Software Quality Engineering

Exercises

Exercise 1

The following program computes the greatest common divisor of two natural numbers by Euclid's algorithm.

```
Begin
   input(x,y);
   While(x>0 and y>0) Do
      If (x>y)
         Then x:=x-y
         Else y:=y-x
      Endif
   Endwhile
   output(x+y);
End
```

Questions:

1. Build the corresponding control flow graph
2. Devise test sets (in the form of paths), and then test inputs (i.e., values for `x` and `y`) in order to achieve:
   a. statement coverage
   b. edge coverage
   c. condition coverage
   d. path coverage

Exercise 2

Reuse the control flow built in Question 1, and devise test sets and test inputs in order to achieve (each time you will first identify the elements you have to cover in the graph):

a. 100% all-uses
b. 100% all-definitions
c. 100% all-definition-uses

Exercise 3

Consider the following Boolean expression and use criterion Modified Condition Coverage in order to devise inputs to be tested (recall that ∨ is the `OR`, ∧ the `AND`, and ¬ the `NOT`).

$$A \vee (B \wedge C)$$

Exercise 4

Consider the following Boolean expression and use criterion Modified Condition Coverage in order to devise inputs to be tested (recall that ∨ is the OR, ∧ the AND, and ¬ the NOT). Do you notice anything special?

```
A ∨ (A ∧ B)
```

Exercise 5

Consider the following Boolean expression and use criterion Modified Condition Coverage in order to devise inputs to be tested (recall that ∨ is the OR, ∧ the AND, and ¬ the NOT). Use Boolean algebra to simply the expression.

```
A ∧ (B ∨ ¬(A ∧ (A ∨ ¬C) ) )
```

Exercise 6

Consider the following Boolean function: `X = AB(CD+E)`
Use the Variable Negation strategy to derive test cases for `X`.

Number of variants = 2 power 5 = 32

Exercise 7

A `Java` class, named `OrdSetSimple`, implements an ordered set of integers (upper bounded, without duplicates). Among other things, it supports a basic sets operation, named `difference`, which determines the difference between two sets: the difference between sets `s1` and `s2` contains all the elements in `s1` that are not in `s2` (e.g., `{1,3,4,5}-{2,3,4}={1,5}`).
The upper bound of the set of integer is given as an argument of the constructor. Once created with a given upper bound `UB`, if a client of `OrdSetSimple` tries to add more than `UB` elements, the element is not added and an error message is written on the standard output.

Apply Category-Partition and devise test cases (and test data) on the difference operation. Clearly indicate what are the categories, choices, … when applying the technique. Note: you may need up to 5 different categories.

Exercise 8

Below is the source code for operator `difference`, as well as additional explanations (methods in class `OrdSetSimple`). Verify the structural coverage of your black-box test, and update them if necessary (according to Marick's principle), according to the following white-box criteria:

- All-Uses
- All-Definitions
- All-DU

Class `OrdSetSimple` has two public methods, namely `getSize()` and `getElementAt()` (names are self-explanatory), which are used in operator `difference`. Operator `difference` also uses method `binSearch()`, which returns the position of the element passed as an argument or `–1` if the element cannot be found (using a binary search algorithm). Last, the only mechanism provided by class `OrdSetSimple` to add elements is named `addElement()` (see how it is used below). This last method adds the element only if it is not already in the set, and writes an error message on the standard output if the set is already full.

When a variable is a reference to an object and method calls are performed on that reference: If the method modifies an attribute of the referenced object, consider this a *definition* of the object reference, otherwise consider this a *usage* of the object reference.

```
1     public OrdSetSimple difference(OrdSetSimple s2) {
2        OrdSetSimple s1 = this;
3        int size1 = s1.getSize();
4        int size2 = s2.getSize();
5
6        OrdSetSimple set = new OrdSetSimple(size2);
7        // creating a new set for the result
8
9        int k;   // for the visit of array s1
10
11       for(k = 0; k < size1; k++)
12          if ( s2.binSearch(s1.getElementAt(k)) < 0 )
13             set.addElement(s1.getElementAt(k));
14
15       return set;
16    }
```

Exercise 9

Next is a partial specification of an on-board software that controls the thrusters when an aircraft is landing:

> If the pilot selects the reverse-thrusters and either landing-gear is not down or not locked then the reverse-thrusters command should be over-ruled. In addition, if the landing-gear is down and locked, but the wheels are not spinning or the spinning speed is below a threshold, then the reverse-thruster command should also be over-ruled. Otherwise, the reverse-thruster command should be accepted.

1. Analyze this specification and produce the corresponding cause-effect graph. Clearly identify causes and effects (only consider the over-rule thruster effect).
2. Derive a decision table from the cause-effect graph (recall to use “don’t care” values), and provide test cases.

Exercise 10

Consider functions `Main()` and `B()` below. (For the purpose of this exercise these functions use a specific Integer class, that is also provided.)
Build the control flow graphs for `Main()` and `B()`. In each of these control-flow graphs, for the sake of clarity, add a node for the entry in the function and a node for the exit of the function showing formal parameters and return variables (if they exist).
Derive control flow (sub-)paths and associated test data according to the all-uses integration strategy described in class.

```
Main() {
  Integer i,j;
  Integer sum=new Integer(0);
  read i, j;
  while (i.getInt() < 10) {
    B(i, j, sum);
  }
  System.out.println("Sum: "+sum);
}
```

```
void B(Integer x, Integer y, Integer z) {
  …
  if (y.getInt() >= 0 ) then {
    z.setInt(z.getInt() + y.getInt());
    read y;
  }
  x.setInt(x.getInt() + 1);
}
```

```
public class Integer {
  private int i;
  public Integer(int i) { this.i=i; }
  public void setInt(int j) { i=j;}
  public int getInt(){return i;}
  public String toString(){return ""+i};
}
```

Exercise 11

Consider the (incomplete) description of classes `Shape`, `Triangle` and `EquiTriangle` below.

You will assume that you can devise both black-box and white-box test suites for each method in each class: e.g., you assume you have both black-box and white-box test suites for methods in class `Shape` (though this is an abstract class), except for method `area` which is the only abstract method (you only have a black-box test suite in this case).
In order to identify those test suites, use acronyms *TS* (Test suite from Specification) and *TP* (Test suite from Program), and subscript these acronyms with numbers identifying methods: e.g., TS_1 and TP_2 refer to a functional test suite for method `setPreferencePoint` (this method is numbered 1) and a structural test suite for method `getReferencePoint` (this method is numbered 2).

Apply the hierarchical incremental testing strategy we have seen in class and identify:

- For which of the test sets (functional and/or structural) defined for class `Shape`:
  - (1) One needs to execute again in the context of class `Triangle`;
  - (2) One does not need to execute again in the context of class `Triangle`;

(3) One needs to define (new test sets) in the context of class `Triangle`.

- For which of the test sets (functional and/or structural) used for class `Triangle`:
  (1) One needs to execute again in the context of class `EquiTriangle`;
  (2) One does not need to execute again in the context of class `EquiTriangle`;
  (3) One needs to define (new test sets) in the context of class `EquiTriangle`.

Justify your answers.

```
public abstract class Shape {
   private Point referencePoint;

   public void setReferencePoint(Point p) { referencePoint = p; }
   public Point getReferencePoint() { return referencePoint; }
   public void moveTo(Point p) {…}  // calls erase() and draw()
   public void erase() {…}          // calls draw()
   public void draw() {…}           // calls getReferencePoint()
   public abstract float area()     // abstract
   public Shape(Point p) {…}        // Constructor: sets the value of
                                    // referencePoint and calls draw()
   public Shape() {…}               // Constructor: sets the value of
                                    // referencePoint (default value)
                                    and
                                    // calls draw()
}

public class Triangle extends Shape {
   private Point vertex2;
   private Point vertex3;

   public Point getVertex1() { return getReferencePoint(); }
   public Point getVettex2() { return vertex2; }
   public Point getVertex3() { return vertex3; }
   public void setVertex1(Point p) { setReferencePoint(p); }
   public void setVertex2(Point p) { vertex2 = p; }
   public void setVertex3(Point p) { vertex3 = p; }
   public void draw();        // new version
   public float area();       // new version
   public Triangle();
   public Triangle(Point p1, Point p2, Point p3);
}

public class EquiTriangle extends Triangle {
   public float area();       // new version
   public EquiTriangle(Point p1, Point p2, Point p3);
   public EquiTriangle();
}
```

Exercise 12

Consider a type `IntBag`, with operations to `insert` and `remove` elements, as well as all the observers of `IntSet`. Bags are like sets except that elements can occur multiple times in a bag.
Is `IntBag` a legitimate subtype of `IntSet`? Explain by arguing that either the Liskov substitution principle is violated (for a non-subtype) or that it holds (for a subtype).

Exercise 13

Consider the source code for the CCoinBox class provided in the course slides
Apply Bashir&Goel technique for class testing. Add reporters where needed. Any comment?