ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 16 — February 7, 2013

Patrick Lam *version 2*

More on Unit Testing

Now that we've introduced general information on testing, we are going to talk about the most popular unit testing framework for Java, JUnit[1]. In Assignment 6, you will develop both standalone JUnit tests and JUnit tests for Android. This material will reinforce your understanding of assertions.

Objective. You will learn how to write simple JUnit tests for Java code, which usually use `assertTrue` or `assertEquals` to verify their results.

Practical Information. JUnit tests are organized into test classes. Typically, you'd have a test class for each class that you would like to test. You would label tests with the `@org.junit.Test` attribute[2].

A unit test makes calls to the class that you're testing and verifies that the class is doing the right thing, using assertions (as discussed previously).

After you've written the unit tests, you can just press a button in your IDE and it will run the tests automatically for you, so that you'll know whether the code passes the tests (green bars) or not (yellow or red bars). The biggest benefit of unit tests is that they can run automatically; you'll never run tests that are annoying to run.

[1] `http://junit.sourceforge.net`

[2] This is accurate for JUnit 4. There is more overhead for JUnit 3 tests.

Account Example. Here is a simple example.

```
public class Account
{
  private float balance;
  public void deposit (float amount) {
    balance += amount;
  }

  public void withdraw (float amount) {
    balance -= amount;
  }

  public void transferFunds(Account destination, float amount) {
  }

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

Let's write a unit test for this class:

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

public class AccountTest {
  @Test
  public void transferFunds() {
    Account source = new Account();
    source.deposit(200.00f);
    Account destination = new Account();
    destination.deposit(150.00f);

    source.transferFunds(destination, 100.00f);
    assertEquals("destination balance", 250.00f, destination.getBalance(), 0.01);
    assertEquals("source balance", 100.00f, source.getBalance(), 0.01);
  }
}
```

We've labelled the `transferFunds()` method as a test with the @Test annotation. Then, we do operations on the `source` and `destination` accounts, and *assert* that, after we transfer `100.00f` from the `source` to the `destination`, both the `source` and `destination` contain the right amount of money.

If we run this test, we'd get a red bar; `transferFunds` doesn't do anything yet. Adding this code:

```
public void transferFunds(Account destination, float amount) {
    destination.deposit(amount);
    withdraw(amount);
}
```

makes the bar green, since the test now passes.

Fixtures: sharing resources among tests

Sometimes you have a number of objects that get used by more than one test. Setting up these objects can be repetitive. JUnit provides the notion of test fixtures to help alleviate this problem.

```
public class AccountTest {
  private Account account1;

  @Before public void setup() {
    account1 = new Account(); account1.deposit(200.00f);
```

```
  }
}
```

Similarly, you can free any resources (like files) that you open in a `@Before` by writing a teardown method and annotating it with `@After`.

Beware. Note that tests should never rely on any persistent state of the objects (like the account balance). JUnit does not guarantee that it will execute tests in any particular order.