CITS5501 Software Testing and Quality Assurance
Semester 1, 2022
Week 3 workshop – Data-driven tests and test design – solutions

1. Parameterized tests

Normal test methods in JUnit don't take any parameters at all – for instance, the `testAdd` method from last week's code:

```
@Test
public void testAdd() {
    Calculator c = new Calculator(3, 4);
    int result = c.add();
    assertEquals(result, 7, "result should be 7");
}
```

But in last week's lab, we briefly saw an example of a *parameterized test* in JUnit – these *do* take parameters. The parameterized test was `addZeroHasNoEffect`, which checks, for a range of `int` values, that using the `Calculator` to add 0 to the number gives back the original `int`:

```
/** Adding zero to a number should always
  * give us the same number back.
  */
@ParameterizedTest
@ValueSource( ints = { -99, -1, 0, 1, 2, 101, 337  })
void addZeroHasNoEffect(int num) {
    Calculator c = new Calculator(num, 0);
    int result = c.add();
    assertEquals(result, num, "result should be same as num");
}
```

In effect, annotating the test with `@ParameterizedTest` says to JUnit "You need to call this test repeatedly, each time passing it an `int` from a list of `int`s I'm going to give you". And the `@ValueSource` annotation says "Here is the list of `int`s I mentioned".

So the test method, `addZeroHasNoEffect`, takes *one* parameter, an int, and each time the test method is called, it is passed a different int from the list given by `@ValueSource`.

The code for this week's lab is the same as last week's, but with some new test methods, so you can try out the `addZeroHasNoEffect` test if you haven't already:

Exercises

a. Run the tests in BlueJ or your IDE to confirm that all those ints are used – how can you tell?
b. Try changing them and/or adding to the list.
c. `addZeroHasNoEffect` is a single method. But (based on the material from lectures and the textbooks) is it also a single *test case*? If not, how many test cases does it comprise?

Sample solutions:

a. Most IDEs should show you exactly what tests cases have been run – check the documentation for your IDE if it is not clear.

c. Seven test cases, because there are seven `int`s in the list.

Hopefully the use of `@ValueSource` seems straightforward. A question may now arise: What if we have multiple parameters? Or what if we wish to specify not just the *test* values, but the *expected* values?

In this week's code, we have a new test method, `tableOfTests`. It is designed to get test values and expected values from some other source, and then run them as tests.

```
1  @ParameterizedTest
2  @MethodSource("additionTestCasesProvider")
3  void tableOfTests(int num1, int num2, int expectedResult) {
4      Calculator c = new Calculator(num1, num2);
5      int result = c.add();
6      assertEquals(expectedResult, result, "result should be same as as
   ↪  expected result");
7  }
```

The `@ValueSource` annotation is used when you have a straightforward list of literal values you want JUnit to iterate over. But in `tableOfTests`, we're using a new annotation, `@MethodSource`. When we annotate our test method with `@MethodSource("additionTestCasesProvider")` we are effectively saying to JUnit, "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 on writing parameterized tests.

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).

Exercises

a. How many *test cases* would you say `tableOfTests` and `additionTestCasesProvider` comprise?

b. Read through the JUnit documentation on writing parameterized tests.

c. In Java, `enum` classes are used to represent types that can take on values from only a distinct set. By convention, the values are given names in ALL CAPS.

For instance,

```
public enum Weekday {
  MON, TUE, WED, THU, FRI, SAT, SUN
}
```

or

```
public enum Color {
  RED, ORANGE, YELLOW, BLUE, GREEN, INDIGO, VIOLET
}
```

(For more on Java enums, including how to create enums with constructors and fields, see the Java documentation on enum types.)

Suppose we need to run a test which should be passed each `Weekday` in turn. Which JUnit annotation should we use for this?

d. If you have used testing frameworks in languages other than Java – how do they compare with JUnit? Do they offer facilities for creating data-driven tests? Are these more or less convenient than the way things are done with JUnit?

Sample solutions:

a. Together, they comprise four test cases: `tableOfTests` will get invoked four times, each time with different test values and expected result.

c. The JUnit documentation explains how the `@EnumSource` annotation can be used with enum types. By default, an `@EnumSource` will iterate over all the possible values of an enum type, calling the test method once with each value.

d. Writing data-driven tests tends to be more convenient (and less verbose) in languages which are not statically type-checked (for instance, JavaScript, Python and Ruby and Lisp-family languages) or which have rich type systems and *type*

inference (like Scala, Ocaml and Haskell).
But as we have seen, it is certainly possible in Java.

Extension exercises

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

2. Preconditions and postconditions

Work through and discuss the following scenario and exercises if there is sufficient time. If there is not, work through these in your own time.

Consider the following scenario:

Enrolment database

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);
}
```

If possible, it's recommended you discuss the following questions with a partner or in a small group, and come up with an answer for each. But if that is not feasible, spend several minutes thinking about the questions yourself before sharing ideas with the class.

Exercises

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`.)

Tip: If asked in assingments or the exam to specify preconditions for a method, we will nearly always be using the Java language. If you specify preconditions like "The parameter `String s` must be a `String`" you generally (a) will not get any marks for saying so, and (b) will make the markers think you don't understand how Java works.

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. (This can be found in the "Unit Readings" via the LMS.)

Sample solutions:

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.

d. 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.