Thank you!

SYSC2004C
Object Oriented Software Development

Grade / ???

Milestone 3

Date Due: 21-03-2021

Time Due: Before 20:00 EDT

Student: Gabriel Benni K. Evensen [101119814]

Student: Ngo Huu Gia Bao [101163137]

- The InventoryTest class
- The StoreManagerTest class
- The ShoppingCartTest class

---

1. **What were the testing methods/strategies that you used for this milestone? Be detailed. Use the testing terminology presented in lectures.**

   (a) **Exception Handling**
   We used **try-catch** block for the Exception Handling.

   First, when **choosing the StoreView**, if the user input a String or any invalid input, the program will catch the error and return a message to the user.

   Second, in the **addDisplay** method. The method first will prompt the user to enter the ID of the item which they wish to purchase, followed by the amount they want to purchase. If the user enters the invalid ID, or the amount (ex: String), the method will catch that error and return a message to the users.

   Third, in the **removeDisplay** method. The method first will prompt the user to enter the ID of the item which they wish to remove from their cart, followed by the amount they want to remove. If the user enters the invalid ID or the amount (ex: String), the method will catch that error and return a message to the users.

   (a) **Testing Methods**
   We used **AssertEquals** for testing methods in the Inventory and StoreManager class. We hard coded the expected value and compared it to the actual value which is called in the methods we want to test.

   i. **Inventory class**

   We have three testing methods for this class. First, **testGetAvailableID** which will check if the **getAvailableID** method returns the available ID of the products.

   Second, **testGettingQuantity** will check the if **gettingQuantity** method returns the correct quantity of the products when the user input the ID of the product.

   Finally, **testRemovingQuantity** will check if the **removingQuantity** method removes the quantity of the product with the given ID and the amount of the product.

   i. **StoreManager class**

   We have five testing methods for this class. First, **testAssignNewCartID** which will check if the **assignNewCartID** method increments the cartID by one and store it as a key, and the ShoppingCart object as a value inside the HashMap.

   Second, **testRemoveCartInventory** will check the if **removeCartInventory** method adds the

wanted amount to the customer cart and remove the correspond amount from the inventory.

Third, **testAddCartInventory** will check if the **addCartInventory** method removes the quantity of the product with the given ID and the amount from the customer cart and adds them back to the inventory.
Fourth, **testProcessTransaction** will compare the expected string with the returning actual string of the **processTransaction** method.

Finally, **testEmptyCustomerCart** will check if the **emptyCustomerCart** method will remove all the items inside the customer cart and put them back to the inventory.

2. **Were there some things you were unable to test with JUnit? What were they, and why were you not able to? (Think about the levels of testing.)**

   The methods inside the StoreView class were not tested as in large they were statements for printing. Not only would adding test cases for these methods introduce a lot of overhead, but it could also lead to reuse-ability problems down the road as the print statements are prone to change. However, the methods that were called (from other classes) inside StoreView have all been tested. Moreover, testing print statements does not typically produce anything meaningful.

   We did not test setters and getters as well. Due to the fact that those methods are straightforward and syntax.

3. **With respect to Question 2, should these parts of the code be tested? Should every inch of code be tested in general?**

   Those parts should not be tested as those tests are redundant. Epanding, testing print statements is not important in JUnit testing as the test would not throw an error if something as insignificant as a whitespace, or a newline, were misplaced. Moreover, testing print statements does not prove or disprove our code correctness. Finally, there is not any benefit to test setters and getters as they are straightforward and syntax. In general, not every inch of code should be tested because some of the codes neither prove or disprove our codes correctness, but rather add overhead.

4. **Change Log:**

   (a) **MILESTONE 1 CHANGELOG:- Kindly find the milestone two log below, dearest TA.**

      i. StoreManager Log:-

      For the **StoreManager** we chose to make *checkStock void* since having to manually type `System.out.println(...` each time you wish to check the stock could get tiring, very quickly. Additionally, the wording of the guideline makes it seem as this is a method the user would call when they wish to check how much of an item is left in the stock, thus this is the easier implementation for them.

      We chose to make the *processTransaction* method have a parameter consisting of a 2D array. This 2D array is then converted into a hashmap (of the same length as the array) using one of the iterator methods learned in Lab 3, and simple array operations. The variable *total* keeps track of the cumulative running total price (in an unspecified dollar currency) of the items so far. We use a simply `if` statement to check if there is adequate wares for the transaction to take place, **iff** there is, then the *total* will be incremented.

      ii. Inventory Log:-

      We chose to declare and initialize products in the *Inventory* constructor for now. Although, we are hoping to eventually be able to add products from the terminal, and have them dynamically added to the inventory. This implementation plan would ideally not change our StoreManager class much.

      For *gettingQuantity* we decided to return an int since we are returning the number of items within the inventory. Using the knowledge gained from Lab 3, we have implemented an iterator, that looks for a specific key, within the keyset. If a match is found, we get the quantity, and return it. If there is not a match found, we simple `return -1`.

      iii. HashMap:-

      We decided to choose HashMap because HashMap can store different type of values which can be easily to access and get. In addition, we do not need to looping through the entire HashMap like ArrayList to find a specific value since there is the support syntax **.containsKey()**. Finally, we can

add new items to the HashMap effective and quick with the syntax **.put(key,value)**

(b) **MILESTONE 2 CHANGELOG:-**

i. StoreManager Log:-

We chose to add the method **assignNewCartID** which is responsible for the designating each new cart with a distinct ID. The way by which we created distinct ID's is described below.

We chose to add **removeCartInventory** which is responsible for removing items from a shopping cart; of a designated quantity.

We **moved the main** to our StoreView.java class as is outlined in the milestone two handout. It does also make sense to this do this as StoreView might be considered our class of the greatest scope.

ii. Inventory Log:-

We chose to add **getAvailableID** method when we need a full Set of Integers of our keys in the infoProduct attribute.

iii. ShoppingCart Log:-

We chose to have an Integer ID for each shopping cart instance. We initialized this variable to zero; upon calling the constructor it is incremented.

We chose to implement **addCustomerProduct** method which takes two Integers; an ID, an amount and an Inventory object. This method is responsible for handling a customers request to add items to their shopping cart. This method must check if the amount that the customer wishes to add is greater than the current available amount; if so an error is printed.

We chose to implement **removeCustomerPrducts** method which takes two Integers; an ID and an amount. This method is responsible for handling a customers request to remove items from their shopping cart. This method must check if the amount that the customer wishes to remove is lesser than the current available amount; if so an error is printed.

iv. StoreView Log:-

We chose to implement **helpDisplay** such as to help with displaying the different options when a user tupes "help".

We chose to implement **browseDisplay**, which serves as the GUI when a user makes the decision to browse the different products. It lists all products; their ID's; their respective names; and their price per unit.

We chose to implement **addDisplay**, which serves as the GUI when a user makes the decision to purchase an item (i.e., add an item to their shopping cart). It lists all the available products, and their respective information. It also prompts the user to enter the ID of the item which they wish to purchase, followed by the amount they which to purchase.

We chose to implement **removeDisplay**, which serves as the GUI when a user makes the decision to return an item (i.e., remove an item from their shopping cart). It prompts the user to enter the ID of the item they wish to return; and the amount they wish to return.

We chose to implement **displayUI** to navigate, according to the user's input, through our different methods, and store.

5. **MILESTONE 3 CHANGELOG:-**

(a) ShoppingCart Log:-

We changed the **addCustomerProduct** method which now takes two Integers; an ID and an amount. The method used to take two Integers; an ID, an amount and an Inventory object. The purpose is the Shopping Cart and the Inventory class will be independent and the managing of the Store Manager class will be thoroughly. The function of the method is still the same. This method is responsible for handling a customers request to add items to their shopping cart. This method must check if the amount that the customer wishes to add is greater than the current available amount; if so an error is printed.

(b) StoreView Log:-

We added the handling error in the **addDisplay** method. The method first will prompt the user to enter the ID of the item which they wish to purchase, followed by the amount they want to purchase. If the user enters the invalid ID or the amount (ex: String), the method will catch that error and return a message to the users.

We added the handling error in the **removeDisplay** method. The method first will prompt the user to enter the ID of the item which they wish to remove from their cart, followed by the amount they want to remove. If the user enters the invalid ID or the amount (ex: String), the method will catch that error and return a message to the users.

We added one more if statement in the **displayUI** to check if the user enters invalid command, the support message will be printed out.

(c) Testing Methods

i. **Inventory class**
We have three testing methods for this class.

- **testGetAvailableID** which will check if the **getAvailableID** method returns the available ID of the products.
- **testGettingQuantity** will check the if **gettingQuantity** method returns the correct quantity of the products when the user input the ID of the product.
- **testRemovingQuantity** will check if the **removingQuantity** method removes the quantity of the product with the given ID and the amount of the product.

i. **StoreManager class**
We have five testing methods for this class.

- **testAssignNewCartID** which will check if the **assignNewCartID** method increment the cartID by one and store it as a key, and the Shooping Cart object as a value inside the HashMap.
- **testRemoveCartInventory** will check the if **removeCartInventory** method adds the wanted amount to the customer cart and remove the correspond amount from the inventory.
- **testAddCartInventory** will check if the **addCartInventory** method removes the quantity of the product with the given ID and the amount from the customer cart and adds them back to the inventory.
- **testProcessTransaction** will compare the expected string with the returning actual string of the **processTransaction** method.
- **testEmptyCustomerCart** will check if the **emptyCustomerCart** method will remove all the items inside the customer cart and put them back to the inventory.