

# Software Testing, Quality Assurance & Maintenance (ECE453/CS447/SE465): Assignment 1

Patrick Lam

Due: January 18, 2010

You may discuss the assignment with others, but I expect each of you to do the assignment independently. I will follow UW's Policy 71 if I discover any cases of plagiarism.

## Question 1 (20 points)

Learn about quality assurance in some other domain of engineering. Write a short essay (1 page) comparing software quality assurance with the domain of your choice. Include about 2 references<sup>1</sup>. CS647 students: I expect at least one of the references to be more specialized than a textbook.

## Question 2 (10 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;
```

---

<sup>1</sup>Wikipedia is not an acceptable reference.

```

A o = null;

if (i == 0)
    q = 4;
q++;
switch (arg.length()) {
    case 0: q /= 2; break;
    case 1: o = new A(); q = 10; break;
    case 2: o = new B(); q = 5; break;
    default: o = new B(); break;
}
if (arg.length() > 0) {
    o.m();
} else {
    System.out.println("zero");
}
}
}

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

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

```

Write JUnit test sets (i.e. a JUnit test class) which achieves, for method `M.m()`, (1) node coverage; (2) edge coverage; and (3) edge-pair coverage.

## Question 3 (20 points)

(Original version by Sarfraz Khurshid, with modifications by Patrick Lam.)

You are to construct a partial<sup>2</sup> control-flow graph from the bytecode<sup>3</sup> of a given Java class using the ASM framework, version 3.2<sup>4</sup>.

---

<sup>2</sup>This assignment ignores the labels that are traditionally annotated on nodes and edges, as well as the edges that correspond to method invocations or `jsr [ w]` bytecodes.

<sup>3</sup><http://java.sun.com/docs/books/jvms/secondedition/html/VMSpecTOC.doc.html>

<sup>4</sup><http://asm.ow2.org>

To illustrate, consider the following class C:

```
public class C {
    int max(int x, int y) {
        if (x < y) {
            return y;
        } else return x;
    }
}
```

which can be represented in bytecode as (e.g. the output of `javap -c`):

```
Compiled from "C.java"
public class ee379k.pset1.C extends java.lang.Object{
    public C();
        Code:
            0:   aload_0
            1:   invokespecial   #8; //Method java/lang/Object.<init>:()V
            4:   return

    int max(int , int);
        Code:
            0:   iload_1
            1:   iload_2
            2:   if_icmpge      7
            5:   iload_2
            6:   ireturn
            7:   iload_1
            8:   ireturn
}
```

You can easily draw the corresponding control-flow graph.

**Graph representation of control-flow.** Implement the class `CFG` to model control-flow in a Java bytecode program. A `CFG` object has a set of nodes that represent bytecode statements and a set of edges that represent the flow of control (branches) among statements. Each node contains:

- an integer that represents the position (bytecode line number) of the statement in the method.
- a reference to the method (an object of `org.objectweb.asm.tree.MethodNode`) containing the bytecode statement; and
- a reference to the class (an object of class `org.objectweb.asm.tree.ClassNode`) that defines the method.

Represent the set of nodes using a `java.util.HashSet` object, and the set of edges using a `java.util.HashMap` object, which maps a node to the set of its neighbors. Ensure the sets of nodes and edges have values that are

consistent, i.e., for any edge, say from node  $a$  to node  $b$ , both  $a$  and  $b$  are in the set of nodes. Moreover, ensure that for any node, say,  $n$ , the map maps  $n$  to a non-null set, which is empty if the node has no neighbours.

You can find a partial implementation of the CFG class at

[http://patricklam.ca/stqam/files/CFG\\_partial.java](http://patricklam.ca/stqam/files/CFG_partial.java);

you'll need to rename it to `CFG.java` to continue. Your first task is to fill in the implementation of this class.

**Adding a node (2 points).** Implement the method `CFG.addNode` such that it creates a new node with the given values and adds it to nodes as well as initializes edges to map the node to an empty set. If the graph already contains a node that is `.equals` to the new node, `addNode` does not modify the graph.

**Adding an edge (2 points).** Implement the method `CFG.addEdge` such that it adds an edge from the node  $(p1, m1, c1)$  to the node  $(p2, m2, c2)$ . Your implementation should update nodes to maintain its consistency with edges as needed.

**Reachability (4 points).** Implement the method `CFG.isReachable` such that it traverses the control-flow graph starting at the node represented by  $(p1, m1, c1)$  and ending at the node represented by  $(p2, m2, c2)$  to determine if there exists any path from the given start node to the given end node. If the start node or the end node are not in the graph, the method returns false.

**Testing (12 points).** I've also provided a class `CFGTest`, at

<http://patricklam.ca/stqam/files/CFGTest.java>

which exercises your `CFG` class. Examine this test class. Which coverage criteria does it satisfy? (Justify your answer.) If it doesn't satisfy any interesting coverage criteria, write additional JUnit test cases which make it satisfy some coverage criterion.