SFWRENG 3S03 Software Testing and Measurement, Winter Term 2021-22
Assignment #1 – due 11 February 2021 by 5pm Hamilton time (submit on Avenue)

This assignment contains three questions. Answer all questions. If you are working in a group of up to three people and submitting one assignment, please make sure **all** people are named clearly on the assignment so that marks are correctly allocated to everyone in the group!

1. [10 marks] Object-oriented programs in Python, Java etc, make use of class inheritance. Inheritance is extremely useful for programmers but has consequences for testing.

When we test software, it may be easy or difficult to detect faults in the code. The testability of code is a reflection on how hard it is for faults to evade detection – even if you have really good tests. Testability is largely governed by two properties: how easy it is to observe the behaviour of a program ("observability"), and how easy it is to provide a program with needed inputs ("controllability").

Give a maximum two paragraph explanation for how the use of an inheritance hierarchy in a program in your favourite language can affect controllability and observability.

2. [10 marks] The following JUnit test method for the `sort()` method has a non-syntactic flaw. Find the flaw and describe it precisely and concisely using the following terminology:

- *Reachability:* the test must reach the location/locations in the program that contain the fault.
- *Infection:* after the location is executed, the state of the program must be incorrect.
- *Propagation:* the infected state must cause some output or final state of the program to be incorrect.
- *Revealability:* the tester must observe part of the incorrect portion of the final program state

That is, explain the flaw in terms of reachability, infection, propagation and revealability.

In the test method, `names` is an instance of an object that stores strings and has methods `add()`, `sort()`, and `getFirst()`, which do exactly what you would expect from their names. You can assume that the object `names` has been properly instantiated and the `add()` and `getFirst()` methods have already been tested and work correctly.

```
    @Test
    public void testSort()
    {
       names.add ("Laura");
       names.add ("Han");
       names.add ("Alex");
       names.add ("Ashley");
       names.sort();
       assertTrue ("Sort method", names.getFirst().equals ("Alex"));

}
```

3. [20 marks] This exercise is to give you a simple introduction to Test-Driven Development, which will be covered in more detail a bit later in the term.

Consider the simple Java program Calc, listed below. Calc currently implements one function: it adds two integers. This is about as simple (or stupid!) a program you can write!

```
public class Calc
{
   static public int add (int a, int b)
   {
      return a + b;
   }
}
```

Use Test-Driven Development (TDD) to add additional functionality to subtract two integers, multiply two integers, and divide two integers. Do this as follows. First create a (failing) test for one of the new functionalities (the test fails because you HAVEN'T implemented the functionality yet), modify the class until the test passes, then perform any refactoring (improvements to the code, moving code around, deleting unnecessary code, etc) needed.

Repeat until all of the required functionality has been added to your new version of Calc, and all tests pass.

Remember that in TDD, the tests determine the requirements. This means you must encode decisions such as whether the division method returns an integer or a floating point number in automated tests before modifying the software.

Hand in your tests, your final version of Calc, and a screen shot showing that all tests pass. Most importantly, include a short narrative describing each TDD test created, the changes needed to make it pass, and any refactoring that was necessary.