CITS5501 Software Testing and Quality Assurance
Semester 1, 2022

Week 3 workshop – Data-driven tests and test design

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:

```java
@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`:

```java
/** 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 `ints` I'm going to give you". And the `@ValueSource` annotation says "Here is the list of `ints` 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?

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.

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

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 `MethodSources` 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,

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

or

```java
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?

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:

```java
/** 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.