**School of Computer Science**

**COMP-3220 Object-Oriented Software Analysis and Design**

**Assignment 1**

**Due: October 18, 2021 – 11:59 pm**

**100 marks – 15% of grade. Groups of 1-3 students allowed.**

**Objective:** Help students understand the types of testing, and techniques for testing programs written using object-oriented programming. Give students an opportunity to practice Java code testing using the JUnit framework.

**Tasks:**

The Lecture 3 slides describe some means and methods of unit testing in Java. In this assignment, students will write a series of tests to ensure correctness of the classes and program they develop, as well as the corresponding classes.

You will develop a small program for managing inventory and sales for a pretend store that sells books, movies, games, and CDs. After developing the program, you will then write a set of test cases for the code and report your results for testing the code. A description of the store program, a description of the test cases to write for this assignment, as well as output from a silly tester program (similar to those provided or written in COMP2120) are provided below (the tester program is just to help you understand the context in which the classes would be used and provide some basic samples of output – obviously it does not sufficiently ensure that the code you write adheres to the specs below – that’s your job! 😊 ). The tester program is provided along this assignment description on Blackboard.

1. **(10 marks, 1 for constructor and properties, one for each point below)** Develop a class called Store. This class has an ArrayList of type Item (defined below) called inventory, a double called totalSalesValue, and an int called totalNumberOfSales. totalSalesValue and totalNumberOfSales should be initialized to zero. The constructor should take nothing and only initialize the empty ArrayList inventory. The class should have the following methods:  
   1. addItem which is void, it takes an Item and adds it to inventory
   2. makeSale which returns boolean, and takes as input an int itemID, an int called quantity, and a java.time.LocalDate called sellDate. The method returns false if the item cannot be found in the store (when calling getItemByID, the item is null). Otherwise, it calls a method on item called “sell”, giving the quantity and sellDate to that method, capturing its return value called “value”. If value > 0, then it increments the stores totalNumberOfSales and totalSalesValue accordingly, and returns true. Otherwise, it returns false.
   3. getInventoryString which returns String, which iterates over the Store’s inventory and for each item adds to a String “\n” + item.toString() + “\n”. It then returns this string,
   4. listInventory which is void, which simply prints into the console “Here are all the inventory items:” followed by the output from getInventoryString.
   5. checkForItem which returns Item and takes an int called itemID. It iterates over the inventory and if it finds an Item with the given ID, it returns a copy of the item. Otherwise, it returns null. This method is public, to check if an item exists in the inventory before making a sale.
   6. getItemByID which returns Item and takes an int called itemID. It iterates over the inventory and if it finds an Item with the given ID, it returns the item. Otherwise, it returns null. Ensure that the Item’s properties can only be modified by the Store class (how?).
   7. A getter for totalSalesValue.
   8. A getter for totalNumberOfSales.
   9. A toString override, which contains “Total sales value: “, then the total sales value, “Total number of sales: “, then the total number of sales, “Current inventory”, and then the output of getInventoryString.
2. **(15 marks, 3 for constructors and properties, one for each point below)** Develop a class called Item. Item should have a String property called title, a double called regularPrice, an int called itemID, a double called discount initialized to 0, a LocalDate called releaseDate, an int called quantity, and a class variable called currentItemID, initialized to 9000.  
     
   The first constructor for Item takes a String called title, a double called regularPrice, a LocalDate called releaseDate, and an int called quantity, and initializes the respective instance variables. Quantity, however, is capped at 100 (simply check the value of quantity and reduce it to 100 if necessary). In addition, it sets the Item’s itemID to currentItemID, and increments currentItemID by 100 (currentItemID goes 9000, 9100, 9200, and so on as new items are created). The second constructor is only for copying items – in addition to the above properties, it should also take an int itemID (note: this constructor should not increment the currentItemID value!). Item has the following methods:  
   1. Getter for itemID.
   2. Getter for title.
   3. Getter for releaseDate.
   4. Getter for regularPrice.
   5. Getter for discount.
   6. Getter for quantity.
   7. getPrice, which returns a double and takes a sellDate. It computes the difference between the item’s releaseDate and sellDate (hint: use a java.time.Period), and if the item is over a year old it returns the item’s regular price, discounted by 50%, discounted further by the Item’s discount variable. Otherwise, it returns the item’s regular price discounted by the given discount.  
      \*\*note\*\* discount is a proportion of the price that should be discounted, e.g. if discount = 0.3, that means 30% off (reduce the price by 30%).
   8. setDiscount, a setter for discount, which takes a double called discount and returns a boolean. We can’t sell things for free and we don’t want to sell things for more than 50% of their regular price. Returns true if successful, false otherwise.
   9. sell, which returns a double and takes an int called amount and a LocalDate called sellDate. If sellDate is before releaseDate, the sale cannot be made so simply return 0. If amount > 0 and <= quantity, then we can sell some of this Item (reduce quantity accordingly, compute the total sales value and return it). Otherwise, we can’t make this sale so we return 0 and do nothing else.
   10. addStock, which takes an int called amount, and increases quantity of the Item accordingly, provided we end up with at most 100 of the item. Return true if successful, false otherwise.
   11. A toString override, which creates a multiline string, which on the first line contains “ITEM” followed by its ID and “:”, and then “Title: ”, “Regular Price: ”, “Release Date: ”, “Discount: ”, and “Quantity: ”, each on a separate line and one tab over, each followed by their respective properties.
   12. makeCopy, which returns a deep copy of the Item object (hint: should this be defined here? What is an Item, really? Maybe this should be deferred to concrete classes?).
3. **(6 marks, 2 for constructors and properties, one for each point below)** Develop a class called Book, which is a type of Item. In addition to Item properties, it has an ArrayList of Strings called authors. Its first constructor should take everything required for Item, but also an ArrayList of Strings to allocate to authors. Its second constructor is only for copying items – in addition to the above properties, it should also take an int itemID. It has the following methods:  
   1. getAuthors, which is a getter for a copy of the authors ArrayList.
   2. A toString override, which returns Item’s toString followed by “Authors:” on a new line and tabbed over once, followed by each author’s name on a separate line, tabbed over twice.
   3. An override for getPrice, which simply returns the regular price after applying the discount for the book. Books are timeless so we won’t discount them further because of time passing.
   4. An override for makeCopy, which returns a deep copy of this book using the second constructor mentioned above.
4. **(4 marks, 2 for constructors and properties, one for each point below)** Develop a class called CompactDisc, which is also a type of Item. In addition to Item properties, it has a String called bandName and another called recordLabel. CompactDisc has three constructors. The first takes everything required for Item, but also a String for bandName and a String for recordLabel. The second only takes a bandName, and initializes recordLabel to “Indie” by calling the first constructor. The third constructor is only for copying items – in addition to the above properties from the first constructor, it should also take an int itemID. It also has the following methods:  
   1. toString override, which returns Item’s toString followed by “Band: “, then the bandName, then “Record Label” and the recordLabel, each on their own lines and tabbed over once.
   2. makeCopy override, which returns a deep copy of this CompactDisc using the third constructor mentioned above.
5. **(6 marks, 2 for constructors and properties, one for each point below)** Develop a class called Game, which is also a type of Item. In addition to Item properties, it has a String called studio and a boolean called discontinued. It has a constructor that takes all the properties required for Item, but also a String called studio and a boolean called discontinued. The second constructor is only for copying items – in addition to the above properties from the first constructor, it should also take an int itemID. As for methods, it has:  
   1. A setter and getter for discontinued.
   2. An override for toString, which returns the toString of Item followed by “Studio: “, the studio, and then “Discontinued: “, and discontinued, each on their own lines and tabbed over once.
   3. An override for getPrice. This is because games become vintage after they are discontinued. Vintage games sell for 20 times their regular price. If the game is not discontinued, it sells for the same price as a regular Item.
   4. An override for makeCopy, which, as you can probably guess by now, returns a deep copy of this game using the second constructor mentioned above.
6. **(4 marks, 2 for constructors and properties, one for each point below)** Develop a class called Movie, another type of Item. Movies have a String for director and another for producer, in addition to the other Item properties. The first constructor for Movie needs all the Item properties as well as Strings for director and producer. The second constructor is only for copying items – in addition to the above properties from the first constructor, it should also take an int itemID. Movie has only two methods:  
   1. A toString override, which returns the toString of Item followed by “Director: ”, the director name, “Producer: “, and the producer name, each on their own lines and tabbed over once.
   2. A makeCopy override, returning a deep copy of this Movie using the second constructor mentioned above.
7. **(55 marks, two marks for each task below, 5 marks total for report containing proof of tests passing)** Now for the fun part. You must make unit tests for each of the following:
   1. Each constructor (for Store, Item, Book, CompactDisc, Game, and Movie).
      1. Item – ensure that you test the boundary cases for constructor and incrementation of currentItemID.
      2. CompactDisc – ensure that you test both constructors.
      3. All – ensure that you test the copy constructor which includes itemID.
   2. Each getPrice (one each for Item, Book, CompactDisc, Game, and Movie).
      1. Item – ensure that you test the boundary for difference between sellDate and releaseDate – if the difference is exactly 1 year, then the price should not be reduced, but if the difference is 1 year and 1 day then the price should be reduced accordingly.
      2. Book – ensure that Item’s getPrice is overridden.
      3. CompactDisc – it should be the same as Item’s getPrice – you can use the same test you created for Item.
      4. Game – ensure that you check both values for discontinued.
      5. Movie – it should be the same as Item’s getPrice – you can use the same test you created for Item.
   3. Each makeCopy (one each for Book, CompactDisc, Game, and Movie).
      1. Ensure that each is a ***deep*** copy.
   4. Store:
      1. makeSale
         1. Ensure that false is returned if Item is null.
         2. Ensure correct behaviour when there is insufficient quantity of Item.
         3. Check that totalNumberOfSales and totalSalesValue are correct after a successful sale.
      2. addItem
         1. Check the inventory before and after an addItem call – ensure that nothing occurs if Item is null.
      3. checkForItem
         1. Check both possible cases – when ID is found, and when ID is not found. Ensure that a copy of the item is returned when found.
      4. getItemByID
         1. Check both possible cases – when ID is found, and when ID is not found. Ensure that the actual item is returned when found.
      5. getTotalSalesValue
         1. Ensure correctness after making a few sales.
      6. getTotalNumberOfSales
         1. Ensure correctness after making a few sales.
   5. Item:
      1. setDiscount
         1. Check boundary cases, ensure that discount is non-zero and no greater than 0.5
      2. sell
         1. Check boundary cases for amount and quantity (check amount <= 0, amount == quantity, amount > quantity). Check boundary cases for sellDate and releaseDate.
      3. addStock
         1. Check boundary cases for amount and quantity (amount + quantity < 100, amount + quantity == 100, amount + quantity > 100).
   6. Book:
      1. getAuthors
         1. Check that the returned ArrayList is a ***deep*** copy.

You must arrange your test cases by class – that is, you should have TestStore, TestItem, TestBook, and so on, each having methods testing those listed above (testMakeSale, testAddItem, and so on). Use the techniques discussed in class to rigorously test the methods listed above. That is, if there are any decisions in your code, test all of them (see MCDC testing). This may require the use of e.g. orthogonal array testing (e.g. if there are several boolean factors) and boundary value testing (if there are boundaries or thresholds that need testing). **Provide proof of your code passing all your tests (e.g. screenshots) in a small report explaining your tests.**

**Silly tester program output:**

Here are all the inventory items:

ITEM 9000:

    Title: Into the Wilds 2: Return of the Wilds

    Regular Price: 19.99

    Release Date: 1999-11-22

    Discount: 0.0

    Quantity: 10

    Authors:

        Jim

        Julie

        Karen

ITEM 9100:

    Title: Into the Wilds

    Regular Price: 22.0

    Release Date: 1999-11-22

    Discount: 0.0

    Quantity: 10

    Director: Bob

    Producer: Joe

ITEM 9200:

    Title: The Tide, The Thief, and River's End

    Regular Price: 40.0

    Release Date: 2013-10-04

    Discount: 0.0

    Quantity: 100

    Band: Caligula's Horse

    Record Label: Indie

ITEM 9300:

    Title: Rise Radiant

    Regular Price: 50.0

    Release Date: 2020-05-22

    Discount: 0.0

    Quantity: 100

    Band: Caligula's Horse

    Record Label: Inside Out Music

ITEM 9400:

    Title: Halo 7: Even More Infinite - Diamond Edition

    Regular Price: 1099.99

    Release Date: 2026-01-17

    Discount: 0.0

    Quantity: 100

    Studio: Bungie

    Discontinued: false

Found item: ITEM 9000:

    Title: Into the Wilds 2: Return of the Wilds

    Regular Price: 19.99

    Release Date: 1999-11-22

    Discount: 0.0

    Quantity: 10

    Authors:

        Jim

        Julie

        Karen

Found item: ITEM 9100:

    Title: Into the Wilds

    Regular Price: 22.0

    Release Date: 1999-11-22

    Discount: 0.0

    Quantity: 10

    Director: Bob

    Producer: Joe

Found item: ITEM 9200:

    Title: The Tide, The Thief, and River's End

    Regular Price: 40.0

    Release Date: 2013-10-04

    Discount: 0.0

    Quantity: 100

    Band: Caligula's Horse

    Record Label: Indie

Found item: ITEM 9300:

    Title: Rise Radiant

    Regular Price: 50.0

    Release Date: 2020-05-22

    Discount: 0.0

    Quantity: 100

    Band: Caligula's Horse

    Record Label: Inside Out Music

Item with ID 9350 not found.

Found item: ITEM 9400:

    Title: Halo 7: Even More Infinite - Diamond Edition

    Regular Price: 1099.99

    Release Date: 2026-01-17

    Discount: 0.0

    Quantity: 100

    Studio: Bungie

    Discontinued: false

Store information:

Store total sales value: 1780.0

Store total number of sales: 2

Store current inventory:

ITEM 9000:

    Title: Into the Wilds 2: Return of the Wilds

    Regular Price: 19.99

    Release Date: 1999-11-22

    Discount: 0.0

    Quantity: 10

    Authors:

        Jim

        Julie

        Karen

ITEM 9100:

    Title: Into the Wilds

    Regular Price: 22.0

    Release Date: 1999-11-22

    Discount: 0.0

    Quantity: 10

    Director: Bob

    Producer: Joe

ITEM 9200:

    Title: The Tide, The Thief, and River's End

    Regular Price: 40.0

    Release Date: 2013-10-04

    Discount: 0.0

    Quantity: 76

    Band: Caligula's Horse

    Record Label: Indie

ITEM 9300:

    Title: Rise Radiant

    Regular Price: 50.0

    Release Date: 2020-05-22

    Discount: 0.0

    Quantity: 74

    Band: Caligula's Horse

    Record Label: Inside Out Music

ITEM 9400:

    Title: Halo 7: Even More Infinite - Diamond Edition

    Regular Price: 1099.99

    Release Date: 2026-01-17

    Discount: 0.0

    Quantity: 100

    Studio: Bungie

    Discontinued: false