Practice Test
CSCI 2600 Principles of Software

- DO NOT OPEN THIS EXAM UNTIL TOLD TO DO SO!
- READ THROUGH THE ENTIRE EXAM BEFORE STARTING TO WORK.
- YOU ARE ALLOWED ONLY 1 “CHEAT” PAGES. NO OTHER MATERIAL IS ALLOWED.

This exam is worth 100 points.

Make sure you have 9 pages counting this one. There are 4 parts, each including multiple questions. If you need more room for an answer than is provided, please use the back of the page and indicate that you have done so. If you re-do a question, please make clear what is your final answer.

Be clear and brief in your explanations—rambling and lengthy answers will be penalized. All questions have short answers.

The following is for the use of graders

1. ________/20
2. ________/20
3. ________/40
4. ________/20

TOTAL:__________/100
Part I. Abstraction Functions and Rep Invariants

1) Short answer (fill in the blanks or answer briefly).
   a) The representation invariant maps __________ to __________.
   b) The abstraction function maps __________ to __________.
   c) Why representation exposure must be avoided?

2) (question due to Mike Ernst, UW) Consider the IntMap interface and unrelated IntStack on the following page.

Willy Wazoo wants to write his own implementation for IntMap, but the only data structure he knows how to use is an IntStack! So he started out like this before he got stuck:

```java
class WillysIntMap implements IntMap {
    // Represents the IntMap
    private IntStack theRep;
}
```

Help Willy write a rep invariant and abstraction function for his implementation. Don't change his representation. Do not write the implementation itself. It must be possible to implement IntMap using the rep invariant and abstraction function, but don't worry about the efficiency of the implementation. If you don't see how to implement IntMap, answer the questions as well as you can without regard for the implementation.

(a) Rep invariant

(b) Abstraction function
/** An IntMap is a mapping from integers to integers.
* It implements a subset of the functionality of Map<int,int>.
* All operations are exactly as specified in the documentation for Map.
*
* IntMap can be thought of as a set of key-value pairs:
*
* @specfield pairs == { <k1, v1>, <k2, v2>, <k3, v3>, ... } */

interface IntMap {
/** Associates the specified value with the specified key in this map. */
bool put(int key, int val);
/** Removes the mapping for the key from this map if it is present. */
int remove(int key);
/** Returns true if this map contains a mapping for the specified key. */
bool containsKey(int key);
/** Returns the value to which specified key is mapped, or 0 if this
* map contains no mapping for the key. */
int get(int key);
}

/**
* An IntStack represents a stack of ints.
* It implements a subset of the functionality of Stack<int>.
* All operations are exactly as specified in the documentation for Stack.
*
* IntStack can be thought of as an ordered list of ints:
*
* @specfield stack : List<int>
* stack == [a_0, a_1, a_2, ..., a_k]
*/

interface IntStack {
/** Pushes an item onto the top of this stack.
* If stack_pre == [a_0, a_1, a_2, ..., a_(k-1), a_k]
* then stack_post == [a_0, a_1, a_2, ..., a_(k-1), a_k, val]. */
void push(int val);
/**
* Removes the int at the top of this stack and returns that int.
* If stack_pre == [a_0, a_1, a_2, ..., a_(k-1), a_k]
* then stack_post == [a_0, a_1, a_2, ..., a_(k-1)]
* and the return value is a_k.
*/
int pop();
}
Part II. Testing

1) Draw the control-flow graph (CFG) for the function below:

```c
// requires a,b ≥ 0
// returns: the gcd of a and b
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - 2b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

2) What is the % statement coverage and the % branch coverage for test suite
   gcd(15, 6), gcd(4, 8)?

3) What test reveals the bug in gcd?

4) True/False questions.

   (a) (True/False) 100% statement coverage implies 100% branch coverage.
   (b) (True/False) 100% branch coverage implies 100% statement coverage.
   (c) (True/False) Given a path from start to end in a CFG, we can always write a
       (white-box) test that covers (i.e., executes) this path. (We require that the test
       covers the nodes/edges in the order specified by the path.)
   (d) (True/False) If implementation I passes all tests written for specification S1,
       then I passes all tests for a weaker specification S2.
   (e) (True/False) You can tell by looking at a test case, whether it was written using a
       black-box or white-box strategy.
Part III. LSP, Java Subtyping and Equality

1) True/False questions

(a) (True/False) An overriding method in Java must declare the same return type as the method it overrides.
(b) (True/False) An overriding method must have parameters that are the exact same type as the parameters of the method it overrides.
(c) (True/False) Java subtyping guarantees that the runtime type of reference \( x \) is a true subtype of \( x \)'s declared class.
(d) (True/False) Java subtyping guarantees that at call \( a.m() \) if \( a \) refers to object \( o \), \( o \) has an implementation of \( m \).

2) Short answers

(a) Explain briefly the benefits of true subtyping.

(b) Java does not allow subclassing of multiple classes (C++ does). Java allows subclassing of at most one class and implementation of multiple interfaces. Give at least two advantages of the Java approach.
3) Consider the following Java hierarchies and the clients below. At the designated places, write which method gets called, or indicate compile-time or runtime error.

```java
class Y extends X {
    ...}
class A {
    X m(Object o) {
    }
}
class B extends A {
    X m(Z z) {
    }
}
class C extends B {
    Y m(Z z) {
    }

A a = new B();
Object o = new Object();
X x = a.m(o); // Which m is called here?

A a = new C();
Object o = new Z();
X x = a.m(o); // Which m is called here?
```

4) Now consider these slightly different Java hierarchies and the clients below:

```java
class Y extends X {
    ...}
class W extends Z {
    ...}
class A {
    void m(Z z, Y y) {
    }
}
class B extends A {
    void m(W w, X x) {
    }
}
class C extends B {
    void m(W w, X x) {
    }

A a = new B();
Z z = new W();
Y y = new Y();
a.m(z,y); // Which m gets called here?

B b = new C();
W w = new W();
X x = new Y();
b.m(w,x); // Which m gets called here?
```
B b = new C();
W w = new W();
Y y = new Y();
b.m(w,y); // Which m gets called here?

Several questions below are based on the following code for a 2D point with integer coordinates:

class TwoDPoint {
    int x;
    int y;
    public boolean equals(Object o) {
        if (o instanceof TwoDPoint) {
            TwoDPoint p = (TwoDPoint) o;
            return p.x == x && p.y == y;
        } else
            return false;
    }
}

5) Let's implement a 3D point:

class ThreeDPoint extends TwoDPoint {
    int z;
    public boolean equals(Object o) {
        if (o instanceof ThreeDPoint) {
            ThreeDPoint p = (ThreeDPoint) o;
            return super.equals(p) && p.z == z;
        } else
            return false;
    }
}

Indicate which of the following properties holds for the equals method.

(a) Is equals reflexive? If not, give a counterexample below.

(b) Is equals symmetric? If not, give a counterexample.
(c) Is `equals` transitive? Again if not, give a counterexample.

6) Another implementation of 3D point `equals`:

```java
public boolean equals(Object o) {
    if (o instanceof TwoDPoint) {
        return super.equals(o);
    }
    else if (o instanceof ThreeDPoint) {
        ThreeDPoint p = (ThreeDPoint) o;
        return super.equals(p) && p.z == z;
    }
    else
        return false;
}
```

Again, indicate which of the following properties holds for the `equals` method.

(a) Is `equals` reflexive? If not, give a counterexample below.

(b) Is `equals` symmetric? If not, give a counterexample.

(c) Is `equals` transitive? Again if not, give a counterexample.

7) Yet another implementation of 3D point `equals`:

```java
public boolean equals(Object o) {
    if (o instanceof ThreeDPoint) {
        ThreeDPoint p = (ThreeDPoint) o;
        return super.equals(p) && p.z == z;
    }
    else if (o instanceof TwoDPoint) {
        return super.equals(o);
    }
    else
        return false;
}
```
(a) Is equals reflexive? If not, give a counterexample below.

(b) Is equals symmetric? If not, give a counterexample.

(c) Is equals transitive? Again if not, give a counterexample.

8) Rank the implementations of TwoDPoint.hashCode from best to worst:

(a) int hashCode() { return super.hashCode(); }
(b) int hashCode() { return 1; }
(c) int hashCode() { return x+y; }
(d) int hashCode() { return 31*x+y; }

Ranking: __, __, __, __

Part IV. Design Patterns

1) Short answers.

(a) What pattern(s) allows the creation of objects of varying types?

(b) What pattern(s) allows a client to treat collection objects and unit objects uniformly, i.e., through a common interface?

(c) What pattern restricts a class to a single instance?

(d) What pattern changes the interface of an existing class to an expected, different interface?

(e) Why can interning be applied only on immutable types?