Unit Testing
F-22 Raptor Fighter
F-22 Raptor Fighter

- Manufactured by Lockheed Martin & Boeing

![F-22 Raptor Fighter diagram]
F-22 Raptor Fighter

• What would happen if Lockheed assembled an F-22 with "untested" parts (i.e., parts that were built but never verified)?

• It wouldn't work, and in all likelihood you would never be able to make it work – Cheaper and easier to just start over
Managing implementation complexity

• Individual parts should be verified before being integrated with other parts

• Integrated subsystems should also be verified

• If adding a new part breaks the system, the problem must be related to the recently added part

• Track down the problem and fix it

• This ultimately leads to a complete system that works
2 approaches to programming

• Approach #1
  – "I wrote ALL of the code, but when I tried to compile and run it, nothing seemed to work!"

• Approach #2
  – Write a little code (e.g., a method or small class)
  – Test it
  – Write a little more code
  – Test it
  – Integrate the two verified pieces of code
  – Test it
  – ...

Unit testing

• Large programs consist of many smaller pieces
  – Classes, methods, packages, etc.

• "Unit" is a generic term for these smaller pieces

• Three important types of software testing are:
  – Unit Testing (test units in isolation)
  – Integration Testing (test integrated subsystems)
  – System Testing (test entire system that is fully integrated)

• Unit Testing is done to test the smaller pieces in isolation before they are combined with other pieces
  – Usually done by the developers who write the code
What unit tests do

- Unit tests create objects, call methods, and verify that the returned results are correct

- Actual results vs. Expected results

- Unit tests should be automated so that they can be run frequently (many times a day) to ensure that changes, additions, bug fixes, etc. have not broken the code
  - Regression testing

- Notifies you when changes have introduced bugs, and helps to avoid destabilizing the system
Test driver program

• The tests are run by a "test driver", which is a program that just runs all of the unit test cases

• It must be easy to add new tests to the test driver

• After running the test cases, the test driver either tells you that everything worked, or gives you a list of tests that failed

• Little or no manual labor required to run tests and check the results

• Tools like Ant or Make are often used to automate the building and running of the test driver (e.g., $ ant test)
Android testing framework

• Android provides a framework for writing automated unit tests
  – Based on the popular JUnit unit testing framework

• There are two types of Android unit tests
  – Instrumented Unit Tests
    • These tests depend on Android-specific classes, and so must be run on an Android device
  – Local Unit Tests
    • These tests depend only on standard Java classes, and so can be run on the development computer instead of on an Android device
Android instrumented unit tests

- Require a device or emulator to run
- App’s primary source code is located in the folder
  - app/src/main/java/<app-package>
- Instrumented unit test code is located in the folder
  - app/src/androidTest/java/<app-package>
Android instrumented unit tests

• For each primary class, you can create a corresponding test class

• Put the test class in the same package as the primary class
  – app/src/main/java/<app-package>/SomeClass.java
  – app/src/androidTest/java/<app-package>/SomeClassTest.java

• Putting both classes in the same package (although different folders) gives the test class greater access to the primary class’s members
Android instrumented unit tests

• Examples
• SuperAsteroids (basic test)
  – app/src/main/java/edu/byu/cs/superasteroids/core/GraphicsUtils.java
  – app/src/androidTest/java/edu/byu/cs/superasteroids/core/GraphicsUtilsTests.java

• BookClub (database test, code under database lecture)
  – app/src/main/java/edu/byu/cs240/bookclub/database/BookDAO.java
  – app/src/androidTest/java/edu/byu/cs240/bookclub/database/BookDAOTest.java
Android instrumented unit tests

• Test class is subclass of
  – `android.test.AndroidTestCase`

• Put test cases in `test*` methods

• Test methods follow this outline:
  – Initialize test objects/data
  – Invoke methods on test objects
  – Use `assert*` methods to compare expected and actual results
  – Cleanup test objects/data
Android instrumented unit tests

- Test methods often have redundant (i.e., duplicated) initialization and cleanup code
- You can override AndroidTestCase’s setUp() and tearDown() methods to centralize redundant initialization/cleanup code
  - setUp() is run before each test* method
  - tearDown() is run after each test* method
- Test methods should not influence each other (i.e., they should not depend on each other or have “cross talk”)
- It should be possible to run the test methods in a random order without affecting the results
Running instrumented unit tests

• Make sure a device or emulator is available
• In Android Studio, open the “Build Variants” tool window in the bottom-left corner, and set the “Test Artifact” setting to “Android Instrumentation Tests”
• To run a single test class, in the “Project” tool window right-click on a test class name, and select “Run *Tests”
• To run all of your instrumented unit tests, right-click on the “androidTest/java” folder, and select “Run All Tests”
Android local unit tests

• Can run on the development computer without a device or emulator
• App’s primary source code is located in the folder
  – app/src/main/java/<app-package>
• Local unit test code is located in the folder
  – app/src/test/java/<app-package>
Android local unit tests

• Example
• SuperAsteroids (local test)
  – app/src/test/java/edu/byu/cs/superasteroids/core/LocalTests.java
Android local unit tests

• Local test classes are written using the JUnit 4 unit test framework
• Include the following in app/build.gradle
  
dependencies {
    ...
    testCompile 'junit:junit:4.12'
  }

• Import JUnit 4 classes
  import org.junit.*;
  import static org.junit.Assert.*;
Android local unit tests

- Test classes are just regular classes (no special superclass)
- Test methods may have any name (need not be test*), but must have the @Test annotation on them
- Common initialization code can be placed in a method (any name) with the @Before annotation
- Common cleanup code can be placed in a method (any name) with the @After annotation
- Use JUnit assert* methods to implement test cases
Running local unit tests

• No device or emulator is needed
• In Android Studio, open the “Build Variants” tool window in the bottom-left corner, and set the “Test Artifact” setting to “Unit Tests”
• To run a single test class, in the “Project” tool window right-click on a test class name, and select “Run *Tests”
• To run all of your local unit tests, right-click on the “test/java” folder, and select “Run All Tests”
JUnit 4 unit testing framework

- [JUnit 4 Documentation](#)
- Use JUnit 4 annotations to mark test methods

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Test public void method()</td>
<td>The annotation @Test identifies that a method is a test method.</td>
</tr>
<tr>
<td>@Before public void method()</td>
<td>Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).</td>
</tr>
<tr>
<td>@After public void method()</td>
<td>Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).</td>
</tr>
</tbody>
</table>
JUnit 4 unit testing framework

- Use JUnit 4 annotations to mark test methods

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@BeforeClass public void method()</td>
<td>Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database.</td>
</tr>
<tr>
<td>@AfterClass public void method()</td>
<td>Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database.</td>
</tr>
<tr>
<td>@Test (expected = Exception.class)</td>
<td>Fails, if the method does not throw the named exception.</td>
</tr>
<tr>
<td>@Test(timeout=100)</td>
<td>Fails, if the method takes longer than 100 milliseconds.</td>
</tr>
</tbody>
</table>