CSE 331 Final Exam 6/5/17 **Sample Solution**

Question 1. (26 points) Testing, specification, debugging, and proof – four questions in one! Consider the following method, which is supposed to compute the dot product (also called the scalar product) of two arrays of doubles. The dot product of two vectors {x1, x2, …, xn} * {y1, y2, …,yn} is the single number x1*y1 + x2*y2 + … + xn*yn.

Please do not remove this page from the exam. You will need to modify and prove this code in a later part of this question.

(Proof from part (d) written on this page, below. Bug fix is underlined.)

```
public static double dotprod(double[] x, double[] y) {
  { pre: x.length == y.length }
  double ans = 0.0;
  int k = 0;
  { inv: ans = x[0]*y[0] + ... + x[k-1]*y[k-1] }
  while (k < x.length) {
    { inv && k < x.length }
    ans = ans + x[k]*y[k];
    { ans = x[0]*y[0] + ... + x[k]*y[k] }
    k = k + 1;
    { inv }
  }
  { inv && k = x.length } =>
  { ans = x[0]*y[0] + ... + x[x.length-1]*y[x.length-1] }
  return ans;
}
```

The code would have been better if the loop condition were `k!=x.length`. The proof above relies on a bit of induction to conclude that `k=x.length` when the loop terminates.

(continued on next page – do not detach this page from the exam)

Question 1. (cont.) (a) (6 points) Ignoring any other possible problems with this method, the method only works if the two arrays have the same length (same number of elements). There are several ways we can deal with this issue when we write the specification for the method. Here is a list of some of the possibilities. Assuming that this method appears in a public library for anyone to use, you should rank these choices with 1 being the best choice, 2 being the second-best, etc. Write your numbers in the blanks provided. If there is a tie (i.e., two choices are equally good or equally bad) you should give them the same ranking using the same number:

__**3**__ Specify a precondition (`@requires`) `x.length==y.length`. Do nothing else.

__**2**__ Specify a precondition (`@requires`) `x.length==y.length`. Use an `assert` statement to terminate execution if the precondition is not met.

__**2**__ Specify a precondition (`@requires`) `x.length==y.length`. Throw an `IllegalArgumentException` to terminate execution if the precondition is not met.

__**1**__ Specify that the method will throw (`@throws`) an `IllegalArgumentException` if the two arrays have different lengths.

__**5**__ Specify that the method will throw (`@throws`) an `IndexOutOfBounds` exception if `y` is shorter than `x` and may return an incorrect result if `x` is shorter than `y` (i.e., excess elements in one of the vectors might be ignored or might lead to an error).

__**4**__ Do not say anything about the array lengths in the specification since it is implicit in the definition of dot product that they should be the same.

Notes: For a public method, specifying behavior for all inputs is best. If we require a precondition, it is best to check defensively for violations, and in that case using `assert` or explicitly throwing an exception are basically the same (throwing an exception might be a bit better since it cannot be disabled, and when grading we accepted that preference as well). Simply ignoring the situation is somewhat better than the bizarre specification that reflects accidents of how the code is implemented.

(b) (2 points) Even if both arrays have the same length, there is (are) bug(s) in this code. What bug(s) is (are) present? (Very briefly) Hint: if you don't see any problems right away, work on other parts of the question, which may reveal the bug.

In the loop the variable `ans` is assigned the value `x[k]*y[k]` each time through the loop, rather than having that product added to its previous value.

(continued on next page)

Question 1. (cont.) (c) (6 points) Complete the following two JUnit tests for this method. The first test should execute successfully using the original code for the method, in spite of any bug(s) you may have identified. The second test should reveal a bug that you have identified. For this exam question, it's fine to compare `doubles` for equality and not worry about roundoff errors.

```
// Does not reveal any bugs
@Test
public void test1() {
  double x[] = {0.0, 2.0};
  double y[] = {0.0, 3.0};
  assertEquals(6.0, dotprod(x, y));
}
```

Note: Arrays that have either one element or arrays where the sum of products except for the last add up to 0.0 will miss the bug.

```
// Reveals a bug given in your answer to part(b)
@Test
public void test2() {
  double x[] = {1.0, 2.0, 3.0};
  double y[] = {4.0, 5.0, 6.0};
  assertEquals(32.0, dotprod(x, y));
}
```

(d) (12 points) Now go back to the original code at the beginning of this question, fix the bug(s) you have identified by writing corrections in the code, and then prove that the resulting method is correct. You will need to provide a suitable loop invariant, plus any necessary preconditions, postconditions, assertions, and anything else needed to show that the repaired code works properly. You can omit some trivial assertions and logic steps to save writing, but your proof should contain all essential details.

See solution on earlier page.

Question 2. (10 points) A generic question. The code below is from the solution to a midterm exam problem involving the definition of a `Node` class that could be used in a `Graph` ADT.

We would like to change `Node` into a generic class, where the type of the node label is a generic parameter instead of `String`. We have already filled in part of the solution for you by adding the generic parameter `<E>` at the beginning of the Node class (in **Bold**).

Your job is to make all of the other changes needed to use this generic type instead of `String` inside `Node`, and then to make any changes needed in the enclosing `Graph` class and its `main` method so it successfully creates and uses generic `Node`s with `String` as the actual type parameter for the `Node` objects. Do *not* add any type parameters to class `Graph` itself, just to the `Node` class, and then change code that uses `Node`. Write your changes in the code below.

Changes are shown in bold green below.

```
Import java.util.*;
public class Graph {
  // inner node class
  private class Node<E> { // change this class to use E
    public final E label;
    public Node (E label) {
      this.label=label;
    }
    @Override
    public boolean equals(Object other) {
      if (! (other instanceof Node<?>))
        return false;
      Node<?> n = (Node<?>)other;
      return this.label.equals(n.label);
    }
    @Override
    public int hashCode() {
      return this.label.hashCode();
    }
  }  // end of Node<E>
```

(Problem continued on next page. Make any further needed changes there.)

Question 2. (cont.) Additional code from `Graph` that uses `Node<E>`. Make any necessary changes below.

```
  // Graph instance variable
  private Set<Node<String>> nodes;

  // constructor
  public Graph() {
    nodes = new HashSet<Node<String>>();
  }

  // methods
  public void addNode(String label) {
    nodes.add(new Node<String>(label));
  }

  public int getNumNodes() {
    return nodes.size();
  }

  public static void main (String[] args) {
    Graph g = new Graph();
    g.addNode("Waffles");
    g.addNode("Pancakes");
    g.addNode("Pancakes");
    g.addNode("Maple Syrup");
    System.out.println(g.getNumNodes());
  }
}  // end of Graph
```

The academic year is almost over and it's time for many people to move to a new place. We've been working with Fly-By-Night Movers to help them with their database for keeping track of items during moves. The next several questions refer to the following hierarchy of classes, which describes the household items being moved. The first two classes show some of the variables and methods that would be included in the final application. Other details have been omitted to save space.

You can remove this page for reference while working on the following problems.

```
public class Property {
  String description;  // property description
  int weight;          // weight
  public String getDescription() { return description; }
  public int getWeight() { return weight; }

  @Override
  public boolean equals(Object o) {
    if (! (o instanceof Property))
      return false;
    Property p = (Property) o;
    return this.description.equals(p.description) &&
                                this.weight == p.weight;
  }
  /* .... */
}

public class Furniture extends Property {
  String color;
  public String getColor() { return color; }

  @Override
  public boolean equals(Object o) {
    if (! (o instanceof Property))
      return false;
    if (! (o instanceof Furniture))
      return super.equals(o);
    Furniture f = (Furniture) o;
    return super.equals(f) && this.color.equals(f.color);
  }
  /* ... */
}

public class Table extends Furniture { /* ... */ }
public class Chair extends Furniture { /* ... */ }
public class Electronics extends Property { /* ... */ }
public class Phone extends Electronics { /* ... */ }
public class Computer extends Electronics { /* ... */ }
```

Question 3. (6 points) Hashcodes. (By popular demand based on *your* answers to the last question on the midterm!). Here are six possible `hashCode` methods for `Property`. Your job is to decide which ones are legal (satisfy the contract for `hashCode`), and, among the ones that are legal, to rank them from best to worst.

In the blank space to the left of each method, you should put an X if that `hashCode` is not appropriate or correct for class `Property`. For the ones that are legal, you should write 1 to the left of the best one, 2 next to the next-best one, and so forth. If two methods are about equally good or bad, you should rank them with the same number (i.e., ties might be possible). Use your best judgment – we will be fairly generous with partial credit when appropriate.

Note: `Math.random()` returns a real number in the range 0.0-1.0.

__**2**__ (i) `int hashCode() { return description.hashCode(); }`

__**2**__ (ii) `int hashCode() { return 13*description.hashCode(); }`

__**X**__ (iii) `int hashCode() { return (int)(Math.random()* description.hashCode()); }`

__**4**__ (iv) `int hashCode() { return 17; }`

__**3**__ (v) `int hashCode() { return weight; }`

__**1**__ (vi) `int hashCode() { return 13*description.hashCode() + weight; }`

Notes: Answers (i) and (ii) will have an identical distribution of values. They are probably somewhat better than (v), although we did not deduct points if all three of these were treated as equally good.

Question 4. (6 points) The `Property` and `Furniture` classes above both include `equals` methods. Part of the contract for `equals` is that it should implement an equivalence relation (i.e., reflexive, symmetric, and transitive). Do these `equals` methods implement correct equivalence relations? If your answer is yes, give a brief, convincing argument why they do. If your answer is no, give a counterexample that shows why one or both of these fail to implement an equivalence relation.

You should assume that methods and constructors are available to create `Property` and `Furniture` objects with whatever instance variable values you wish to use in your answer. You do not need to write Java code to create these objects, just argue whether the `equals` methods do or don't implement proper equivalence relations for whatever object(s) you use in your examples.

No, the `equals` methods are not transitive if `Property` and `Furniture` objects are compared to each other.

Suppose we have the following objects (assuming the existence of the obvious constructors for each class):

```
Property p1 = new Furniture("table", 25, "red");
Property p2 = new Property("table", 25);
Property p3 = new Furniture("table ", 25, "black");
```

Then `p1.equals(p2)` is true and `p2.equals(p3)` is true, but `p1.equals(p3)` is false.

Continuing with the app that we are developing for Fly-By-Night Movers, we need a class (ADT) that can store an inventory (list) of items that are being moved from one location to another. An `Inventory` stores a collection of <key, value> pairs, where the key is a string (a unique identifying tag) that identifies a single item in the inventory, and the associated value is a `Property` object describing that item. Here is the core part of the code for that class.

```
public class Inventory {
  // instance variable
  private Map<String, Property> items;  // <tag, property> pairs

  // construct an empty Inventory
  public Inventory() {
    items = new HashMap<String, Property>();
  }

  // add a new property with the given tag to this
  // Inventory.  Return null if this tag was not previously
  // used, otherwise return the property previously
  // associated with this tag.
  public Property add(String tag, Property p) {
    return items.put(tag, p);
  }

  // if tag appears as a key in this Inventory, remove
  // the <tag, property> from the inventory and return the
  // associated property; otherwise return null indicating
  // the tag was not found in the inventory
  public Property remove(String tag) {
    return items.remove(tag);
  }

  // return the <id tag, property> pairs in this Inventory
  public Map<String, Property> getItems() {
    return items;
  }

  // return the total weight of everything in this Inventory
  public int totalWeight() {
    int result = 0;
    for (Property p: items.values()) {
      result += p.getWeight();
    }
    return result;
  }
}
```

(Answer questions about this code on the following pages. You may remove this page from the exam for reference if you wish.)

Question 5. (10 points) Class specification, RI & AF. As always seems to be the case with CSE 331 exams, code samples often do not have adequate documentation. That's true of the `Inventory` class on the previous page.

(a) (3 points) Give a suitable abstract description for class `Inventory` as would be written at the beginning of the JavaDoc comment above the `Inventory` class heading.

An `Inventory` is a collection of items that are being moved from one location to another. Each item is a <key, value> pair, where key is a unique `String` tag identifying that item, and the value is a `Property` description.

(b) (4 points) Give a suitable Representation Invariant (RI) for this class. (Remember that the RI should be sufficient to guarantee that the existing code executes successfully.)

`items` is not `null`, and the key and value in each <key, value> pair in `items` are both non-`null`.

(Note that we have to require that the `Property` values are not `null` so that the code in `totalWeight` will work correctly. Requiring that key values are never null makes sense given the application.)

(c) (3 points) Give a suitable Abstraction Function (AF) for this class that relates the representation (and RI) to the abstract value of an `Inventory` object.

`items` stores the collection of items in this `Inventory`. Each <key, value> pair in `items` identifies a unique `String` tag (the key) and the associated `Property` value.

Question 6. (8 points) As usual, the methods in class `Inventory` need better documentation. Below, complete the JavaDoc comments for the constructor and method `totalWeight` from class `Inventory`. Leave any unneeded parts blank. You should use your best judgment based on the existing code to decide what to include. (Implementations omitted to save space – see previous pages for detailed code if needed.)

```
/** Construct a new Inventory
 *
 * @param
 *
 *
 * @requires
 *
 *
 * @modifies
 *
 *
 * @effects makes a new, empty Inventory
 *
 *
 * @throws
 *
 *
 * @returns
 *
 */
public Inventory() { ... }


/** Return the total weight of everything in this Inventory
 *
 * @param
 *
 *
 * @requires
 *
 *
 * @modifies
 *
 *
 * @effects
 *
 *
 * @throws
 *
 *
 * @returns the total weight of all the items in this
 *
 */
public int totalWeight() { ... }
```

Question 7. (6 points) Representation exposure. One of the new summer interns, who has just taken CSE 331 at UW, is convinced that the `getItems` method in `Inventory` creates a representation exposure problem. Is the intern right? If so, give a brief explanation of the problem and then suggest two (2) distinct ways that the problem could be solved while still making it possible for clients to retrieve a `Map` describing the contents of an `Inventory` object. If the intern is wrong, give a brief explanation of why there is no potential representation exposure problem with this method.

Yes, there is a problem. Returning this instance variable gives client code direct access to the `items` object, and that allows the client to modify the internal representation, including placing `null` keys or values in the `items` map.

Two reasonable solutions are (i) make a copy of `items` and return that copy to the user, and (ii) wrap `items` in an `unmodifiableMap` object and return a reference to that object.

(Note: there are additional issues if we worry about whether `Property` objects can be compromised, but we decided those were secondary enough that we did not mark off if there was no discussion of deep vs. shallow copies, etc. The code defining `Property` should have been more careful to protect fields so those objects would basically be immutable.)

Question 8. (10 points, 1 each) Generics revisited. Recall that the Property class hierarchy defines several related Java classes:

```
class Property
class Furniture extends Property
class Table extends Furniture
class Chair extends Furniture
class Electronics extends Property
class Phone extends Electronics
class Computer extends Electronics
```

Now suppose we have the following variables:

```
Object o;
Furniture f;     Electronics e;
Chair c;         Phone p;

List<? extends Furniture> lef;
List<? extends Phone> lep;
List<? super Chair> lsc;
```

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(c);`

OK **(ERROR)** `lef.add(o);`

(OK) ERROR `lsc.add(c);`

OK **(ERROR)** `lsc.add(f);`

(OK) ERROR `lep.add(null);`

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

OK **(ERROR)** `f = lsc.get(1);`

OK **(ERROR)** `c = lsc.get(1);`

OK **(ERROR)** `c = lef.get(1);`

(OK) ERROR `e = lep.get(1);`

Question 9. (8 points, 2 each) Comparing specifications. Suppose we want to add a `send` method to the app we are developing for Fly-By-Night Movers that will send a `Property` object to a destination. Here are four possible specifications for `send`:

S1: @param p – Property to send
@return true if p was successfully sent to the destination

S2: @param p – Property to send
@requires p.getWeight() <= 10
@return true if p was successfully sent to the destination

S3: @param p – Property to send
@return true if p was successfully sent to the destination
@throws IllegalArgumentException if p.getWeight() > 10

S4: @param p – Property to send
@requires p.getWeight() <= 5
@return true if p was successfully sent to the destination

In the answers below, you do not need to include each specification in the list of ones that are stronger or equal to itself. Just list other specifications that are stronger or equal.

(a) List all of the specification that are stronger than or equal to S1. __**none**___________

(b) List all of the specification that are stronger than or equal to S2. __**S1, S3**_________

(c) List all of the specification that are stronger than or equal to S3. __**none**___________

(d) List all of the specification that are stronger than or equal to S4. __**S1, S2, S3**______

Question 10. (10 points, 2 each) One last question: design patterns. Here is are some of the design patterns we discussed this quarter: Adapter, Builder, Composite, Decorator, Dependency Injection, Factory method, Factory object, Iterator, Intern, Model-View-Controller (MVC), Observer, Prototype, Proxy, Singleton

Below are short descriptions of several common design situations, each of which corresponds to one of the above design patterns. Below each description, write the name of the design pattern from the list above that most closely corresponds to that description.

(a) In a graphical user interface, when a user clicks a button on the screen, that causes a method callback to an object that has registered interest in being notified of those events.

Observer (not MVC, which is about separating viewer/controller from models)

(b) In HW8 and HW9, we were able to replace a text-based user interface with a graphical one without having to modify the core logic that stored the campus map and computed shortest paths.

MVC (We gave partial credit for Adaptor, since there is a possible argument that it is possible to build a graphical user interface on top of a text-based one, although this is likely not a good design. But MVC is the key pattern for this.)

(c) In the Swing user interface library, containers like `JFrame` can contain other components, some of which are also containers and can contain further subcomponents. Software dealing with these components can treat the overall collection or the individual components inside it in uniform ways since they all have similar interfaces and implement common methods like `getWidth` or `paintComponent`.

Composite

(d) An application accesses a database by calling methods belonging to a local object. That object forwards the work via an encrypted communications channel to a remote object that does the work and returns the results to the local object, which then returns the results to the application, without the application being aware of the existence of the remote object.

Proxy

(e) One optimization that Java's `String` class provides is the ability to have several variables share a single copy of a string value like "hello", rather than having multiple copies of the same immutable value stored separately in memory.

Intern

Congratulations from the CSE 331 staff!!
Have a great summer and best wishes for the future.