

Lecture 15 – Testing: Intro & JUnit

Jeff Zarnett & Patrick Lam

`jzarnett@uwaterloo.ca` & `p.lam@ece.uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

December 29, 2014

Testing attempts to verify the functionality of software.

It is somewhat like testing physical systems.

But software can fail in many bizarre ways.

Most problems: design problems, not manufacturing problems.

Software does not corrode; so problems were always there.

Software bugs are a fact of life. Not because programmers are careless; because software's complex.

Human ability to deal with complexity is limited.

We can have problems of design or implementation.

Programs are dynamic: a change can have side effects.

A test that previously passed might now fail.

It will never be possible to test software completely.

We cannot guarantee a bug-free program.

But testing makes our software better.

NIST says that in 2002, software bugs cost the US economy: \$59.5 Billion.

They estimate $1/3$ could be saved with better testing practice.

The earlier a bug is found, the cheaper it is to fix.

When 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.

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

We have to know what is the correct output, independent of the actual output of the software

Dangerous if we don't know that answer.

Tests are often written after implementation and just use the current output as correct.

Imagine if the function returns $2 + 2 = 5$ and a programmer wrote a test that expects the answer 5!

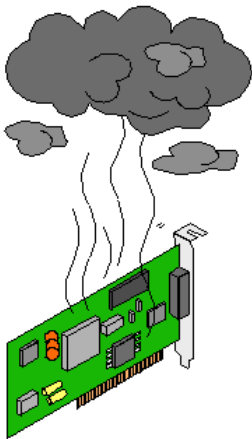
Common view: testing is only for finding defects.

Recent research allows proving correctness with testing.

Testing is often performed at several different levels:

- Smoke tests
- Unit tests
- Integration tests
- Stress tests
- System tests

From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.



Units: small parts of a software system
(methods, classes, sets of classes).

Two Key Ideas:

- 1 Test units independently.
- 2 Specify desired behaviour using tests.

- Focus on the unit being tested
- Easy to run
- Easy to write

Some advice on Unit Testing:

- specify and document the requirements of the unit.
- test behaviour, not state; use mock objects to verify behaviour.
- Test-Driven Development: write tests before the code.

Most popular unit testing framework for Java.

Tests depend on the notion of **assertions**.

An assertion is a statement about the world.

In code, logical expression that should always be true.

If not, throw `AssertionError`.

This usually stops the program.

Learn how to write simple JUnit tests for Java code.

Use `assertTrue` or `assertEquals` to verify test results.

JUnit tests belong to test classes.

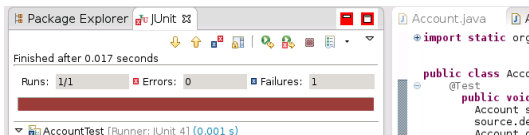
A test:

- is labelled `@org.junit.Test`;
- is a method with no parameters;
- makes calls to the class under test;
- verifies that the class is doing the right thing using assertions.

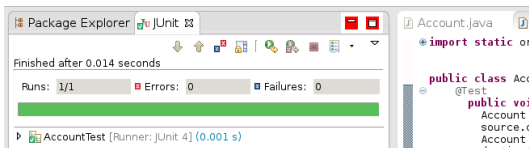
After writing the tests:

- 1 press a button in your IDE;
- 2 it will run the tests automatically for you.

If the tests fail:



If the tests pass:



Biggest benefit of unit tests:

- they can run automatically.

You'll never run tests that are annoying to run.

```
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);
    }
}
```

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

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

makes the bar green, since the test now passes.

Tests often share objects.

Repeatedly allocating these objects is terrible.

Solution: test fixtures.

```
public class AccountTest {  
    private Account account1;  
  
    // runs before any tests  
    @Before public void setUp() {  
        account1 = new Account();  
        account1.deposit(200.00f);  
    }  
}
```

Use `@After` to free resources afterwards.

Caveat: Don't rely on object state between tests: JUnit may shuffle order.

How do you test error-checking code?

How do you test error-checking code?

It should throw exceptions!

Make the test case expect an exception.

From the JUnit cookbook:

```
@Test(expected=IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```


So far, we saw the (better) JUnit 4 syntax.

Android uses JUnit 3 syntax. Key differences:

- test classes must extend `TestCase`;
- test names must start with `test`;
- fixture setup method must be called `setUp()`;
- fixture teardown method must be called `tearDown()`;
- testing exceptions is harder.

Useful Concept (not just for Testing)

Imagine a JUnit test for a piece of code that is used for data retrieval.

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''};
```

Useful Concept (not just for Testing)

Check: index in the array of Student IDs is the same as the array of Student Names and that the two arrays line up.

Call the function `getName(studentIDs[i])`.

Check with an assertion statement that the name returned matches `studentNames[i]`.

Can we have a relation between the student IDs and names without `i`?

A **Key-Value Pair**.

Key on the left; value on the right.

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

The Java data structure to represent this is a HashMap.

A HashMap holds a set of keys and values.

Keys must be unique. A key corresponds to exactly 1 value.

A list definition has a type: `ArrayList<String>`.

A HashMap has two types:
the type for the Key and the type for the Value.

Example: `HashMap<String, Money>`.

The types of the key and value can be anything, e.g., Student and Address.

An `List` of type `T` is effectively a `HashMap<int, T>`.

Give in an integer (0) and get a value (object of type `T`).

The methods used have different names and signatures.
But they are functionally the same.

Their internal representations may be different.

To retrieve some data from the HashMap:

`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.

`put("20000003", "Mel Mustermann")` → new entry in the map.

Already a value for the key? replaces the value:

`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")` → remove the entry ("20000001", "Jane Doe") from the map.

To get a list of all the keys: `keySet()`.

(There are other methods: `clear()`, `containsKey()`, `containsValue()`, `isEmpty()`, `size()`...)

Test our `getName(String studentID)` function using a `HashMap` called `hm`.

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