CIS 573 Software Engineering
Fall 2015
Homework #3

Introduction

This assignment has two parts, both related to software testing.

In the first part of the assignment, you will write JUnit tests for an Android app, using a "mocking" library called Robolectric to mimic some of the underlying Android behavior.

In the second part of the assignment, you will conduct white-box testing on methods in a Java Swing app. The detailed specifications will not be available, but there will be a high-level description of the app.

You must work **alone** on this assignment. As before, although you are free to discuss the intent of the assignment and ask your fellow classmates for clarification, the code and report that you write must be your own.

This assignment is worth a total of 100 points.

Part 1. Android Testing

In this part of the assignment, you'll write tests for an Android app.

Implementation

The app that you'll test is a slightly modified implementation of the Traveling Salesman game from Homework #2. You can download the app as an Android Studio project in Canvas. Please use this implementation, and not the one from Homework #2; also, please note that this is *not* a solution to Homework #2, of course.

The classes in this app are more or less the same as in Homework #2, except that the GameView class now has a separate method for detecting circuits (*detectCircuit*).

Setting Up

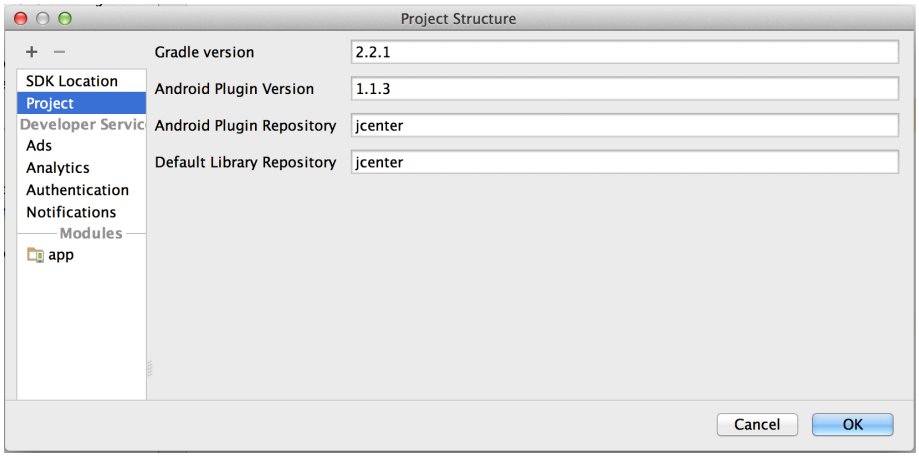
As discussed in class, there are generally two ways to write tests for Android apps:

- "Android Instrumentation Tests," which run in the context of the full Android OS and require some sort of underlying device (physical or virtual)
- "JUnit Tests," which can still test Android classes but require some library to mock the underlying OS behavior

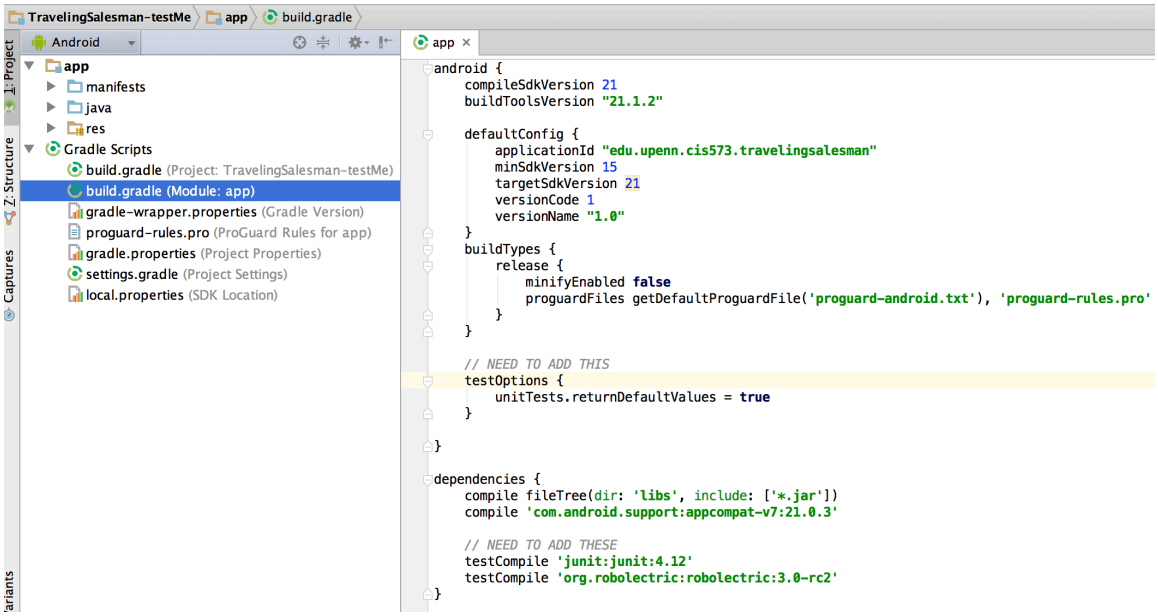
In this assignment, you will write JUnit Tests, using the Robolectric library for mocking the Android components. Be sure to review your class notes from the lecture on Android testing if you need help.

To get your JUnit Tests set up in Android Studio, follow these steps:

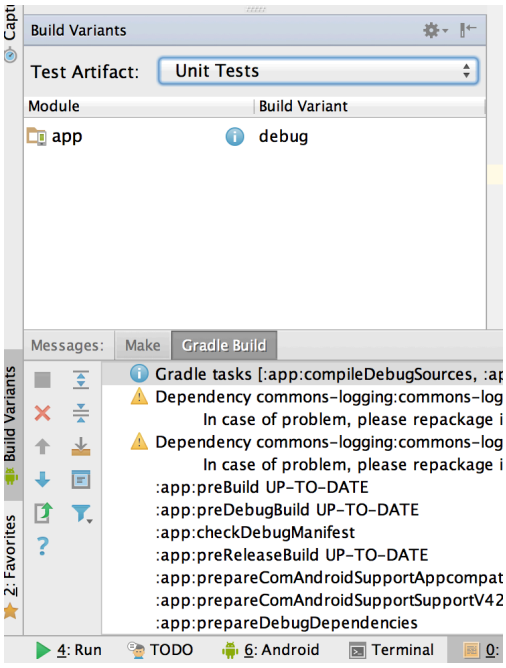
1. Build the TravelingSalesman-testMe project and make sure you can run it in a virtual device (just like in Homework #2) before proceeding.
2. In Android Studio, choose File -> Project Structure… from the main menu. Then choose Project from the menu on the left and then set **Android Plugin Version to 1.1.3** as shown below (if it's already higher, leave it as-is):



3. Modify the build.gradle file for the "app" module as shown in the screen capture on the next page:
    - Add `testOptions {unitTests.returnDefaultValues = true}` to the "android" configuration. This tells Android Studio that we are going to mock the underlying Android components.
    - Add `testCompile 'junit:junit:4.12'` to the "dependencies" configuration. This tells Android Studio that we're going to need JUnit 4.12.
    - Add `testCompile 'org.robolectric:robolectric:3.0-rc2'` to the "dependencies" configuration. This tells Android Studio that we're going to use the Robolectric library.
4. After making these changes, **choose Build -> Make Project from the main menu**. This will likely cause Android Studio to download JUnit and Robolectric, so be sure you are connected to the Internet when you do this. This may take a few minutes to finish and you should eventually see "Gradle build finished" in the bottom left message bar (if you get warnings, choose the "Messages" tab and if there's something there about commons-logging, it's okay to ignore that).

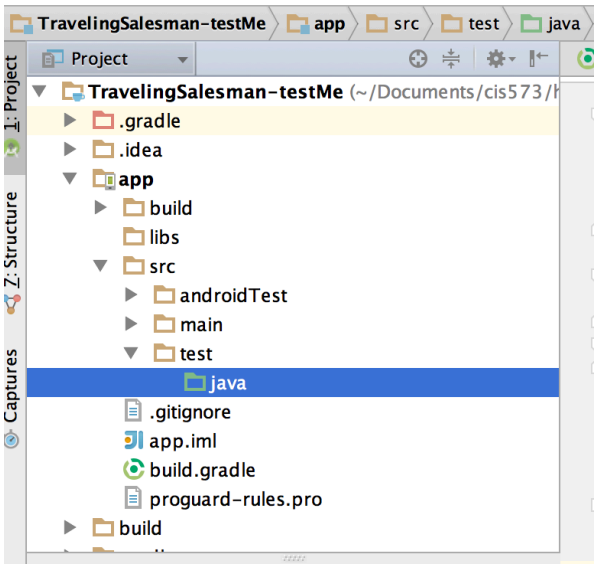


5. Next we'll tell Android Studio that we're going to use "JUnit Tests" instead of "Android Instrumentation Tests." Select the Build Variants tab (it's probably on the bottom left of the window, and the words are vertical). In the Build Variants window, **change Test Artifact from "Android Instrumentation Tests" to "Unit Tests"** as shown below:

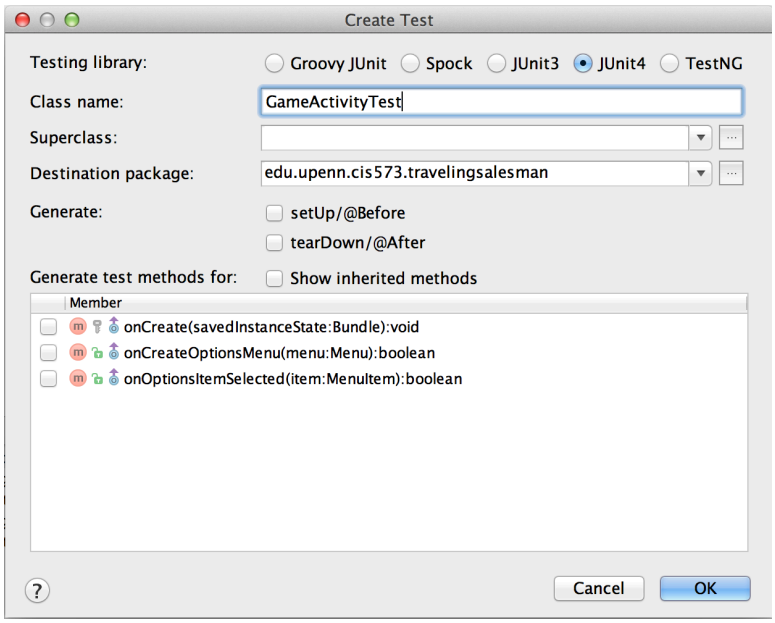


6. Choose **Build -> Make Project** from the main menu to rebuild the project.

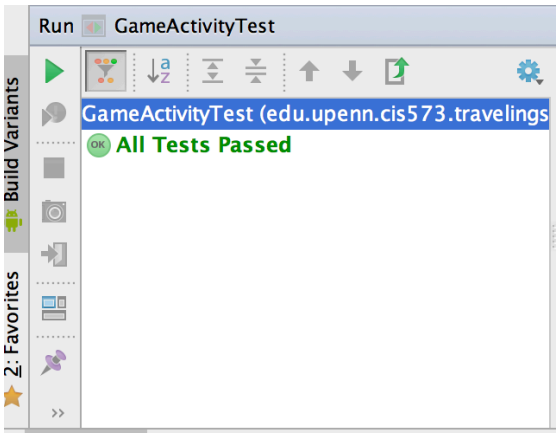
7. Next you need to create a directory to hold your JUnit tests. You should *not* use the built-in "androidTest" directory, but rather should create a new one. Open the Project view in the top left (it may currently be set to the Android view) and then **create a subdirectory called test, then test/java/ in the app/src/ directory**. You should see the java subdirectory turn green.



8. Now go back to the Android view and open the GameActivity class in the code editor. Right-click on the class name in the code (i.e., the line that reads "public class GameActivity extends ActionBarActivity"), then **select Go To from the context menu, then choose Test, then Create new test…**
9. **Select JUnit 4 as the Testing library**, name the class **GameActivityTest**, and click OK.



10. **Replace the code for GameActivityTest with the code from Canvas**. This code is configured to use the Robolectric library but otherwise is just a standard JUnit test class. It has placeholders for the test methods you need to implement.
11. Choose Build -> Make Project from the main menu to rebuild the project.
12. In the Android view, **right-click GameActivityTest.java and choose Run GameActivityTest.** You should see the unit test run in the console on the bottom left and then see the message "All Tests Passed":



If you made it this far, you're ready to start writing your own tests!

Troubleshooting:

- Error running GameActivityTest: Class edu.upenn.cis573.travelingsalesman.GameActivityTest not found in module 'app'
  - Be sure to select "Unit Tests" for the Test Artifact in the Build Variants panel, and not "Android Instrumentation Tests"
- java.lang.RuntimeException: Method setUp in android.test.ActivityUnitTestCase not mocked.
  - Make sure you add the testOptions configuration in build.gradle for the app module
- java.lang.RuntimeException: build/intermediates/bundles/debug/AndroidManifest.xml not found or not a file
  - This may happen on a Mac. To fix it, do the following:
    - Go to Run -> Edit Configurations
    - On the left, choose JUnit then GameActivityTest
    - Set Working Directory to $MODULE_DIR$
- Gradle: Error: Cannot find symbol class GameActivity
  - You should be able to make this go away by choosing Build -> Clean Project and then Build -> Make Project
- Exception in thread "main" java.lang.NoClassDefFoundError: junit/textui/ResultPrinter
  - You can ignore this… just re-run your test

Testing Activities

Implement these unit tests in GameActivityTest:

1. *testCreateSegment*: tests that a line segment is properly created and added to the collection of segments in the GameView when the user touches the screen close to one of the map locations (within 30 pixels), drags their finger close to another location, and releases
2. *testDontCreateSegmentFirstPointNotCloseToMapLocation*: tests that a line segment is *not* added to the collection of segments when the user's touch is not close (within 30 pixels) to one of the map locations, then they drag their finger close to another location and releases
3. *testDontCreateSegmentLastPointNotCloseToMapLocation*: tests that a line segment is *not* added to the collection of segments when the user's touch is close to one of the map locations, then they drag their finger and release at a point that is not close to a map location
4. *testDontCreateSegmentFirstAndLastPointsSame*: tests that a line segment is *not* added to the collection of segments when the user's touch is close to one of the map locations, then they drags their finger and release close to the same map location
5. *testDontCreateCircuit*: tests that the *detectCircuit* method returns false when the number of line segments equals the number of map locations, but the line segments do not form a single circuit (you do not have to worry about the "two circuit" problem that you fixed in Homework #2).
6. *testCreateCircuitNotShortestPath*: test that the *detectCircuit* method returns true when the number of line segments equals the number of map locations and the line segments form a circuit, but that the value returned by *myPathLength* is not the same as the distance for the shortest path for that set of map locations.
7. *testCreateCircuitShortestPath*: test that the *detectCircuit* method returns true when the number of line segments equals the number of map locations and the line segments form a circuit, and that the value returned by *myPathLength* is the same (within 0.01) as the distance for the shortest path.

Notes:

- For tests 1-4, you can directly manipulate the *mapPoints* field and then directly inspect the *segments* field. You do, however, need to use MotionEvents to simulate the user touching the screen and moving their finger. See the Android Testing lecture slides for an example of how to do this.
- For tests 5-7, you can directly manipulate the *segments* field, i.e. you don't have to simulate MotionEvents.
- For test 1 (*testCreateSegment*), keep in mind that it is not sufficient to check that another line segment has been added to the *segments* field; you must check that the segment connects the correct points. It's a little tricky, though: the values held in *mapPoints* represent the top left corner of the square that is drawn onscreen, whereas the values held in *segments* refer to the center of the square and are shifted by 10 pixels on the x- and y-axes from the values in *mapPoints*.

- You may *not* refactor or modify the implementation of the TSP app in order to write your unit tests. Please notify the instructor if you feel that it is necessary to do so.
- If written correctly, all of these tests should pass! If you believe that your test is sound but the test fails, please notify the instructor.

Part 2. White-Box Testing

Presumably you have heard of the game Hangman, in which one player thinks of a word and the other tries to guess it by suggesting letters. If the player cannot figure out the word in a certain number of guesses, then he or she loses. See http://en.wikipedia.org/wiki/Hangman_(game) if you are not familiar with this game.

You may have had something like this as an introductory programming assignment, in which the computer program randomly picks a word and the human player has to guess it.

In 2011, a variation of this game was presented at the SIGCSE conference on computer science education. In this variation, called "Evil Hangman" (http://nifty.stanford.edu/2011/schwarz-evil-hangman/), the computer program changes the target word according to what letters the human player has suggested, making it very hard to correctly guess the word. Sneaky, huh?

The "evil" program works by maintaining a list of words and then, each time the player suggests a letter, removing all words containing that letter from its list (assuming that some words would still remain), essentially dodging the human player's guesses as much as possible. Although it is possible for the human player to win, it is certainly tricky to do so.

In this assignment, you are given an implementation of Evil Hangman and you need to conduct white-box testing of three methods.

Implementation

The code that you will test is available in Canvas. We recommend that you use Eclipse for this part of the assignment.

The app that you will test uses the Java Swing API for its user interface; if you haven't encountered Swing before, it's pretty similar to Android in that you have objects to represent the window itself (in Swing, these are called JFrames), you have the various UI widgets (e.g. JLabel, JComboBox, JButton, etc.), and you have methods that are called whenever the user interacts with a widget (this method is usually called "actionPerformed").

There are four classes that represent the different windows that appear in the app:

- **Start**: This is shown when the application starts. It contains drop-down lists to allow the user to choose the number of letters in the target word and the

maximum number of incorrect guesses allowed. This contains the “main” method for the program, too.
- **GUI_PlayGame**: This is the window in which the user plays the game by clicking on buttons for the different letters.
- **GUI_Winner**: This window is shown when the user correctly guesses the word. You won't be seeing it much. ;-)
- **GUI_Loser**: This is shown when the user runs out of letters to guess.

There are three other classes that represent the data and logic for playing the game:
- **HangmanGame**: This is an interface that defines the methods needed in order to play hangman.
- **NormalHangMan**: This is an implementation of the interface that plays hangman by the normal rules.
- **EvilHangMan**: This is an implementation that changes the target word each time the player makes a guess.

This app starts out by using EvilHangMan as the HangmanGame implementation; however, if the player guesses a letter, and removing all words with that letter would mean there are no legal words left, then that guess is considered to be correct (EvilHangMan may be evil, but it's not a cheater). At that point, the app chooses one of the legal words from the dictionary and switches over to NormalHangMan rules.

Testing Activities
In this part of the assignment, you need to write **JUnit tests** for these three methods:
- EvilHangman.makeGuess
- NormalHangman.makeGuess
- GUI_PlayGame.controller

Please use JUnit 4 for your test cases, and follow the naming conventions discussed in class.

Your goal is to create a set of unit tests that achieves **100% statement coverage** for each method.

Note that GUI_PlayGame.controller calls makeGuess, but you still need separate unit tests for the two makeGuess methods anyway. That is, when you run your tests for controller, you’ll also cover some statements in makeGuess, but you cannot count those toward the statement coverage for makeGuess. You are expected to have a separate set of tests for each makeGuess method, and those tests should achieve 100% statement coverage.

Keep in mind that it is not enough just to have tests that call each method with various inputs so that the different statements are covered; **your tests must be sound** and **must check for any visible side effects** (i.e., changes to public or protected fields) of calling the method, as well as its return value. This means that for GUI_PlayGame.controller, your tests must check that the JLabels have the right values (hint: use the “getText”

method) and that the HangmanGame field is properly set up; you do not, however, need to check that the Winner or Loser screen was displayed.

Please note that you should *not* change the implementation of any classes that we provided in order to write your tests. If you think a change is necessary, please speak with a member of the instruction staff.

Measuring Coverage

To measure the statement coverage achieved by your unit tests, we recommend you use CodePro (https://developers.google.com/java-dev-tools/codepro/doc/), which may already be bundled with your Eclipse distribution. Open the Package Explorer of your Java project and right-click on a .java file to open the context menu. If you see "Coverage As" toward the bottom of the menu, you're all set.

Otherwise, to install the CodePro plugin on your own machine, follow these instructions:

- In Eclipse, click the Help menu.
- Click "Install New Software..." or a similar option in older versions
- Click "Add..."
- For the "Name" field, put "CodePro"
- In the "Location" field, enter "http://dl.google.com/eclipse/inst/codepro/latest/3.7" if you are using Eclipse 3.7 or later, and "http://dl.google.com/eclipse/inst/codepro/latest/3.6" if you have 3.6
- Click OK
- You should now see CodeCoverage, CodePro, and Infrastructure as installation options
- Click "Select All"
- Click "Next"
- Agree to all the license agreements
- Click "Finish" and wait a few minutes for it to install
- Exit and restart Eclipse (it should prompt you to do this, if not, do it manually).

Once CodePro is installed, you can measure coverage by right-clicking on the name of your JUnit test class, then choosing "Coverage As" from the context menu, and then "JUnit Test."

You should see a new panel open at the bottom of Eclipse that allows you to navigate to your source code and see the amount of coverage that was achieved.

If you open the code in the code editor window, you'll see covered statements in green and uncovered statements in red. Yellow statements are decision points for which branches have only been partially covered; you can consider these as "covered" for our purposes.

Some students have had difficulty getting CodePro to work on Eclipse 3.7 and later. Feel free to use other coverage tools such as EclEmma (http://www.eclemma.org) or eCobertura (http://ecobertura.johoop.de/). Instructions will be posted in Canvas.

If some statements cannot be covered, create a plain-text README file and explain why the statements are unreachable, e.g. by stating the unsatisfiable path condition.

A few words about using Piazza...

As always, please be careful about how you use Piazza for this assignment. It is important that you do not reveal (accidentally or intentionally) which test cases you've created or how you wrote them.

Please use Piazza for (a) clarification questions regarding the intent of the assignment, (b) clarification questions regarding the specification of the code you're testing, and (c) getting help with problems using Android Studio, Robolectric, JUnit, coverage tools, etc.

Before you post anything on Piazza, think to yourself "is my question giving away an answer?" If you think it is, please email the instruction staff directly and we'll post it if we think it's appropriate.

Academic Honesty and Collaboration

You are expected to work **alone** on this assignment.

You are free to discuss the intent of the assignment with your classmates and ask clarification questions, and also to help other students with setting up the testing tools and environments.

However, you may not discuss or share solutions or findings with other students. In particular, *you absolutely must **not** share or discuss your test code or specific test cases*. Please see the course policy on academic honesty (posted in Canvas on the "Syllabus" page) for more information.

As always, if you run into problems, please ask a member of the teaching staff for help before trying to find solutions online or asking your classmates!

Deliverables

Your final deliverable should consist of a *single* zip file containing the test classes you created: GameActivityTest.java for Part 1; and the three test classes for Part 2. If you find unreachable statements in Part 2, describe those in a README file and put that in your zip file, too.

Please do *not* submit the entire Android Studio or Eclipse projects! Submit *only* your test classes!

Submission

All deliverables are due in Canvas by Friday, October 16, at 5:00pm. Late submissions

will be penalized by 10% if 0-24 hours late, 20% for 24-48 hours late, and so on, up to one week, after which they will no longer be accepted.

Last updated: 1 Oct 2015, 5:07pm