CSE 331 Final Exam 12/12/16

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.

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. ______ / 10	6. ______ / 6
2. ______ / 10	7. ______ / 10
3. ______ / 10	8. ______ / 12
4. ______ / 12	9. ______ / 12
5. ______ / 12	10. ______ / 6

CSE 331 Final Exam 12/12/16

The code on this page and the next implement two classes: one describes an individual cooking recipe, and the other is a collection of recipes. The code is not up to CSE 331 standards, alas, so several questions concern possible changes and issues with the code.

```
/** A single, immmutable recipe */
public class Recipe {
 String name;              // recipe name
 List<String> ingredients; // unique items (“flour”, “eggs”, ...)
 int servings;             // number of people served by recipe

  /**
    * Construct a new recipe
    * @param n string representing the name of the recipe
    * @param i collection of strings listing individual,
    *          unique ingredients in this recipe
    * @param s number of servings this recipe can serve
    * @requires n and i are not null and s > 0
    */
  public Recipe(String n, List<String> i, int s){
    name = n;
    ingredients = i;
    servings = s;
  }

  /** Return a list of the ingredients in this recipe */
  public List<String> getIngredients(){
    return ingredients;
  }

  /** Return the name of this recipe */
  public String getName(){
    return name;
  }

  /** Return the number of servings in this recipe  */
  public int getServings(){
    return servings;
  }
}
```

(Code continued on next page. You may remove this page and the next from the exam if you wish.)

Recipe and RecipeFolder code (cont.)

```
/** A mutable collection of recipes */
public class RecipeFolder {
  String owner;
  List<Recipe> recipes;

  /**
   * Construct a new, empty RecipeFolder object with owner o.
   * @requires - o != null and o != ""
   */
  public RecipeFolder(String o){
    owner = o;
    recipes = new ArrayList<Recipe>();
  }

  /** Add a recipe to this folder */
  public void addRecipe(Recipe r){
    recipes.add(r);
  }

  /** Return a recipe with the given name */
  public Recipe getRecipe(String name){
    for(Recipe item : recipes){
      if(item.getName().equals(name)){
        return item;
      }
    }
    return null;
  }

  /**
   * Return a list of the ingredients required to make the given
   * list of recipes. If the same ingredient is needed in more
   * than one recipe, include it once for each recipe that
   * requires it.
   * @param r - the list of recipes whose ingredients are needed
  public static List<String> getIngredients(List<Recipe> r){
    List<String> shoppingList = new ArrayList<String>();
    for (Recipe item : r){
      shoppingList.addAll(item.getIngredients());
    }
    return shoppingList;
  }
```

(Answer questions about these classes on the following pages. You can also remove this page from the exam if you'd like.)

Question 1. (10 points) RI & AF. The comments in the `Recipe` class code imply some properties of `Recipe` objects (immutable, lists of unique ingredients, maybe others).

(a) (6 points) Give a suitable Representation Invariant (RI) for class `Recipe`. Use the information in the code and comments, and consider possible uses of the class, to come up with the most appropriate choices.

(b) (4 points) Give a suitable Abstraction Function (AF) for class `Recipe` based on the representation invariant you gave above and the code for the class.

Question 2. (10 points) Most of the methods in the code do not have proper documentation. Below, complete the JavaDoc comments for methods `addRecipe` and `getIngredients` from class `RecipeFolder`. Leave any unneeded parts blank. You should use your best judgment based on the existing code to decide what to include.

```
/** Add a recipe to this folder
 *
 * @param r Recipe to be added
 *
 * @requires
 *
 *
 * @modifies
 *
 *
 * @effects
 *
 *
 * @throws
 *
 *
 * @returns
 *
 */
public void addRecipe(Recipe r){ recipes.add(r); }

/**Return a list of the ingredients required to make the given
 *list of recipes. If the same ingredient is needed in more
 *than one recipe, include it once for each recipe that uses it
 *
 * @param r
 *
 *
 * @requires
 *
 *
 * @modifies
 *
 *
 * @effects
 *
 *
 * @throws
 *
 *
 * @returns
 *
 */
public static List<String> getIngredients(List<Recipe> r){...}
```

(code omitted because of space – see previous page)

Question 3. (10 points) Representation exposure. Examine the code for classes `Recipe` and `RecipeFolder`.

Are there any potential representation exposure problems in either class? If so, give a brief description of where the problem(s) is (are).

If there are representation exposure problems, give a brief description of how to fix them. Your answer should be limited to the minimal changes that are needed to fix the problem(s) – i.e., don't make unnecessary copies of data or unneeded modifications to the code "just in case". You do not need to rewrite or copy all of the code here, just explain what changes need to be made and where.

If there are no representation exposure problems, simply write "no changes needed" below.

Question 4. (12 points, 4 points each) Testing. Describe three distinct "black box" tests for method `getIngredients` in class `RecipeFolder`. Note that this is a `static` method. That probably won't have any effect on your answers, but it might. For each test, describe the input or test setup, the test code that is to be executed, and the expected results. You do not need to include detailed junit or CSE 331-like specification test code.

(a)

(b)

(c)

(End of questions about Recipes and related code. But don't stop now! Continue on the next page with the rest of the exam.)

Question 5. (12 points) Specifications. Here are four possible specifications for a `find` method to locate an item x in a list that contains no duplicate entries:

S1: @param x – item to locate
@param list – list to search
@return location of x in the list or -1 if not found

S2: @param x – item to locate
@param list – list to search
@requires list is sorted
@return location of x in the list or -1 if not found

S3: @param x – item to locate
@param list – list to search
@return location of x in the list
@throws NotFoundException if x is not found in the list

S4: @param x – item to locate
@param list – list to search
@requires x is contained somewhere in the list
@return location of x in the list

We have four different implementations of the `find` method, A, B, C, and D, and each of these implementations is known to satisfy at least one specification, as shown in the following table (i.e., A satisfies S1, B satisfies S2, etc.)

Your job is to add additional X's in the table for the cases where we can conclude that an implementation satisfies additional specifications given that it is known to satisfy the specification already given in the table. (i.e., given the specifications above, and the known "implementation *xi* satisfies S*i*" information, which other specifications are also satisfied by each of the implementations?)

	S1	S2	S3	S4
A	X			
B		X		
C			X	
D				X

Question 6. (6 points, 1 each) Generics. Suppose we have the following Java classes:

```
class Furniture                     class CoffeeTable extends Table
class Table extends Furniture       class DinnerTable extends Table
```

For each of the following statements, circle T (true) or F (false). (Hint: recall that type `LinkedList` implements `List`.)

T F `List<Table>` is a Java subtype of `List<Furniture>`

T F `List<Furniture>` is a Java subtype of `List<Table>`

T F `CoffeeTable[]` is a Java subtype of `Furniture[]`

T F `Furniture[]` is a Java subtype of `Table[]`

T F `LinkedList<CoffeeTable>` is a Java subtype of `List<CoffeeTable>`

T F `LinkedList<CoffeeTable>` is a Java subtype of `List<Table>`

Question 7. (10 points, 1 each) Generics continued, or the "we-ran-out-of-new-ideas-for-this-one" question. We have the following classes from the previous problem:

```
class Furniture                    class CoffeeTable extends Table
class Table extends Furniture      class DinnerTable extends Table
```

Now suppose we have the following variables:

```
Object o;
Furniture f;              CoffeeTable ct;
Table t;                  DinnerTable dt;

List<? extends Furniture> lef;
List<? extends Table> let;
List<? super   Table> lst;
```

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 `lef.add(t);`

OK ERROR `lef.add(o);`

OK ERROR `lst.add(t);`

OK ERROR `lst.add(o);`

OK ERROR `let.add(dt);`

OK ERROR `o = lef.get(1);`

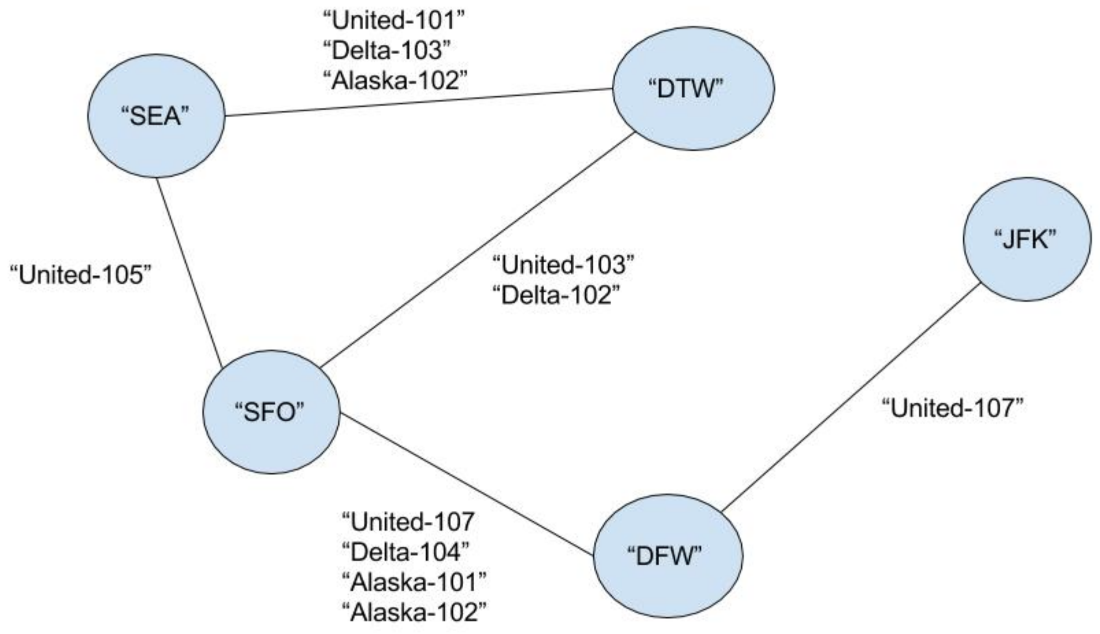
OK ERROR `f = let.get(1);`

OK ERROR `ct = let.get(1);`

OK ERROR `t = lst.get(1);`

OK ERROR `o = lst.get(1);`

Question 8. (12 points) Graphs. The following graph shows a small set of airline flights between five major airports in the United States. Each node is represented by an airport code (e.g., "SEA"), and the edges between the nodes are labeled with the available flights between them, all represented as strings. (To keep the diagram from being too cluttered, only one line is drawn between each pair of nodes.) We would like to find paths in this graph using the algorithms we explored in the project assignments.



Answer questions about this graph on the next page. You can remove this page from the exam if you wish and can use the rest of this page for scratch work if you want.

Question 8. (cont.) (a) (6 points) What is the shortest path between SEA and JFK using BFS (breadth-first search), as done in homework 6? Fill in the blanks below with the path segments that make up the shortest path. If there are ties, you should pick the "lexicographically least path" (i.e., use alphabetical ordering) as we did in HW6. If you need fewer lines than are provided, leave the remaining ones blank. If you need more lines, add them at the end.

__________ to __________ via flight _______________

__________ to __________ via flight _______________

__________ to __________ via flight _______________

__________ to __________ via flight _______________

__________ to __________ via flight _______________

(b) (6 points). Now suppose we treat the graph as having a single pair of directed weighted edges between each pair of nodes, where the weight of each edge is calculated as we did with the Marvel data for homework 7. In other words, the weight associated with each edge between two airports is the inverse of the number of flights between those two airports.

With this change, use Dijkstra's Algorithm to compute the shortest (least-cost) path from SEA to JFK. Fill in the blanks below with the path segments that make up the shortest path. If there are two or more minimum-cost paths from SEA to JFK, write down any one of them – which one is not specified. As before, leave unneeded lines at the end blank, or add additional lines if you need them.

__________ to __________ with weight _______________

__________ to __________ with weight _______________

__________ to __________ with weight _______________

__________ to __________ with weight _______________

__________ to __________ with weight _______________

Question 9. (12 points, 3 each) Design patterns. The main hw5-hw9 project sequence that resulted in the Marvel comics social graph and eventually in the Campus Maps application used several of the design patterns we discussed at the end of the quarter. For each of the following design patterns, identify a specific place or places where it appeared in your project (i.e., Which software component or components? How did they interact? Identifying specific assignments might be useful if it is not clear otherwise). Be brief and to the point.

(a) Model-View-Controller:

(b) Observer:

(c) Iterator:

(d) Composite:

Question 10. (6 points, 3 each) When we discussed system integration at the end of the course, we identified two strategies for organizing the building and integration of modules in a large system: top-down and bottom-up.

(a) Give one advantage of a top-down strategy compared to a bottom-up strategy. Be brief.

(b) Give one advantage of a bottom-up strategy compared to a top-down strategy. Again, brief and to the point is good.

Congratulations & best wishes for the holidays & the new year!!
The CSE 331 staff