Software Testing, Quality Assurance & Maintenance (ECE453/CS447/SE465): Assignment 2 Solutions

[A TA] and Bob Zhang, edited by Patrick Lam

Question 1: TestGenerator (25 points)

[A TA] marked this question. He used the following marking scheme:

- Working program: 13 points
- Code readability: 2 points
- (Good) comments : 5 points
- Quality (Modularity, Algorithm, etc.) : 5 points

He also writes:

> The solution of Samuel Grossman is a very good example of good software coding practice.
>
> I have also noticed that students are not aware that writing good comments, having a modular and well structured code is part of the Software Maintenance process.

I have reproduced Samuel's solution below with his permission.

```java
/*
 * JUnit Test Case Generator
 * ECE 453 Assignment 2 Question 1
 * Samuel Grossman
 */

import java.util.List;
import java.util.Vector;
import org.objectweb.asm.*;
import org.objectweb.asm.tree.*;

public class TestGenerator {
  public String createTests(String className) throws ClassNotFoundException {
    // The ASM-related startup code was borrowed from the supplied code for
    // Assignment 1's CFGTest implementation.
    ClassReader cr;
        try {
          cr = new ClassReader(className);
        } catch (java.io.IOException e) {
          throw new ClassNotFoundException("Class \"" + className + "\" is undefined.");
        }

    final ClassNode cn = new ClassNode();
        cr.accept(cn, ClassReader.SKIP_DEBUG);

    // This string will list ALL the generated test cases for the given class.
    StringBuffer outputString = new StringBuffer();

    // Loop through all the static methods in the class.
    for (final MethodNode m : (List<MethodNode>)cn.methods) {
      if ((m.access & Opcodes.ACC_STATIC) != 0) {
        Type[] argumentTypes = Type.getArgumentTypes(m.desc);

        // Now we are ready to start the recursion.
        // For each parameter, recursively generate parameter lists
        // over the entire domain of that type.
        // If we find a type we do not recognize,
        // bail by throwing an exception.
        outputString.append(createTestsInternal(m.name, argumentTypes));
      }
    }

    return outputString.toString();
  }

  private StringBuffer createTestsInternal(String methodName, Type[] argumentTypes) {
    // These are the template strings that will serve as the base
    // for each test case generated.
    // It will be basicStringStart + # + basicStringMiddle +
    //       params list + basicStringEnd.
    String basicStringStart = "@Test public void test_" + methodName + "_";
    String basicStringMiddle = "() { " + methodName + "(";
    String basicStringEnd = "); }\n";

    // This string will list ALL the test cases generated for the given method.
    StringBuffer outputString = new StringBuffer();

    // This list is used by all recursive calls to allow them to build upon one another.
    Vector buildParams = new Vector<String>();
    buildParams.setSize(argumentTypes.length);

    // This will help keep track of the number of test cases generated.
    // It is used to name each test case.
    int[] numCases = new int[]{ 0 };
```

```java
    // Now the recursion has been initialized.
    // Call the recursive method to start.
    createTestsRecursive(basicStringStart, basicStringMiddle,
      basicStringEnd, outputString, buildParams, 0, argumentTypes, numCases);

    return outputString;
  }

  private void createTestsRecursive(String start, String mid, String end,
    StringBuffer out, Vector<String> build, int index, Type[] args, int[] numCases)
      throws RuntimeException {
    // First determine the parameter type.
    // Then iterate over the domain for that type.
    switch (args[index].getSort()) {
      case Type.INT:
        for (int i = 0; i < Domain.INT.length; ++i) {
          // For each integer in the domain, add it as a possible parameter.
          build.setElementAt(Integer.toString(Domain.INT[i]), index);

          // Recursion ends when there are no more parameters to check.
          if ((index + 1) == args.length)
            createTestsRecursiveDone(start, mid, end, out, build, numCases);
          else
            createTestsRecursive(start, mid, end, out, build, (index + 1), args, numCases);
        }
        break;

      case Type.BOOLEAN:
        for (int i = 0; i < Domain.BOOLEAN.length; ++i) {
          // For each boolean in the domain, add it as a possible parameter.
          build.setElementAt(Boolean.toString(Domain.BOOLEAN[i]), index);

          // Recursion ends when there are no more parameters to check.
          if ((index + 1) == args.length)
            createTestsRecursiveDone(start, mid, end, out, build, numCases);
          else
            createTestsRecursive(start, mid, end, out, build, (index + 1), args, numCases);
        }
        break;

      case Type.OBJECT:
        if (args[index].getClassName().equals("java.lang.String")) {
          for (int i = 0; i < Domain.STRING.length; ++i) {
            // For each String in the domain, add it as a possible parameter.
            if (Domain.STRING[i] == null) {
              build.setElementAt("null", index);
            } else {
              build.setElementAt("\"" + Domain.STRING[i] + "\"", index);
            }

            // Recursion ends when there are no more parameters to check.
            if ((index + 1) == args.length)
              createTestsRecursiveDone(start, mid, end, out, build, numCases);
            else
              createTestsRecursive(start, mid, end, out, build, (index + 1), args, numCases);
          }
        } else {
          for (int i = 0; i < Domain.OBJECT.length; ++i) {
            // For each Object in the domain, add it as a possible parameter.
            build.setElementAt(Domain.OBJECT[i], index);

            // Recursion ends when there are no more parameters to check.
            if ((index + 1) == args.length)
              createTestsRecursiveDone(start, mid, end, out, build, numCases);
            else
```

```java
            createTestsRecursive(start, mid, end, out, build, (index + 1), args, numCases);
          }
        }
        break;

      default:
        throw new RuntimeException("Method contains an unrecognized type.");
    }
  }

  private void createTestsRecursiveDone(String start, String mid, String end,
      StringBuffer out, Vector<String> build, int[] numCases) {
    // Build the test case string from the basic strings, the number, and the parameters list.
    StringBuffer testCaseString = new StringBuffer();
    testCaseString.append(start);
    testCaseString.append(++numCases[0]);
    testCaseString.append(mid);

    for (int i = 0; i < (build.size() - 1); ++i) {
      testCaseString.append(build.get(i));
      testCaseString.append(", ");
    }
    testCaseString.append(build.get(build.size() - 1));
    testCaseString.append(end);

    out.append(testCaseString);
  }

  public static void main(String[] argv) {
    if (argv.length != 1) {
      System.out.println("Usage: TestGenerator [className]");
    } else {
      try {
        TestGenerator t = new TestGenerator();
        String output = t.createTests(argv[0]);
        System.out.println("Class \"" + argv[0] + "\" generated these tests:\n");
        if (output.isEmpty()) {
          System.out.println("[no tests generated]");
        } else {
          System.out.println(output);
        }
      } catch (Exception e) {
        String message = e.getMessage();
        if (message != null) {
          System.out.println("Error: " + e.getMessage());
        } else {
          System.out.println("An unknown error has occurred.");
        }
      }
    }
  }
}

// For testing only; can pass this as a command-line argument.
class Demo {
  // Expect 3 test cases
  static void m(int x) {
    return;
  }

  // Expect 8 test cases
  static void n(boolean b, String s) {
    return;
  }
```

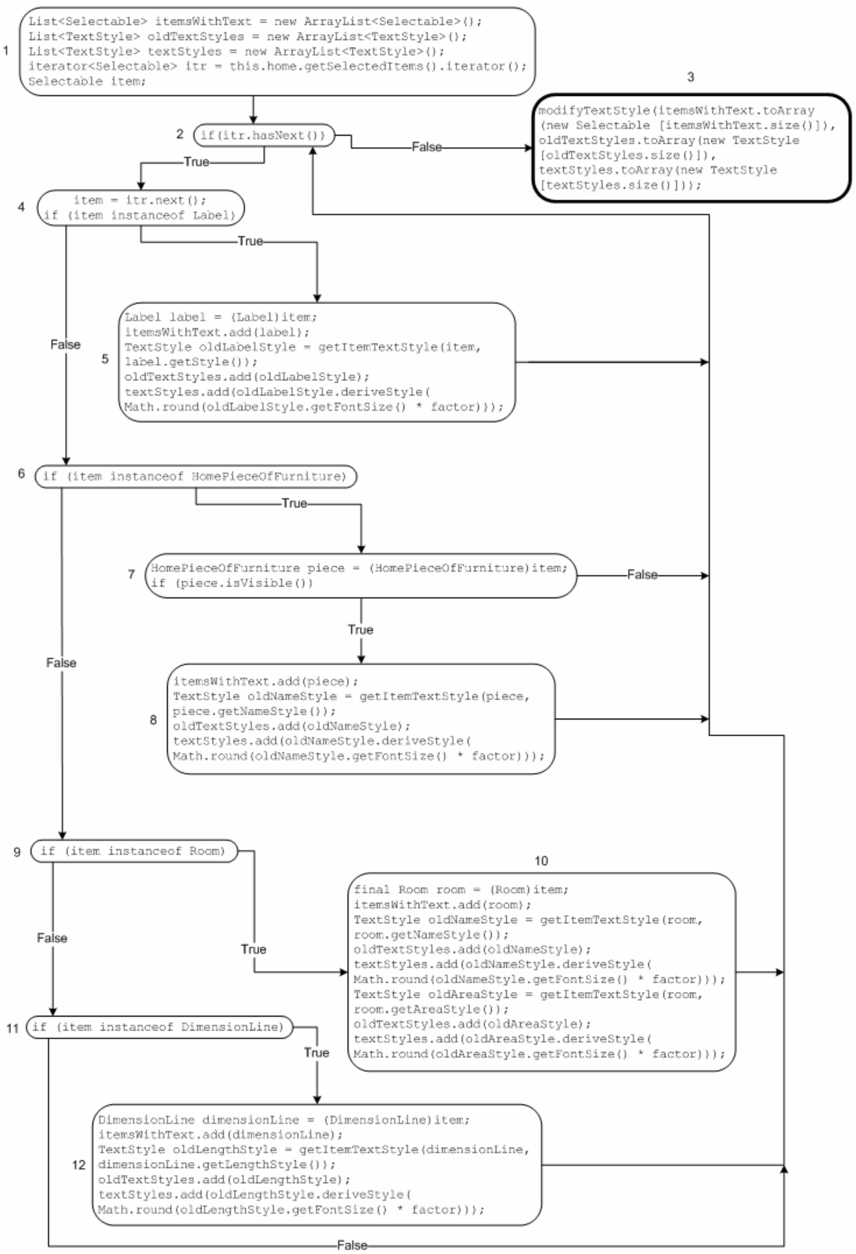
```
  // Expect 2 test cases
  static float p(Object o) {
    if (o == null)
      return 0.0f;
    else
      return 1.0f;
  }

  // Expect NO test cases (since this is non-static)
  private void bad(boolean b) {
    return;
  }
}
```

Question 2: PPC (50 points)

Solutions due to Bob Zhang.

Creating a CFG (10 points).



Identifying requirements for PPC (5 points). List all paths that start at the start node (1):

```
1. 1, 2, 3
2. 1, 2, 4, 5
3. 1, 2, 4, 6, 7, 8
4. 1, 2, 4, 6, 9, 10
5. 1, 2, 4, 6, 9, 11, 12
```

List all paths that involve full cycles.

```
6. 2, 4, 5, 2 and all related cycles
7. 2, 4, 6, 7, 2 and all related cycles
8. 2, 4, 6, 7, 8, 2 and all related cycles
9. 2, 4, 6, 9, 10, 2 and all related cycles
10. 2, 4, 6, 9, 11, 2 and all related cycles
11. 2, 4, 6, 9, 11, 12, 2 and all related cycles
```

List all paths that end at the end node (3):

```
12. 4, 5, 2, 3
13. 4, 6, 7, 2, 3
14. 4, 6, 7, 8, 2, 3
15. 4, 6, 9, 10, 2, 3
16. 4, 6, 9, 11, 2, 3
17. 4, 6, 9, 11, 12, 2, 3
```

List all paths that transition from one branch inside a loop in one iteration to another branch inside the loop in another iteration

```
18. 5, 2, 4, 6, 7, 8
19. 5, 2, 4, 6, 9, 10
20. 5, 2, 4, 6, 9, 11, 12
21. 6, 7, 2, 4, 5
22. 6, 7, 8, 2, 4, 5
23. 6, 9, 10, 2, 4, 5
24. 6, 9, 11, 2, 4, 5
25. 6, 9, 11, 12, 2, 4, 5
26. 7, 2, 4, 6, 9, 10
27. 7, 2, 4, 6, 9, 11, 12
28. 7, 8, 2, 4, 6, 9, 10
29. 7, 8, 2, 4, 6, 9, 11, 12
30. 9, 10, 2, 4, 6, 7, 8
31. 9, 11, 2, 4, 6, 7, 8
32. 9, 11, 12, 2, 4, 6, 7, 8
33. 10, 2, 4, 6, 9, 11, 12
34. 11, 2, 4, 6, 9, 10
35. 11, 12, 4, 6, 9, 10
```

Identifying requirements for ADC, AUC, ADUPC (10 points). The defs and uses for each variable are:

```
itemsWithText: defs = {1} uses = {3,5,8,10,12}
oldTextStyles: defs = {1} uses = {3,5,8,10,12}
textStyles: defs = {1} uses = {3,5,8,10,12}
itr: defs = {1} uses = {2,4}
item: defs = {4} uses = {4,5,6,7,9,10,11,12}
label: defs = {5} uses = {5}
piece: defs = {7} uses = {7,8}
room: defs = {10} uses = {10}
dimensionLine: defs = {12} uses = {12}
oldNameStyle: defs = {8,10} uses = {8,10}
oldAreaStyle: defs = {10} uses = {10}
oldLabelStyle: defs = {5} uses = {5}
oldLengthStyle: defs = {12} uses = {12}
factor: defs = {1} uses = {5,8,10,12}
```

To satisfy ADC the following paths are required for each variable:

```
itemsWithText, oldTextStyles, textStyles: [1,2,3]
itr: [1,2]
item: [4]
label, oldLabelStyle: [5]
piece: [7]
room: [10]
dimensionLine, oldLengthStyle: [12]
oldAreaStyle: [10]
oldNameStyle: {[8],[10]}
factor: [1,2,4,5]
```

To satisfy AUC the following paths are required for each variable:

```
itemsWithText, oldTextStyles, textStyles:
 {[1,2,3], [1,2,4,5], [1,2,4,6,7,8], [1,2,4,6,9,10],
  [1,2,4,6,9,11,12]}
itr: {[1,2],[1,2,4]}
item:
 {[4], [4,5], [4,6], [4,6,7], [4,6,9], [4,6,9,10],
  [4,6,9,11], [4,6,9,11,12]}
label, oldLabelStyle: [5]
piece: {[7], [7,8]}
room: [10]
dimensionLine: [12]
oldAreaStyle: [10]
oldNameStyle: {[8], [10]}
factor: {[1,2,4,5], [1,2,4,6,7,8], [1,2,4,6,9,10], [1,2,4,6,9,11,12]}
```

All paths satisfying AUC would also satisfy ADUPC.

Comparing coverage criteria (5 points). In this question we are looking for your understanding of the different test criteria. Simply saying PPC subsumes ADUPC, AUC and ADC is not enough. For example one could state PPC covers the whole program structure but cannot identify problems with the underlying data (variables). ADC, AUC, and ADUPC on the other hand are designed to cover data. They focus on def and use of variables and reveal more detailed information. Any reasonable answer on those lines got full marks. Many of you forgot to discuss whether the additional number of test requirements is worth it.

Creating tests (20 points). The following test cases achieve prime path coverage.

```
1. Empty
2. label label
3. room room
4. visible piece visible piece
5. dim dim
6. wall wall
7. invisible line invisible line
8. label room
9. label visible piece
10. label dim
11. label wall
12. label invisible piece
13. room label
14. room visible piece
15. room dim
16. room wall
```

17. room invisible piece
18. visible piece label
19. visible piece room
20. visible piece dim
21. visible piece wall
22. visible piece invisible piece
23. dim label
24. dim room
25. dim visible piece
26. dim wall
27. dim invisible piece
28. wall label
29. wall room
30. wall visible piece
31. wall dim
32. wall invisible piece
33. invisible piece label
34. invisible piece room
35. invisible piece visible piece
36. invisible piece dim
37. invisible piece wall