CPSC 211

MIDTERM PRACTICE EXERCISES

Note: These questions are intended to help you practice and review the course material. Do not consider these questions as typical midterm questions; in particular, many would be difficult to finish within the length of the midterm.

Note: Challenging exercises are indicated with a * before their number.

Software Design.

*1. You have been asked to help design the "CPSC 211 Student Transcript System", whose purpose is to display student transcripts. A transcript lists the courses a student took, sorted by session. For each session, the transcript lists all courses taken by the student; both UBC courses for term 1 and term 2, and transfer courses (courses taken at another college or university, and whose credit can be used towards a UBC degree). Finally, each object listed on a transcript may render itself (i.e., may prepare to write itself out) if it is given a Renderer object. Each Renderer object knows how to print tables, lines, string, etc… in a specific output format. For the moment we are interested in HTML documents and Java GUI formats.

Suppose we have already determined that we likely need the following classes and interfaces for this problem:

- Transcript - a student transcript
- Session – an academic session for the transcript (i.e. 2007s, 2008w)
- Term – a term of an academic session (term1, term2)
- Course – a university course that appears in a term of a session in the transcript
- UBCCourse – a course taken at UBC
- TransferCourse – a course taken outside UBC and transferred to UBC
- Renderer – the interface of a renderer that can render a transcript
- JavaGUIRenderer – a renderer for Java applications
- HTMLRenderer – a renderer for Web applications
- Renderable – an interface for any object that can be rendered by a renderer

Draw a UML class diagram to describe the basic design for the Student Transcript System. Your diagram should include the given classes and interfaces and should show the relationships (with appropriate multiplicities) among them. Interfaces and classes must show the most important methods that are required for the functionality mentioned in the problem description. Make sure that your design satisfies the design principles we discussed in class.

2. Consider the following *partial* class specifications:

```
class GroceryOrder {
   // Each order includes a map of
   // items which have been ordered.

   protected GroceryBill bill;
   protected Map<GroceryItem,Integer>
         itemCount;
   protected double totalAmount;
   protected int totalItems;

   /**
    * Add an item to the map.
    * @pre newItem != null
    * @post newItem's count incremented
    * @post totalItems incremented
    */
   public void addItem(
               GroceryItem newItem) {…}

   /**
    * Compute current bill.
    * @pre true
    * @post getAmount() >= 0
    */
   public void computeBill() {…}


   /**
    * Finalize order.
    * @pre totalItems > 0
    * @post getAmount >= 0
    */
   public void checkOut() {…}

   /**
    * Gets total amount of order.
    * @pre true
    * @returns totalAmount
    */
   public double getAmount() {…}
}
```

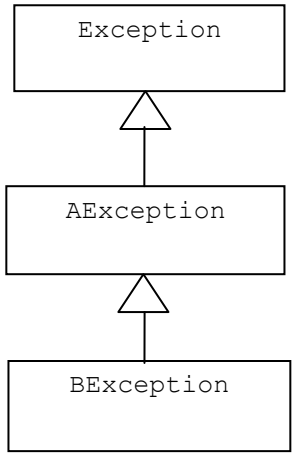
```
class DeliveredGroceryOrder
                  extends GroceryOrder {
   // orders which will be delivered to
   // customer's home use a special
   // delivery inventory, have a minimum
   // order and a delivery charge is added.

   private static final double
       MinDeliveryCharge = 5.00;
   private static final double
       MinOrderAmount = 25.00;
   private List<GroceryItem> delivInventory;
   // list of deliverable items

  /**
   * @pre newItem != null &&
   *      delivInventory.contains( newItem )
   * @post newItem's count incremented
   * @post totalItems incremented
   */
  public void addItem(GroceryItem newItem) {…}

   /**
    * Compute bill including delivery charge.
    * @pre true
    * @post getAmount() >= MinDeliveryCharge
    */
   public void computeBill()
     {…}

  /**
   * Finalize order.
   * @pre totalItems > 0
   * @pre getAmount() >= MinOrderAmount
   * @post getAmount() >= MinOrderAmount
   *                       + MinDeliveryCharge
   */
  public void checkOut() {…}
}
```

a) Complete the following table inserting the word "same", "weaker" or "stronger" for the pre- and postcondition of each method of the **`DeliveredGroceryOrder`** class to indicate whether the condition is the same, weaker or stronger than the corresponding condition in the super class.

	precondition	postcondition
`addItem`		
`computeBill`		
`checkOut`		

b) Is `DeliveredGroceryOrder` a proper subtype of `GroceryOrder` according to the Liskov Substitution Principle? Briefly explain your answer.

Exceptions



1. Assume that classes `AException` and `BException` are related as shown in the UML diagram to the right.

   What is the output produced when this program is run?

```
public class ExceptionTester {
    public static void main( String[] args ) {
        Catcher theCatcher = new Catcher();

        for( int val = -10; val <= 10; val += 10 ) {
            try {
                theCatcher.catchIt( val );
            }
            catch( BException e ) {
                System.out.println( "main caught: " + e.getMessage());
            }
        }
    }
}
public class Catcher {
   public void catchIt( int send ) throws BException  {
        Pitcher aPitcher = new Pitcher();

        try {
            aPitcher.throwIt( send );
        }
        catch( AException e ) {
            System.out.println( "catchIt caught: " + e.getMessage());
        }
    }
}
public class Pitcher {
    public void throwIt( int a ) throws AException, BException  {
        if( a < 0 )
            throw new AException("an exception ");
        else if( a == 0 )
            throw new BException("exceptional! ");
        else
            System.out.println( "In throwIt a is: " + a );
    }
}

class AException extends Exception {
   public AException (String arg0) {
         super("wow "+arg0);
   }
}
class BException extends AException {
   public BException (String arg0) {
         super("amazing "+arg0);
   }
}
```

Software Testing

1. Consider a class that represents a ticket purchased for an event at a theatre.

```
class TheatreTicket {

   // The price of the ticket
   private double price;

   // The location of the seat for which the ticket has been bought
   private int row;
   private int seat;


   /**
    * Set the price of a ticket
    * @pre true
    * @post the ticket's price = amount
    * @throws IllegalValueException (a runtime exception) when price <= 0
    */
    public void setPrice( double amount ) { … }

   /**
    * Set the location of the seat for which the ticket is purchased
    * @pre 0 < theRow <= 50 AND 0 < theSeat <= 100
    * @post the ticket's row = theRow AND the ticket's seat = theSeat
    */
    public void setLocation( int theRow, int theSeat ) { … }

    // The rest of the class is not shown
}
```

a. List the equivalence classes for the *amount* parameter of the `setPrice` method.

b. Write *four* test cases that result from applying the equivalence class partitioning and boundary condition technique to the `setLocation` method. Your test cases must include at least one typical case and at least one boundary case. For each test case, indicate the type of the test case (i.e. typical or boundary).

Test Case	Type

Java Collections, etc.

1. Using the methods in the Java Collection Framework, write a method
   `public static <E> void deleteAll(List<E> list, E obj)`
   which iterates through the list using an `Iterator` and deletes all the occurrences of the object `obj` (i.e. all objects that are `equals` to `obj`). What is the time complexity of your implementation in the cases that the method is passed an ArrayList and a LinkedList?

2. Using the methods in the Java Collection Framework, write a method
   `public static <E> List<Integer> getIndices(List<E> col, E obj)`
   which returns a list of the indices of the list that contain an occurrence of the object `obj`.

3. Using the methods in the Java Collection Framework, write a method
   `public static <E> List<E> subst(List<E> list,E old,E new)`
   which accepts a list `list`, and two objects of type E and returns a new list containing the elements in `list` with every occurrence of `old` replaced by `new`. The original list must be unchanged.

4. Based on the class structure on page 2, implement
   `public void addItem(GroceryItem newItem)`
   which increments the count for `newItem` in the map `itemCount`. (Hint: you need to do something different when `newItem` is in the map and when it isn't).

5. Assume that a Dog class is defined as following:

```
public class Dog {
   private String breed;
   private String name;
   private String gender;

   public Dog(String aBreed, String aName, String aGender)
   { … }
   public String getBreed() { … }
   public String getName() { … }
   public String getGender() { … }


   /* Two Dog objects are equal if they have equal breeds,
genders, and names.
  */
   public boolean equals(Object o) {…)

   public int hashCode() { … }
   …
}
```

Write the code for the equals method of this class.