SEGR 5210 Software Testing
Fall 2014

Unit Testing on Android

Prerequisites

Before completing the tutorial, you need to do the following:

1. Download and install the Eclipse ADT (http://developer.android.com/sdk/index.html). You may need to install JDK and Apache Ant (check System Requirements on the Eclipse ADT website).
2. It is recommended to go through "Building Your First App" tutorial (http://developer.android.com/training/basics/firstapp/index.html) if you have no experience with Android app development. In particular, it is helpful to understand the activity life cycle and the Android device emulator as this document does not cover these in detail.
    - When creating the emulator, you need to select a device. I used a Nexus 7.
3. Download `NumberGuessing.zip` from the course website and extract the contents.
4. Import the NumberGuessing project into Eclipse: File → Import → Android → Existing Android Code into Workspace
5. Run the app in the emulator (refer to the "Building Your First App" tutorial on setting up the emulator).

Troubleshooting

- If you have a problem building the app because it cannot find the libraries:
    1. Select the menu option: Project→Properties
    2. In the dialog box, select Android. Then check the box next to Android.
    3. Press Apply and the press OK.
- If you get a build error because it cannot find the resource "crunch": Clean the project (Project → Clean…) and make sure the emulator is running before running the app.

About the Sample Program

This tutorial uses a simple number guessing game app to illustrate the different types of tests you can use for Android apps. The game asks the player to guess a randomly generated four digit number from 0000 to 9999. If the player is incorrect, the game returns the sum of the differences for each digit. For example, if the mystery number is "3871" and the player guessed "8675", the result would be 11 = 5 + 2 + 0 + 4. Once the player is correct, a winning message appears. The player may reset the game by pressing the Restart button.

The program consists of three classes (in the `src` directory):

- `MainActivity.java`: The main GUI for the program.
- `WinningMsgActivity.java`: The screen that appears when the player wins the game.
- `GuessingGame.java`: A class that handles the playing of the game (game logic).

The first two classes are Android activities. An activity within Android typically refers to a "screen". The third class (`GuessingGame.java`) is a normal Java class with no Android specific functionality.

Testing is done using JUnit – a unit testing framework for Java that is integrated with Eclipse. In order to test Android-specific functionality, the Android SDK extends the JUnit framework with additional classes and methods that allow tests to interact with the GUI elements and Android functions.

Test Project

The first step is to create a test project:

1. In Package Explorer, right-click on the NumberGuessing project and select New →Project...
2. In the wizard:
    a. Select Android → Android Test Project. Then press Next.
    b. Type in the name of the project – a good convention is to use the name of the project followed by Test. In this case, use NumberGuessingTest. Then press Next.
    c. Now choose a test target. Select "An existing Android project:" → NumberGuessing. Then press Next.
    d. Choose an SDK to target. On my machine, I only had one choice (Android 4.4.2). If you have one choice, select that. If you have two choices, pick the newest Android version. Don't worry if your Android version is not 4.4.2 unless it is really old. Then press Finish.
3. The test project will now appear in Package Explorer.

Creating JUnit Tests

We will start by creating JUnit tests for `GuessingGame.java`. This will get you familiar with the Java unit tests before introducing Android-specific functionality.

To start, you need to create a class for testing. Each class that you test in the project will have a corresponding class in the test project. Here is the procedure for creating the new class:

1. In Package Explorer, right-click on the NumberGuessingTest project and select New → Junit Test Case.
2. In the wizard's first dialog box:
    a. Select New JUnit 3 Test (Android uses JUnit 3 for their test cases so we'll use JUnit 3 for everything)
    b. Package: `edu.seattleu.numberguessing.test`
    c. Name: `GuessingGameTest`
    d. Superclass: `junit.framework.TestCase`
    e. Method stubs to create: Select setUp(), tearDown(), and constructor.
    f. Class under test: `edu.seattleu.numberguessing.GuessingGame`
    g. Click Next when done.
3. In the wizard's second dialog box:
    a. For available methods – click on the top level `GuessingGame` entry. This will cause all of the methods to be clicked.
    b. Leave the Object and its methods blank.
    c. Then click Finish.
4. This will create and display the file `GuessingGameTest.java`. You will see the constructor, `setUp`, tearDown, and skeleton functions for each of the methods.

JUnit works by running each function that starts with the prefix "`test`". Before each function is called, JUnit will first call `setUp`. This is used to do setup and initialization that are common to all tests. Similarly, JUnit will call t`earDown` after executing each test. This allows the app to free up resources.

Write `setUp` and `tearDown` using these functions:

```
protected void setUp() throws Exception {
	super.setUp();
	game = new GuessingGame(4);
}

protected void tearDown() throws Exception {
	super.tearDown();
	game = null;
}
```

You will also need to add the private data member game:

```
private GuessingGame game;
```

If this statement has an error (red squiggly line), hover over the error and select to Import 'Guessing Game'.

The `setUp` function initializes the game to create a 4 digit number. The `tearDown` function sets game to null allowing the memory associated with the game to be freed up. There is some debate on whether this is necessary or not. One source claimed it was necessary for JUnit 3 but not JUnit 4. For the purposes of this course, it suffices to use the default `tearDown` function unless there is an issue (which is unlikely).

Now, we will implement the test functions as follows. Note these tests are shown to introduce you to using the unit testing tools. They should not be construed as a "complete" test suite.

```
public void testGuessingGame() {
	assertEquals(4, game.getNumDigits());
	assertEquals(0, game.getNumGuesses());
}

public void testSetMysteryNumber() {
	game.setMysteryNumber("5913");
	assertEquals("5913", game.getMysteryNumber());
}

public void testGetMysteryNumber() {
	game.setMysteryNumber("2987");
	assertEquals("2987", game.getMysteryNumber());
}

public void testGetNumDigits() {
	assertEquals(4, game.getNumDigits());
}
```

```
public void testGetNumGuesses() {
      game.setMysteryNumber("5913");
      game.makeGuess("1234");
      game.makeGuess("5678");
      int numGuesses = game.getNumGuesses();
      assertEquals(2, numGuesses);
}

public void testReset() {
      game.reset();
      int numGuesses = game.getNumGuesses();
      assertEquals(0, numGuesses);
}

public void testMakeGuess() {
      game.setMysteryNumber("5913");
      int result = game.makeGuess("5678");
      assertEquals(14, result);
}
```

First, look at the last of the functions `testMakeGuess`. This is a typical structure for a test:

1. Perform additional initialization (beyond `setUp`). In this case, we are setting the mystery number to 5913.
2. Call the function under test: `game.makeGuess`
3. Check the result. The check is done using an assertion called `assertEquals`. `assertEquals` will cause the test to fail if the two arguments are not equal. The statement has no effect if the two arguments are equal.

A test is considered to pass unless an assertion triggers a failure. There are other types of assertions beyond `assertEquals`. Other common assertions include `assertTrue`, `assertFalse`, `assertNull`, and `assertNotNull`.

Here's a brief of description of some of the other tests:

- The test `testGetNumGuesses` is formatted similarly to `testMakeGuess` except that the setup is more complicated: setting the mystery number and making two guesses.
- The test `testReset` is difficult in that the `reset` method does not return a value. For these types of methods, it is necessary to use other functions to determine if the method under test is working correctly. In this situation, we use `getNumGuesses` and check to see if it has been reset to zero.
- The test for the constructor often only includes assertions and is basically testing that the initial configuration after `setUp` is correct.

Running the Tests

To run the tests:

1. Select Run → Run As... → JUnit Test.
2. You may get a window to select preferred launcher – select Android JUnit Test Launcher. This window only appears the first time you run the tests.
3. You see a JUnit tab by Package Explorer that shows the results of the tests. All tests should have passed: Runs: 7 / 7

To see a failure, change the assertion in `testMakeGuess` such that the expected value is 12 instead of 14:

```
assertEquals(12, result);
```

Rerun the tests – this is easily accomplished using the Rerun Test button on the JUnit panel. Now you will see that `testMakeGuess` fails. Clicking on the test name in JUnit panel will show a failure trace. The first line of the trace is typically the most interesting:

```
JUnit.framework.AssertionFailedError: expected:<12> but was:<14>
```

Before proceeding, go back and correct the assertion from 12 back to 14 and rerun the tests. They should all pass again.

Additional notes:

- The order in which the tests are run is completely up to JUnit. The programmer has no control. In particular, the tests are not run in the order they appear in the file (except by chance). This is a common criticism of JUnit.
- Some programmers will use declare and use constants for hard-coded input (such as the strings "5913" and "5678" in `testMakeGuess`) and expected output (such as the 12 in `testMakeGuess`). For the purposes of this project, it is acceptable to use hard-coded values like presented here. One reason is why this is acceptable is that the test cases need to appear in tabular form in the project report.
- It is acceptable and encourage to create additional functions in the test project – simply give them a name that does not start with the word "`test`". Such a function can be useful for functions where you have several values to test. Here is an example for `testMakeGuess`:

```
private void checkGuess(String mysteryNum, String guess, int
  expectedResult) {
      game.setMysteryNumber(mysteryNum);
      int result = game.makeGuess(guess);
      assertEquals(expectedResult, result);
}

public void testMakeGuess() {
      checkGuess("5913", "5678", 14);
      checkGuess("5913", "5913", 0);
      ...
}
```

Creating Android Unit Tests

Android unit tests are an extension of JUnit tests. Here are some similarities:

- Tests are stored in a test class. Any function that starts with the prefix "`test`" will be considered a test.
- Tests will execute `setUp` function before execution and `tearDown` after execution.
- The process of running tests and analyzing the results are the same.

Here are some key differences:

- The unit test class extends `android.test.ActivityInstrumentationTestCase2` instead of `JUnit.framework.TestCase`
- There are functions provided to interact with the GUI for both input and output.
- Execution of tests is done through the emulator or the device itself. This tutorial will only cover use of the emulator.

Let's get started by creating a test class:

1. In Package Explorer, right-click on the NumberGuessingTest project and select New → Class.
2. In the dialog box:
    a. Package: `edu.seattleu.numberguessing.test`
    b. Name: `MainActivityTest`
    c. Superclass: `android.test.ActivityInstrumentationTestCase2<MainActivity>` (Note: The ActivityInstrumentationTestCase2 class is a template class and requires to specify the name of the target class.)
    d. Click Finish.
3. This will create a blank class with two errors.
    a. No import of the subject under test (`edu.seattleu.numberguessing.MainActivity` in this case). This error can be fixed by hovering over the red squiggly line and picking the first option in the quick fix menu.
    b. No constructor. Add the generated constructor with this:

```
public MainActivityTest() {
   super(MainActivity.class);
}
```

Set Up

The next step is to create a `setUp` function:

```
@Override
protected void setUp() throws Exception {
  super.setUp();
  setActivityInitialTouchMode(false);
  mActivity = getActivity();

  mActivity.setMysteryNumber("5913");

  mSubmitButton = (Button)
    mActivity.findViewById(edu.seattleu.numberguessing.R.id.submitButton);
  mRestartButton = (Button)
    mActivity.findViewById(edu.seattleu.numberguessing.R.id.restartButton);
  mGuessEntry = (EditText)
    mActivity.findViewById(edu.seattleu.numberguessing.R.id.guessEntry);
  mResult = (TextView)
    mActivity.findViewById(edu.seattleu.numberguessing.R.id.result);
  mPrevGuess = (TextView)
    mActivity.findViewById(edu.seattleu.numberguessing.R.id.prevGuess);
  mGuessCount = (TextView)
    mActivity.findViewById(edu.seattleu.numberguessing.R.id.guessCount);
}
```

The first three lines should be the same for the tests you create in this course:

- `super.setUp()` calls the parent constructor.
- `setActivityInitialTouchMode(false)` disables input from the user during the test.
- `mActivity = getActivity()` stores the activity for current use.

The remainder of the function has two purposes: (1) initialize state common to all tests and (2) initialize data members for activity components so they can easily be accessed by the tests.

In this example, the only initialization is set the mystery number to "5913". It is not necessary to test with different mystery numbers since we are only testing the GUI. The game logic should be thoroughly tested with different mystery numbers when testing `GuessingGame.java`.

The remainder of the function involves the initialization of six data members. The format for these initializing statements is:

```
<member name> = (<type>) mActivity.findViewById(<package>.R.id.<element name>);
```

The package is the name of the package as indicated by the package line (typically the first line) in the source code file of the class / activity you are testing.

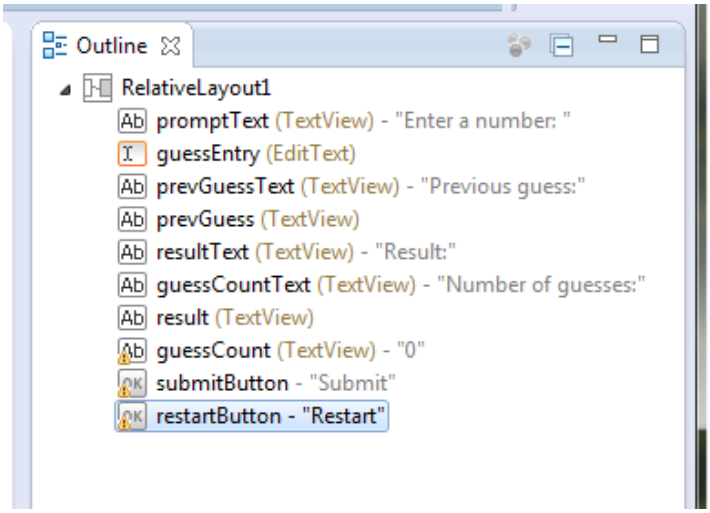
To find the type and element name, you need to first find the layout file:

1. Look at the source code of the class you are testing (`MainActivity.java` in this case) and find the `onCreate` method.
2. Within the `onCreate` method, there should be a call like this (or similar):

   ```
   setContentView(R.layout.activity_main);
   ```

   The parameter indicates the name of the layout (`activity_main` in this case).
3. Traverse to the `res/layout` directory of the app within Package Explorer. You should see some `.xml` files there.
4. One should have the same name as the layout name from step 2 (`activity_main.xml` in this case). Double click on it.

The graphical layout tool will open with several panes. The pane you are most interested is the outline pane (typically in the upper right) as shown below. This pane gives a list of the names of GUI elements. In the Linux version of Eclipse, selecting a name will highlight the corresponding GUI element in the layout pane and vice versa. This unfortunately did not work in the Windows version for some reason. Hopefully the names are clear enough to figure out which component is which.



To get the type of the element, double click on the entry in the outline pane to bring up the raw XML code. The type is given by the XML tag it is enclosed in. For example, `submitButton` has type `Button`.

```
<Button
        android:id="@+id/submitButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@+id/guessEntry"
        android:layout_toRightOf="@+id/guessCountText"
        android:onClick="processGuess"
        android:text="Submit" />
```

Additional notes:

- For some unknown reason, the Outline pane lists the type of `EditText` and `TextView` in parentheses but does not do this for other elements.
- It is not necessary to create a data member for every element in the GUI. In particular, it is not necessary to create data members for static images and text strings that never change. For instance, a data member is not created for the `TextView resultText`, the string that always displays "`Result:`".
- `TextView` elements that change during the course of execution may be blank in the graphical layout so it is necessary to find the appropriate element in the Outline pane. For instance, the `TextView` elements `result` and `prevGuess` are blank.

Once the type and name are found, completing the initialization statement is straightforward. The only thing left is the name of the data member. The convention here is to use the same name preceded by an `m` (for member). The data members must be declared within the class like this:

```
private MainActivity mActivity;
private Button mSubmitButton;
private Button mRestartButton;
private EditText mGuessEntry;
private TextView mResult;
private TextView mPrevGuess;
private TextView mGuessCount;
```

You may need to import the types `Button`, `EditText`, `TextView`. This can be done by hovering over the errors and selecting Import from the Quick Fix menu.

Writing Tests

A common first test is to test that the app is setup properly. The convention is to call this test `testPreConditions`:

```
public void testPreConditions() {
      assertEquals("", mPrevGuess.getText().toString());
      assertEquals("", mResult.getText().toString());
      assertEquals("0", mGuessCount.getText().toString());
}
```

This simply checks that the text views have the proper default values.

Our next test will test the restart button:

```
public void testRestart() {
	TouchUtils.clickView(this, mSubmitButton);
	TouchUtils.clickView(this, mRestartButton);
	assertEquals("", mPrevGuess.getText().toString());
	assertEquals("", mResult.getText().toString());
	assertEquals("0", mGuessCount.getText().toString());
}
```

This test clicks the submit button updating the `result`, `prevGuess`, and `guessCount` text views. Then the test clicks the restart button and checks whether these text views go back to their initial state. The command `TouchUtils.clickView` is the function used to click a button. The first parameter is `this` and the second parameter is the data member corresponding the button.

Here is a similar test for testing the `testGuessCount` display. It makes four guesses and checks that the guess count shows that four guesses are made.

```
public void testGuessCount() {
	for (int i = 0; i < 4; i++)
		TouchUtils.clickView(this, mSubmitButton);
	assertEquals("4", mGuessCount.getText().toString());
}
```

Next up is the submit button:

```
public void testSubmit() {
	makeGuess("1234");
	checkGuess("1234", "14", "1");
}
```

This test consists of two helper functions `makeGuess` and `checkGuess`. These functions will be used in additional tests beyond `testSubmit`. Here is the function `makeGuess`:

```
private void makeGuess(final String guess) {
	Instrumentation instr = this.getInstrumentation();

	instr.runOnMainSync(new Runnable() {
		@Override
		public void run() {
			mGuessEntry.requestFocus();
		}
	});
	instr.waitForIdleSync();
	instr.sendStringSync(guess);
	instr.waitForIdleSync();
	TouchUtils.clickView(this, mSubmitButton);
}
```

The process of sending a screen to a text box is a bit convoluted. Just follow this pattern and everything should just work. Here is a longer explanation:

- This requires using instrumentation (it is an `ActivityInstrumentationTestCase2` after all).
- The test runs on a separate thread from the main thread but only the main thread can interact with GUI elements. Using the instrumentation's `runOnMainSync` method, you can run commands that interact with the GUI on the main thread.
  - In this case, the only thing that is done is selecting the `guessEntry` and giving it focus so that it receives the string instead of some other GUI element.
- The command `waitForIdleSync` waits for the GUI to settle down and guarantees that previous updates have been made. This is a best practice for writing robust tests. As noted above, there are multiple threads and this command essentially causes the thread running the test and the main thread running the GUI to be synch up before proceeding.
- The command `sendStringSync` sends the string to the GUI.
- Why don't we have to this with clicking a button? The `TouchUtils.clickView` handles all of this behind the scenes for you. Unfortunately, a comparable function does not exist for entering text.

The `checkGuess` helper function is simply a set of three assertions:

```
private void checkGuess(String prevGuess, String result, String guessCount)
{
    assertEquals(prevGuess, mPrevGuess.getText().toString());
    assertEquals(result, mResult.getText().toString());
    assertEquals(guessCount, mGuessCount.getText().toString());
}
```

Note: It is not necessary to create helper functions like this but they are recommended as there is a lot of repeated code between different tests.

There are several ways of testing UI operations:

- `TouchUtils` class: This class provides methods for generating input event such as clicking, dragging, and scrolling. Check this class first as if `TouchUtils` supports the desired input, as it is the easiest to use.
- `runOnMainSync`: This method from the instrumentation class will allow you to manipulate the GUI. Common tasks include putting the focus on an element (as seen in the `makeGuess` function above) or manipulating the GUI element by calling one of its methods (an example is calling the `setChecked()` method for a `CheckBox`).
  - One challenge with this technique is determining what needs to be run on the main thread as the documentation is not clear. One approach, that is admittedly not very satisfying, is to assume the function does not need to be run on the GUI. While running the test, a run-time error will occur when a violation occurs. Then you can fix the error.
- `UiThreadTest`: This annotation can be applied to an entire test. The entire test case will be run on the main thread. However, many instrumentation routines cannot be used in the test. The use of this annotation is not covered in this tutorial – please consult other sources for more information.

Our last test tests what happens when the game is won. A new "you win" activity is created. Testing this requires using a instrumentation monitor:

```
private static final int TIMEOUT_IN_MS = 5000;

public void testWinGame() {
      Instrumentation instr = this.getInstrumentation();
      ActivityMonitor winActivityMonitor =
            instr.addMonitor(WinningMsgActivity.class.getName(),
            null, false);

      makeGuess("5913");
      WinningMsgActivity winActivity = (WinningMsgActivity)
            winActivityMonitor.waitForActivityWithTimeout(TIMEOUT_IN_MS);
      assertNotNull("WinningMsgActivity is null", winActivity);
      assertEquals("Monitor for WinningMsgActivity has not been called",
                  1, winActivityMonitor.getHits());
      assertEquals("Activity is of wrong type",
                  WinningMsgActivity.class, winActivity.getClass());

     instr.removeMonitor(winActivityMonitor);
}
```

Some details:

- The monitor is added using `instr.addMonitor()` using the name of the resulting activity (`WinningMsgActivity` in this case). Use the `.class.getName()` method to get the name.
- Once the test has executed a command that is supposed to cause the start of the new activity, call the `waitForActivityWithTimeout` method.
- Use the three asserts as shown in the example to determine if the activity has been properly started or not.
- Always remove the monitor at the end of the test. The instrumentation engine is not restarted so the monitor will be present in other tests if not removed. This can be problematic especially if other tests employ monitors.

Running the Tests

To run the tests: Select Run → Run As... → Android JUnit Test. This should bring up the emulator if one is not starting already. This should execute both the `GuessingGame` tests and the `MainActivity` tests (a total of 12 tests). You can watch the Android tests execute in the emulator. All tests should pass.

Unfortunately, running the tests is a bit flaky at times. Here are some things to do if you encounter problems:

- *The Run --> Run As... --> menu is empty.* Go to Package Explorer, highlight NumberGuessingTest, right-click and select Run → Run As... from there.
- *Only the GuessingGame tests run, the MainActivity tests do not run.* Go to Package Explorer, highlight `MainActivityTest.java`, right-click and select Run → Run As... from there. This will run the `MainActivity` tests but not the `GuessingGame` tests. After doing this once, then select Run from the right-click menu on the NumberGuessingTest folder in Package Explorer. Now all 12 tests should run.
- *The emulator does not start automatically.* Run the NumberGuessing program in the emulator first and then run the tests.
- *The tests fail due to an INJECT_EVENTS error.* This error occurs because the screen is locked in the emulator. Unlock the screen by swiping and rerun the tests.

- *Tests fail due to a permission problem.* This error is likely caused by trying to run a method that manipulates the GUI. The method may need to be run using the `runOnMainSync` method described earlier in this document. There might be other situations that cause this error too.
- *Other errors (or the solutions above do not work).* Can try some of the following:
  - Clean the test project (Project→Clean).
  - Close the emulator.
  - Restart Eclipse (File→Restart).

Measuring Test Code Coverage

To measure code coverage, we will using the tool EMMA. This tool is provided with the Android SDK but it is not integrated with Eclipse so some command-line setup is necessary using command prompt or Cygwin in Windows, terminal in Mac, or a shell in Linux. Note the following directories:

- `<ADT>`: Directory where Eclipse ADT is placed. This directory subdirectories `eclipse` and `sdk`.
- `<App>`: Directory where subject under test is located (NumberGuessing).
- `<Test>`: Directory of test project (NumberGuessingTest).

These commands will create build scripts (`build.xml`) for the application and the test project. In Windows command prompt, you will need to use "\" instead of "/" (it was also fairly slow on my laptop).

```
cd <App>
<ADT>/sdk/tools/android update project --path .
cd <Test>
<ADT>/sdk/tools/android update test-project --main <App> --path .
```

Go back to Eclipse and do the following:
1. Expand the test project in Package Explorer. The file `build.xml` should be present. If it is not visible, press F5 to refresh the project directory.
2. Right-click the `build.xml` file and select "Run As Ant Build...".
3. The Edit Configuration dialog box will appear. Select the Main tab and set the arguments to: `clean emma debug install Test`
4. Select Run.

The above steps only need to be done once. To run subsequent times, simply right-click the `build.xml` file and select "Run As Ant Build".

The tests will run and create a report. To view the report:
1. Press F5 to refresh the project directory.
2. Expand the bin folder in the test project.
3. Double click on `coverage.html`. (If it missing, trying right-clicking on the bin folder and selecting Refresh.)
4. You will see statistics for class, method, block, and line. Click on `edu.seattleu.numberguessing`
5. You will now see the coverage for each class. Click on `MainActivity.java`
6. The coverage for each method is displayed followed by the source code. Red lines were not covered and green lines were covered. You can jump to a method by clicking on its name in the table.

To run the unit tests without getting coverage, you will need to clean the project first: Select Project→Clean and select both the project under test (NumberGuessing) and the test project (NumberGuessingTest).