CITS5501 Software Testing and Quality Assurance
Semester 1, 2020
Week 3 workshop – Data-driven tests and test design

1. Parameterized tests

In last week's lab, we briefly saw an example of a *paramaterized test* in JUnit. Normal test methods in JUnit don't take any parameters at all.

The test method, `addZeroHasNoEffect`, took *one* parameter, an int, from a list of ints specified by the `@ValueSource` above it. Run the tests in BlueJ or your IDE to confirm that all those ints are used; try changing them or adding to them.

But what if we have multiple parameters? Or what if we wish to specify not just the test values, but the expected values? The code for this week is the same as last week's but with two new methods – `tableOfTests` and `additionTestCasesProvider`.

`tableOfTests` is designed to get test values and expected values from some other source, and then run them as tests. What source? Another method – the annotation `@MethodSource("additionTestCasesProvider")` says "Go and call the `additionTestCasesProvider` method in order to get a list of test values and expected values. You can read more about `MethodSource`s in the JUnit documentation, here.

This sort of testing is called *data-driven* testing – we have basically the same test being run, but with different values each team; so it makes sense to write the logic for the test just once (rather than three times).

Question: Looking at the code, how many *test cases* would you say `tableOfTests` and `additionTestCasesProvider` comprise?

Exercise: Based on this example, try writing your own parameterized tests for other methods (for instance, subtraction).

2. Preconditions and postconditions

Consider the following scenario:

> A database has a table for students, a table for units being offered, and a table for enrolments. When a unit is removed as an offering, all enrolments relating to that unit must also be removed. The code for doing a unit removal currently looks like this:

```java
/** Remove a unit from the system
 */
void removeUnit(String unitCode) {
  units.removeRecord(unitCode);
}
```

It is recommended you discuss the following questions with a partner, and come up with an answer for each:

a. What *preconditions* do you think there should be for calling removeUnit()?
b. What *postconditions* should hold after it is called?
c. Does the scenario give rise to any system *invariants*?
d. Can you identify any problems with the code? Describe what defects, failures and erroneous states might exist as a consequence.

If you don't recall what preconditions, postconditions, and invariants are, you might wish to review the week 1 readings.

Sample solutions:

a. Preconditions

Preconditions for `removeUnit` to complete properly, and bring about the postconditions, might include:

- `unitCode` is not `null`
- `unitCode` represents a valid, existing unit code
- The receiver object for `removeUnit()` (i.e., the object reference it is being called on) is not `null`
- A valid database and database connection exist

However, note that we would not necessarily mention all of these in the Javadoc comment for `removeUnit()`:

- Although it may be a precondition that `unitCode` is not `null`, it is true for nearly all Java methods that their arguments must not be null; we are more likely to document the *opposite* case, where a `null` value *is* allowed.
- It is likely that "A valid database and database connection exist" are preconditions for many of the methods in the class we are considering. So we probably would mention this in the Javadoc comment for the class as a whole, to save repeating ourselves.
  (For example, take a look at the documentation for Java's `java.util.TreeMap` ↪ class. It says "This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations", rather than repeating that statement four times.)
- It is a property of the Java language that a method can only be successfully be called when the receiver object is not `null`, else a `NullPointerException` will be thrown. So this need not be mentioned.

Other preconditions we need not document:

- That `unitCode` is of type `String`. If `unitCode` were not of type `String`, the source code couldn't possibly have compiled, and we couldn't be running the program.
  (Or: if, somehow, `unitCode` referred to a spot in memory that did *not* contain a String object, this would be an indication that the Java Runtime Environment had somehow become corrupted.)
  It is a guarantee of the Java language that parameters always have the correct types.

  (In Python, the situation is different. We might require that `unitCode` supports particular string operations, since at runtime, the parameter passed need not be of type `str`.)

Alternative solutions:

- If we adopt the solution above, then that means that if `unitCode` does not refer to a valid, existing unit code, either this method or one of the methods we call should throw an exception. (We would document *that* in our Javadoc as well, so callers know what exceptions can be thrown.)
  But an alternative design is to simply do nothing when the unit code does not exist. In that case, we should *not* throw an exception.

⚠ If you are not clear on what preconditions are from the lecture slides, you might want to read chapter 3 of *Object-Oriented Design and Patterns* (2nd edn) by Cay S. Horstmann. (Password-protected; I'll provide logon details next lecture.)

b. Postconditions

The postcondition here could be stated as:

- "The unit with code `unitCode` does not exist in the database, and no records in the enrolment table exist that refer to it."

c. System invariants

Recall that invariants are assertions that should hold true before and after every method call of a class (or, if we are describing invariants for a whole subsystem or system, for all methods of classes in the subsystem or system).

From what we are told, we can infer that there is at least the following invariant (presumably, for the whole system):

- If an entry in the enrolments table refers to a unit code, then a corresponding entry in the units table must exist.

There might be others as well.

c. Problems with the code

We are not told what `units.removeRecord()` does, exactly.

However, if we make the following assumption:

- `removeRecord()` removes an entry from the units table, and does not alter any other tables

then there *is* a problem with the code: it should have updated the enrolments table, to remove any references to the deleted unit.

So a *defect* in the code is that it does not call whatever methods are necessary to delete records from the enrolments table. (This is a static property of the code.)

The system enters an *erroneous state* immediately after the `removeUnit` method call, because now one of the system invariants is not satisfied – the system is inconsistent.

As for *failures*: recall that these are ways in which the system observably departs from its specification. It is likely that no failure will occur *directly* after the `removeUnit` method call. But the next time someone queries the system to see what units a student was enrolled in: they may be recorded as being enrolled in a non-existent unit, which likely *is* a failure.

⚠ If you are not clear about the difference between *faults*, *failures*, and *erroneous states*, you might want to read chapter 11 of Bruegge and Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java* (3rd edn), particularly section 11.3, "Testing Concepts".

Alternative solution:

- On the other hand, if we make the assumption that `removeRecord` *does* correctly remove all enrolment records which mention the deleted unit, then there is no fault. (This is called "cascading a deletion", in database terminology.)
  But we have to make one assumption or the other, and say which one we are making and why.