Milestone 3
Questions

Bardia Parmoun 101143006

Guy Morgenshtern 101151430

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

Using JUnit 5, testing was split into individual methods and also into individual method cases. For each edge case of a function, a separate test function was created to specifically test the outcome of that case. Edge cases were determined by looking at the extreme cases of each function.

In order to save time and repeated code when testing the StoreManager classes we decided to take a sequential approach and have the tests build up on each other. This also helps simulate an actual instance of using the program since most of the methods that deal with the shopping carts often build up on each other.

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.)

We were not able to test every function with JUnit as some of the functions dealt with user inputs and as a result unit testing will not be an efficient approach when it comes to dealing with those functions. Another group of functions that could not be tested were those that were private and as a result the unit testing class would not have to access to those functions. Finally, in this testing process only the StoreManager and the Inventory classes were tested. These classes make up the majority of the methods in the project and other classes often rely on them. As a result testing only these two classes should give a good understanding of whether the entire program works properly. It is also worth noting that many of the methods that are created in the other programs are also used in these two classes and by testing these classes we are also testing the functionality of those methods.

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

The parts above should not be tested the same way the unit tests were conducted. User inputs can be scrubbed using try catch and error handling. Private methods are used internally and thus are tested when testing the public methods that employ them. Since unit testing provides tests on function output, private tests are not always needed to be tested because their output will not be seen. Many other sections of code are either very basic one-liners (accessor and modifier methods) are methods/classes that are used within others. This means that not all parts of code need to be tested.

Changelog

Product

- The method “equals” and “hashCode” were added so the object instances can be easily compared

Inventory

- Constructor chaining was used for the two constructors in Inventory.
- Conditions were added to addStock and removeStock to make sure negative stock would not change product stock

StoreManager

- The variable “id_count” was changed to “idCounter” to follow the camelCase conventions of Java.

StoreView

- Try Catch statements were added for every instance that the user is prompted for error handling to avoid sudden errors

StoreManagerTest

- This class was added to the test the methods of StoreManager
- To avoid repeated code the testing for class was done sequentially so functions build up from each other.
- The init method initializes an array of carts. It also creates 6 sample products which can be used for the methods.
- testCheckStock method is used to check if the checkStock method is working properly and it matches the expected values.
- testGetAvailableProducts is used to check the available products in the inventory are correct.
- testAssignNewCartID checks to see if the carts were initialized properly
- testAddToCartNormal checks to see if the addToCart method works properly for valid values
- testGetCartProducts checks to see if the carts that are returned from cart are the same as what they are expected.
- testAddToCartProductNotThere checks to see if the addToCart method works properly if the product does not exist in the inventory
- testAddToCartIllegalAmount checks to see if the addToCart method works with edge values such as 0, -1, more than the total amount
- testAddToCartTotalAmount checks to see if the addToCart method works properly when the total amount of a certain product is ordered.
- testRemoveFromCartNormal checks to see if the removeFromCart method works properly for valid values

- testRemoveFromCartProductNotThere checks to see if the removeFromCart method works properly if the product does not exist in the inventory
- testRemoveFromCartIllegalAmount checks to see if the removeFromCart method works with edge values such as 0, -1, more than the total amount
- testRemoveFromCart TotalAmount checks to see if the removeFromCart method works properly when the total amount of a product is removed from the cart.
- testGetCartTotalPrice is used to check to see if the getCartTotalPrice works properly
- testEmptyCart is used to empty the cart once the cart is empty
- testCheckout is used to checkout the cart once the cart is done. It adds up the total price and removes the cart once it is done.

InventoryTest

- Global variables
    - inv - Inventory object for testing
    - initialProducts - HashMap<Product, Integer> for initializing inv and comparisons in testing
    - invalidID - a Product ID that is never used in the test Inventory, used for testing invalid ID functions (getProductInfoProductDoesntExist, getStockProductDoesn'tExist, etc..)
- setUp initializes a new inventory with products after every test to ensure tests don't interfere with eachother
- tearDown notifies user that the test is complete
- testGetProductInfoProductDoesntExists checks to see if getProductInfo works properly if the product isn't in the inventory
- testGetProductInfoProductExists checks to see if getProductInfo works properly when the Product is in the inventory
- testGetStockNoProductExists checks if getStock works if the Product is not in the inventory
- testGetStockProductExists checks if getStock works if the Product is in the inventory
- testAddNegativeStock checks if addStock works properly when called with negative stock
- testAddStockNoProductExists checks if addStock creates a new product when called with a product that does not exist in inventory yet
- testAddStockProductExists checks if the addStock works under normal conditions (non negative stock, product that exists)
- testRemoveNegativeStock checks to see if removeStock works properly when called with a negative new stock argument
- testRemoveStockNoProductExists checks if removeStock works properly when the product in the argument isn't in inventory

- testRemoveTooMuchStock checks if removeStock works properly if more stock is asked to be removed than currently available for that product
- testRemoveAllStockAvailable checks if removeStock works properly when all of a Product’s stock is removed
- testRemoveStockProductExists checks if removeStock works properly when called under normal conditions (non negative stock that is less than available stock, product exists)