Software Testing, Quality Assurance & Maintenance (ECE453/CS447/CS647/SE465): Final

April 20, 2010

This open-book final has 7 questions worth 100 points. Answer the questions in your answer book. You may consult any printed material (books, notes, etc).

Question 1: Short Answer (10 points)

Fill in the blank (in your answer book). You shouldn't need more than about 15 words per answer. There may be more than one right answer.

1. Regression test suites should be __________.
2. If a test suite satisfies a set of test requirements, there must be at least as many test cases in the test suite as there are test requirements. (True/False)
3. Quality assurance for web applications developed with short release cycles should focus on ____________________.
4. Usage models enable inferences about ____________________.
5. Alloy's transitive closure support is especially useful for checking implementations of _________________________.
6. The main difficulty in applying software model-checking tools arises from the fact that _________________________.
7. A potential problem with reporting more than one bug in a bug report is: _________________________.
8. The most conspicuous part of a bug report is the ________________.
9. Test plans can help with ____________________.
10. If coverage criterion A subsumes criterion B, then any test suite satisfying A is always better than a test suite satisfying B. (True/False)

Question 2: Graph Coverage (20 points)

Consider the following code from `weka`'s `LogitBoost` class. We'll use it for questions on both graph and logic coverage.

```java
/**
 * Select only instances with weights that contribute to
 * the specified quantile of the weight distribution
 *
 * @param data the input instances
 * @param quantile the specified quantile eg 0.9 to select
 * 90% of the weight mass
 * @return the selected instances
 */
protected Instances selectWeightQuantile(Instances data, double quantile) {
  int numInstances = data.numInstances();
  Instances trainData = new Instances(data, numInstances);
  double [] weights = new double [numInstances];

  double sumOfWeights = 0;
  for (int i = 0; i < numInstances; i++) {
    weights[i] = data.instance(i).weight();
    sumOfWeights += weights[i];
  }
  double weightMassToSelect = sumOfWeights * quantile;
  int [] sortedIndices = Utils.sort(weights);

  // Select the instances
  sumOfWeights = 0;
  for (int i = numInstances-1; i >= 0; i--) {
    Instance instance = (Instance)data.instance(sortedIndices[i]).copy();
    trainData.add(instance);
    sumOfWeights += weights[sortedIndices[i]];
    if ((sumOfWeights > weightMassToSelect) &&
        (weights[sortedIndices[i]] != weights[sortedIndices[i-1]])) {
      break;
    }
  } // ...
  return trainData;
}
```

Draw a control-flow graph for this method. List defs and uses for each CFG node. Enumerate the test requirements for all-defs coverage (ADC). Choose any three test requirements and describe how you would satisfy them. (Assume that `Instances` has a no-args constructor which creates an empty set, and that `Instance` has a constructor which sets the `weight()`.)

Question 3: Logic Coverage (18 points)

Enumerate the predicates in `selectWeightQuantile`, list the test requirements for CACC, and propose a test suite which achieves CACC. (Hint: a test case looks like `d = new Instances(); d.add(new Instance(2.0)); d.add(new Instance(4.0)); selectWeightQuantile(d, 0.5);`.)

Question 4: Comparing ADUPC and PPC (2 points)

Draw a control-flow graph, annotated with the relevant definitions and uses, where ADUPC and PPC impose the same test requirements. (List these test requirements.) **Your CFG must contain a loop.**

Question 5: Mutation (20 points)

Consider the following sorting function from `en.literateprograms.org`.

```
public static void insertionSort(int[] a) {
    for (int i=1; i < a.length; i++) {
        /* Insert a[i] into the sorted sublist */
        int v = a[i];
        int j;
        for (j = i - 1; j >= 0; j--) {
            if (a[j] <= v) break;
            a[j + 1] = a[j];
        }
        a[j + 1] = v;
    }
}
```

Propose two non-stillborn mutations to this function. Write down test inputs which kill these mutants (syntax doesn't matter) and the expected output of your test cases on the original code and on the mutant.

Question 6: Input-Space Testing (10 points)

Consider the following `Polygon` class:

```
class Point { int x, y; }
class Polygon {
  Point[] points;
}
```

Propose three partitionings for the space of `Polygon` objects; describe the characteristics which you are using for these partitionings and the resulting blocks. Choose a base choice block and write a test suite which achieves base choice coverage.

Question 7: Stub/mock Objects (20 points)

Consider the following class definitions.

```
class A {
  int x, y;
  Object n;

  public A(int x, int y, Object n) {
    this.x = x; this.y = y; this.n = n;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public Object getN() { return n; }
}

class B {
  H h;

  public B(H h) { this.h = h; }

  private int state = 0;
  public void advanceState() { state++; }
  public void action() {
    if (state == 6) {
      h.foo(this); // foo() is expected to call this.respond().
    }
    if (state == 9) h.bar();
  }

  public void respond() { // ...
  }
}

interface C {
  String getStringValue(String key);
  void setStringValue(String key, String sessionValue);

  DataSet getDataSetValue(String key);
  void setDataSetValue(String key, DataSet data);
}
```

```
// source: class PlaceOrder from
// http://java.sun.com/developer/technicalArticles/Database/
//             dukesbakery/DukeBakery.java

class D {
   private DataPanel screenvar;
   private JTextArea msgout;
   private Connection dbconn;
   private OrderBox obox;

   public D(Connection dbc, DataPanel scv,
            JTextArea msg, OrderBox ob ) {
      dbconn = dbc; screenvar = scv; msgout = msg; obox = ob;
   }

   public void actionPerformed( ActionEvent e ) {
     int iwheat=0; int icake=0;
     Statement statement = dbconn.createStatement();
     iwheat = Integer.parseInt(obox.inwheat.getText());
     icake = Integer.parseInt(obox.incake.getText());
     java.util.Date date = new java.util.Date();
     SimpleDateFormat fmt = new SimpleDateFormat ("yyyy.MM.dd-HH:mm z");
     String dtstr = fmt.format(date);
     if( !screenvar.id.getText().equals("") ) {
       if (iwheat !=  0 || icake != 0) {
         String query = "INSERT INTO orders " +
                        "(LinkAddrTbl,OrderDate,wheat,cake)" +
                        "VALUES ("+ screenvar.id.getText() + "," +
                        "'" + dtstr           + "'," +
                        String.valueOf(iwheat) + "," +
                        String.valueOf(icake)  + ")";
         int result = statement.executeUpdate( query );
         if ( result == 1 )
           msgout.append( "\nOrder Placed\n" );
         else
           msgout.append( "\nInsertion failed\n" );
      }
   }
}
```

You are writing unit tests for a class `X` which depends on classes or interfaces `A` through `D`. (I'm not telling you how `X` uses these classes. Assume that it receives instances of these classes as parameters.) Describe which kinds of test doubles (if any) are appropriate for testing each of these classes (and why they are appropriate), and explain the challenges you expect to face in developing these test doubles.