CSE 331 Final Exam 12/12/16 Sample Solution

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.

There might be some small variations, but this is a reasonable choice based on what needs to be true for the code to work properly:

`name != null` and `name` is not ""

`ingredients != null` and for all items in `ingredients`, the item is not `null` and is not "", and for 0 <= i, j < ingredients.size(), if i != j, `ingredients.get(i).equals(ingredients.get(j)` is false (i.e., ingredients are unique – it was sufficient to just say that much for full credit)

`servings > 0`

(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.

This class represents a recipe where name is the title of the recipe, ingredients is a list of the ingredients required (for example, "flour", "butter"), and servings is the number of portions the recipe will produce.

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 r != null
 *
 *
 * @modifies this
 *
 *
 * @effects Recipe r is added to this RecipeFolder
 *
 *
 * @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 A list of recipes whose ingredients are to be
 *          returned
 *
 * @requires r != null and all elements of r are not null
 *
 *
 * @modifies
 *
 *
 * @effects
 *
 *
 * @throws
 *
 *
 * @returns A list of all ingredients found in the recipes in r
 *
 */
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.

Yes, there are several problems. Here is a list with possible solutions for each.

Problem: Both classes have package-visible instance variables, so client code can access the representation and change it.

Solution: Add "private" to all of the instance variable declarations

Problem: The `Recipe` constructor stores a reference to a client-provided list in one of its instance variables. If the client later changes that list, the change could invalidate the `Recipe` rep invariant or immutability.

Solution: The constructor has to create a clone of the list of ingredients and store a local copy in the `Recipe`. The strings in the list do not need to be copied since they are immutable.

Problem: `Recipe` `getIngredients` returns a reference to the `ingredients` list. Clients could modify this list, invalidating the rep invariant or mutating an immutable object.

Solution: Either return a copy of the `ArrayList` to the client (`clone` would be suitable here), or return an `unmodifiableList` view of the original list. Again, the strings in the ingredient list do not need to be copied.

Note: a non-problem is that `RecipeFolder` returns a `Recipe` object. Since `Recipe` objects are immutable (assuming we fix the other problems), there is no harm in allowing client code to have a reference to a `Recipe` object. Similarly, there is no need to copy a `Recipe` when adding it to a `RecipeFolder`.

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.

There are many possible answers. Here are some.

Create an empty List of Recipes r.
Let i = getIngredients(r);
Verify that i is empty

Create a List of Recipies r with a single Recipe having ingredients "flour" and "egg"
Let i = getIngredients(r)
Verify that i contains only "flour" and "egg"

Create a List of Recipes r with two copies of a single Recipe having ingredients "flour" and "egg"
Let i = getIngredients(r)
Verify that i contains "flour", "flour", "egg", "egg" in some order

Create a List of Recipes r with two recipes: one with ingredients "flour" and "egg", and another with ingredients "sugar" and "chocolate"
Let i = getIngredients(r)
Verify that i contains "flour", "egg", "sugar", "chocolate" in some order

Create a List of Recipes r with two recipes: one with ingredients "flour" and "egg", and another with ingredients "egg" and "chocolate"
Let i = getIngredients(r)
Verify that i contains "flour", "egg", "egg", "chocolate" in some order

(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	**X**		**X**
B		X		
C			X	**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);`

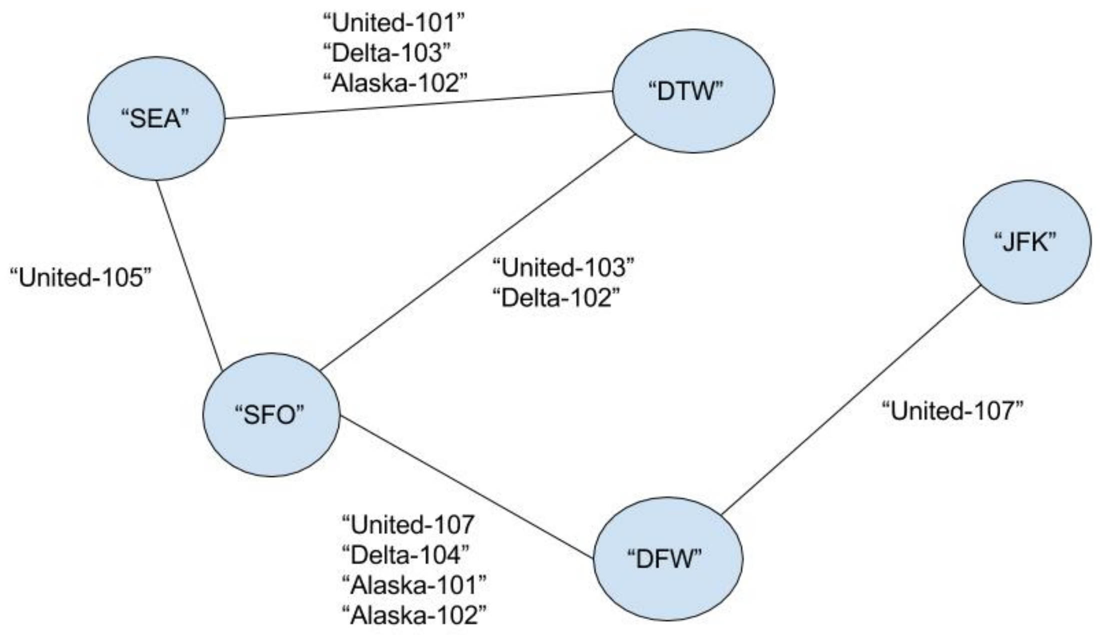
[✓ OK] ERROR `f = let.get(1);`

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.

___**SEA**____ to ___**SFO**____ via flight **United-105**

___**SFO**____ to ___**DFW**___ via flight **Alaska-101**

___**DFW**___ to ___**JFK**_____ via flight **United-107**

__________ 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.

___**SEA**___ to ___**DTW**___ with weight **0.33**

___**DTW**__ to ___**SFO**____ with weight **0.5**

___**SFO**___ to ___**DFW**___ with weight **0.25**

___**DFW**__ to ___**JFK**____ with weight **1.00**

__________ 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.

These answers are typical places where these design patterns appeared in most projects. Other answers received full credit if they had a reasonable description of the use of a particular design pattern.

(a) Model-View-Controller:

HW8 and HW9 – The model was the graph and related methods for finding paths and other data manipulation. The text interface (HW8) and GUI (HW9) were two implementations of the view/controller components.

(b) Observer:

HW9 user interface widgets are a great example. For instance, the reset button was an observable object, and the code that was notified when the button was clicked was an observer. (Note that this question was asking about the observer design pattern, not the notion of observer methods in an ADT, which is a different concept.)

(c) Iterator:

Many places in the code – pretty much anywhere that a `for(x:container)` loop would appear, such as iterating through a list of edges to process all nodes adjacent to a particular node.

(d) Composite:

Any object that had instance variables that were themselves ADTs or similar items. One example would be a `Graph` that included a hash table of `Node` information. Another example would be a `JPanel` subclass that contained multiple `JButtons` or drop-down lists or other user interface components.

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.

Top-down is particularly good both for making progress visible and for validating global design decisions early in the process when they are easier and cheaper to fix.

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

Bottom-up is especially useful for exposing resource or infrastructure efficiency problems early in the process. For instance, if we expect a database or networking infrastructure layer to support a particular transaction capacity, implementing that layer and stress-testing it early in the implementation will show if the needed performance can be obtained.

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