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:

```
/** 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.