CSE 331 Final Exam 12/14/15 Sample Solution

Question 1. (20 points, 1 each) Warmup. For each statement, circle T if it is true and F if it is false.

a) T / (F) A well-written test suite for a piece of software can guarantee that there are no bugs in the software.

b) T / (F) Two examples of wrapper patterns are Adapter and Builder

c) (T) / F One way to ensure that only a single instance of a class ever gets created is to use the singleton pattern.

d) (T) / F Two different sets of concrete variable values in an ADT can be mapped to the same abstract value by the Abstraction Function (AF).

e) T / (F) A client can add `DirectedEdge` instances to a `List<Edge>`, provided that `Edge` extends `DirectedEdge`.

f) T / (F) In unit testing, it is best to test all of a class's or module's functionality in one test method

g) T / (F) If A and B are two different types, and A is a subtype of B, then `Queue<A>` is a subtype of `Queue<B>`.

h) (T) / F Declaring a variable to have type `List<? extends Object>` is the same as declaring a variable to have type `List<?>`.

i) (T) / F In Java, it is possible to use `try..catch` to recover from exceptions and continue program execution.

j) (T) / F If the client fails to satisfy a `@requires` clause, you can throw an exception of your choice to let the client know.

k) T / (F) In a Swing GUI, the `actionPerformed` method that responds to a button click must call `paintComponent` if the screen needs to be redrawn.

l) (T) / F If two objects are equal as defined by their `equals` methods, then it must be that their hash codes returned by their `hashCode` methods are equal.

m) T / (F) A `hashCode` implementation that always returns a constant value is the best choice in terms of simplicity and performance.

n) (T) / F B is a true subtype of A if B has a stronger specification than A.

o) T / (F) A loop invariant can be considered too weak, but there is no such thing as a loop invariant that is "too strong".

p) (T) / F A specification is a two-way contract between the implementer and the client.

q) T / (F) An implementation must include code to check each `@requires` clause.

r) T / (F) The only way to avoid representation exposure is to make deep copies of all data that crosses the abstraction barrier.

s) T / (F) The equals specification requires that for any non-null reference value `x`, execution of `x.equals(null)` must throw a `NullPointerException`.

t) (T) / F Adding a precondition always weakens a specification.

Question 2. (10 points, 1 each) Complete each of the following sentences by filling in the word or phrase from the following list that best completes the sentence. Note that some of these words may be used more than once, and some won't be used at all.

abstract state, abstraction function, black-box, checked, cohesion, concrete state, coupling, creator, deep copy, deep equality, division of work, double checking, doubling, @effects, layering, logical equivalence, mixing, @modifies, mutator, observer, overview, producer, reading the specification, reference equality, reflexive equality, representation invariant, @requires, @returns, @throws, unchecked, validation, value equality, verification, white-box

a) Determining whether you have built the right system is called **validation**.

b) Determining whether you have built the system right is called **verification**.

c) In a JavaDoc specification, the **@effects** tag specifies or gives guarantees about the final state of all modified objects.

d) In ADTs, a(n) **producer** method returns a new value of the same type.

e) In ADTs, a(n) **mutator** method modifies the value of the ADT.

f) A(n) **representation invariant** denotes which concrete values of an ADT represent well-formed abstract values.

g) The `equals` method in class `Object` implements this kind of equality: **reference equality**.

h) When considering the general design of software modules, we want to increase **cohesion** and decrease **coupling** .

i) A `NullPointerException` is classified as a **unchecked** exception.

Question 3. (20 points) Testing/debugging/proving. The following method is supposed to swap pairs of elements in its integer array argument starting with arr[0] and arr[1]. If there is an odd number of array elements, the last element is not changed.

```
public void swapPairs(int[] arr) {
  int i = 0;
  while (i != arr.length) {
    int temp = arr[i];
    arr[i] = arr[i+1];
    arr[i+1] = temp;
    i = i + 2;
  }
}
```

Unfortunately there is at least one bug in the code.

(a) (8 points) Write two different tests for this method. One test should reveal the bug. The other test should check some other behavior and should succeed. You do not need to give JUnit or other code for the tests, although that is one way to answer the question clearly, but you should give a precise description of the test inputs, output expected, and actual output observed when used to test the above code.

```
// Will reveal the bug - throws ArrayIndexOutOfBounds exception
@Test
public void testSwapOnOddSizeArray {
   int[] arr = [3, 5, 2, 9, 4];
   swapPairs(arr);
}

// Does not reveal the bug
@Test
public void testSwapOnEvenSizeArray {
   int[] arr = [3, 1, 9, -4];
   swapPairs(arr);
   assertEquals(1, arr[0]);
   assertEquals(3, arr[1]);
   assertEquals(-4, arr[2]);
   assertEquals(9, arr[3]);
}
```

(Problem continued on the next page.)

Question 3. (cont) Now that we've verified that there is at least one bug in this code we'd like to fix it up and show that our fix is correct. Here is the original code again:

```
public void swapPairs(int[] arr) {
  int i = 0;
  while (i != arr.length) {
    int temp = arr[i]; arr[i] = arr[i+1]; arr[i+1] = temp;
    i = i + 2;
  }
}
```

(b) (12 points) Rewrite this method to fix the bug(s) and prove that your code is correct. You will need to provide suitable preconditions, postconditions, assertions, and a loop invariant to go with the proof. You may assume that the sequence `temp=a; a=b; b=temp;` will swap the contents of variables `a` and `b` and you do not need to provide intermediate assertions to prove that this swap operation itself is correct.

```
{ pre: arr != null }
public void swapPairs(int[] arr) {
  int i = 0;
  { inv: pairs a[k] and a[k+1] have been swapped for
    k even and k < i && i even }
  while (i < arr.length-1) {
    { inv && i < array.length - 1 =>
      arr[i] and arr[i+1] exist and should be swapped }
    int temp = arr[i];
    arr[i] = arr[i+1];
    arr[i+1] = temp;
    { pairs a[k] and a[k+1] swapped for k even
                                          and k < i+2 }
    i = i + 2;
    { inv }
  }
  { inv && i >= arr.length - 1 =>
              pairs a[k] and a[k+1] swapped for k even &&
              k < arr.length - 1 }
}
```

Question 4. (12 points) hashCodes. Consider the following class, which represents an immutable point on the 2-D plane:

```
public class Point2D {
  // rectangular coordinates of the point
  private final double x;
  private final double y;

  // Construct point at (0,0)
  public Point2D() { this.x = 0.0; this.y = 0.0;   }

  // construct point at (x,y)
  public Point2D(double x, double y) {
    this.x = x; this.y = y;
  }

  // observers
  public double getX() { return this.x; }
  public double getY() { return this.y; }

  // equality
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Point2D))
      return false;
    Point2D other = (Point2D) o;
    return this.x == other.x && this.y == other.y;
  }
}
```

(a) (10 points, 2 each) Here are five possible `hashCode` methods for this class. For each one, circle OK if this is a legal `hashCode` method for class `Point` as given above, or circle ERROR if it is not correct. All of the functions do compile without errors.

(OK) ERROR `int hashCode() {return (int)x;}`

(OK) ERROR `int hashCode() {return (int)(x + y);}`

(OK) ERROR `int hashCode() {return (int)(x*x + y*y);}`

(OK) ERROR `int hashCode() {return (int)(13*x + y);}`

(OK) ERROR `int hashCode() {return (int)Math.max(x,y);}`

(b) (2 points) Of the legal `hashCode` methods above, which one would be the best choice and why?

The 4th one (13*x+y). This uses both pieces of information but is not symmetric so it is more likely to produce different `hashCodes` for different x, y pairs.

Question 5. (10 points, 2 each) We'd now like to create a new class `PointBag` that holds a collection of Point2D objects with possible duplicates:

```
/** A PointBag is an unordered collection of non-null
 *  Point2D objects p1, p2, ... pn */
public class PointBag { … }
```

One of the methods we want to include in this class is one that adds a new `Point2D` to a `PointBag`. We are considering several possible ways to specify that method. All of the specifications will include the following:

```
  * @param    p the Point2D object to add
  * @modifies this
  * @effects  p added to this
```

But there are differences in the remaining parts of each possible specification. Here are the additional parts included in five possible specifications:

S1: `@requires p != null`

S2: `@throws IllegalArgumentException if p == null`

S3: `@throws NullPointerException if p == null`

S4: `@throws RuntimeException if p == null`

S5: *no additional clauses in this specification*

(Recall that `NullPointerException` and `IllegalArgumentException` are both subclasses of `RuntimeException`.)

For each of these five specifications, list all other specifications that are *stronger than* or *equal to* the named specification. Since each specification is equal to itself, you do not need to include each specification in its own list.

S1: **S2, S3, S4, S5**

S2: **none**

S3: **none**

S4: **S2, S3**

S5: **none**

Question 6. (10 points) Specifications. We would like to add a method to our `PointBag` class to compute and return a point that represents the centroid of the `Point2D` objects currently in the `PointBag`. The centroid is simply another point whose x and y coordinates are computed by averaging the coordinates of all of the points in the collection. We've got an implementation of this method, but it has not been properly documented yet.

Complete the JavaDoc comments for `centroid` below to provide the most suitable specification. Leave any unneeded parts blank. We have provided the method summary for you at the beginning. You may have to use your best judgment based on the implementation to decide how to specify some details. Hint: the answer probably won't need nearly this much space.

```
/**
 * Return a Point2D object that represents the centroid
 * of the points in this PointBag
 *
 * @param
 *
 * @requires number of items in PointBag > 0
 *
 * @modifies
 *
 * @effects
 *
 * @throws
 *
 * @returns new Point2D(x,y) where x is the average of
 *          the x coordinates of all points in the PointBag
 *          and y is the average of the y coordinates
 */
public Point2D centroid() {
  double xsum = 0.0; double ysum = 0.0;
  for (Point2D p : points) {
    xsum += p.getX();
    ysum += p.getY();
  }
  return new Point2D(xsum/points.size(),
                     ysum/points.size());
}
```

Note: the `@requires` clause is needed given this particular implementation, which will generate division by zero errors if the `PointBag` is empty.

Question 7. (18 points) Subclassing/subtypes. Suppose we define a new class `Point3D` that extends our previous `Point2D` class to represent 3-dimensional points:

```
public class Point3D extends Point2D { ... }
```

Because of this declaration we know that `Point3D` is a (Java) subtype of `Point2D`. Also recall that `Point2D` implicitly extends Java's `Object` type.

Further, recall that `ArrayList` is a Java subtype of `List` since it implements `List`.

(a) (6 points, 1 each) For each of the following, circle T (true) or F (false)

T / (F) `ArrayList<Point3D>` is a Java subtype of `ArrayList<Point2D>`

T / (F) `ArrayList<Point2D>` is a Java subtype of `ArrayList<Point3D>`

(T) / F `ArrayList<Point2D>` is a Java subtype of `List<Point2D>`

T / (F) `List<Point2D>` is a Java subtype of `ArrayList<Point2D>`

(T) / F `Point3D[]` is a Java subtype of `Point2D[]` (arrays)

T / (F) `Point2D[]` is a Java subtype of `Point3D[]` (arrays)

(question continued on the next page)

Question 7. (cont) (b) (12 points, 1 each) Now suppose we declare the following objects and lists using the `Point2D` and `Point3D` classes:

```
Object o;
Point2D two;
Point3D three;

List<Point2D> lst2d;
List<? extends Point2D> ext2d;
List<? super Point2D>  sup2d;
```

For each of the following, circle OK if the statement has correct Java types and will compile without type-checking errors; circle ERROR if there is some sort of type error.

(OK) ERROR `lst2d.add(two);`

OK (ERROR) `ext2d.add(two);`

OK (ERROR) `ext2d.add(three);`

(OK) ERROR `sup2d.add(two);`

(OK) ERROR `sup2d.add(three);`

OK (ERROR) `sup2d.add(o);`

OK (ERROR) `ext2d.add(o);`

(OK) ERROR `two = lst2d.get(0);`

(OK) ERROR `two = ext2d.get(0);`

OK (ERROR) `two = sup2d.get(0);`

OK (ERROR) `three = ext2d.get(0);`

OK (ERROR) `three = sup2d.get(0);`

Question 8. (10 points, 2 each) Design patterns. Here is are some of the design patterns we discussed this quarter:

Adapter, Builder, Composite, Decorator, Factory method, Factory object, Flyweight, Iterator, Intern, Interpreter, Model-View-Controller (MVC), Observer, Procedural, Prototype, Proxy, Singleton, Visitor, Wrapper

For each of the following code fragments or descriptions, give the name of the design pattern that is the *best* match to the description or to the result of the given code.

(a)
```
CurrencyConverter cc1 = CurrencyConverter.getInstance();
CurrencyConverter cc2 = CurrencyConverter.getInstance();
assert cc1 == cc2;   // always passes
```

singleton

(b)
```
CurrencyConverter cc1 = CurrencyConverter.incomingCurrency("USD")
                            .outgoingCurrency("CAD") // Canadian $
                            .build();
cc1.convert(50.00);
```

builder

(c) We have an existing class that counts coins and returns the total amount in CAD (Canadian dollars). A developer is writing a new class that uses the existing one to do the work but needs answers in US dollars.

adapter

(d) A method in the ATM software at the airport does currency conversions by forwarding the data to an identical method running on a computer at the bank's data center downtown where the calculations are done and results returned to the airport ATM computer.

proxy (we allowed half credit for MVC here, even though it isn't the best match)

(e) A method in the Campus Maps graphical user interface is called whenever a button on the screen is clicked because the method has previously been registered with the button to be notified whenever a click occurs.

observer

Question 9. (10 points) A couple of questions about testing.

(a) (3 points) Test Driven Development is a strategy where the tests for a module or function are always written before the actual code. Give a main reason why this is a useful strategy other than "it ensures that the tests will actually get written eventually."

Writing the test helps the author understand better what the code is supposed to do. The effort spent writing tests should ease the work needed to actually write correct code for the implementation later.

(b) (4 points) We observed that 100% *statement coverage* wasn't sufficient to guarantee that a test suite caught all possible bugs. A more comprehensive metric is 100% *path coverage*, where every possible execution path is executed at least once. But we also said that 100% path coverage is not realistic in most systems. Give two distinct reasons why this is true:

Here are three:

(i) **Loops. (Most programs contain loops that will execute an unknown number of times. It is impossible to execute all possible paths, which would mean all possible numbers of iterations.)**

(ii) **Combinatorial explosion. (Even without loops, the number of paths through the code is an exponential function of the number of branches in the code. In real code it is usually not feasible to execute all possibilities in reasonable amounts of time.)**

(iii) Rarely used or unusual paths through the code. Some code exists to handle "should never happen" or "almost never could happen" situations and devising tests that exercise all of these paths may not be feasible.

(c) (3 points) A guideline for testing and debugging is that once a test is found that reproduces a bug, that test must be added to the test suite and retained forever. Why? What value is that test after the bug is fixed?

A bug occurs because of a defect in the code due to some human error or misunderstanding. If it happened once, it could happen again as the code evolves, either because knowledge of the original problem fades with time or new people work on the code. Keeping the test in the test suite ensures that if the error is re-introduced it will be detected when the tests are run.

Best wishes for the holidays! See you after the break!
The CSE 331 staff