ECE155: Engineering Design with Embedded Systems Winter 2015

Lecture 15 — Testing: Introduction and JUnit

Patrick Lam & Jeff Zarnett

Software Testing

In 4th year, the university offers a complete course on software testing, ECE453. Unfortunately, that's really late in the program, so we'll give you a preview in the hopes that it helps you write better software as a co-op (and in your other classes).

Testing attempts to verify the functionality of your software. Testing software is somewhat like testing physical systems, but physical systems tend to have a small number of ways in which they fail, but software can fail in many bizarre ways. Also unlike physical systems, software defects are often design errors and not manufacturing defects. Software does not rust or corrode, so bugs don't develop over time, they were always there, even if it took a long time to discover them [Pan99].

Software bugs are a fact of life, not because programmers are careless, but because software is extremely complex and the ability of the human mind to deal with complexity is limited. This is not just for implementation, but applies also to design: most software's design is complex and understanding the design is also a problem. But it gets worse: programs are dynamic entities: when a programmer finds a bug and fixes it there is a possibility that a new bug has been introduced alongside that fix. So a test that previously passed will suddenly start to fail [Pan99].

Because of the complexity of software and the limitations of testing (which we will discuss a bit later on), it will never be possible to test software completely and never possible to guarantee that a (non-trivial) program does not have bugs in it. But we test anyway, to improve the quality of the software. We can make it better, even if we can't make it perfect.

A study by the US National Institute of Science and Technology (NIST) showed that in 2002, software bugs cost the United States economy $59.5 billion annually, and that more than one third of this cost could be saved if testing practices were better [NIS02].

The earlier a bug is found, the cheaper it is to fix it. If a problem is found by internal testing, the software is updated and a new build is created. If the bug is found by the customer, it might cost 10-100x more to fix it [McC04]. Some ways it might be costly: support calls from the customer, recovering corrupted data, deploying the new version to all customers, and many more.

Test Cases

When you're testing, you'll run *test suites* consisting of *test cases*.

A *test case* contains:

- what you feed to software; and
- what the software should output in response.

The second point is not as simple as it sounds. To really be sure that the software does what it is supposed to do, we have to know what the software should output as the correct response, independent of the actual output of the software. If the function is supposed to add two numbers, we know in advance

what the expected output of $2 + 2$ is. It is dangerous, however, when the programmer does not know what the correct answer should be.

It is unfortunately common that when people write tests, they take the current output of the software as correct and the test is then written to expect that, even if it is wrong. Imagine if the computer said $2 + 2 = 5$ and a programmer wrote a test to check the add function, and the test expected the answer 5! This is a comical example, but for more complex input and outputs this can happen.

You can organize the collection of test cases you need to write according to a *test plan*.

Testing versus correctness. Common wisdom used to be that you could only find defects using testing, and not prove correctness, which would require exhaustive testing. However, there have been recent research advances which make it possible to prove correctness using testing.

Levels of Testing

Testing is often performed at several different levels:

- *Smoke Tests* are the most basic of checks.
- *Unit tests* are low-level tests which verify the functionality of a single class (or unit) at a time.
- *Integration tests* combine more than one unit and verify the functionality of the interfaces between components.
- *Stress tests* run to see how components or systems behave when under pressure.
- *System tests* run on the entire system, after it has been integrated, and verify that everything works right.

Smoke Testing

The term comes from electrical engineering... if you turn on your circuit and smoke comes out, something is catastrophically wrong. Sometimes we do this in software as well; a minimal set of tests to be sure that the system starts up and runs without anything going disastrously wrong. However, smoke tests do not ensure anything about the program other than it turns on without smoke streaming out.

Unit Testing

We will discuss principles behind unit testing and how and when to best deploy unit testing. We'll provide definitions of what people usually mean when they say unit tests.

1. Test small parts—units—of a software system (method, class, set of classes) independently.
2. Specify the desired behaviour of the unit using tests.

Recall from earlier: test-driven development, which is currently the most-advocated way to write unit tests; mock objects; and automation. Some basic properties of unit tests:

- focus on the unit being tested; for instance, don't depend on a database, network, or other external resources;

- easy to run by anyone (e.g. by setting them up as JUnit tests): they must incur no setup costs to run nor require user input; they can therefore serve as regression tests;
- easy to write (a few minutes per test) and focus on one aspect of behaviour; they should tell you something about the unit.

Unit tests come with a lot of baggage and people telling you how to do things, e.g.:

- specify and document the requirements of the unit;
- test behaviour, not state, and use mock objects to verify behaviour;
- some say that unit tests ought to be created during development (TDD) and written first, even before the code to be tested exists.

Assertions. Before we begin, one important definition. An assertion is a statement about the world; for instance, I assert that PowerPoint is often harmful to education. In the context of code, assertions are logical expressions that should always be true. When the program executes an assertion, it verifies that the logical expression indeed evaluates to true. If not, then it throws an `AssertionError`, which will usually stop the program. You can also supply a second parameter to the assertion, containing a message to be reported with the error in the event of assertion failure.

Unit Testing Example: JUnit

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].

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.

Account Example. Here is a simple example.

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

[1] `http://junit.sourceforge.net`
[2] This is accurate for JUnit 4. There is more overhead for JUnit 3 tests.

```
  }

  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:

```
[language={Java}]
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:

```
[language={Java}]
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.

```
[language={Java}]
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.

Useful Concept (and Helpful for Testing)

Imagine that you are writing a JUnit test for a piece of code that is used for data retrieval, and you have a set of student IDs and student names. You might have two `Array` objects, defined like this:

```
private ArrayList<String> studentIDs = {''20000000'', ''20000001'', ''20000002''};
private ArrayList<String> studentNames = {''John Doe'', ''Jane Doe'', ''Max Mustermann''};
```

You might store your data in two `Array`s and when you test the code, you need to make sure your index in the array of Student IDs is the same as the array of Student Names and that the two arrays line up. So you call the function `getName(studentIDs[i])` and check with an assertion statement that the name returned matches `studentNames[i]`. It would be nice if we could have a relation between the student IDs and names without needing to maintain the index variable `i`.

Eliminating the index variable would leave us with what we call a *Key-Value Pair*. The related Arrays could then be rewritten into a set of pairs as below. The key is on the left, and the value is on the right.

```
(''20000000'', ''John Doe'')
(''20000001'', ''Jane Doe'')
(''20000002'', ''Max Mustermann'')
```

The data structure we use to represent this in Java is called a `HashMap`. A `HashMap` holds a set of keys and values. Keys must be unique, and each key corresponds to exactly one value. A list definition has a type, such as `ArrayList<String>`. A `HashMap` has two types, the type for the Key and the type for the Value. If the Key is type `String` and the Value is type `Money`, it is written in Java as `HashMap<String, Money>`. The types of the key and value can be anything, such as having a relation between `Student` and `Address`.

Interesting insight: a `List` of type `T` is effectively a `HashMap<int, T>`. You give in an integer (e.g., 0) and get a value (the object of type `T`). The methods used have different names and signatures (get/set/add vs. get/put/remove), but they are functionally the same. Their internal representations may be different.

How to use a HashMap. To retrieve some data from the `HashMap`, we use the `get()` method with the key of the item you want: `get(''20000000'')` returns `''John Doe''`.

To add something to the `HashMap`, use the `put()` method, with the key and the value for that key: `put(''20000003'', ''Mel Mustermann'')` puts a new entry into the map. If there is already a value for the key that is given in the `put()` method, it replaces the value: if we `put(''20000000'', ''Michael Doe'')`,`''Michael Doe''` replaces `''John Doe''` in the map.

To remove something from the map, use the `remove()` method with the key of the item to remove: `remove(''20000001'')` will remove the entry `(''20000001'', ''Jane Doe'')` from the map.

To get a list of all the keys, use the method `keySet()`, which will provide all of the keys. This will let you step through all of the items of the map.

(There are, of course, other methods, like `clear()`, `containsKey()`, `containsValue()`, `isEmpty()`, `size()`, et cetera, but we won't go over them in detail.)

How would we test our `getName(String studentID)` function using a `HashMap` called `hm`?

```
for (String key : hm.keySet()) {
    assertEquals(‘‘Student Name’’, hm.get(key), getName(key));
}
```

References

[McC04] Steve McConnell. *Code Complete, 2nd Edition*. Microsoft Press, 2004.

[NIS02] NIST. Software errors cost U.S. economy $59.5 billion annually, 2002. Online; accessed 26-December-2013.

[Pan99] Jiantao Pan. Software testing, 1999. Online; accessed 26-December-2013.