CSE 331 23wi Final Exam 3/14/23 Sample Solution

Question 1. (10 points) Specifications and implementations. Here are four possible specifications for a method that returns the location of the first occurrence of a character `ch` in a string `s`.

Specification A
@requires s != null
@return the index of the first occurrence of ch in s or -1 if ch is not found in s

Specification B
@requires s != null and ch appears at least once in string s
@return the index of the first occurrence of ch in s

Specification C
@return the index of the first occurrence of ch in s or -1 if ch is not found in s
@throws IllegalArgumentException if s == null

Specification D
@requires s != null
@return the index of the first occurrence of ch in s
@throws NoSuchElementException if ch does not appear in s

Now suppose we have four different implementations I1 through I4, each of which is known to satisfy one of the above specifications. Since an implementation that satisfies a stronger specification will also satisfy a weaker specification, it's entirely possible that some of the implementations might satisfy additional specifications beyond the one already shown below in the table.

Add an X in the following table in every square where the implementation given in the left column will also satisfy the specification shown in the top row. An X has already been supplied for each implementation and the specification it is known to satisfy.

Hint: it might (or might not) be useful to think about which specifications are stronger or weaker than others.

impl \ spec	spec. A	spec. B	spec. C	spec. D
impl. I1	X	**X**		
impl. I2		X		
impl. I3	**X**	**X**	X	
impl. I4		**X**		X

The next two question concern a set of classes that represent magical creatures including elfs*, gremlins, and dragons. The code for these questions is on the pages at the end of the exam. You should tear off those pages and use them with this question.

Question 2. (12 points) Overloading and Overriding. The following table contains in the left column lines of code from a main program that uses the magical creature classes from the code at the end of the exam. Your job is to write in the right column the output produced when the corresponding line of code is executed. (All of the code compiles and executes without errors.) If a line of code produces more than one line of output, write all of the output in that table entry in the correct order. If a line of code produces no output, leave the corresponding entry in the table blank. The lines of code are executed in order they appear in the table.

a) `MagicalCreature legolad = new Elf();`	
b) `legolad.demeanor();`	**`Pleasant.`**
c) `legolad.useMagic();`	**`Alakazam!`**
d) `Gremlin bojo = new Gremlin();`	
e) `bojo.useMagic();`	**`No magic for you!`**
f) `bojo.useMagic("42");`	**`Three wishes 42`**
g) `Elf sproot = new Gremlin();`	
h) `sproot.useMagic();`	**`No magic for you!`**
i) `MagicalCreature fafner = new Dragon();`	
j) `fafner.useMagic("451");`	**`Alakazam!`** **`Fire breathing magic 451`**

***Note: proper spelling, of course, is "elves", but we didn't want to confuse things since there is an Elf class in the code but not an Elve class. ☺**

Question 3. (12 points, 1 each) And now for the dreaded generics question. ☺ Using the creature class hierarchy from the previous question, assume we have the following variables:

```
Object o; MagicalCreature mc; Elf e; Gremlin g; Dragon d;

List<?> lq;
List<Elf> le;
List<? super Elf> lse;
List<? extends Elf> lee;
```

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. (Notes: the question only asks about type checking, so it doesn't matter whether the argument 0 in `get(0)` is out-of-bounds or not. The type checker also does not consider the actual values stored in variables when deciding if they are being used properly.)

a) OK (ERROR) `lq.add(e);`

b) (OK) ERROR `le.add(e);`

c) (OK) ERROR `lse.add(e);`

d) OK (ERROR) `lee.add(e);`

e) (OK) ERROR `le.add(g);`

f) (OK) ERROR `lse.add(g);`

g) OK (ERROR) `lee.add(g);`

h) (OK) ERROR `o = le.get(0);`

i) (OK) ERROR `mc = le.get(0);`

j) OK (ERROR) `e = lse.get(0);`

k) (OK) ERROR `e = lee.get(0);`

l) OK (ERROR) `g = lee.get(0);`

Specifications and things. It's preregistration time for next quarter and we have been working on a program to represent a list of students in a course. The code has two classes: `StudentInfo` and `ClassList`. The code for these classes is on a separate page at the end of the exam. You should detach that page and refer to it while reading and answering these questions.

Class `StudentInfo` represents the information about one student: a first and last name, and a student ID number. The names can be changed since a student can change their preferred name, but the ID number is immutable. To save space on the exam, the fields in this object are `public` so we can omit get and set methods.

The `ClassList` class stores a list of `StudentInfo` objects representing the students who are enrolled in a single course. The representation of this class is a simple list with a representation invariant that the list does not contain any `null` elements and no two `StudentInfo` objects in the list have the same ID number.

Question 4. (12 points) `equals` and `hashCode`. Two `StudentInfo` objects are considered to be equal if they have identical `idNum` fields, ignoring the first and last name fields. Below, write the Java code for an appropriate `equals` and a `hashCode` method to be included in class `StudentInfo`. You do not need to include JavaDoc comments (to save time).

```
@Override
public boolean equals(Object o) {
  if (! (o instanceof StudentInfo)) {
    return false;
  }
  StudentInfo other = (StudentInfo) o;
  return this.idNum == other.idNum;
}


@Override
public int hashCode() {
  return this.idNum;
}
```

Question 5. (10 points, 5 each) Method specifications. The methods in `ClassList` do not have proper specifications. Give appropriate CSE331-style JavaDoc specifications for methods `isRegistered` and `add`. For CSE331 custom tags like @requires (or any others) you are free to write @requires or @spec.requires. You should base your specifications on the intended behavior you can infer from the given code. If more than one specification is reasonable, give the one that would be best if this code were included in a publicly distributed library. Be sure to include a method summary (synopsis) at the beginning of each comment.

```
/**
 *  Return true if student s is registered in this course
 *
 *  @param s the student we are looking for
 *
 *  @requires s != null
 *
 *  @returns true if some student in this Classlist has an idNum
 *           that is equal to s.idnum
 */
public boolean isRegistered(StudentInfo s) { ... }

/**
 *  Add student s to this course if they are not already
 *  registered for it
 *
 *  @param s Student to add to this course
 *
 *  @returns true if s is added to this course, or false if the
 *           student was already registered in this course
 *
 *  @modifies this
 *
 *  @effects student s added to this if not already present
 *
 *  @returns true if s is added to this course, or false if
 *           student s already was registered for this course
 *
 *  @throws IllegalArgumentException if s == null
 *
 */
public boolean add(StudentInfo s) { ... }
```

Notes: for `isRegistered`, it is clear that `s` should not be `null`, but since this is only checked with an assertion, which could be disabled at runtime, this can only be a precondition for the method, and the method behavior if `s` is `null` cannot be guaranteed.

Method `add`, however, will always throw an exception if `s==null`, so best practice is to include this as part of the specified behavior.

Question 6. (8 points) Rep Invariants and forward reasoning. A rep invariant acts as both a precondition and a postcondition for a public method of an ADT like `ClassList`. For the ADT `ClassList`, the rep invariant is

```
// RI: students contains no nulls, and no two StudentInfo objects
// in students have the same idNum values.
```

We would like to use forward reasoning to verify that the `add` method preserves the rep invariant. Add annotations (proof assertions) in the following code to show that this is true.
Notes: (i) Your proof will be somewhat informal, since it will not include the same level of detail as we had in problems where we were reasoning about the values of integer variables. The key thing is to be sure to assert facts that can be combined and used to show that the rep invariant holds when the method returns to the caller.
(ii) Method `add` calls `isRegistered`. You can treat that inner method call as a single statement whose properties are given by the specification of `isRegistered`, and you do not need to trace or prove properties of that method.
The code for `add` is given below, with space for you to add assertions as needed. You should show that the rep invariant holds right before each `return` statement.

```
public boolean add(StudentInfo s) {
  { RI }  // rep invariant holds here
  if (s == null) {
    { RI is still true here as the method exits }
    throw new IllegalArgumentException();
  }
  { RI and s != null }
  if (isRegistered(s)) {
    { RI is still true here as the method exits }
    return false;
  }
  { RI and s != null and s.idNum does not match the idNum of
    any element in students (because isRegistered returned false). }
  students.add(s);
  { RI }
  return true;
}
```

Note: the steps in the proof need to include facts that we need to carry forward to use in later assertions, such as RI being true at various points in the code. It cannot magically (re-)appear in the final assertion if it has not been included in intermediate ones previously.

Question 7. (8 points) Representation exposure (the question that didn't get included in the midterm ☺)

We've decided that the `ClassList` ADT would be much more useful if it included a method that would return the list of `StudentInfo` objects stored in the `ClassList`. So we've added the following method:

```
// return the list of students in this course
public List<StudentInfo> getStudentList() {
  return Collections.unmodifiableList(students);
}
```

(a) (5 points) Does this method create a representation exposure problem for the class `ClassList`? Why or why not? (briefly – one or two sentences.)

Yes. Although client code cannot modify the unmodifiable list that is returned from `getStudentList`, the client can access the individual `StudentInfo` objects in that list and change the strings in those objects. That would modify the data that is the representation of `ClassList` and that should be owned privately by `ClassList`.

(b) (3 points) If there is a representation exposure problem (given in your answer to part (a)), how could we fix it? If there is no representation exposure possible in the original code, as explained in your answer to part (a), simply write "no changes needed" below.

The only real fix would be to modify `getStudentList` to create a deep copy of the students list, including copies of all of the `StudentInfo` objects in that list.

Note: We cannot make the strings in a `StudentInfo` object final or immutable, because those are supposed to be changeable if a student changes their preferred names. But we cannot allow the representation exposure that would let client code that calls `getStudentList` modify the internal private state of the `StudentList` object.

Question 8. (14 points) React. At the end of this exam, you will find the code from two React files `App.tsx` and `Item.tsx`. This React app is a very simple storefront for the CSE 331 Café, which sells Blueberry Muffins and Chocolate Chip Cookies. The web interface shows the name of each item and how many are available, and has a `Buy` button for each item that the user can click to buy one item. The store interface also shows the total number of items sold so far (cookies plus muffins).

(a) (7 points). Suppose we have the following two files included as part of this React app.

Index.html

```
<html>
   <body>
      <div id="root"></div>
   </body>
</html>
```

Index.tsx

```
ReactDOM.render(<App />, document.getElementById('root'));
```

When this React app is loaded into a browser, what is the HTML code that is generated by the React program and sent to the browser? You should show the HTML as it would appear initially before any buttons have been clicked to buy any muffins or cookies. The outer tags have been written for you. You need to add the HTML that would be generated by the React program.

```
<html>
   <body>
      <div id="root">
         <div>
            <h1>CSE331 Cafe</h1>
               <div>
                  <p>Item: Blueberry Muffin, Remaining: 2</p>
                  <button>Buy</button>
               </div>
               <div>
                  <p>Item: Chocolate Chip Cookie, Remaining: 3</p>
                  <button>Buy</button>
               </div>
            <p>Sold 0 items so far</p>
         </div>
      </div>
   </body>
</html>
```

Grading note: Each button would have some sort of callback ("onclick") action associated with it in the generated React/JavaScript code, but since we don't really know what that generated code would look like, we didn't expect to see it in the above html code answer.

(continued on next page)

Question 8. (cont.) (b) (6 points) There are initially two (2) Blueberry Muffins in stock. Suppose we click the `Buy` button for Blueberry Muffins three (3) times. What messages are written to the console log when we do this? (i.e., trace through execution of the React code and write down all of the `console.log` messages that are produced as we click the Blueberry Muffins `Buy` button three times.)

```
Selling a Blueberry Muffin
Updating quantity to: 1
CSE331 Cafe has sold one more item!
Selling a Blueberry Muffin
Updating quantity to: 0
CSE331 Cafe has sold one more item!
Selling a Blueberry Muffin
Cannot update quantity, no items to sell.
```

(c) (1 point) An ordinary user, of course, does not read messages written to the JavaScript `console.log`. When the user clicks the `Buy` button for Blueberry Muffins the third (3rd) time, what do they observe happening as a result of that third click?

An alert appears with the message "Sorry, sold out!"

Question 9. (12 points, 2 each) Design patterns. Here are some of the design patterns we have discussed or used this quarter:

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

For each of the following design problems or situations, write the name of the design pattern from the above list that is the best or most appropriate pattern to use or is used in this situation.

Note: We probably should have included Strategy in the list of patterns since we did discuss it in class, but it is not the best match for any of these problems.

(a) In a React program, the props passed to a button component includes a function that the button is supposed to call when the button is clicked.

Observer

(b) In a simulation of an aquarium, we want to be sure that all of the fish objects share the same random number generator object to be sure that there is only one source of random numbers in the program.

Singleton

(c) We have a library function that performs calculation using metric units (meters, etc.) and we want to use it to implement a function that does the same thing, only with U.S. units (feet, etc.)

Adapter

(d) We have a program that uses `Strings` and, to save space, we want to be sure that each unique `String` value is only stored once in the program.

Intern

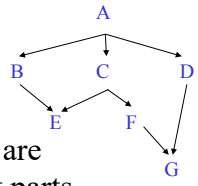
(e) We have a program that uses Complex number objects, but we have two possible implementations of Complex – one uses rectangular coordinates, the other uses Polar. We want the program to be able to select during execution which version to use when a new Complex object is created, and not have that decision fixed when the program is compiled.

Factory method

Note: Strategy would not be appropriate here, even if it had been included in the list. This situation involves creating a new object, and Strategy is not a creation pattern.

(f) We have a complicated object with many configuration options. We would like to organize the creation of this object so we can do it in multiple steps rather than needing to have a constructor with 12 parameters to set all of the configuration options all at once.

Builder



Question 10. (10 points, 2 each) System integration. When building a large system, there are two common strategies for the order in which to implement, combine, and test the different parts of the system: top-down and bottom-up. These two strategies have different characteristics and strengths.

For each of the following, circle "top-down" or "bottom-up" if that strategy is the best match to the description. If both strategies are a good match or are equally effective at solving the problem, circle "both". If neither strategy matches the description or neither is particularly effective at or relevant to solving the problem, circle "neither".

(a) Best at discovering early whether the network module that processes transactions will be able to handle the anticipated network traffic volume.

top-down **bottom-up** both neither

(b) Best for discovering early if there are major usability or human interface design problems in the system specification.

top-down bottom-up both neither

(c) Requires building "mock objects" or stubs in order to test a component that is being developed.

top-down bottom-up both neither

(d) Requires a good set of unit tests for each module to verify that it works as expected in isolation before combining it with other modules.

top-down bottom-up **both** neither

(e) Best for showing visible progress that can be observed by clients and funding agencies or investors.

top-down bottom-up both neither

Question 11. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

Draw a picture of something that you plan to do during spring break!

Congratulations from the CSE 331 staff!
Have a great break and we'll see you when you get back!!

Code representing various enchanted creatures for questions 2 and 3. **Remove this page from the exam** and use it while answering those questions.

```
abstract class MagicalCreature {
  public abstract void demeanor();
  public void useMagic() { System.out.println("Alakazam!"); }
  public void useMagic(String password) {
            System.out.println("Secret magic " + password); }

class Elf extends MagicalCreature {
  public void demeanor() { System.out.println("Pleasant."); }
  public void useMagic(String password) {
            System.out.println("Three wishes " + password); }
}

class Gremlin extends Elf {
  public void demeanor() { System.out.println("Mischievous."); }
  public void useMagic() {
                      System.out.println("No magic for you!"); }
}

class Dragon extends MagicalCreature {
  public void demeanor() { System.out.println("Angry"); }
  public void useMagic(String password) {
       useMagic();
       System.out.println("Fire breathing magic " + password); }
}
```

`StudentInfo` and `ClassList` code to be used to answer questions 4 through 7. **Remove this page from the exam** and use it while answering those questions.

```
// Simple data record for one student.
// Student names are mutable; id numbers are immutable.
class StudentInfo {
  public String firstName;
  public String lastName;
  final public int idNum;

  // insert equals() and hashCode() methods here.  See question ###.

  @Override
  public String toString() {
    return "StudentInfo("+lastName+", "+firstName+" ("+idNum+")";
  }
}

// collection of students in a course
public class ClassList {
  // rep invariant: students contains no nulls and no two students
  // have the same idNum
  List<StudentInfo> students;

  // construct a new, initially empty ClassList
  public ClassList() {
    students = new ArrayList<StudentInfo>();
  }

  // return true if a student with s.idNum is in this course
  public boolean isRegistered(StudentInfo s) {
    assert(s != null);
    for (StudentInfo student: students) {
      if (student.idNum == s.idNum) {
        return true;
      }
    }
    return false;
  }

  // add student to course if not already registered
  public boolean add(StudentInfo s) {
    if (s == null) {
      throw new IllegalArgumentException();
    }
    if (isRegistered(s)) {
      return false;
    }
    students.add(s);
    return true;
  }
}
```

React code to be used to answer question 8. **Remove this page from the exam** and use it while answering that question.

App.tsx

```
import React, {Component} from "react";
import Item from "./Item";

interface AppState {
  totalSold: number;   // total number of items sold
}

class App extends Component<{}, AppState> {

  constructor(props: {}) {
    super(props);
    this.state = {
        totalSold: 0
    }
  }

  increaseSoldCount = () => {
    console.log("CSE331 Cafe has sold one more item!");
    this.setState({
        totalSold: this.state.totalSold + 1
    });
  }

  render() {
    return (
        <div>
          <h1>CSE331 Cafe</h1>
          {/* Cafe inventory: */}
          <Item name={"Blueberry Muffin"} quantity={2}
                onSale={this.increaseSoldCount}></Item>
          <Item name={"Chocolate Chip Cookie"} quantity={3}
                onSale={this.increaseSoldCount}></Item>
          <p>Sold {this.state.totalSold} items so far!</p>
        </div>
    );
  }
}

export default App;
```

(additional code for this problem appears on the next page)

Additional React code for question 8. **Remove this page from the exam** and use it while answering that question.

Item.tsx

```
import React, { Component } from "react";

interface ItemProps {
  name: string;           // Name of item
  quantity: number;       // Initial number of item
  onSale: () => void;     // callback for when object is sold
}

interface ItemState {
  quantity: number;       // Current number of this item in stock
}

class Item extends Component<ItemProps, ItemState> {
  constructor(props: any) {
    super(props);
    this.state = {
      quantity: this.props.quantity,
    };
  };

  sellItem = () => {
    console.log("Selling a " + this.props.name);
    // Only sell the thing if there are some left
    if (this.state.quantity > 0) {
      console.log("Updating quantity to: " + (this.state.quantity-1));
      this.setState({
        quantity: this.state.quantity - 1
      });
      this.props.onSale();
    } else {
      console.log("Cannot update quantity, no items to sell.");
      alert("Sorry, sold out!");
    }
  };

  render() {
    return (
        <div>
          <p>Item: {this.props.name},
                   Remaining: {this.state.quantity}</p>
          <button onClick={this.sellItem}>Buy</button>
        </div>
    );
  }
}

export default Item;
```