ECE453/CS447/ECE653/CS647/SE465
Software Testing, Quality Assurance, and Maintenance
Assignment/Lab 1 (70 Points), Version 1

Instructors: Patrick Lam & Lin Tan
Release Date: January 12, 2015

Due: 11:59 PM, Monday, February 2, 2015
Submit: via ecgit

After setting up your ssh key at `http://ecgit.uwaterloo.ca`,
fork the provided git repository at `git@ecgit.uwaterloo.ca:se465/1151/a1`:

```
ssh git@ecgit.uwaterloo.ca fork se465/1151/a1 se465/1151/USERNAME/a1
```

and then clone the provided files. (You can also download the provided files at `http://patricklam.ca/stqam/files/provided_a01.tar.gz`, but don't do that if you're in the course—it'll make submitting harder.)

We expect each of you to do the assignment independently. We will follow UW's Policy 71 for all cases of plagiarism.

Submission Instructions:

Please read the following instructions carefully. **If you do not follow the instructions, you may be penalized up to 5 points.** Illegible answers receive no points.

Electronic submission: Please push the following files to your ecgit fork:

- a single pdf file "a1_sub.pdf" including all written answers. The first page must include your full name, 8-digit student number, your course number (one of ECE453, CS447, ECE653, CS647, & SE465) and section (for SE465), and your uwaterloo email address. The provided LaTeX sample meets these requirements (if you add your own information, of course).
- a directory "q1" that contains your code (source code only; no binaries or object files) for Question 1,
- "q3/TestM.java" for Question 3, and
- "q4/distribute/track/tests.py", along with your fix, for Question 4.

It's git, so you can submit multiple times. After submission, it is in your best interest to **re-clone your submissions to make sure you have uploaded (git added) all necessary files**.

Question	TA in Charge
1	Edmund, Thibaud
2	Taiyue, Abdel
3	Jeff, Jinqiu
4	Mike, Song

Question 1 (10 points)

Write one simple program that prints out the value of -5 modulo 2 (e.g., -5%2 in C) in each of the following 4 programming languages: C/C++, Java, Perl, and Python. You will write 4 programs in total. Report what you have found and the reason for this discrepancy. Discuss at least 2 different strategies to cope with such a problem. Hint: think about what can change: developers, compilers, standards, etc.

Question 2 (10 points)

Below is a faulty *Java* program, which includes a test case that results in a failure. The if-statement needs to take account of negative values. A possible fix is:

```
if (x[i]%2 != 0)
```

Answer the following questions for this program.

(a) If possible, identify a test case that does not execute the fault.

(b) If possible, identify a test case that executes the fault, but does not result in an error state.

(c) If possible, identify a test case that results in an error, but not a failure.

(d) For the given test case, identify the first error state. Describe the complete state.

```
public static int odd(int[] x)  {
// Effects: if x==null throw NullPointerException,
// else return the number of elements in x that are odd
        int count = 0;
        for (int i =0; i < x.length; i++) {
                if (x[i]%2==1) {
                        count++;
                }
        }
        return count;
}
// test: x = [-10, -9, 0, 99, 100]
// Expected: 2
```

Question 3 (20 points)

Consider the following (contrived) program:

```
class M {
        public static void main(String [] argv){
                M obj = new M();
                if (argv.length > 0)
                        obj.m(argv[0], argv.length);
        }

        public void m(String arg, int i) {
                int q = 1;
                A o = null;
                Impossible nothing = new Impossible();
                if (i == 0)
                        q = 4;
                q++;
                switch (arg.length()) {
                        case 0: q /= 2; break;
                        case 1: o = new A(); new B(); q = 25; break;
                        case 2: o = new A(); q = q * 100; // no break
                        default: o = new B(); break;
                }
                if (arg.length() > 0) {
                        o.m();
                } else {
                        System.out.println("zero");
```

```
                }
                nothing.happened();
        }
}

class A {
        public void m() {
                System.out.println("a");
        }
}

class B extends A {
        public void m() {
                System.out.println("b");
        }
}

class Impossible{
        public void happened() {
                // "2b||!2b?", whatever the answer nothing happens here
        }
}
```

(a) Using the minimal number of nodes (hint: 11), draw a Control Flow Graph (CFG) for method `M.m()` and include it in your a1_sub.pdf. The CFG should be at the level of basic blocks. See the lecture notes on *Structural Coverage* and *CFG* for examples.

(b) List the sets of Test Requirements (TRs) with respect to the CFG you drew in part (a) for each of the following coverages: node coverage; edge coverage; edge-pair coverage; and prime path coverage. In other words, write four sets: TR_{NC}, TR_{EC}, TR_{EPC}, and TR_{PPC}. If there are infeasible test requirements, list them separately and explain why they are infeasible.

(c) Using `q3/TestM-skeleton.java` as a starting point, write one JUnit Test Class that achieves, for method `M.m()`, each of the following coverages: (1) node coverage but not edge coverage; (2) edge coverage but not edge-pair coverage; (3) edge-pair coverage but not prime path coverage; and (4) prime path coverage. In other words, you will write four test sets (groups of JUnit test functions) in total. One test set satisfies (1), one satisfies (2), one satisfies (3), and the last satisfies (4), if possible. If it is not possible to write a test set to satisfy (1), (2), (3), or (4), explain why. For each test written, provide a simple documentation in the form of a few comment lines above the test function, listing which TRs are satisfied by that test. Consider feasible test requirements only for this part.

Our ECE Linux systems (ecelinux.uwaterloo.ca) have JUnit installed (in /opt/). If you have any questions regarding access to these machines, please contact our lab instructor Bernie. JUnit is included in standard Java code development environments such as Eclipse.

The discussion about JUnit 3 and JUnit 4 may be useful for Q3:
`http://www.ibm.com/developerworks/java/library/j-junit4.html`

Question 4 (30 points)

This question involves writing tests for the `distribute` app, which you will also find in your virtual machine, underneath the `/home/vagrant/distribute` directory. It is also live-linked on your host system (e.g. your laptop) in your checkout of your git repository, under the `q4/distribute` directory.

Part 1 (5 points). Write down (in English text) a set of test requirements describing the aspects of the system that you intend to test. This goes into `a1_sub.pdf`. Reasonable subcomponents to test are the two calculations (although you can also describe some other breakdown of the system). I'm looking for some overall integration-level system tests along with unit tests for the subcomponents.

Part 2 (20 points). Implement a test suite that covers test requirements for the integration tests and one component. Discuss your coverage results. You can either use integration-like tests that exercise the `get()` code paths as in

the `isin` example, or unit tests that directly call the view methods being tested. It is OK to use string comparisons for comparing the state of model objects. Submit the resulting `tests.py` file as part of your assignment submission.

Part 3 (5 points). There are at least four bugs in `distribute`. Find one bug in the system. Don't use invalid inputs to trigger the bug (e.g. well-formed inputs only). Point out which case from part 2 triggers the bug. (Yes, you have to have a test case for the bug.) Show the output of running your test cases before and after the fix, in `a1_sub.pdf`. Fix the bug, commit it to your repository, and explain the fix.

Does the triggering test case improve statement coverage, as reported by the coverage tool? That is, would you have the same coverage without the triggering test case? Discuss.

Q4 Grading scheme

We will allocate 50% of the marks in parts 1 and 2 for covering all of the obvious happy-case functionality (as in the "isin" example), and an additional 25% for one non-obvious test case per component (e.g. explore some weird behaviour). Integration tests will count for 60% of the marks while the unit tests of a component will count for 40% of the marks.

To get full marks for those parts: your tests must succeed; you must cover the test requirements or have a reasonable explanation about uncovered requrements; your tests must have sensible (and non-shadowing!) names; and you must have a discussion of your coverage results.

For part 3, we will give full points for a correct answer, with generous definitions of "a bug".