CompSci 230
Software Construction

Lecture Slides #22: Black Box testing with JUnit S1 2016

Agenda

- Topics:
  - Testing software
    - Testing techniques overview
  - Black Box Testing
    - Picking test cases
  - Writing test cases in JUnit

Testing

- Different types of testing
  - Purpose of testing?
- Black-box testing
  - Treat implementation under test as black box (look at it how it interacts with the outside but not at how it is implemented internally)
  - Typically design tests based on which output / behavior we expect for certain input
- White-box testing
  - Design tests based on implementation itself (structure of code)
  - Try to test all execution paths
- Usability testing
  - Get user opinion about an implementation

Black-box testing

- Implementation under test (IUT) is regarded as a "black box"
  - E.g., IUT could be a class, a method, a module, a service, etc.
- Tests interact with the IUT's interface
  - E.g., if the IUT is a class, tests interact with the class (or objects of the class) by calling its methods and inspecting their return values
- Objectives:
  - Design tests so any faults reveal themselves (i.e., trigger failures)
  - Minimise the number of tests required
- From here, we will assume that the IUT is a class or method and that we will pass parameters and receive return values (or trigger exceptions)

Designing black box tests

- Design tests so they *fail* when a fault triggers a failure
  - Tests that trigger intended behavior must not fail
- Example:
  - A method throws SomeCustomException when invoked with a certain combination of parameters. Suppose this is intended behaviour (i.e., it is what the method *should* do)
  - A test supplying such a combination must not fail when this exception is thrown (the exception is *not* a failure in this case). That is:
    - The test must not flag a fail if the exception is thrown and must catch the exception instead
    - The test must fail if the exception is *not* thrown

Choosing suitable test cases

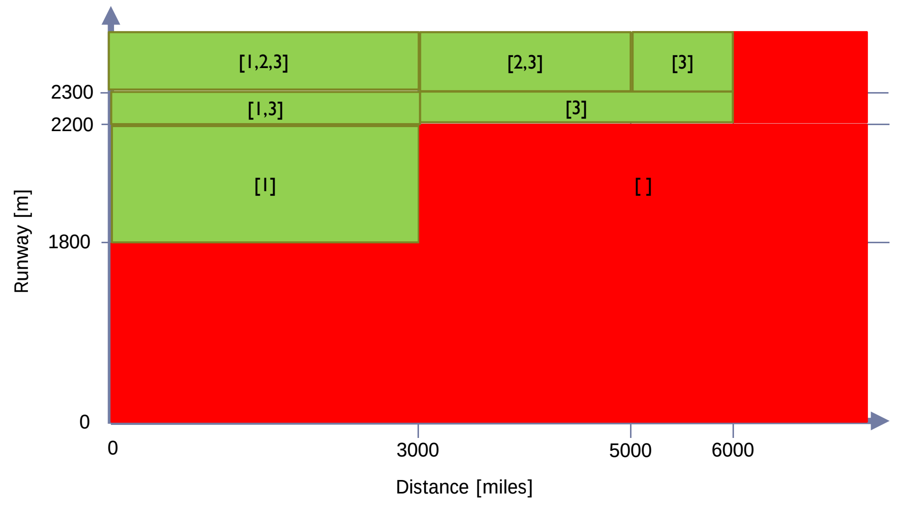
- In most cases, the number of possible parameter values that we can pass to a method is practically unlimited
  - Which ones do we choose for our tests?
- As a general rule, we want to choose those test cases that are most likely to trigger a failure
- We want to keep the number of test cases small (minimise effort)
- Writing test cases, identifying suitable test cases, and minimising the number of test cases all require effort.
  - Need to find some sort of compromise

Input partitioning

- In a lot of cases, we can partition the possible input space
- Example: An airline flies between a variety of destinations using a variety of aircraft. We have a method that computes which aircraft types can be used on which flight.
  - Input parameters: the shortest runway length of the two airports concerned and the distance between the airports
  - Return value: an array of suitable aircraft types

  We also know that:
  - Aircraft type 1 requires a minimum runway length of 1800 m and can fly 3000 miles
  - Aircraft type 2 requires a minimum runway length of 2300 m and can fly 5000 miles
  - Aircraft type 3 requires a minimum runway length of 2200 m and can fly 6000 miles

Input partitioning



Input partitioning - observations

- All parameter value combinations in a partition should give the same return value
  - Testing one value combination from each partition should give us all possible return values
  - Which of these value combinations are worth testing?
  - For some partitions, it's definitely worth testing several combinations, e.g., the red partition
- Observation from experience: Faults tend to occur mostly at boundaries between partitions
  - Test either side of each boundary

Good and bad test cases

- Which of the following would make a good test set for our example?
  1. (1799 m, 3000 nm), (1799 m, 3001 nm), (1800 m, 1 nm), (1800 m, 3000 nm), (2199 m, 3000 nm), (2199 m, 3001 nm), (2200 m, 3000 nm), (2200 m, 3001 nm), (2200 m, 5000 nm), (2200 m, 5001 nm), (2200 m, 6000 nm), (2200 m, 6001 nm), (2299 m, 3000 nm), (2299 m, 3001 nm), (2299 m, 6000 nm), (2299 m, 6001 nm), (2300 m, 3000 nm), (2300 m, 3001 nm), (2300 m, 5000 nm), (2300 m, 5001 nm), (2300 m, 6000 nm), (2300 m, 6001 nm)
  2. (1000 m, 1000 nm), (1000 m, 2000 nm), (1000 m, 3000 nm), (1000 m, 4000 nm), (1000 m, 6000 nm), (1000 m, 7000 nm), (1500 m, 1000 nm), (1500 m, 2000 nm), (1500 m, 3000 nm), (1500 m, 4000 nm), (1500 m, 6000 nm), (1500 m, 7000 nm), (2000 m, 1000 nm), (2000 m, 2000 nm), (2000 m, 3000 nm), (2000 m, 4000 nm), (2000 m, 6000 nm), (2000 m, 7000 nm) , (2500 m, 1000 nm), (2500 m, 2000 nm), (2500 m, 3000 nm), (2500 m, 4000 nm), (2500 m, 6000 nm), (2500 m, 7000 nm)
- Which of the "better" test cases would you consider the most important ones?

Test fixtures

- A *test fixture* is code that we write for the purpose of testing the IUT
  - Test fixtures set up the environment for a test, e.g., create and configure objects and other variables that we may wish to pass to the IUT
  - Test fixtures can bring the application into a known stable state so test results become reproducible
- *Drivers* are pieces of code that invoke methods in IUT
- *Stubs* are fake methods that the IUT can call while it is under test
  - give tester full control over the return value
  - may replace methods that have yet to be implemented
  - allow for in-test analysis of the parameters that the IUT actually passes to the method

JUnit

- JUnit is a test framework for Java
- Included in Eclipse (but not just Eclipse)
- JUnit tests are methods in classes
  - At top of the class .java file, we put: **import static org.junit.Assert.*;**
- Apart from that, the syntax is as for any other class, with some special additions:
  - Test methods are public void
  - Each method that represents a JUnit test is preceded by an @Test
  - A test class can contain multiple test methods
- JUnit tests usually also contain a number of special functions
- Tests pass if the method returns without a call to fail() or a failed assertion

Junit fail() and assertions

- A call to fail() in a test causes the test to fail
  - We use this if we detect ourselves that the test has failed
- An assertion describes a condition that must be true in order for the test not to fail
  - A test may still fail even if an assertion in the test is true
  - Junit provides a number of assertion methods: assertEquals(), assertTrue(), assertNull(), assertNotNull(), assertSame(), assertArrayEquals(), …
  - A test may contain more than one assertion (but shouldn't contain too many or it becomes difficult to tell why the test failed)

JUnit tests and exceptions

- If the IUT throws a custom exception (or we expect it to throw a built-in exception under normal operation), then it may be desirable to test if the exception is actually thrown in appropriate test cases
  - If so, we must catch the exception (and then do nothing – except maybe check that it was thrown for the expected reasons).
  - If the exception is unexpectedly not thrown, then we must explicitly fail() the test

JUnit example - IUT

```java
public class DodgyClass {

    public int addOne(int x) {
        return x+1;
    }

    public int addAbsoluteValues(int x, int y) {
        return x + y;
    }

    public double reciprocalIfSmallerThan10(double x) throws ArgumentIs10OrLargerException {
        if (x < 10) {
            return 1/x; // note this fails for x=0
        }
        else
        {
            throw new ArgumentIs10OrLargerException();
        }
    }

}
```

JUnit tests

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

public class TestDodgyClass {

    @Test
    public void testAddOne() {
        DodgyClass d = new DodgyClass();
        assertEquals(5,d.addOne(4)); // test fails if assert fails
    }

    @Test
    public void test1AddAbsoluteValues() {
        DodgyClass d = new DodgyClass();
        assertEquals(8,d.addAbsoluteValues(3,5)); // this assert succeeds by coincidence
    }

    @Test
    public void test2AddAbsoluteValues() {
        DodgyClass d = new DodgyClass();
        assertEquals(8,d.addAbsoluteValues(3,-5)); // this assert fails: test fails
    }
…
```

JUnit tests with exceptions

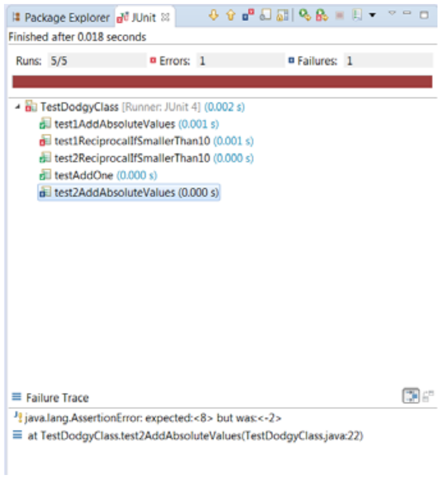
```
@Test
public void test1ReciprocalIfSmallerThan10() {
     DodgyClass d = new DodgyClass();
     try {
          assertEquals(0.33,d.reciprocalIfSmallerThan10(3),0.01);
     } catch (Exception e) {
          // we shouldn't get here if test succeeds, so explicit fail
          fail("Unexpected exception!");
     }
}

@Test
public void test2ReciprocalIfSmallerThan10() {
     DodgyClass d = new DodgyClass();
     try {
          d.reciprocalIfSmallerThan10(10); // this should throw an exception
          fail("Didn't throw ArgumentIs10OrLargerException for argument 10");
     }
     catch (ArgumentIs10OrLargerException e) {
          // just what we wanted, nothing to do here
     }
}
```

All exceptions should be caught; expected exceptions must be caught with no further action

Running JUnit tests in Eclipse

- Right-click on the test class in the Package Explorer -> Run as… -> JUnit Test
- This runs all tests in the class
- Results are available on the JUnit tab



Finished after 0.018 seconds

Runs: 5/5 Errors: 1 Failures: 1

```
TestDodgyClass [Runner: JUnit 4] (0.002 s)
    test1AddAbsoluteValues (0.001 s)
    test1ReciprocalIfSmallerThan10 (0.001 s)
    test2ReciprocalIfSmallerThan10 (0.000 s)
    testAddOne (0.000 s)
    test2AddAbsoluteValues (0.000 s)
```

Failure Trace

```
java.lang.AssertionError: expected:<8> but was:<-2>
at TestDodgyClass.test2AddAbsoluteValues(TestDodgyClass.java:22)
```

Review

- Give an example of something that black box testing does not do
- What are we trying to achieve when we design black box tests?
- In COMPSCI230, students pass right out if they achieve a minimum of 50% of marks in the theoretical part and the practical part. Imagine a method that take the marks for the theoretical and practical parts and returns true for a guaranteed pass and false for a potential fail. Can you draw a diagram showing the input partitioning for these cases?
- Are JUnit tests drivers or stubs?
- What happens when an assertion fails?
- What happens when an assertion holds?
- What do we need to do if an IUT is supposed to throw an exception in the course of correct operation?