CITS5501 Software Testing and Quality Assurance
Semester 1, 2022
Workshop 1 – Testing introduction

0. Accessing required software

We will be using the Java language for the bulk of the workshops, so you should make use of a Java IDE (Integrated Development Environment).

If you are using a university computer, you should be able to access the BlueJ IDE.

If you are using a laptop or home computer, you may install and use BlueJ, or you may install another Java IDE if you prefer.

Some freely available options are:

- Netbeans, downloadable from https://netbeans.org
- Eclipse IDE for Java Developers, downloadable from https://www.eclipse.org/downloads/packages/release/2020-12/r/eclipse-ide-java-developers
- IntelliJ IDEA Community, downloadable from https://www.jetbrains.com/idea/download/

As a first step for today's workshop, ensure you can install and/or access at least one of these.

We will give instructions for BlueJ version 5.0, but it should be straightforward to adapt these to other IDEs.

1. JUnit tests

Download and compile workshop code

Download the `workshop-01-code.zip` file, and unzip it somewhere on your computer.

Open the code as a "project" in your IDE. For BlueJ, this is done by selecting "Project" / "Open Non BlueJ", and selecting the directory containing the Workshop 1 code.

Ensure you can compile the project – in BlueJ, by selecting "Tools" / "Compile".

If using an IDE other than Java: you may need to instruct the IDE to add the "JUnit 5" libraries to the project; typically, viewing the project *properties* in your IDE will reveal some way of doing this.

Take a look at the `Calculator` class, in `Calculator.java` – this class has trivial functionality, but is useful as an example of a *class under test*.

Take a look at the `CalculatorSimpleTest` class, in `CalculatorSimpleTest.java`. This class defines a number of *JUnit tests* for our `Calculator` class.

Test classes can be called anything, but by convention, *unit tests* (which are written to test a single class) usually start with the same name as the *class under test*, followed by a description of the test (or just the word "Test").

Run the JUnit tests

Run the tests in the `CalculatorSimpleTest` class.

In BlueJ, this is done by right-clicking on the class (after compiling) and selecting "Test All".

You should see that some tests "pass" (with green ticks) and some "fail" (with red crosses) – see if you can work out what the failing `testSubstract` test is telling you about what the problem is.

Inspect the JUnit tests

Look at the parts of the `CalculatorSimpleTest` test class, using the JUnit User Guide (https://junit.org/junit5/docs/current/user-guide/) as a reference.

- Test classes can be called anything.
- Test cases are written in methods annotated `@Test`
- For each test, the methods annotated `@BeforeEach` and `@AfterEach` are run before the test and after the test, respectively.
  These methods can be used to create and destroy test *fixtures* – in Java, fixtures are normally a set of objects in a known state. (The state *can* include things outside the Java program, however – databases, files on a remote system, anything we like. But for unit tests, the fixtures will only be Java objects.)
- The `testSubtractThrowsException()` test is intended to discover whether the `Calculator.subtract()` method throws an *exception* in circumstances where it should.

  The code

```
47      Throwable exception = assertThrows(
48          ArithmeticException.class,
49          () -> c.subtract()
50      );
```

calls the `assertThrows` method, which is used to assert that when its second paramater (a bit of executable Java code, called a *lambda expression*) is run, it throws the exception specified by its first parameter.

We will look at these more later.

- Note that the first few test methods take no arguments, but the test `addZeroHasNoEffect` is what JUnit calls a parameterized test – unlike other test methods, it does take arguments.
  We will look at these more later; but JUnit's parameterized tests are designed to make it easy to run what are called *data-driven tests* (see Wikipedia on Data-driven testing), as well as a subset of data-driven testing called *property-based testing* (see the explanation given by the Hypothesis Python-based library for doing this sort of testing).

Consider the following question: if you want to get all the tests passing, how do you determine what each method is supposed to *do*, and when it is correct? (After all, someone writing the test could have made a mistake in the test code.)

2. API documentation

Look at the `Calculator.java` class from the workshop 1 code.

Can you identify

a. A Javadoc comment, which documents the API?
b. A Java comment which is *not* Javadoc?

Use your IDE to run the `javadoc` tool, which generates API documentation from source code.

- In BlueJ, select "Tools" / "Project Documentation", then look for a directory called "`doc`" which should be created within the source code directory.
- BlueJ should automatically open the generated documentation in your browser.

View the generated documentation in your browser.

Identify one class member marked `private`, and make it `public` and write a Javadoc comment for it. Re-run `javadoc` – what changes do you see in the generated documentation?

3. Fix the code

See if you can fix the code in the `Calculator` class so that all the tests pass.

For the `subtract` method – aside from other changes you might need to make, you might want code something like the following:

```
if (/* some condition goes here */) {
  throw new ArithmeticException("can't return a negative result");
}
```

Try creating your own new tests. In BlueJ, if you right click on a class, there should be an option to create a test class. Use the existing tests as an example – can you think of other tests we might add?

5. Concepts review questions

Answer the following questions to test your understanding of concepts introduced in the lectures and prescribed reading.

For each of the following scenarios, explain whether you think a *failure*, a *fault* or an *erroneous state* (or none of these, or more than one) has occurred, and explain why. If it is a failure – is it non-conformance with a functional or a non-functional requirement?

a. The social media site "Witter" allows users to specify that their email and date of birth should not be displayed publicly. But after a system update, that information is now visible for all users.

b. The ride-sharing app Habari runs on a user's mobile phone, and communicates with Habari's servers to find nearby divers and arrange a ride. However, the communications are not encrypted, meaning a tech-savvy user could manipulate the system and obtain free rides.

c. Your colleague Mila is writing a function which should return the arithmetic mean of numbers in a list:

```
double total = 0;
for (double num : number_list) { total += num ; }
return total / number_list.length
```

However, when the list is of length 0, this code return the result "`INFINITY`".