Effective Testing

COMSM0086

Dr Simon Lock & Dr Sion Hannuna

Typical Conversation (After DB Feedback)

```
Student: Why did I lose marks for the DELETE command ?
Marker:  Well, does it work in ALL situations ?
Student: I've got lots of tests - it passes all those !
Marker:  Yes, but does it work in ALL situations ?
Student: I spent much time on it and wrote many tests
Marker:  What about if you try to DELETE *****
Student: Yes, I'm sure it works in that situation
Marker:  Why not try it and see what happens ?
                    [Some time later]
Student: That's strange, I'm sure it *should* work
Marker:  But it didn't ? That's why you lost the mark !
```

Perhaps some advice to improve testing ?

Testing

In other units you'll learning the theory of testing
In this unit you get the chance to apply it in practice

Nobody actually *enjoys* writing tests (anyone ?)
But it's *essential* part of professional development

We've already encountered the JUnit test framework
You should already have a feel for the way it works
But it would be good to think about HOW to use it

Levels of Abstraction

Devise your tests to target multiple levels:

Element testing: Exercise each & every line of code
For example, make sure all branches are executed

Unit Testing: Fully test operation of each method
Test different combinations of various parameters

Integration Testing: Test all high-level features
Ensure all user goals can be achieved using system

Marking

During marking, we only perform integration testing

Other levels require knowledge of code structure

But everyone's code is different (or should be ;o)

IMPORTANT THOUGHT

If you skip "element" and "unit" testing however

The chances are your code will be pretty buggy

And will thus fail OUR integration test cases

Equivalence Partitions

Fundamental principle of Equivalence Partitions is...
We don't need to test EVERY SINGLE input value
We can cluster input values together by similarity

Just pick representative values from each "cluster"
This drastically reduces the number of test cases
(Without significantly impacting the test coverage)

The crucial concept is: Test ALL clusters (partitions)
Rather than all cases in JUST A FEW clusters

Example Unit Test Scenario

Let's imagine we are writing a 'Month' class:

```
Month currentMonth = new Month(MARCH,2024);
```

We want to Unit test the 'getDayOfWeek' method:

```
String dayOfWeek = currentMonth.getDayOfWeek(20);
```

You *could* test method with ALL possible integers

```
for(int i=Integer.MIN_VALUE; i<=Integer.MAX_VALUE; i++)
```

But would take a long time (and is unnecessary)
Instead we pick a set of representative numbers...

Suitable Equivalence Partitions

Negative numbers: check invalid inputs are trapped

Zero: a "special" boundary case, in an EP on its own

One: a "special" boundary case, in an EP on its own

Numbers from 2 to 28: should all work the same

Borderlines: 29, 30, 31 (number of days in month)

Big numbers: >31 check invalid inputs are trapped

-2 0 1 5 31 32 50

Example of Poor Testing

Test cases I would have chosen...

- Both extremes: Black and White
- A selection from the across greyscale range
- Primary light colours (Red, Green, Blue)
- Primary print colours (Cyan, Yellow, Magenta)
- A typical dark colour (low brightness)
- A typical light colour (high brightness)
- A typical dull colour (low saturation)
- A typical vibrant colour (high saturation)
- Different fabrics/materials !
- Different lighting (sun, halogen, redhead, LED, UV)

But I digress, let us return to software...

Top Ten Tips for Writing Test Cases

1. Keep tests short & simple - test just one feature
   Easier to keep track of what is being tested

2. It's fine to have more that one assertion in a test
   It's good to check feature from different angles

3. OK to have more than one test case in a method
   Clustering for same feature limits method count

4. Don't waste time with brute force coverage
   Use equivalence partitions to target test cases

5. Often need to perform a number of "setup" steps
   Which is where @BeforeEach is useful

Top Ten Tips for Writing Test Cases

6. Give test methods detailed and descriptive names
   So it is clear which tests have failed in the output

7. Include useful and informative failure comments
   So you know what has failed and why it failed

8. Be sure to cover all core feature (obviously)
   But also spend time checking the edge cases

9. Ensure software DOESN'T do what it SHOULDN'T
   As well as it DOES do what it SHOULD

10. Use sub-functions to perform common tasks
    "Divide and conquer" as with normal code

Failure Comments

All assertions take a 'failure comment' parameter
A message which gets printed out if assertion fails

These can be extremely useful (if used well)
Try to write specific and detailed messages

Include parameters passed in to provide more detail

```
"Interpreter failed on " + incommingCommand
```

There is a lot of information printed during testing
Careful for-thought will make debugging a lot easier

Failure Comment Examples

Bad Failure Comments

```
error claiming cell
wrong exception occurred
incorrect return value
```

Good Failure Comments

```
cell owner still null after attempting to claim cell A2
expected an IOException, but got a NullPointer instead
expected 100 to be returned, but 10 returned instead
```

Note: Failure report includes the test method name
So be sure to give it a clear and descriptive name

Importance of Testing

We have accumulated a set of marking test cases
We will use these to assess your final submissions

You won't get the opportunity to see OUR test cases
Only way to ensure your code performs well is to...
Make sure YOUR test cases as extensive as OURS

Remember that this unit is not just about "coding"
We aim to teach 'development' (that includes testing)

Each Year Someone Always Asks...

Why can't we see the actual marking test cases ?

It's like you aren't giving us the full assignment !

To help explain why we don't make them available...

Let's consider the DB template example test script:

```
sendCommandToServer("INSERT INTO marks VALUES ('Simon', 65, TRUE);");
sendCommandToServer("INSERT INTO marks VALUES ('Chris', 20, FALSE);");
response = sendCommandToServer("SELECT * FROM marks;");
assertTrue(response.contains("[OK]"), "A valid query was made, but [OK] tag not returned");
assertTrue(response.contains("Simon"), "Simon added to table, but not returned by SELECT *");
assertTrue(response.contains("Chris"), "Chris added to table, but not returned by SELECT *");
response = sendCommandToServer("SELECT * FROM libraryfines;");
assertTrue(response.contains("[ERROR]"), "Access non-existent table, [ERROR] not returned");
```

A Suitable Solution ?

Following code would pass all of those tests just fine:

```
public String handleCommand(String command) {
    if(command.contains("SELECT id")) return "[OK]\n5\n";
    if(command.contains("libraryfines")) return "[ERROR]\n";
    if(command.contains("SELECT *")) return "[OK]\nSimon\nChris\n";
    return "";
}
```

An extreme example, but illustrates a key point...

There is natural reaction to focus on just passing tests

Rather than writing "correct" and "error-free" code

The Dichotomy of Testing

Good code will *always* pass the tests

BUT

Code that passes the tests may not *always* be good