Tutorial 2: Testing something slightly horrible

We warned you in the previous tutorial sheet that this one, and the next, will “give you some uglier code to work on”. They will. They’ll then ask you to:

1. Apply the basic techniques from last week on this uglier code
2. Expand the resulting test set in order to meet basic code coverage measures

Time budget for this tutorial: We’d recommended taking 2-4 hours in tutorials plus 4 hours of private study. If you’ve not made much progress by the end of that, consider spending some more – this is less important than the first tutorial, but still closely related to what you’ll need to demonstrate in the final exam.

Exercise 1 – Get the SUT running

New Challenge: get the Equations program to run

(This exercise is slightly out of order, but it’s critical that you have time to do this while you’re in class and we’re here to help you)

1. Download the “Equations” program from GitHub
2. Get it to run
    a. Unzip the program and launch a command prompt.
    b. In the command prompt navigate to the directory in which you unzipped the archive, then to the *bin* subdirectory.
    c. Launch the program using the following command line:

```
java -cp . uk.ac.york.modules.testing.EquationsView Help
```

    d. To actually render graphs, replace “Help” with one of the options shown in the help text

Exercise 2 — Define some requirements for the SUT

New Challenge: Turn a vague specification into one you can test

This testing will be a bit trickier than that in the exercise last time, partly because the program mixes GUI and command line features, but largely because you’re not going to be given a good clear specification. You will have to work out what correct behaviour should be as part of the testing process. (The reference book by Cem Kaner has some advice on this.)

Imagine that the specification you've been given is this:

```
We want to make a program that plots different types of
equations on a graph and can be easily extended to support
more.

Equations to support:

First order      y=ax+b

Second order     y=ax²+bx+c

Sinus            y=a sin(bxᶜ)+d

Fraction         y=a/(x+b)
```

This is all you've got. No doubt the client has lots of other things they want that they've not expressed (and perhaps have never expressed in words), and even if the client is easily pleased, the eventual end users (high school maths teachers? Client hasn't told us that either!) will certainly have some views on what such a system should do.

So, in preparation for the next tutorial, it would be a good idea to take the boxed text above and write down a list of requirements that are precise enough to be tested.

Exercise 3 — Test if the SUT meets your requirements

New Challenge: Specify a test suite that covers a set of requirements, and carry it out

Now that you have some reasonably precise requirements, you can create tests that check them.

1. Create a written list of tests that between them check all the requirements you have listed.
2. Where appropriate, create automated JUnit tests
3. Execute your tests (by hand or automatically, as they require) and note the results.
4. Assess each failed test. Does it indicate a fault? Or is there a problem with your requirements, or with your test?

Exercise 4 — Expand your tests to achieve a measure of code coverage

New Challenge: Specify an automated test suite that achieves 100% code coverage for a basic coverage measure

Code coverage measures are dubious when used on your own, but when combined with other criteria (here, requirements-based testing) they can be valuable in increasing the power of your tests.

1. Get EclEmma working for your tests on Equations
2. Observe which branches are not being covered by your tests

3. Create additional tests to target those branches until you achieve 100% branch coverage
4. Double-check that all your tests have value in themselves i.e. they check some program behaviour worth checking and are not just there for the coverage score

Note that you may well run into the weaknesses in EclEmma noted in the Tutorial 1 exercise. If you do, you will have to manually check the missed coverage points.

Exercise 5 — Control flow graphs

Draw the control flow graphs of the following programs.

```
public static double computeHarmonicSum(int N){
  double sum = 0.0;
  for (int i = 1; i <= N; i++) {
    sum += 1.0 / i;
  }
  return sum;
}
```

```
public static String taskScheduler(List<Task> taskList, int timeLimit) {
  if (taskList == null || taskList.isEmpty() || timeLimit <= 0) {
    System.out.println( "No tasks scheduled");
    return false;
  }

  int totalTime = 0;
  List<String> scheduledTasks = new ArrayList<>();

  for(Task task : taskList) { // B
    String taskName = task.getName();
    int duration = task.getDuration();

    if(totalTime + duration > timeLimit) { \\C
      break;
    }

    if(duration > timeLimit / 2) { \\D
      continue;
    }

    scheduledTasks.add(taskName);
    totalTime += duration;
  }

  if(scheduledTasks.isEmpty()) { \\ E
    System.out.println("No tasks fit within the time limit");
    return false;
    else{
      System.out.println(String.format("Scheduled %d tasks.", scheduledTasks.getSize());
      return true;
  }
}
```