass will be called or executed only once after all test cases in a particular class are run.
6. **@Rule:** It is used to add behavior to the test cases. We can only annotate the fields of type **TestRule**

**Let’s look at the components of the Espresso Instrumentation Testing Framework:**

**Android:** <https://developer.android.com/training/testing/espresso/cheat-sheet>

1. **ViewMatchers:** allows us to find a view in the view hierarchy.
2. **ViewActions:** allows us to perform automated actions on the view. Such as clicks etc.
3. **ViewAssertions:** allows us to assert the state of the view. Assertion is a way of checking whether the expected value of the test case matches the actual value of the test case result.

onView(ViewMatcher)

.perform(ViewAction)

.check(ViewAssertion)

1. **onView(ViewMatcher)**
2. withID(): Pass the unique id of view widget
3. withText(): Pass the text of the view. It searches for a view with the specified text.

### withContentDescription():

1. **perform(ViewAction)**
2. click(): Clicks the view passed in onView.
3. typeText(): Pass the string to be entered in the View. This is especially used in EditText.
4. replaceText(): Replaces the current text with the string that’s passed
5. closeSoftKeyboard(): Dismisses the keyboard.

### doubleClick()

### longClick()

### pressBack()

### openLink()

### scrollTo()

### swipeDown()

### swipeUp()

### swipeRight()

### swipeLeft()

### clearText()

### pressKey()

### pressMenuKey()

1. **check(ViewAssertion)**
2. **match**()
3. **doesNotExist**()
4. **fail()** − To explicitly make a test case fail.
5. **assertTrue(boolean test\_condition)** − Checks that the test\_condition is true
6. **assertFalse(boolean test\_condition)** − Checks that the test\_condition is false
7. **assertEquals(expected, actual)** − Checks that both values are equal
8. **assertNull(object)** − Checks that the object is null
9. **assertNotNull(object)** − Checks that the object is not null
10. **assertSame(expected, actual)** − Checks that both refers same object.
11. **assertNotSame(expected, actual)** − Checks that both refers different object.

## **Appium**

**Appium** is an open-source automation mobile testing tool, which is used to test the application. It is developed and supported by **Sauce Labs** to automate native and hybrid mobile apps. Appium is an HTTP server that is written in **node.js**. Appium is very much similar to the **Selenium Webdriver** testing tool.Appium has **NO dependency** on mobile device OS because it has a framework that converts the Selenium Webdriver commands to UIAutomator and UIAutomation commands for Android and iOS respectively.

<https://www.youtube.com/playlist?list=PLUDwpEzHYYLsx_2JFNBMITjHqTnuszhb_/>

## Features of Appium

* Appium does not require application source code or library.
* Appium provides a strong and active community.
* Appium has multi-platform support i.e., it can run the same test cases on multiple platforms.
* Appium allows the parallel execution of test scripts.
* In Appium, a small change does not require re-installation of the application.
* Appium supports various languages like C#, Python, Java, Ruby, PHP, JavaScript with node.js, and many others that have Selenium client library.

## Prerequisites to use Appium

* Install **Java**(JDK): <https://www.oracle.com/in/java/technologies/javase-downloads.html/>
* Install **Android Studio** <http://www.automationtestinghub.com/download-and-install-android/>
* Install **Appium jar** file: <http://appium.io/downloads.html/>
* js (Not required - It by default comes with "node.js" and "NPM", whenever the Appium server is installed. Therefore, it is not required to install node.js and NPM separately. It is already included in current version of Appium.)
* Install **Appium Desktop Client:** [http://appium.io/](https://appium.io/)
* Install **Eclipse IDE** for Java: <https://www.eclipse.org/downloads/> <https://www.pavantestingtools.com/2019/02/eclipse-configuration.html>

### 

### **Configure Eclipse Editor**

Downloadable's:

[Appium Client Library (java-client-7.0.0.jar)](https://appium.io/downloads.html) : <https://appium.io/downloads.html/>

[Selenium stand-alone server( Selenium Java jars)](https://docs.seleniumhq.org/download/): <https://docs.seleniumhq.org/download/>

[Commons\_lang3 (commons-lang3-3.8.1.jar)](http://central.maven.org/maven2/org/apache/commons/commons-lang3/3.8.1/commons-lang3-3.8.1.jar): <https://mvnrepository.com/artifact/org.apache.commons/commons-lang3/3.8.1/>

### Add all the jar files to Project Build Path

* Right click the project name and navigate to **Build Path**.
* Select Configure **Build Path**... and the project properties window will appear showing your **build path** configurations.
* Select the Libraries tab.
* Click Add **JARs**...
* Locate and select the **JARs** you want and click OK. ...
* Click OK to close the properties window.