Android Testing Fundamentals

Contents

Chapter 1

Android Testing Guidelines

What should I test?

1.1 Activity Testing

Screen Sizes and Resolutions Verify that interfaces display correctly on all target devices you support.

- Test it on all of the screen sizes and densities on which you want it to run.
- You can test the application on multiple sizes and densities using AVDs, or you can test your application directly on the devices that you are targeting.

Run-Time Configuration Changes Test that each activity responds correctly to the possible changes in the device's configuration while your application is running.

- These include a change to the device's:
  - Orientation
  - Language and region
  - Day/night mode
  - Docking status
  - Physical keyboard hidden or visible

1.2 Activity Testing (cont.)

Input Validation Test that each Activity responds correctly to input values.

- Verify key events are reflected correctly in the Views.
- Verify that context-sensitive Views change their state correctly in response to input.
- Verify error messages and dialogs in response to invalid input.
- Test different input methods, such as touch events.

1.3 Activity Testing (cont.)

Lifecycle Events Test that each Activity handles lifecycle events correctly.

- In general, lifecycle events are actions, either from the system or from the user, that trigger a callback method such as `onCreate()` or `onClick()`.
- For example, an Activity should respond to pause or destroy events by saving its state.
- Remember that even a change in screen orientation causes the current Activity to be destroyed, so you should test that accidental device movements don't accidentally lose the application state.

INTENTS

- Test that each Activity correctly handles the Intents listed in the intent filter specified in its manifest.
- Test valid and invalid permission, if you restrict access by permission.
- Test URI and extra data information associated with Intents. Verify correct behavior if URI or extras are missing or invalid.

1.4 Service Testing

Lifecycle Events Test that each Service handles lifecycle events correctly.

- Ensure that the `onCreate()` is called in response to `Context.startService()` or `Context.bindService()`.
- Ensure that `onDestroy()` is called in response to `Context.stopService()`, `Context.unbindService()`, `stopSelf()`, or `stopSelfResult()`.
- Test that the Service correctly handles multiple calls from `Context.startService()`. Only the first call triggers `Service.onCreate()`, but all calls trigger a call to `Service.onStartCommand()`.
- Remember that `startService()` calls don't nest, so a single call to `Context.stopService()` or `Service.stopSelf()` (but not `stopSelf(int)`) will stop the Service. Test that the Service stops at the correct point.

1.5 Service Testing (cont.)

Business Logic Test any business logic that your Service implements. Business logic includes checking for invalid values, financial and arithmetic calculations, and so forth.

INTENTS

- Test that each Service correctly handles the Intents listed in the intent filter specified in its manifest.
- Test valid and invalid permission, if you restrict access by permission.
- Test URI and extra data information associated with Intents. Verify correct behavior if URI or extras are missing or invalid.

1.6 Content Provider Testing

Public Contract If you intend your provider to be public and available to other applications, you should test it as a contract.

- Test with constants that your provider publicly exposes. For example, look for constants that refer to column names in one of the provider's data tables. These should always be constants publicly defined by the provider.
- Test all the URIs offered by your provider. Your provider may offer several URIs, each one referring to a different aspect of the data.
- Test invalid URIs: Your unit tests should deliberately call the provider with an invalid URI, and look for errors. Good provider design is to throw an `IllegalArgumentException` for invalid URIs.

Standard Provider Interactions Test the standard provider interactions.

- Most providers offer six access methods: query(), insert(), delete(), update(), getType(), and onCreate().

Business Logic Test the business logic that your provider should enforce.

- Business logic includes handling of invalid values, financial or arithmetic calculations, elimination or combining of duplicates, and so forth.

1.7 Other Testing: External Resource Dependencies

If your application depends on network access, SMS, Bluetooth, or GPS, then you should test what happens when the resource or resources are not available.

You can use the emulator to test network access, bandwidth, and GSP. The test can be performed manually, or scripted using the emulator console.

Chapter 2

JUnit Basics

2.1 Why Focus on Testing?

- Testing is a critical activity for ensuring that a software system meets its functional and non-functional requirements.
- Testing activities are a part of a source software development lifecycle where testing is done at different levels of granularity and complexity.
- We cannot make reasonable claims about quality of a particular software product or product line without systematic testing results.

Testing methods may span different dimensions:

Granularity
: Including system, subsystem, component, unit

Coverage
: Such as line coverage, module coverage, feature coverage

Component Dependence
: Including black box testing, white box testing, black/white (grey) box testing

Automation Level
: Such as manual, semi-automated, automated

Validation Method
: Including no validation, using assertions, formal methods

Metric Dependence
: Including product metrics (e.g., robustness, security, etc.), business metrics (e.g., budget and time constraints)

2.2 Test, Test, and Test Some More

How frequently should we test our code?

- We should test our code whenever we have time available for it.
- Unfortunately, testing activities require addition time and result in additional development costs.
- They also detract from business metrics of productivity (e.g., number of features implemented, amount of chargeable hours invested).
- We code, compile, then run, thereby testing the code written.
- We test individual lines of code or we test particular behavior (e.g., selecting a particular command or entering specific input).

Most developers have their own "testing patterns," such as running a small testing suite of test cases after each significant change.

- The testing activities can be viewed as a break from the analytical and problem solving programming tasks.
- As a result, testing is rarely given equal importance as programming, and exhaustive testing is seen as a secondary development activity.
- Because test cases reflect software requirements, they must be updated and expanded along with corresponding code changes.

2.3 About JUnit

JUnit is a *testing framework* intended to promote structured approach to automated unit testing of software written in Java.

- It represents a reusable pattern (structure) for testing Java code.
- It was developed by Erich Gamma (co-author of *Design Patterns*) and Kent Beck (known for work on Extreme Programming) in 1997.
  - It is an instance of *xUnit* testing architecture that offers instances for testing other languages such as C/C++, Python, and Perl.
- JUnit is now an open-source project hosted by GitHub: https://github.com/KentBeck/junit
  - The JUnit home page is: http://www.junit.org
  - It is distributed under the Common Public License, thereby making it easier to be incorporated into commercial tools.
- In the context of JUnit, a *unit test* refers to testing a "unit of work," such as a method, a group of related methods, or even a class.
  - The key distinction is **isolation** of the unit from other units.
  - JUnit is **not** intended for testing unit interaction such as *integration testing* of components, *feature testing*, or *system testing*.

2.4 Advantages of JUnit Testing

JUnit tests are easy to develop since they are written in Java, and they are more reflective of the software requirements.

- For each test, typically we need to decide how to define the test, what test variables to instantiate, and how to analyze the output.
- With JUnit, we need to focus on the test itself, as the framework resolves questions about the test case instantiation and output.

Unit test cases cost less since much of the creation and maintenance overhead is eliminated through the framework.

- The need for defining a new test suite structure for every module is eliminated.
- Individual test cases run independently of each other so the need for testing coordination is decreased.
- JUnit structure does not change but additional suites can be added or existing ones extended.
- Overhead of deciding how to create individual test cases is reduced through a familiar test-suite assertions interface.

2.5 Advantages of JUnit Testing (cont.)

JUnit provides a method for creation of hierarchical test suites.

- Individual test classes can be composed into test suits.
- Test suits can be composed into other test suits to create a hierarchy of testing modules that reflects the code structure.

JUnit tests internally check results and instantly provide feedback.

- With the use of assertions, the need for an output language and its analysis is eliminated.
- JUnit provides a summary of passed and failed tests, with corresponding details for each one of the tests that failed.

The use of JUnit tests leads to increase in software quality.

- By promoting simplicity of writing test cases and decreasing cost of creating, running, and maintaining test suites, the appeal of testing to developers should increase.
- Given that the JUnit suites can be rerun inexpensively after each change or a set of changes, disruption to the expected software operation can be detected earlier.
- Test-driven development promotes focus on satisfying software requirements, and in turn leads to higher quality software.

2.6 Disadvantages of JUnit Testing

JUnit does not guarantee better quality software.

- It is a framework that promotes the use of testing in software development.
- Software systems are inherently complex and their inherent complexity is present even at the lowest levels of abstraction.
- Creation of test cases and testing suites in highly complex or critical domains may require a more dedicated approach to testing such as the use of formal methods or simulations.
- Moreover, using JUnit by itself will not improve quality unless it is a part of a structured system testing approach.

JUnit does not invalidate previous testing methods.

- It provides a framework that simplifies test case definition and use, but the analysis algorithms such as boundary-case analysis still apply.

2.7 JUnit 101

JUnit is a combination of two concepts:

Design patterns	The framework in its structure represents an instance of a Command design pattern
Assertions	The framework makes use of assertions to generate output of individual tests

JUnit is meant to follow the structure of the code:

- For each Java class, typically there is a JUnit *test class*.
  - Test classes must have a no-argument `public` constructor.
  - In JUnit 3 and earlier, each JUnit test class extends the `junit.framework.TestCase` base class, which provides the JUnit framework interfaces.
  - JUnit 4 no longer requires test classes to extend any base class.
- Within the JUnit test class, *test methods* are defined to define the actual tests.
  - Test methods must be declared `public void` with no arguments.
  - In JUnit 3 and earlier, test methods must be named starting with the prefix `test`.
  - In JUnit 4, test methods may have any name, but must be annotated with a `@org.junit.Test` annotation.
- Within each test method, `assert`*`Type`* assertions are used to check desired testing conditions.

JUnit Class Relationships

Figure 2.1: JUnit Class Relationships

Important
The Android testing framework is built on JUnit 3, and does not currently support JUnit 4.

2.8 Assertions

JUnit uses *assertions* to determine whether a test method passes or fails.

- An *assertion* raises an `AssertionError` exception if the asserted condition is not met.
- The `assert` statement was added in Java 1.4.

JUnit defines a variety of convenience static methods for testing common conditions.

- JUnit 3 assertion static methods are declared in `junit.framework.Assert`.
- JUnit 4 assertion static methods are declared in `org.junit.Assert`.

The following static assertion methods are available in both JUnit 3 and 4:

- `assertTrue(boolean condition)`
- `assertFalse(boolean condition)`
- `assertNull(Object o)`
- `assertNotNull(Object o)`
- `assertEquals(`*`Type`* `expected, `*`Type`* `actual)`
- `assertEquals(float expected, float actual, float delta)`
- `assertEquals(double expected, double actual, double delta)`
- `assertSame(`*`Type`* `expected, `*`Type`* `actual)`
- `assertNotSame(`*`Type`* `expected, `*`Type`* `actual)`
- `fail()`

Note
The `assertSame()` method tests whether two reference variables refer to the *same object*, whereas `assertEquals()` uses `expected.equals(actual)` to determine equality of two objects.

JUnit 3 and 4 also provide overloaded versions of these methods that take an additional `String` message as their first parameter to provide more meaningful reporting in case of failures.

JUnit 4 added:

- `assertArrayEquals(`*`Type`*`[] expected, `*`Type`*`[] actual)`
- `assertArrayEquals(float[] expected, float actual[], float delta)`
- `assertArrayEquals(double[] expected, double actual[], double delta)`

The JUnit 3 `junit.framework.TestCase` class extends `junit.framework.Assert`, so you are able to use all the JUnit assertion methods without importing them.

For JUnit 4, because you are not required to derive a text class from a specific class, you must access the assertion methods with package-qualified names or use the Java 5 static import feature. For example, to import all of the assertion methods for use in your test class:

```
import static org.junit.Assert.*;
```

2.9 Getting Started with JUnit

Most modern IDEs already come with built-in support for JUnit.

- For example, the Java Development Tools (JDT) plugin for Eclipse includes JUnit.

Recommendations for using JUnit in an Eclipse project:

- Store JUnit test classes in the same Java project as the product code.
- Create a separate *source folder*, by convention named `test`, in your Java project for your test classes.
  - Select File ⇒ New ⇒ Source Folder.
- JUnit test classes should be in the same package as the classes they test.
  - You avoid needing to `import` your product classes into your test class files.
  - Your test classes can test `protected` methods in your product classes.
- Remember to add the JUnit JAR to the Java Build Path for your project.
  - Eclipse's New JUnit Test Case wizard can do this automatically when you create a JUnit test class.

If you need to download JUnit:

JUnit 4	The latest version of JUnit 4 is available for download from: https://github.com/KentBeck/junit/downloads
JUnit 3	The latest version of JUnit 3 is available for download from: http://sourceforge.net/projects/junit/files/junit/

The ZIP files contain the JUnit JAR files, JavaDoc of the APIs, and additional documentation and examples.

2.10 Creating a New JUnit Test Case in Eclipse

To create a new JUnit test case in Eclipse, select File ⇒ New ⇒ JUnit Test Case.

Tip
If you select the class to test before bringing up the New JUnit Test Case wizard, Eclipse automatically populates most fields with values based on the class under test.

Eclipse New JUnit Test Case wizard

Figure 2.2: Eclipse New JUnit Test Case wizard

2.11 Simple Example: Domain Class

This class will serve as the target of our example JUnit tests:

```
package com.marakana.demo;

public class Account {
    private int balance;

    public Account(int balance) {
        this.balance = balance;
    }

    public int credit(int increase) {
        balance += increase;
        return balance;
    }

    public int debit(int decrease) {
        balance -= decrease;
        return balance;
    }

    public int getBalance() {
        return balance;
    }
}
```

2.12 Simple JUnit 4 Test Class

```
package com.marakana.demo;

import org.junit.Test;
import static org.junit.Assert.*;

public class AccountTestV4 {

    @Test
    public void testIncreaseBalance() {
        float delta = 0.001f;

        Account a1 = new Account(10f);
        assertNotNull(a1);

        // A dangerous way to test for floating point equality:
        assertTrue(a1.getBalance() == 10f);

        // A much better way -- equal within a certain tolerance:
        a1.credit(5f);
        assertEquals(a1.getBalance(), 15f, delta);

        a1.credit(-5f);
        assertEquals(a1.getBalance(), 10f, delta);
    }

}
```

2.13 Simple JUnit 3 Test Class

```
package com.marakana.demo;

import junit.framework.TestCase;

public class AccountTestV3 extends TestCase {

    public void testIncreaseBalance() {
        float delta = 0.001f;

        Account a1 = new Account(10f);
        assertNotNull(a1);

        // A dangerous way to test for floating point equality:
        assertTrue(a1.getBalance() == 10f);

        // A much better way -- equal within a certain tolerance:
        a1.credit(5f);
        assertEquals(a1.getBalance(), 15f, delta);

        a1.credit(-5f);
        assertEquals(a1.getBalance(), 10f, delta);
    }

}
```

2.14 Testing for Exceptions in JUnit 4

You should include *negative* test cases as well as *positive* ones.

- Positive test cases verify the correct results for valid inputs.
- Negative test cases verify correct error handling for invalid inputs.

In Java, this means that we need to detect for exceptions thrown under certain circumstances.

- The `@Test` annotation supports an `expected` parameter indicating the expected exception class.
- For example, assume `Account` has a method declared:

```
public void debit(float amount) throws NonSufficientFundsException { // ... }
```

- You can then test for the NSF exception:

```
@Test(expected=NonSufficientFundsException.class)
public void testNSF() throws NonSufficientFundsException {
    Account a = new Account(0f);
    a.debit(100f);
}
```

Note
When testing for a checked exception, the test method must also declare that it throws the exception.

2.15 Testing for Exceptions in JUnit 3

In JUnit 3, to test for an exception properly thrown by the test target:

- `catch` the expected exception type, and
- Use `fail()` to fail the test after the point in your code where the exception should have occurred.

For example, to test for the NSF exception from `Account.withdraw()`:

```
public void testNSF() {
    Account a = new Account(0f);
    try {
        a.debit(100f);
        fail("NSF exception expected");
    }
    catch (NonSufficientFundsException expected) {}
}
```

2.16 Test Method Isolation

When the JUnit framework executes your test class, each test method in the class is executed in isolation.

- The JUnit framework creates a *separate* instance of the class for each test method it executes. For example, if a test class contains three test methods, three separate instances of the class are created, one for each method.
- JUnit makes no guarantee as to the order in which test methods are executed.
- This also implies that test methods must have no dependencies on each other. A test method must not assume that the test target is left in a particular state from a previous test method.

2.17 JUnit Test Results

When JUnit runs, it executes all `test*` methods. The result of each execution is recorded as:

Success	The method passed all asserts and didn't throw any exceptions
Failure	The method failed at least one assertion. The output includes the line number of the failure and a stack trace.
Error	The method failed with an *unexpected* exception (for example, a `RunTimeException`). This can indicate an error in the logic of your test method or an unforeseen exception in your test target. The output includes the line number of the failure and a stack trace.

2.18 Running Tests with Eclipse

To execute a JUnit test class in Eclipse:

- Select the test class
- Select Run ⇒ Run As ⇒ JUnit Test

Eclipse opens a JUnit view to display the test results, for example:

Eclipse JUnit Test Results View

Figure 2.3: Eclipse JUnit Test Results View

2.19 Running Tests from the Command Line

JUnit provides provides a default test runner application class that you can use to execute JUnit tests from the command line.

- In JUnit 4, execute:

```
java org.junit.runner.JUnitCore TestClassName
```

> **Note**
> The JUnit 4 test runner also supports executing legacy JUnit 3 tests.

- In JUnit 3, execute:

```
java junit.textui.TestRunner TestClassName
```

Example output:

```
JUnit version 4.8.2
.E.
Time: 0.009
There was 1 failure:
1) testIncreaseBalance(com.marakana.demo.AccountTestV4)
java.lang.AssertionError:
    at org.junit.Assert.fail(Assert.java:91)
    ....
```

JUnit 3 and 4 allow you to create your own test runner classes to customize test execution and reporting. See the JUnit documentation for more details.

JUnit 3 also included two other graphical test execution classes, which are not supported in JUnit 4:

- `junit.swingui.TestRunner`, using the Swing GUI framework
- `junit.awtgui.TestRunner`, using the AWT GUI framework

2.20 Test Fixtures

Test methods within a test class may require a common setup and/or cleanup (referred to as a *test fixture*).

To avoid redundancy, you optionally can consolidate common setup behavior into one method and common cleanup before into another method.

- The setup method will be executed before **each** test method invocation in that test class.
- The cleanup method will be executed after **each** test method invocation in that test class.
- The setup and cleanup methods must be `public void` methods that take no arguments.

In JUnit 4:

- Annotate your setup method with the `@Before` annotation
- Annotate your cleanup method with the `@After` annotation
- You can choose any names you like for your setup and cleanup methods.

In JUnit 3:

- Name your setup method `setup()`
- Name your cleanup method `tearDown()`

JUnit 4 also gives you the option of designating a method to execute **once** before any of the test class's test methods execute, and another method to execute **once** after all of the test class's test methods have executed.

- This feature is often used to perform time-consuming initialization of a test resource, such as a database connection or an application server.
- This feature is **not available** in JUnit 3.

To designate class-wide setup and cleanup methods:

- Annotate your class-wide setup method with `@BeforeClass`.
- Annotate your class-wide cleanup method with `@AfterClass`.
- You can choose any names you like for your setup and cleanup methods.
- Both methods must be `public static void` methods that take no arguments.

Remember that a separate instance of your test class is created for each test method it contains. Within that instance, JUnit first invokes your `@Before`/`setUp()` method, then the `@Test`/`test` method for that instance, and finally the `@After`/`tearDown()` method.

2.21 Combining Test Cases

You can combine test classes into a *test suites*. Test suites also can be combined into bigger test suites (i.e., a test suite can contain other "nested" test suites).

To define a test suite in JUnit 4:

- Create a class. The class definition can be empty, unless you want to define a `@BeforeClass` and/or `@AfterClass` method, which would be invoked before/after the entire test suite.
- Annotate the class with `@RunWith(Suite.class)`.
- Also annotate the class with `@Suite.SuiteClasses()`, providing as an argument an array of test classes to include in the suite.
- For example:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    CheckingAccountTest.class,
    SavingsAccountTest.class
})
public class AccountTests {}
```

To define a test suite in JUnit 3:

- Create a class with a `public static` method named `suite()` that accepts no arguments and has a `Test` return type.
- The `suite()` method must instantiate and return a `TestSuite` object.
- On the `TestSuite` object, invoke `addTestSuite(Class)` to add individual test classes to the suite.
- For example:

```
public class AccountTests {
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(CheckingAccountTest.class);
        suite.addTestSuite(SavingsAccountTest.class);
        // Etc...
        return suite;
    }
}
```

2.22 Parameterized Test

JUnit 4 introduced support for a *parameterized test*, which is a test class with a test method run many times with different sets of parameters.

Note
Junit 3 does not provide support for parameterized testing.

To create a parameterized test class:

- Declare a test class with the annotation `@RunWith(Parameterized.class)`.
- Define a set of instance fields, one for each test parameter.
- Define a static method providing the parameter values:
  - The method can have any name.
  - The method must have the `@Parameters` annotation.
  - The method must be `public static`, take no arguments, and return a `java.util.Collection` object.
  - Each element of the `Collection` returned must be an array of parameter values for a specific test run.
- Define a public constructor for the class:
  - The constructor must accept the same number of arguments as the number of parameter values being provided.
- Define the test method, annotated with the `@Test` annotation.

For each element in the `Collection` returned by the `@Parameters` method, JUnit will instantiate the test class by invoking the constructor with the array values, then invoke the `@Test` method.

The following is a simple example of a parameterized test class:

```
package com.marakana.demo;

import static org.junit.Assert.*;

import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class ParameterizedTest {

    private int expected;
    private int valueOne;
    private int valueTwo;

    @Parameters
```

```
    public static Collection<Integer[]> getTestParameters() {
        return Arrays.asList( new Integer[][] {
                {2, 1, 1},  // expected, valueOne, valueTwo
                {7, 3, 4},  // expected, valueOne, valueTwo
                {0, -4, 4}, // expected, valueOne, valueTwo
        });
    }

    public ParameterizedTest(int expected, int valueOne, int valueTwo) {
        this.expected = expected;
        this.valueOne = valueOne;
        this.valueTwo = valueTwo;
    }

    @Test
    public void sum() {
        assertEquals(expected, (valueOne + valueTwo));
    }

}
```

2.23 Timeout Testing

JUnit 4 introduced the ability to mark a test as failed if it takes too long to execute.

- The `@Test` annotation accepts a `timeout` parameter, whose value is the maximum number of milliseconds allowed to execute the test.
- For example

```
@Test(timeout=5000)     // Fail if test takes longer than 5 seconds
public void testDatabaseSetup() {
    // Various test operations
    // Potentially other test assertions
}
```

Note
Junit 3 does not provide support for timeout testing.

2.24 Skipping Tests

Occasionally, you might want to skip execution of a test, for example:

- The test hasn't been developed, or bugs have been identified in its coding.
- The feature to test has not been implemented yet.
- The test is time consuming, and you want to run it only occasionally.

In JUnit 4, you can skip a test by annotating it with `@Ignore`.

- The `@Ignore` annotation accepts an optional `value` parameter, which is the reason for disabling the test.

> **Tip**
> Always provide a reason for a disabled test.

- For example:

```
@Ignore("Product feature deferred until version 2.0")
@Test
public void testSomeFeature() {
    // ...
}
```

JUnit 3 does not provide an explicit mechanism for skipping execution of a test.

- To disable a JUnit 3 test, change the method name so that it does not start with `test`. (JUnit 3 executes as test methods only those methods whose name begins with `test`).

> **Tip**
> Always add a comment explicitly stating that the test is disabled and why.

- For example

```
// Product feature deferred until version 2.0
public void disabledTestSomeFeature() {
    // ...
}
```

You also can selectively run sets of test cases by defining multiple test suites containing different combinations of test classes.

Chapter 3

Android Testing Environment

3.1 Android Test Framework

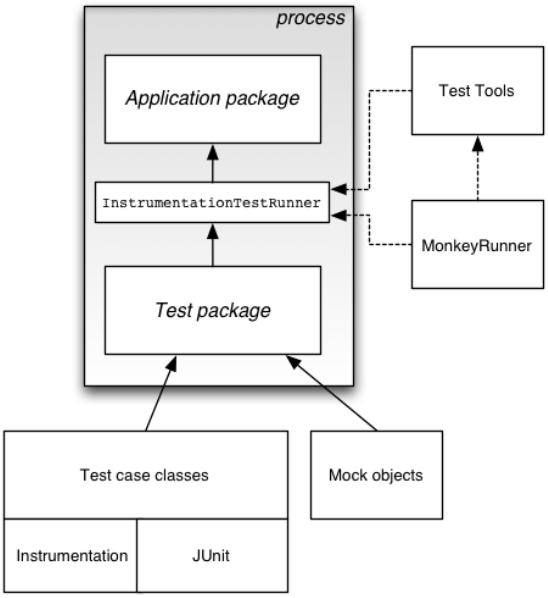

Figure 3.1: New Android Test Project Wizard

Android Test Projects Android test projects are test suites based on JUnit

- Android provides a set of JUnit extensions designed to test Android components such as Activities, Services, and Content Providers
- You can use plain JUnit to test an auxiliary class that doesn't call the Android API

Note
The Android testing API supports only JUnit 3 capabilities, not JUnit 4. For example, test methods must be named starting with `test` rather than annotated with `@Test`.

Other Android Test Tools Other tools provided by the SDK that you can use for for testing include:

- UI/Application Exerciser Monkey, a command-line tool for stress-testing UIs by sending pseudo-random events to a device
- The monkeyrunner tool, a Python API for writing programs that control an Android device or emulator from outside of Android code
- The AVD emulator console

3.2 Android Test Project

Android tests, like Android applications, are organized into projects.

- An Android test project is a directory or Eclipse project in which you create the source code, manifest file, and other files for a test package.
- An Android test project creates a test application (an `.apk` file) that you must install on the test device/emulator along with the target application to test.
- The Android test application contains special code enabling it to *instrument* the target application on the test device/emulator, so that it can be run under the control of the test application.

You should always use Android tools to create a test project. Among other benefits, the tools:

- Automatically set up your test package to use `InstrumentationTestRunner` as the test case runner. You must use `InstrumentationTestRunner` (or a subclass) to run Android JUnit tests.
- Create an appropriate name for the test package. If the application under test has a package name of `com.mydomain.myapp`, then the Android tools set the test package name to `com.mydomain.myapp.test`. This helps you identify their relationship, while preventing conflicts within the system.
- Automatically create the proper build files, manifest file, and directory structure for the test project. This helps you to build the test package without having to modify build files and sets up the linkage between your test package and the application under test.

3.3 Android Test Case Classes

All Android test classes are in the `android.test` package. Some auxiliary classes are in sub-packages.

- `AndroidTestCase` is the base Android test class, derived from `junit.framework.TestCase`.
- In general, avoid using `AndroidTestCase` directly, and instead use one of its subclasses designed to test specific application components:

  Activity testing
  `ActivityInstrumentationTestCase2`
  `ActivityUnitTestCase`
  `SingleLaunchActivityTestCase`

  Service testing
  `ServiceTestCase`

  Content Provider testing
  `ProviderTestCase2`

  Application class testing
  `ApplicationTestCase`

3.4 Additional General-Purpose Asserts

The Android SDK includes the `MoreAsserts` class, which defines a variety of `assert` methods beyond those provided by JUnit 3, including:

- `assertEquals(Object[] expected, Object[] actual)`
- `assertEquals(int[] expected, int[] actual)`
- `assertEquals(double[] expected, double[] actual)`
- `assertEquals(Set<? extends Object> expected, Set<? extends Object> actual)`
- `assertMatchesRegex(String expectedRegex, String actual)`
- `assertNotMatchesRegex(String expectedRegex, String actual)`

Android also provides overloaded versions of these methods that take an additional `String message` as their first parameter to provide more meaningful reporting in case of failures.

3.5 Additional View-Related Asserts

To support Activity testing, the `ViewAsserts` class provides several `asserts` for testing Views, including:

- `assertGroupContains(ViewGroup parent, View child)`
- `assertGroupNotContains(ViewGroup parent, View child)`
- `assertHasScreenCoordinates(View origin, View view, int x, int y)`
- `assertOnScreen(View origin, View view)`
- `assertOffScreenAbove(View origin, View view)`
- `assertOffScreenBelow(View origin, View view)`

Android also provides overloaded versions of these methods that take an additional `String message` as their first parameter to provide more meaningful reporting in case of failures.

An example of using `assertOnScreen()`:

```
ViewAsserts.assertOnScreen(view1.getRootView(), view2);
```

3.6 Mock Object Classes

You can completely or partially isolate the application component under test through the use of *mock objects*.

- The Android SDK provides mock object classes for many system resource classes.
- By default, most or all of the methods implemented by these classes simply throw an `UnsupportedOperationException` if called.
- You can create a subclass of a mock object class and override just those methods required by the test target. For example, these methods could return hard-coded values, simple calculated values, or access information from alternative databases and/or locations on the file system.

The mock object classes are contained in the `android.test.mock` package. The classes provided are:

- `MockApplication`
- `MockContentProvider`
- `MockContentResolver`
- `MockContext`
- `MockCursor`
- `MockDialogInterface`
- `MockPackageManager`
- `MockResources`

Android provides two "pre-built" mock Context classes in the `android.test` package that are useful for testing:

- `IsolatedContext` provides an isolated Context. File, directory, and database operations that use this Context take place in a test area. Though its functionality is limited, this Context has enough stub code to respond to system calls.

  This class allows you to test an application's data operations without affecting real data that may be present on the device.
- `RenamingDelegatingContext` provides a Context in which most functions are handled by an existing Context, but file and database operations are handled by a `IsolatedContext`. The isolated part uses a test directory and creates special file and directory names. You can control the naming yourself, or let the constructor determine it automatically.

  This object provides a quick way to set up an isolated area for data operations, while keeping normal functionality for all other Context operations.

3.7 Creating an Android Test Project in Eclipse

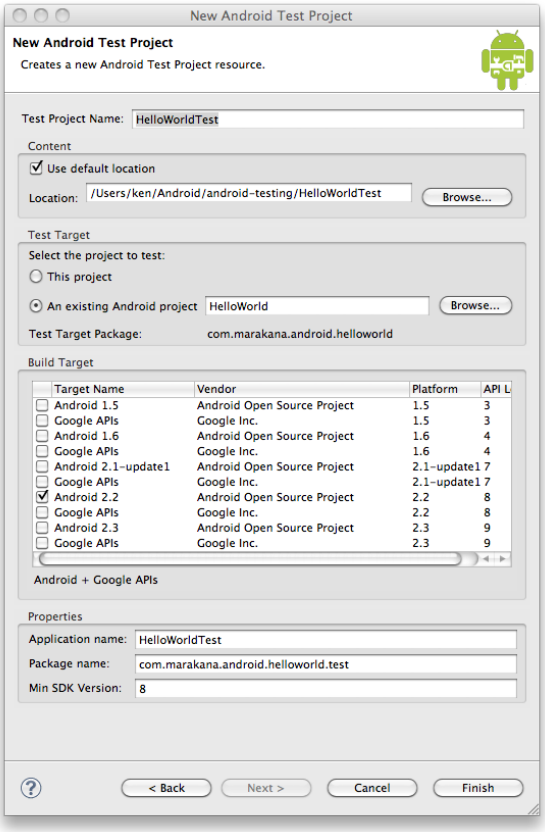

Figure 3.2: New Android Test Project Wizard

To create a test project in Eclipse:

1. In Eclipse, select File ⇒ New ⇒ Other to open the Select a Wizard dialog.
2. In the dialog, expand the entry for Android, then select Android Test Project and click `Next` to display the New Android Test Project wizard.
3. Next to `Test Project Name`, enter a name for the project. A typical convention is to append "Test" to the project name for the application under test.
4. In the Content panel, examine the suggested `Location` to the project and change if desired.
5. In the Test Target panel, set `An Existing Android Project`, click `Browse`, then select your Android application from the list. You now see that the wizard has completed the `Test Target Package`, `Application Name`, and `Package Name` fields for you.
6. In the Build Target panel, select the Android SDK platform that the application under test uses.
7. Click `Finish` to complete the wizard.

3.8 The Test Project Manifest File

As the Android test project creates an application that is installed on the test device/emulator, it must have a manifest file.

- The `<manifest>` element's `package` attribute should be a unique Android package name. By convention, it is the package name of the application under test with "`.test`" appended. This is the default provided by the Android tools when you create a test project.
- The test application has no Activities, Services, Content Providers, or Broadcast Receivers, so there should be no corresponding elements present in the manifest.
- The `<application>` element should contain the following element:

  ```
  <uses-library android:name="android.test.runner" />
  ```

- The `<manifest>` element must also have an `<instrumentation>` child element, with the following attributes set:

`android:targetPackage`	The Android package name of the application to test
`android:name`	The JUnit test runner class, typically `android.test.InstrumentationTestRunner`

Here is an example of a complete manifest file for an Android test project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.marakana.android.currencyconverter.test"
    android:versionCode="1" android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <uses-library android:name="android.test.runner" />
    </application>
    <uses-sdk android:minSdkVersion="4" />
    <instrumentation android:targetPackage="com.marakana.android.currencyconverter"
        android:name="android.test.InstrumentationTestRunner" />
</manifest>
```

3.9 Activity Testing: Activity Test Classes

In practice, most Activity test cases are derived from:

`ActivityInstrumentationTestCase2`
: Designed to do functional testing of one or more Activities in an application, using a normal system infrastructure

`ActivityUnitTestCase`
: Designed to test a single Activity in isolation. Before you start the Activity, you can inject a mock Context or Application, or both.

`SingleLaunchActivityTestCase`
: Designed for testing an activity that runs in a launch mode other than `standard`. It invokes `setUp()` and `tearDown()` only once, instead of once per test method call. It does not allow you to inject any mock objects.

ACTIVITY TEST CLASS HIERARCHY

- `ActivityInstrumentationTestCase2` and `ActivityUnitTestCase` are subclasses of `ActivityTestCase`.
- `ActivityTestCase` and `SingleLaunchActivityTestCase` are subclasses of `InstrumentationTestCase`, which in turn is a subclass of `junit.framework.TestCase`.

For more information on Activity launch modes, see the Android Manifest File reference documentation for the `<activity>` element's `android:launchMode` attribute.

3.10 Overriding the Base Activity Test Class

When creating an Activity test case derived from any of the Activity Test Classes, you must provide a no-argument constructor.

- The constructor must explicitly invoke the base class's constructor, providing the Activity class under test as an argument.
- For example:

```
public class HelloWorldActivityTest
        extends ActivityInstrumentationTestCase2<HelloWorldActivity> {

    public HelloWorldActivityTest() {
        super(HelloWorldActivity.class);
    }
}
```

3.11 Managing Activity Lifecycle and the `Instrument` Class

Android instrumentation is a set of control methods or "hooks" in the Android system.

- These hooks control an Android component independent of its normal lifecycle.
- The Activity test classes use instrumentation to manage an Activity under test, allowing you to invoke the callback methods of an Activity in your test code.

The `getActivity()` method returns the Activity under test, starting it if necessary.

- For each test method invocation, the Activity is not created until the first time this method is called.
- If you want to provide custom setup values to your Activity, you may do so before your first call to `getActivity()`.

The `InstrumentationTestCase` class provides a `getInstrumentation()` method, which returns an `Instrumentation` object, which provides other methods from controlling an Activity, such as invoking its lifecycle methods. For example:

- `callActivityOnPause(Activity activity)`
- `callActivityOnStop(Activity activity)`
- `callActivityOnRestart(Activity activity)`
- `callActivityOnStart(Activity activity)`

3.12 Turning Off Touch Mode

To control the emulator or a device with key events you send from your tests, you must turn off touch mode.

- If you do not do this, the key events are ignored.

To turn off touch mode:

- Invoke `ActivityInstrumentationTestCase2.setActivityInitialTouchMode(false)` before you call `getActivity()` to start the activity.

Note
You must invoke the method in a test method that is not running on the UI thread. For this reason, you can't invoke the touch mode method from a test method that is annotated with `@UIThread`. Instead, invoke the touch mode method from `setUp()`.

- For example:

```
public void setUp() {
    super.setUp();
    setActivityInitialTouchMode(false);
    this.activity = getActivity();
}
```

Thread management in test cases is discussed later in this module.

3.13 Generating Touch Events

The `TouchUtils` class provides many static methods for simulating touch events in the Activity. These include:

- `clickView(InstrumentationTestCase test, View v)`
- `drag(InstrumentationTestCase test, float fromX, float toX, float fromY, float toY, int stepCount)`
- `dragQuarterScreenDown(InstrumentationTestCase test, Activity activity)`
- `dragQuarterScreenUp(InstrumentationTestCase test, Activity activity)`
- `longClickView(InstrumentationTestCase test, View v)`
- `scrollToBottom(InstrumentationTestCase test, Activity activity, ViewGroup v)`
- `scrollToTop(InstrumentationTestCase test, Activity activity, ViewGroup v)`
- `tapView(InstrumentationTestCase test, View v)`

3.14 Sending Key Events

The `InstrumentTestCase` class provides methods for sending key events to the target Activity:

`public void sendKeys (String keysSequence)`
Sends a series of key events through instrumentation and waits for idle. The sequence of keys is a string containing the key names as specified in `KeyEvent`, without the `KEYCODE_` prefix. For instance:

```
sendKeys("DPAD_LEFT A B C DPAD_CENTER");
```

Each key can be repeated by using the `N*` prefix. For instance, to send two `KEYCODE_DPAD_LEFT`, use the following:

```
sendKeys("2*DPAD_LEFT");
```

`public void sendKeys (int... keys)`
Sends a series of key events through instrumentation and waits for idle. For instance:

```
sendKeys(KEYCODE_DPAD_LEFT, KEYCODE_DPAD_CENTER);
```

`public void sendRepeatedKeys (int... keys)`
Sends a series of key events through instrumentation and waits for idle. Each key code must be preceded by the number of times the key code must be sent. For instance:

```
sendRepeatedKeys(1, KEYCODE_DPAD_CENTER, 2, KEYCODE_DPAD_LEFT);
```

3.15 Testing on the UI Thread

An application's Activities run on the application's *UI thread* (also known as the *looper thread*).

The test application runs in a *separate* thread in the *same process* as the application under test.

- Generally, your test application can *read* properties and values from objects in the UI thread (for example, `TextView.getText()`).
- If your test application attempt to *change* properties on objects or otherwise send events to the UI thread from the test thread (for example, `TextView.setTest()`), you'll encounter a `WrongThreadException`.

The `TouchUtils` methods and the `InstrumentTestCase.sendKeys` methods have been written to send these events to the UI thread safely.

To otherwise manipulate the UI, you must explicitly execute that code in the UI thread. There are two ways to do so:

- To run an entire test method on the UI thread, annotate the thread with `@UIThreadTest`. Methods that do not interact with the UI are not allowed; for example, you can't invoke `Instrumentation.waitForIdleSync()`.
- To run a subset of a test method on the UI thread, create an anonymous class of type `Runnable`, put the statements you want in the `run()` method, and instantiate a new instance of the class as a parameter to the method *`activity`*`.runOnUiThread()`, where *`activity`* is the instance of the Activity you are testing. Execute `Instrumentation.waitForIdleSync()` afterwards, to wait until the Runnable has completed executed before continuing with your test method. For example:

```
activity.runOnUiThread(new Runnable() {
    public void run() {
        spinner.requestFocus();
    }
});

getInstrumentation().waitForIdleSync();

sendKeys(KeyEvent.KEYCODE_DPAD_CENTER);
```

3.16 Running Tests from Eclipse

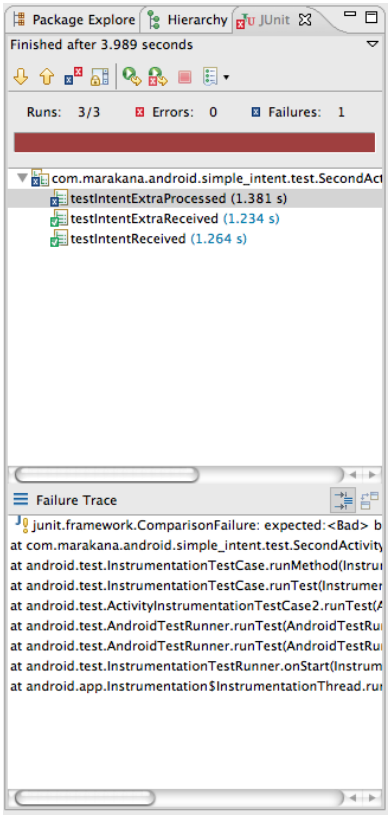

Figure 3.3: Android Test Results in Eclipse

In Eclipse with ADT, you run a test application as an "Android JUnit test" rather than a regular Android application.

To run the test application as an Android JUnit test, in the Package Explorer right-click the test project and select Run As ⇒ Android JUnit Test.

The ADT plugin then launches the test application and the application under test on a the target emulator or device. When both applications are running, the testing framework runs the tests and reports the results in the JUnit view of Eclipse, which appears by default as a tab next to the Package Explorer.

3.17 Running Tests from the Command Line

- To run a test from the command line, you run `adb shell` to start a command-line shell on your device or emulator, and then in the shell run the `am instrument` command.
- As a shortcut, you can start an `adb shell`, call `am instrument`, and specify command-line flags all on one input line. The shell opens on the device or emulator, runs your tests, produces output, and then returns to the command line on your computer.
- To run a test with `am instrument`:

  1. If necessary, rebuild your main application and test package.
  2. Install your test package and main application Android package files (`.apk` files) to your current Android device or emulator
  3. At the command line, enter:

     ```
     adb shell am instrument -w test_package_name/runner_class
     ```

     test_package_name
     The Android package name of your test application

     runner_class
     The name of the Android test runner class you are using, usually `android.test.InstrumentationTestRunner`

The `am instrument` tool passes testing options to `InstrumentationTestRunner` or a subclass in the form of key-value pairs, using the `-e` flag, with this syntax:

```
-e <key> <value>
```

Some keys accept multiple values, specified as a comma-separated list. Common keys include:

Key	Value	Description
`package`	*Java_package_name*	The fully-qualified Java package name for one of the packages in the test application. Any test case class that uses this package name is executed. Notice that this is not an Android package name; a test package has a single Android package name but may have several Java packages within it.
`class`	*class_name*	The fully-qualified Java class name for one of the test case classes. Only this test case class is executed.
	class_name`#`*method name*	A fully-qualified test case class name, and one of its methods. Only this method is executed. Note the hash mark (`#`) between the class name and the method name.
`size`	`small`|`medium`|`large`	Runs a test method annotated by size. The annotations are `@SmallTest`, `@MediumTest`, and `@LargeTest`.

3.18 Service Testing

`ServiceTestCase` extends the JUnit `TestCase` class with with methods for testing application permissions and for controlling the application and Service under test.

- It also provides mock application and Context objects that isolate your test from the rest of the system.

`ServiceTestCase` defers initialization of the test environment until you call `ServiceTestCase.startService()` or `ServiceTestCase.bindService()`.

- This allows you to set up your test environment, particularly your mock objects, before the Service is started.

The `setUp()` method for `ServiceTestCase` is called before each test.

- It sets up the test fixture by making a copy of the current system Context before any test methods touch it.
- You can retrieve this Context by calling `getSystemContext()`.
- If you override this method, you must call `super.setUp()` as the first statement in the override.

The methods `setApplication()` and `setContext(Context)` allow you to set a mock Context or mock Application (or both) for the Service, before you start it.

3.19 Content Provider Testing

You test a Content Provider with a subclass of `ProviderTestCase2`.

- This base class extends `AndroidTestCase`.

The `ProviderTestCase2` constructor performs several important initialization steps for creating an isolated test environment.

- All subclasses must invoke the super constructor.

The `ProviderTestCase2` constructor creates an `IsolatedContext` object that allows file and database operations but stubs out other interactions with the Android system.

- The file and database operations themselves take place in a directory that is local to the device or emulator and has a special prefix.

The constructor then creates a `MockContentResolver` to use as the resolver for the test.

Lastly, the constructor creates an instance of the provider under test.

- This is a normal `ContentProvider` object, but it takes all of its environment information from the `IsolatedContext`, so it is restricted to working in the isolated test environment.
- All of the tests done in the test case class run against this isolated object.

3.20 Application Class Testing

You use the `ApplicationTestCase` test case class to test the setup and teardown of Application objects.

- Application objects maintain the global state of information that applies to all the components in an application package.

The test case can be useful in verifying that the `<application>` element in the manifest file is correctly set up.

Note
This test case does not allow you to control testing of the components within your application package.

Chapter 4

Resource List

4.1 Books

[1] Beck, Kent. JUnit Pocket Guide Sebastopol, Calif.: O'Reilly, 2004.

4.2 Web Sites

[2] Android Developers. http://d.android.com

[3] JUnit. http://www.junit.org