CITS5501 Software Testing and Quality Assurance
Semester 1, 2020

Workshop 1 – Testing introduction

Lab computer access – face-to-face students

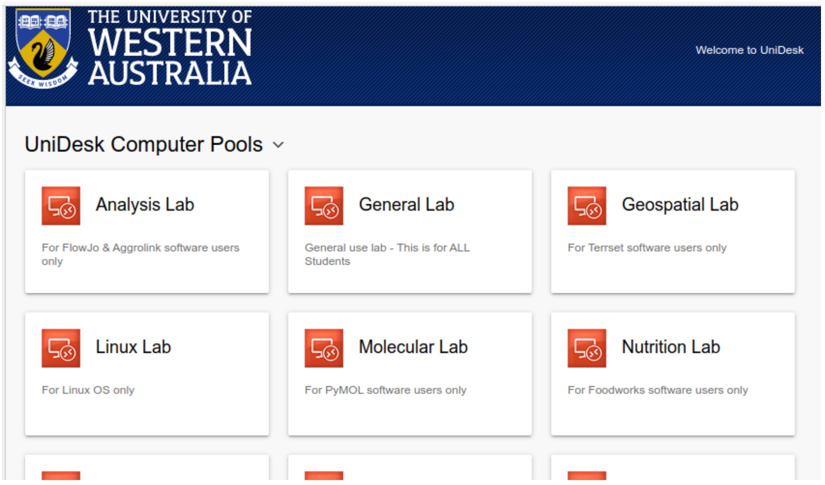
The computers in the labs we will use all have recent Java programming environments installed, which we will use for many of the labs. Some later labs will require different software, in which instructions for install it will be given at the time.

Lab computer access – online students

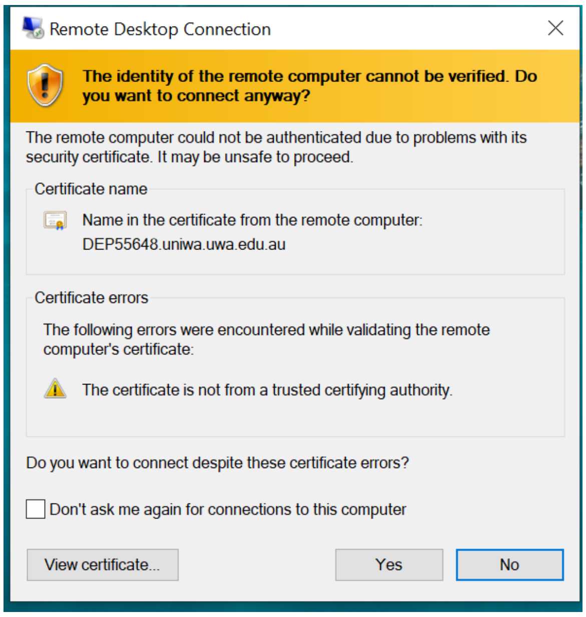
If you are attending a lab *online*, UWA IT's recommended way of accessing a lab computer is to use **Unidesk**.

- Visit the Unidesk site at https://unidesk.uwa.edu.au/

  You should see a web page like the one below:


THE UNIVERSITY OF WESTERN AUSTRALIA
Welcome to UniDesk
UniDesk Computer Pools
Analysis Lab
For FlowJo & Aggrolink software users only
General Lab
General use lab - This is for ALL Students
Geospatial Lab
For Terrset software users only
Linux Lab
For Linux OS only
Molecular Lab
For PyMOL software users only
Nutrition Lab
For Foodworks software users only

- Click on "General Lab".
- You may see a message asking if you want to download or open a file called "`launch9456a51b.rdp`" or similar – click "open".
- You may see an alert asking whether you want to connect (see below) – click "Yes."



Remote Desktop Connection

The identity of the remote computer cannot be verified. Do you want to connect anyway?

The remote computer could not be authenticated due to problems with its security certificate. It may be unsafe to proceed.

Certificate name

Name in the certificate from the remote computer:
DEP55648.uniwa.uwa.edu.au

Certificate errors

The following errors were encountered while validating the remote computer's certificate:

The certificate is not from a trusted certifying authority.

Do you want to connect despite these certificate errors?

Don't ask me again for connections to this computer

- If asked for a user ID and password to log on with, use your student ID (just the number – no `@student.uwa.edu.au` is required), and your normal Pheme password.
- You should then be connected to a UWA lab computer. In the event of any difficulties – it is UWA IT that supports the Unidesk system, so you should contact the UWA Service Desk by calling tel. +61 8 6488 1234 (or extension x1234 if on campus), between 8.00am and 5:00pm weekdays – see the Service Desk page for more details.

1. 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 use another Java IDE if you like.

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/

Ensure you can access at least one of these.

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

2. 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

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

  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.)

3. 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?

4. 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. If you have time available

Consider the following, each of which is supposed to be a system requirement. Discuss with a partner – do you think it would be straightforward to write tests for them? If not, why not?

a. The flight booking system should be easy for travel agents to use.
b. The `int String.indexOf(char ch)` method should return a `-1` if `ch` does not appear in the receiver string, or the index at which it appears, if it does.
c. Internet-aware Toast-O-Matic toasters should have a mean time between failure of 6 months.

Sample solutions:

Fixing errors:

The code for `subtract` has the numbers in the wrong order.

The code for `subtract` also needs to throw an exception when the result would be negative.

e.g.

```
if (num2 > num1) {
  throw new ArithmeticException("can't return a negative result");
}
```

System requirements:

(a) This would be difficult to test.

- "Easy to use" is not a very *precise* requirement. It is the opposite of precise – it is *vague* or *fuzzy*; it is difficult to pinpoint exactly which systems satisfy it and which don't.

  A better requirement might be something like:

  > "Travel agents shall be able to use all the system functions after successful completion of a training course designed by the software provider. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use."

  (This is adapted from *Pressman*.)

- A test like this requires human users, and would often not be done until the *acceptance testing* phase. Prior to that, the software provider might try to come up with a quicker and cheaper test to act as a *proxy* for the acceptance test – they might test it on non-technical staff in their own organisation, for instance.

⚠ If you are not clear about what makes a good requirement, you might want to review the chapter from the *Pressman* textbook on "Understanding Requirements" (in the 9th edition) or "Requirements Engineering" (in earlier editions).
There is also a quick summary (taken from documentation for an IBM requirements management product) available here.

(b) indexOf method

This seems straightforward to test, but leaves some behaviour *unspecified*.

It doesn't specify what happens if the character appears in the string multiple times – it says "the index at which it appears", implying there is only one. It would be better to specify "the *first* index at which it appears".

Once that is done, the method could still be made *more* precise – compare the actual Javadoc for the Java `String.indexOf` method. That documentation clarifies that `ch` represents a Unicode code point, and explains what happens when `ch` falls in various ranges.

Once corrected, the requirement is straightforward to write tests for.

(c) mean time between failure

For many purposes, this is probably precise enough (though one might want to add "under normal operating conditions", as opposed to "when operated in the open in a desert environment frequently subject to sandstorms").

As it stands, it is not easy to test, however, until after the toasters have been sold and are in normal operational use.

Again, the provider might make use of some sort of proxy test to assess the resilience of toasters. (Consider testing of car safety, for instance: do manufacturers "test" the safety of cars by simply selling them, and seeing what accidents occur? No – they do things like simulating wear and tear, and the effects of collisions.)