Final Exam Practice Questions

Fall 2019

1. **Hoare Logic**
   Compute the weakest precondition of the following code **fun** assuming its post-condition is TRUE, meaning that it does not throw any exception and it terminates**.**

```
S1: if (x!=null) {
S2:     n =x.f;
S3: } else {
S4:     n = z++;
S5:     z=2*z+1;
    }
S6: a = new char[n-2];
S7: c = a[z];
```

2. **Hoare Logic**
   Please determine whether each of the following statements is true. You can simply select TRUE or FALSE. No need to justify your answers.
   - If {P}S{Q} and {P}S{R} then {P}S{Q AND R} holds. (TRUE/FALSE)
   - If {P}S{R} and {Q}S{R} then {P AND Q}S{R} holds. (TRUE/FALSE)
   - ({P}S{Q}) $\Leftrightarrow$ (P$\Rightarrow$wp(S,Q)). (TRUE/FALSE)
   - If {P}S{Q} and {Q}T{R} then {P}T;S{R}. (TRUE/FALSE)

3. **Weakest Precondition**
   Suppose your team member Jeanine wrote a Java program and you need to inspect her code. Add an assert statement in the beginning of the method to prevent exceptions.

```
private void doOneThing(int array[], int a, int b, int n) {
    // Add an assert statement to prevent exceptions.
    int result = a/b;
    int i=n-1;
    if(a-b > 0) System.out.println(array[i]);
}
```

4. Regression Testing

Please observe the old and new versions of the code below and answer the questions following it.

Old version

```
public void function(int x, int y) {
1:    if (x<=2)
2:          ++y;
3:    else
4:          --y;
5:    if (y>2)
6:        System.out.println(1);
7:    else
8:        System.out.println(0);
}
```

New Version

```
public void function2(int x, int y) {
1:    if (x<=2)
2:          ++y;
3:    else {
4:          --y;
          y*=2; // added
      }
5:    if (y>2)
6:        System.out.println(1);
7:    else
8:        System.out.println(0);
}
```

```
public void existingtests() {
      function(1,2);  //t1
      function(1,4);  //t2
      function(2,6);  //t3
      function(4,6);  //t4
      function(6,1);  //t5
}
```

a. Measure statement and branch coverage of the old version of function given the tests in the `existingtests` function. (assuming the tests are executed in order)

b. Select a minimum set of test inputs within `existingtests` that must be rerun to test the new version of function. In other words, apply the regression test selection technique discussed in class to identify a subset of tests.

c. Write a set of path conditions for the new version. In other words, what are the test inputs that must be used to test the modified version of the program? Please use symbolic execution technique to justify your answer.

5. Mutation Testing

You are the manager of a software development team. Your team member Mickey has implemented the following code and he told you that he already thoroughly tested his code. Mickey also said two other team members approved his patch during peer code reviews and so the code must be reliable enough. However, you have some doubts about its reliability and would like to assess testing adequacy independently, before deploying the patch in the next release. In order to test adequacy, you added following mutants into the code but Mickey's tests did not kill these mutants. Which test input from the following choices do you suggest Mickey to add? Describe your test input and corresponding expected result (i.e., test oracle) to kill these mutants.

```
public class Segment {
    public int select (int t[], int l, int u) {
        // Assume t is in ascending order, and l<u
        // Count the number of cells where l<t[i]<u
        if (l>=u) return 0;
        if (t==null) return 0;
        int num=0;
        for (int i=0; i<t.length && t[i] <u; i++) {
            // (1) mutate i=0 to i=2 in the above line;
            // (2) mutate t[i]<u to t[i] <=u in the above line
            if (l<t[i])
                num++;
        }
        return num;
    }
}
```

a.
```
public static void test1() {
   int s[] =  {2,2,3};
   System.out.println(select(s, 1, 3));
   if (select(s, 1, 3) == 2)
         System.out.println("PASS");
   }
```

b.
```
public static void test2() {
   int s[] =  {2,2,2};
   System.out.println(select(s, 1, 1));
   if (select(s, 1, 1) == 0)
         System.out.println("PASS");
   }
```

c.
```
public static void test3() {
   int s[] =  {2,2,2};
   System.out.println(select(s, 5, 3));
   if (select(s, 5, 3) == 0)
         System.out.println("PASS");
   }
```

d.
```
public static void test4() {
   int s[] =  {5,6,2};
   System.out.println(select(s, 0, 5));
   if (select(s, 0, 5) == 1)
         System.out.println("PASS");
   }
```

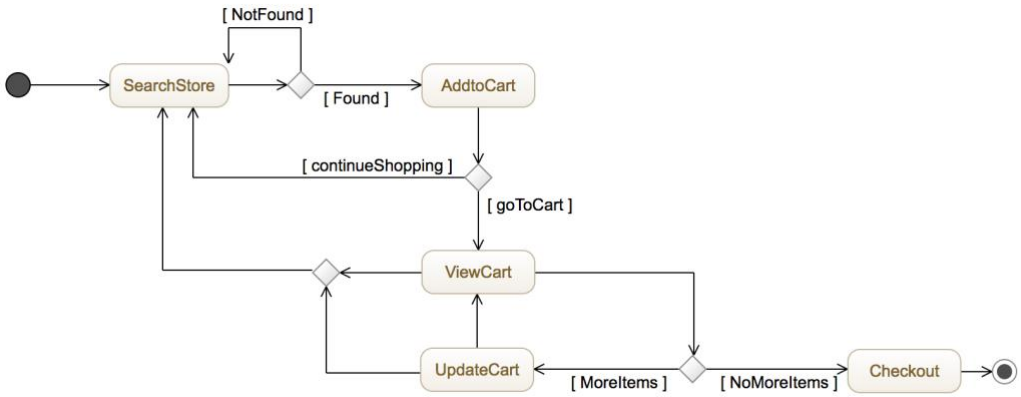
6. **Design Problems and Refactoring**
   Refactor the following code such that
   - When looking at client code, readers must know what all arguments mean.
   - Accidentally reversing arguments of the identical type cannot go undetected.
   - The fields servingSize, serving, and calories cannot be mutated by clients after the construction.
   - Clients cannot directly call NutritionFacts's constructor.

```
public class NutritionFacts {
    private final int servingSize; // required (in mL)
    private final int servings; // required (per container)
    private final int calories; // optional
    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }
    public NutritionFacts(int servingSize, int servings, int calories) {
        this(servingSize, servings, calories, 0);
    }
    public NutritionFacts(int servingSize,
                          int servings, int calories, int fat) {
        this.servingSize = servingSize;
        this.servings = servings;
        this.calories = calories;
    }
    public static void main(String[] args) {
        NutritionFacts cocaCola = new NutritionFacts(240, 8, 100);
    }
}
```

7. UML based Testing

The following diagram shows a UML activity diagram for shopping online. Your manager gives you this diagram and asks you to write test cases based on this UML model.

As shown in the UML activity diagram, actions are symbolized with a round-edged rectangles. Whereas, a conditional branch in the flow is represented with a diamond. It includes a single input and two or more outputs. Each of the output branches could be named, and if taken the next action in flow is taken. Diamonds also represent merges of two or more input, unconditionally, into one output. In this case, SearchStore action could be taken right after either ViewCart action or UpdateCart action is taken.

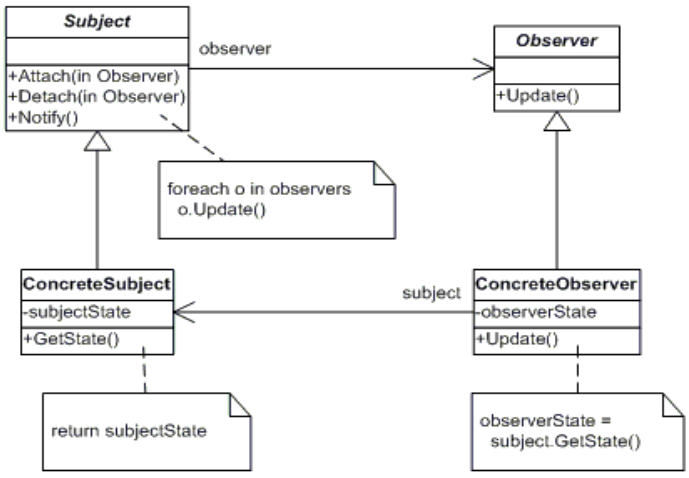


An example testing scenario is as follows:
{SearchStore, if [Found] AddtoCart, if [gotoCart] ViewCart, if [noMoreItems] Checkout}

Please write minimum number of test cases (using the above format) for a test suite such that your test suite achieves 100% node coverage.

8. Design Pattern

The following UML diagram describes the Observer design pattern.



Regarding the following code snippets implementing *Observer*, answer the following questions.

a. What is the name of code element corresponding to Observer and its update method?

b. Which class corresponds to *the participant role of* Subject and which methods correspond to its attach and notify methods respectively?

c. Which classes correspond to the participant of ConcreteObserver?

```
interface AlarmListener {
  public void alarm();
}

class SensorSystem {
  private Vector listeners = new Vector();
  public void register(AlarmListener al) {listeners.addElement(al);}
  public void soundTheAlarm() {
    for (Enumeration e = listeners.elements(); e.hasMoreElements();)
        ((AlarmListener)e.nextElement()).alarm();
  }
}

class Lighting implements AlarmListener {
  public void alarm() {System.out.println("lights up");}
}
```

```
class Gates implements AlarmListener {
   public void alarm() {System.out.println("gates close");}
}

class CheckList {
  public void byTheNumbers() {
    localize();
    isolate();
    identify();
  }
  protected void localize() {
    System.out.println("establish a perimeter");
  }
  protected void isolate()  {
    System.out.println("isolate the grid");
  }
  protected void identify() {
    System.out.println("identify the source");
  }
}

class Surveillance extends CheckList implements AlarmListener {
  public void alarm() {
    System.out.println("Surveillance - by the numbers:");
    byTheNumbers();
  }
  protected void isolate() {System.out.println( "train the cameras");}
}

public class ClassVersusInterface {
  public static void main( String[] args ) {
    SensorSystem ss = new SensorSystem();
    ss.register(new Gates());
    ss.register(new Lighting());
    ss.register(new Surveillance());
    ss.soundTheAlarm();
  }
}
```

9. Symbolic Execution and Path Condition

```
// Assume that abs(x) computes the absolute value of x.
public int pathCondition(int x){
S1:     int value = 0;
S2:     int y = x * x;
S3:     if(x > 4)
S4:         x = x + 1;
        else
S5:         x = abs(x);
S6:     if( 2 * x - 1 >= y)
S7:        y= y * 4;
        else
S8:         y = y / 4;
S9:     if (y > 8 || y < 2)
S10:        return value;

S11:    System.out.print("CS130 is software engineering course.");
}
```

a. Identify all feasible paths and describe a path condition for each feasible path. A path condition is a logical predicate that makes input variables to exercise a particular path in a program. (You may describe each path in terms of a sequence of statement labels and you do not need to include the effect of each path but it is okay to include it if you want.)

b. How many feasible paths do exist in this function?

10. Test Coverage

Answer the following question for the code snippet given below:

```
S1: sum = 0;
S2: for (int i = 0; i < N; i++ ){
S3:     for (int j = 1; j <= pow(2,i); j++){
S4:         if (a[i] < k){
S5:             sum++;
            }
        }
S6:     for (int m = 1; m <= i; m++){
S7:         if (pow(2,m) % 2 == 1){
S8:          sum++;
            }
            else {
S9:             sum--;
            }
        }
    }
```

a. Find the total number of feasible paths. You may assume that an array `a[]` exists, `N` and `k` are arbitrary inputs, and `a[i]` never throws an array out of bound exception. Please include your rationales and steps of your calculation. (Hint: `pow(a,b)` computes a^b).

b. Assume that N=3. For a test suite with two cases {(k=2, a[] = {4, 3, 5, 4}), (k=1, a[]={2,7,5,10})}. What is the cumulative path coverage (%) of the test suite? Your answer must include the total number of feasible paths (in other words a denominator), when N=3.

c. Assume that N=3. For a test suite with two cases, {(k=2, a[] = {2, 3, 1}), (k=3, a[]={2,3,1})}. What is the cumulative statement coverage (%) of the test suite? You may assume that the total number of statements is 13.

11. Hoare Logic and Weakest Precondition

What is the weakest precondition *P* for each of the following statements?

- {P?} z := x / y { z >1 } (Assume that a division by zero will thrown an exception.)
- {P?} y := 2 + z ; x := 3*y + z { x > 0 }
- {P?} assert x = 5 { x = 10 }
- {P?} x := -x + 1; y := x + y { y > 5 }

12. Test Coverage

You are given the following context-free grammar, as specification of the accepted input values to a simple calculator function.

```
Expr := SubExpr '+' Expr | SubExpr

SubExpr := '('Atomic '*' SubExpr')' | Atomic

Atomic := [0..9]+
```

Answer the following questions:

a. "(1 * 2) + 3" is an accepted input based on the given grammar (True, False).

b. "(1 * 2 + 3)" is an accepted input based on the given grammar (True, False).

c. One useful metric for grammar-based testing, is the *production coverage*. Production coverage refers to the percentage of production rules in the grammar that are exercised by the test cases. For instance, if only two of the production rules are exercised, then the production coverage is 40%. Identify the minimum number of following test cases covering the as many as possible production rules.

   T1: "87+63"

   T2: "(2 * 5) + 4"

   T3: "11 + ((8 * 6) * 22) + 81"

d. We can add another test case to cover an uncovered production rule. (True, False)