CSE 331 Final Exam 12/9/13

Name _____

There are 10 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, closed notes, closed electronics, closed telepathy, etc.

You may remove pages during the exam while you're working, but you must return all pages at the end.

Many of the questions have short solutions, even if the question is somewhat long. Don't be alarmed.

If you don't remember the exact syntax of some command or the format of a command's output, make the best attempt you can. We will make allowances when grading.

Relax, you are here to learn.

Please wait to turn the page until everyone is told to begin.

Score _____ / 100

1. _____ / 22
2. _____ / 16
3. _____ / 8
4. _____ / 8
5. _____ / 7
6. _____ / 8
7. _____ / 9
8. _____ / 7
9. _____ / 7
10. _____ / 8

Question 1. (22 points) Class specification. One of last summer's interns was working on a generic version of a graph that used an adjacency matrix representation. There's some existing code, but it's not specified or commented properly and it needs some errors fixed. In this problem we'll figure out what it does and clean it up.

Here is the code:

```
public class AdjacencyMatrixGraph<Edge> {
  private Edge[][] matrix;

  public AdjacencyMatrixGraph(int nodeCount) {
    matrix = new Edge[nodeCount][nodeCount];
  }

  public int getNodeCount() {
    return matrix.length;
  }

  public Edge getEdge(int start, int dest) {
    return matrix[start][dest];
  }

  public void addEdge(int start, int dest, Edge e) {
    matrix[start][dest] = e;
  }
}
```

(a) (3 points) The code for the constructor won't compile. What's wrong and how can it be fixed so it works as intended, but without any compiler errors or warnings. (Hints: an annotation – *@Something* – might be useful along with any other repairs needed. Also, remember that when a Java array is allocated with `new` the elements are initialized to `null`, so that's not the problem.)

Question 1. (cont.) (b) (9 points) Write a suitable JavaDoc comment summarizing the class, and inside the class give a suitable *representation invariant (RI)* and *abstraction function (AF)*. The class declaration and instance variable declaration are repeated here, along with space for the JavaDoc comment above the class declaration. Write the RI and AF comments below the instance variable declaration.

```
/**
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
public class AdjacencyMatrixGraph<Edge> {
  private Edge[][] matrix;

   // write a suitable RI below









   // write the corresponding AF below
```

Question 1. (cont.) (c) (10 points) Give proper CSE331-style JavaDoc specification comments for the `getEdge` and `addEdge` methods of this class. The methods are repeated below for your convenience, with space for your JavaDoc comments.

```
/**
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
public Edge getEdge(int start, int dest) {
  return matrix[start][dest];
}

/**
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
public void addEdge(int start, int dest, Edge e) {
  matrix[start][dest] = e;
}
```

Question 2. (16 points) A bit of design. Ima Hacker, a new Java programmer, was asked to create a small Java program for a text-based tic-tac-toe game, where the user plays against the computer. We eventually want to make the game work on a smartphone, but for now we just want a version that works using a keyboard and monitor. Omitting all the details, here's the outline of the program that Ima hacked up one afternoon.

```
/** A tic-tac-toe game */
public class TicTacToe {
  // instance variables omitted

  /** Initialize a new game object */
  public TicTacToe() { }

  /** Print the state of the game board on System.out */
  public void printGame() { }

  /** Read the player's next move from the keyboard
    * and update the game to reflect that move.      */
  public void getPlayerMove() { }

  /** Calculate the computer's next move and print it on
    * System.out */
  public void computerMove() { }

  /** Calculate the computer and user's current scores
    * and print them on System.out */
  public void printScores() { }

  /** Reset the game back to the same initial state it had
   *  when it was created */
  public void resetGame() { }
}
```

While this set of methods contains all the operations we want for the initial game, the design has problems.

(Question continued on the next page. You may remove this page for reference while working on it if convenient.)

Question 2. (cont.) (a) (4 points) What is (are) the major design flaw(s) in the above design? A brief couple of sentences should be enough to get the point across.

(b) (12 points) Describe how you would refactor (change) the initial design to improve it. Briefly sketch the class(es) and methods in your design, and what each of them do. You do not need to provide very much detail, but there should be enough so we can tell how you've rearranged the design, what the major pieces are, and how they interact.

(There's additional space on the next page if you need it for your answer.)

Question 2. (cont.) (Additional space for the answer to 2(b) if needed.)

Question 3. (8 points) A rather generic question. The following method performs a binary search for an integer value *x* in a sorted array of integers.

```
/** Search a sorted array of integers for a given value
  * @param a Array to be searched
  * @param x Value to search for in array a
  * @return If x is found in a, return largest i such
  *         that a[i]==x. Return -1 if x not found in a
  * @requires a!=null & a is sorted in non-decreasing
  *          order (i.e., a[0]<=a[1]<=...<=a[a.length-1])
  */
public static int bsearch(int[] a, int x) {
  int L = -1;
  int R = a.length;
  // inv: a[0..L] <= x && a[R..a.length-1] > x &&
  //      a[L+1..R-1] not examined
  while (L+1 != R) {
    int mid = (L+R)/2;
    if (a[mid] <= x)
      L = mid;
    else  // a[mid] > x
      R = mid;
  }
  if (L >= 0 && a[L] == x)
    return L;
  else
    return -1;
}
```

We would like to modify this code to change it into a generic method that works with any sorted array whose elements have type `Comparable<T>` (i.e., the elements can be ordered using the `compareTo` method). Mark the code above to show the changes needed to turn this into a generic method.

Question 4. (8 points) There's something fishy about this question. Suppose we have the following class hierarchy:

```
class Creature extends Object { }
class Fish extends Creature {
  /** Return the weight of this Fish */
  public float getWeight() { return ...; }
}
class Salmon  extends Fish { }
class Haddock extends Fish { }
class Tuna    extends Fish { }
```

Class `Creature` does not have a `getWeight` method. Class `Fish` implements that method and all of its subclasses inherit it.

Write a `static` method `collectionWeight` whose parameter can be any Java `Collection` containing `Fish` objects (including objects of any `Fish` subclass(es). Method `collectionWeight` should return the total weight of all the `Fish` in the `Collection`, using `getWeight` to determine the weight of each individual `Fish`.

Hints: Method `collectionWeight` will need a proper generic type for its parameter. Java `Collection`s are things like `List`s and `Set`s that implement the `Iterable` interface. They do not include things like `Map`s, which are not simple collections of objects.

Question 5. (7 points) Testing. (a) (3 points) There are many metrics we can use to try to evaluate how well a test suite does its job. One common metric is *code coverage*. If a test suite achieves 100% code coverage that means that every statement in the code being tested was executed by at least one test.

True or false: 100% code coverage is sufficient to guarantee that if an error is present in the code then it will be detected. If your answer is *true* give a brief justification. If your answer is *false*, give an example that shows why 100% code coverage is not sufficient.

(b) (2 points) Give one advantage that black box tests have compared to white (clear, glass) box tests. Be brief.

(c) (2 points) Give one advantage that white (clear, glass) box tests have compared to black box tests. Be brief.

Some shorter questions.

Question 6. (8 points) Subtyping. Suppose we have a class `A` with a method `m`:

```
class A {
  public T m(S x) { ... }
}
```

Now suppose we create a class `B` that is a subclass of `A` with its own method `m` that is supposed to override the one from class `A`:

```
class B extends A {
  public T1 m(S1 x) { ... }
}
```

(a) (4 points) If we want class `B` to be a true specification subtype of class `A`, what are the possible relationships between `m`'s types `T` and `S` in `A` and types `T1` and `S1` in `B`? Do the types have to match exactly or can one type be a specification (true) subtype or supertype of the other? Remember, this part of the question is asking about the typing rules for true specification subtypes, which might (or might not) be the same as Java's subclass rules.

(b) (4 points) Are Java's rules for subclass and method subtypes the same as the specification subtyping rules in part (a)? If so, it's sufficient to just say that they are the same. If not, what's the difference, and why are the Java rules different? (Be brief)

Question 7. (9 points) The implementation of `equals` in class `Object` returns true if two objects are exactly the same, i.e., `a.equals(b)` returns the result of the comparison `a==b`.

(a) Show that this definition of `equals` defines a proper equivalence relation (i.e., show that this implementation satisfies the properties required of an equivalence relation). A brief answer should be sufficient (and, yes, it's not tricky or complicated).

(b) The actual JavaDoc specification of `equals` in `Object` is fairly complex, enumerating many properties that an equivalence relation should have. Why didn't the specification for `a.equals(b)` in `Object` simply state that it returns the result `a==b`?

(c) The actual implementation of `hashCode` in `Object` returns the memory address of the object. Explain why this implementation of `hashCode` satisfies the necessary properties required of a `hashCode` implementation.

Question 8. (7 points) A couple of questions on usability:

(a) (2 points) If several events occur fast enough, a person perceives them as being a single event. How fast is "fast enough"? Circle the longest time interval that can generally occur between two events for them to be perceived as happening at the same time:

10 sec. 5 sec. 2 sec. 1 sec. 0.5 sec. 0.1 sec. 0.01 sec. 0.001 sec.

(b) (2 points) The complexity of a user interface should often be limited by the number of things that a typical person can hold at once in their short-term working memory. What is the maximum number of things that a typical person can hold in their short-term memory? (circle)

20 15 12 7 3 1

(c) (3 points) A strategy that initially seemed like a good idea was to present the user with a "confirmation dialog" requiring an additional approval before some irreversible action is performed. For example, the dialog might say "delete file?" or "launch missiles?" and require the user to click either yes or no. Why does this turn out not to be particularly effective in practice?

Question 9. (7 points) System design and implementation.

(a) (3 points) Why force everyone on a project to use a build tool like `make` or `ant`? IDEs like Eclipse have built-in tools to do this, why not let people use their IDE's build tools if they have them?

A large project can be built either bottom-up starting with modules that do not depend on others and that provide infrastructure for modules higher up, or top-down, starting at the top layer and adding lower-level modules as the project progresses.

(b) (2 points) Give one advantage of a top-down development strategy over a bottom-up one. (But be brief about it!)

(c) (2 points) Give one advantage of a bottom-up development strategy over a top-down one. (And be brief about this, too!)

Question 10. (8 points) Design patterns. For each of the following design patterns, give a brief explanation of the design problem that it solves and an example of a situation where it would be appropriate to use the pattern. For example,

Singleton: we want to insure that only one instance of a class is created. Examples would be to ensure there is only one random number generator shared throughout a program, or there is only one instance of a class that controls a particular printer or other device.

(a) Observer

(b) Interning

(c) Visitor

(d) Factory

Have a great winter break! See you in January!!