PRACTICE FINAL EXAM
CPSC 210

Consider this scenario: your friend makes handmade *hats*, *scarves* and *handbags*, and sells these items on ebay.com. You want to make an online shop to market and sell these crafts. The starter code is in the **code pack**. This system will be used for most questions on the exam, unless otherwise noted.

Q1. UML Class Diagram [10 marks]

Extract a complete UML class diagram (including associations) for all types in the code pack except for the class **`Main`**. Do not include any types from the Java library. Do not include fields or methods. Do not include dependencies.

Draw your diagram here

Q2. Debugging [7 marks]

(a) The following code will not compile.

```
Wearable hat = new Hat("green hat", 20);
Wearable scarf = new Scarf("green scarf", Clothing.SIZE.M, 30);
System.out.println( hat.hasMatchingScarf() ); // This line will not compile.
```

Investigate the issue and specify what the problem is.

(b) The following assertion will fail.

```
Handbag handbag = new Handbag("test handbag", 220);
assertEquals(220, handbag.getPrice());
```

Assume we want the assertion to pass. Investigate the issue and specify how you would fix the problem (write all the necessary code).

(c) The following assertion will fail.

```
Set<Customer> customers = new HashSet<>();
customers.add(new Customer("Ali", "madooei@cs.ubc.ca"));
customers.add(new Customer("Ali", "madooei@cs.ubc.ca"));
assertEquals(1, customers.size());
```

Assume we want the assertion to pass. What code would you generate and where (in which class), to make the assertion pass. Give the names of the methods and any other information needed or options chosen to generate that code.

Q3. Exception Handling and Testing [9 marks]

Look at the specification of **`makePayment`** in **`Customer`** class. Make its implementation more robust by throwing a *checked* exception called **`InvalidInputException`**.

(a) Specify all the steps and changes that you will make (write the code when necessary).

1. Create a checked exception. Fill in the blank:

```
public class InvalidInputException extends _____________________ {

}
```

2. Write the *specification*, *signature* and *implementation* of the updated **`Customer`**'s **`makePayment`** method

3. Rewrite the code below to accommodate the change to **`Customer`**'s **`makePayment`** method.

```
Customer c = new Customer("Trey", "trey@somewhere.net");
int amount = getPaymentAmount(); // assume this method returns an int
                                 // you do not need to know its details.
if (amount > 0){
    c.makePayment(amount);
}
else {
    System.out.println("Sorry --- invalid payment amount");
}
```

(b) Complete the following two unit tests for testing your new robust **`Customer`**'s **`makePayment`** method designed above. Do not rely on any method tagged `@BeforeEach`. Place any initialisation code in the tests as needed. Do not use the "expected" clause from JUnit 4 (and don't worry if you don't know what the "expected" clause is)

```
//TEST CASE: Make a payment with valid input
@Test
void testMakePaymentValidInput(){









}
//TEST CASE: Make a payment with invalid input
@Test
void testMakePaymentInvalidInput(){









}
```

Q4. Data Abstraction [7 marks]

Provide an implementation for the **`addToShoppingBasket`** in the **`Store`** class.

```
// MODIFIES: this
// EFFECTS: add wearable w to the shopping basket of customer c
public void addToShoppingBasket(Customer c, Wearable w){









}
```

Q5. Bidirectional Association [8 marks]

Your friend often makes matching hats and scarves.

Provide an implementation for **`Scarf.removeMatchingHat`**

```
// MODIFIES: this
// EFFECTS: removes the matching hat for this scarf
public void removeMatchingHat(){









}
```

Provide an implementation for **`Hat.removeMatchingScarf`**

```
// MODIFIES: this
// EFFECTS: removes the matching scarf for this hat
public void removeMatchingScarf(){









}
```

Q6. Design Principles [12 marks]

The **`Store`**'s **`checkout`** method (also available in the code pack) is too long. Refactor its implementation and extract two helper methods - one for each part of the code that is surrounded by a bounding box.

```
// MODIFIES: this
// EFFECTS: omitted
public void checkout(Customer c){

   List<Wearable> basket = shoppingBasket.get(c);

  double totalPrice = 0;
  for(Wearable w: basket) {
      totalPrice += w.getPrice();
  }

   boolean madePayment = c.makePayment(totalPrice);

   if(madePayment){
       System.out.println("Payment received!");
       System.out.println("Shipping to: " + c.getName());

      for(Wearable w: basket) {
          inventory.remove(w);
      }

       basket.clear();
       shoppingBasket.remove(c);

   } else {
       System.out.println("Payment failed!");
   }
}
```

(a) Write the implementation for the two helper methods in the space provided on the next page. Include the method specification (*REQUIRES* / *MODIFIES* / *EFFECTS* as needed), and choose reasonable names for each method based on what they do.

First helper method

Second helper method

(b) Complete the updated implementation of **`Store.checkout`** below. Make use of helper methods that you specified in previous question.

```
// MODIFIES: this
// EFFECTS: omitted
public void checkout(Customer c){

    List<Wearable> basket = shoppingBasket.get(c);
```


```
    boolean madePayment = c.makePayment(totalPrice);

    if(madePayment){
        System.out.println("Payment received!");
        System.out.println("Shipping to: " + c.getName());
```


```
        basket.clear();
        shoppingBasket.remove(c);

    } else {
        System.out.println("Payment failed!");
    }
}
```

(c) Consider the **`BronzeCustomer`** and **`GoldCustomer`** classes in the code pack. Are objects of type **`BronzeCustomer`** and **`GoldCustomer`** substitutable for objects of type **`Customer`**. (Substitutability in terms of the design principles that you learned in this course). In each case clearly state why.

`BronzeCustomer` is substitutable for **`Customer`** (yes or no): _____

Explain:

`GoldCustomer` is substitutable for **`Customer`** (yes or no): _____

Explain:

Q7. Composite Design Pattern [13 marks]

We want to introduce a new class **`Combo`**. We want a combo to be made up of any kind of wearable (but no duplicates of any item), and also other Combos. We want to be able to get the price of a Combo by calling the **`getPrice()`** method on a Combo.

Use the composite pattern to integrate Combos into the project. You may need to add additional classes to carry this out.

(a) Draw a UML class diagram that shows how the composite pattern can be applied in the situation described above. Include only classes that play a role in the application of this pattern. Do not include fields or methods.

Draw your diagram here

(b) What fields would **`Combo`** have?

(c) Implement the **`getPrice`** method for **`Combo`** so that it adds up and returns the price of all the wearable and combos that it holds.

```
// EFFECTS: Returns the price of this collection of wearable
@Override
public double getPrice() {









}
```

(d) What other methods you will have to implement in **`Combo`**? Only write their signatures (neither implementation nor method specification is needed).

Q8. Observer Design Pattern [14 marks]

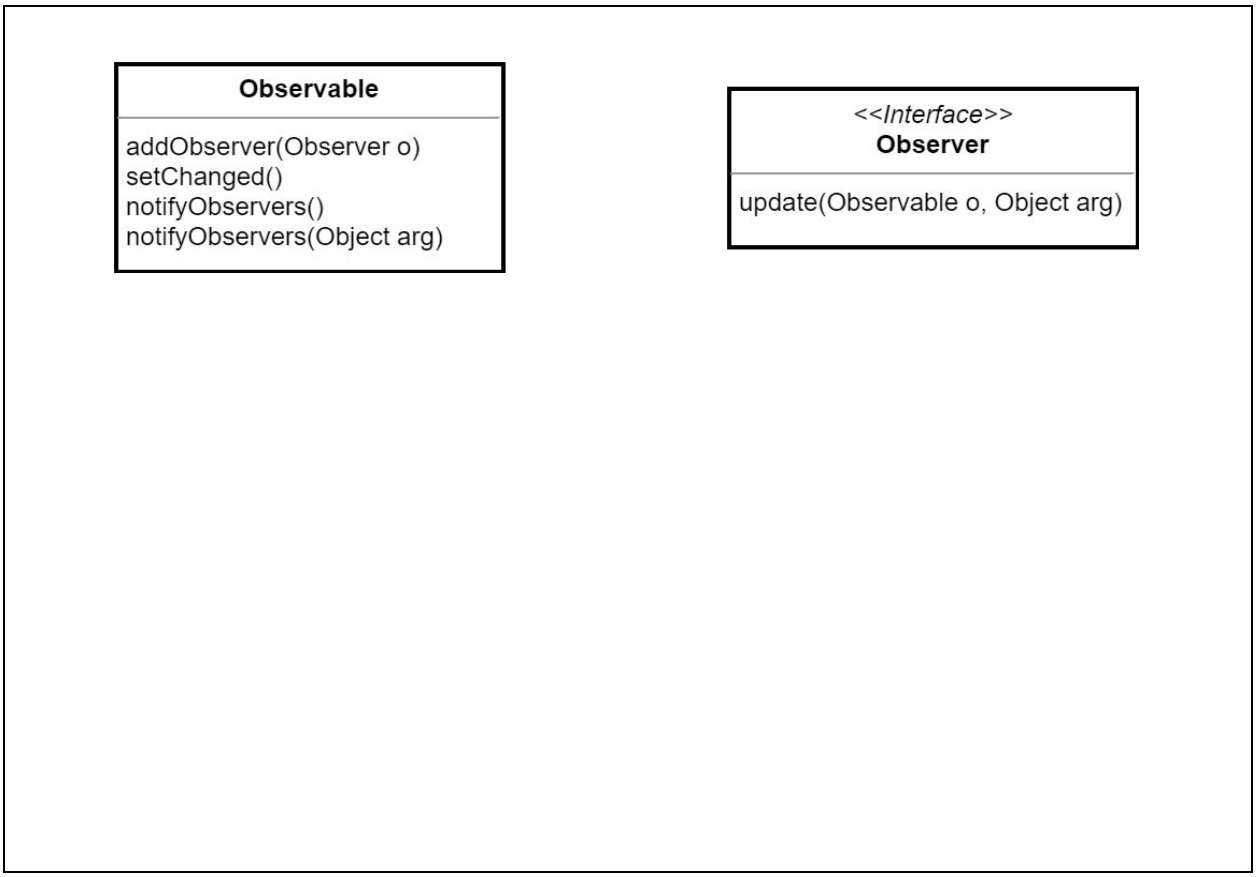
Consider a situation in which Customers can subscribe for updates from a Store when a Wearable is added. The customer wants the item that has been added, and will then invoke getName() on that item. The customer will then print out the message **"I am so excited to see the new ___"** where the blank is filled in with the name of the new item.

Customers are signed up as subscribers when they get added to the store as a customer (whether they like it or not)

Apply the Observer design pattern to make this work. Use Java's **`Observer`**/**`Observable`** to carry out the design.

(a) Draw a UML diagram to show how you would use the observer pattern to inform subscribers about the addition of new Wearables to the store. Include only the **`Observable`** and **`Observer`** types along with any of their subtypes. Include any methods that play a direct role in the application of the pattern.

Complete the diagram below:



(b) What changes would you make to the class that is the **concrete subject**? Do not rewrite the entire class -- but you must include the complete implementation of any modified methods or any new methods.

(c) What changes would you make to the **concrete observer** class? Do not rewrite the entire class -- but you must include the complete implementation of any modified methods or any new methods.

Q9. Iterator Design Pattern [12 marks]

Notice the following code in the **`main`** method:

```
for(Wearable w: store.getInventory())
   System.out.println(w.getName() + "($" + w.getPrice() + ")");
```

Let's change it to iterate over the wearables in **`Store`** without using the **`getInventory`** method, as shown below.

```
for(Wearable w: store)
   System.out.println(w.getName() + "($" + w.getPrice() + ")");
```

(a) Re-implement all the code needed to make the change, including changes to class declarations, if any).

(b) Let's add another collection of Wearables in **`Store`** to keep some wearables for display only (not for sale).

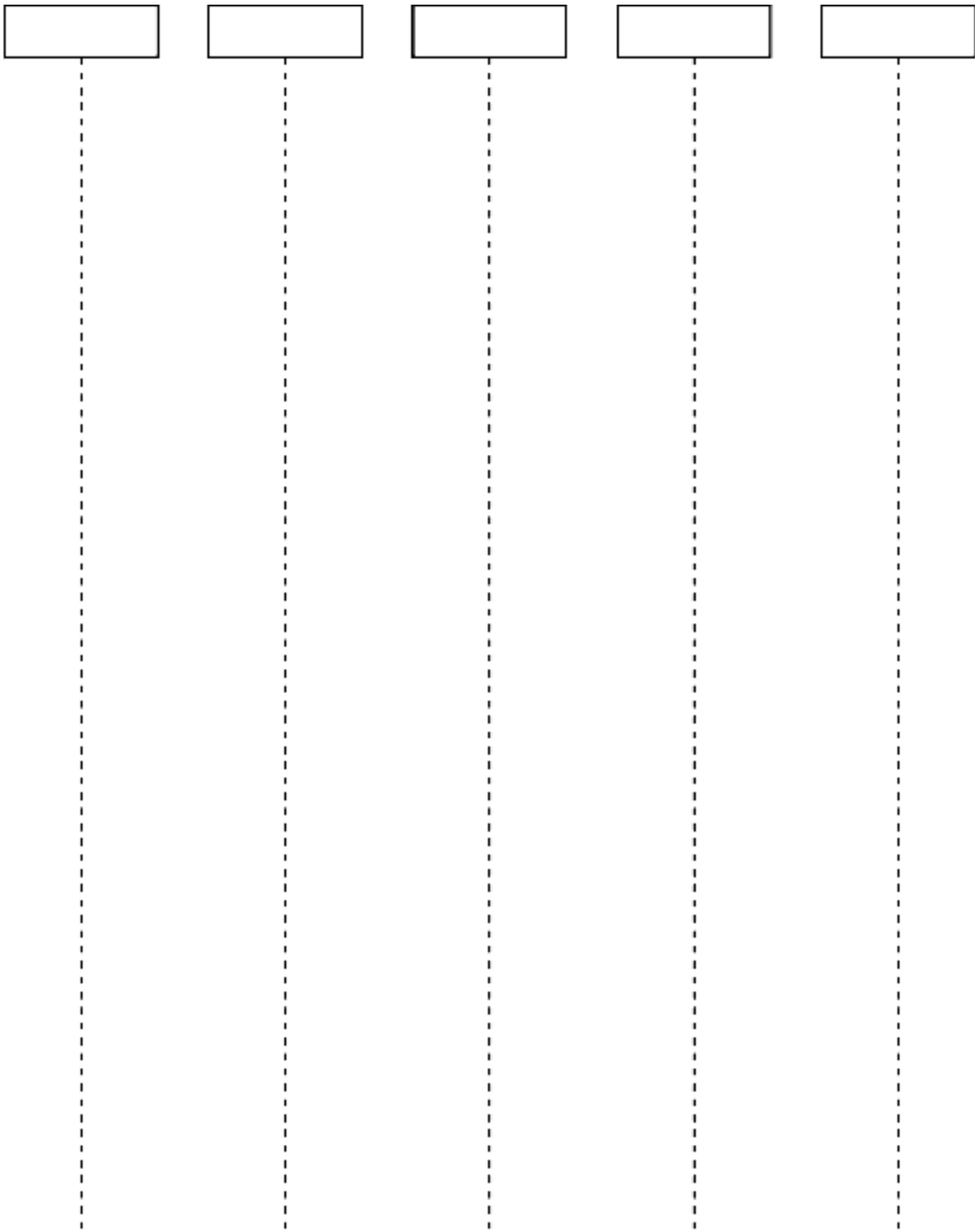
```
public class Store {
   private List<Wearable> inventory;
   private List<Wearable> displayOnly;
   // the rest of the code is not displayed to save space!
}
```

When we iterate over wearables in store using a single foreach loop, we want to first iterate over all elements in **`inventory`**, and then iterate over all the elements in **`displayOnly`**.
Write all code necessary to achieve this.
Note: Do not make a third list that merges the two lists.

Q10. Sequence Diagram [8 marks]

Using the following code, draw a sequence diagram for **`House`**'s **`doNighttimeThings()`** method. Do not include calls to the Java library. For the for-loop you may enclose in a box, or draw two passes through the loop. Use the lifelines on the next page for your diagram.

```
public class House {
   List<Mouse> mice = new ArrayList<>();
   Cat cat = new Cat();

   public House(){
       mice.add(new Mouse());
       mice.add(new Mouse());
   }

   public void doNighttimeThings(){
       cat.prowl();
       for (Mouse mouse : mice){
           cat.chase(mouse);
       }
       cat.knockThingsOver();
   }
}

public class Cat {

   public void prowl() { System.out.println("..."); }

   public void chase(Mouse mouse) { mouse.runAway(); }

   public void knockThingsOver() { breakThings(); }

   private void breakThings() { System.out.println("prrrr"); }
}

public class Mouse {
   public void runAway() { System.out.println("SQUEEK!"); }
}
```



Blank sheet - you can use this as scratch paper