Rajan Saini

**Prototype Kiosk System Design Choices & Afterthoughts**

**Scanner Class**

A simple class that just simulates a real scanner. If this application were to be extended I imagine it would need to be able to decipher barcode information and match it to products in the database.

The input of the program was done through a list of strings that represented scanned products. Thus I decided to represent the “Main” entry point to the program through the Scanner class. It feeds the users input file into a new checkout session which will compute and display the cost. It continuously asks the user for an input file until the user types ‘q’ to quit.

**Assumptions**:

* Because of how the input file represents items that were scanned at a kiosk, I assume items that are not currently in the databases inventory will not be able to be scanned
  + I did not perform tests to check if there is an invalid item in the input file

**Checkout Class**

This class has the responsibility of taking in products “*scanned*” from the Scanner class, and building an itemized receipt. Since the input file is not sorted, Items are added into a dictionary <String, int>. This helps keep track of the products (String) in the cart and how many of each there are (int). This dictionary acts as an electronic cart which makes it easier to apply promotions that are based on the amount of items bought. The alternative would be to calculate costs and apply promotions as each item is scanned, which I decided not to do.

Checkout also checks if there is a promotion available to any of the products and attempts to apply the promotional discount if the requirements are met. Typically requirements are based around the quantity of the item in the cart.

While designing how this class should compute its information, I considered what sort of extensions might require me to modify code in this class. An example comes to mind: Superstores have a promotion where when you spend a certain amount in total on groceries, you receive a specific product for free. If I were to implement that promotion, then I would need to refactor this class. Specifically, I need to add a getter function for the total cost.

**Catalog Class**

The catalog class should not actually be implemented the way it is.

“Each time a checkout transaction begins, the system should utilize the latest pricing rules (both for regular prices and promotions). These can be read in from files.”

Due to the file reading nature of this application I needed a class that would handle the interpretation of these files. The system would be better if catalog was implemented as a GUI system for GroceryCo Sales Administrators to update a central database that houses the Inventory and Promotional information. To simulate database modifications you can swap these text files in between checkout sessions.

Catalog houses functions that imitate SQL lookups, albeit in a very basic sense. This is so if catalog was replaced with a database, than checkout would not have to be modified too much.

**Limitation:**

* Due to how the inventory and promotional data is input using a file rather than a database, the Catalog’s functionality revolves around keeping track of the inventory and promotion data
* At the start of a checkout session, Catalog updates its current information to match whatever is in the current “*database*” (txt file)
* Database updates are represented by swapping out the “***inventory.txt***” and “***promotions.txt***” files in the \ConsoleApplication1\bin\Debug\Data Folder

**Promotion Class**

Promotions for this system all have the same purpose but work towards this goal in different ways. That purpose is to reduce the total cost off of a cart. As such I decided to make promotion an interface. When a new promotional idea is made it can extend the original interface and implement its own unique functionality to reduce the price based on its requirements. The PromotionFactory class builds these different promotions based on a type code.

Some promotions require knowledge of how much of the specified product exists in the cart and some simply require that the item exists in the cart. So I considered making two types of promotion interfaces but I thought this would be bad design. Each promotion has a singular function to fulfill, so there is nothing fundamentally different about them. Thus I decided to redefine a promotion as something that should reduce the total cost based on the contents of the electronic cart.

It’s not currently implemented this way as it wouldn’t be a significant change to functionality and this is more about looking back on the project after completion. However, upon introducing new promotions this change might be vital. “*Superstores have a promotion where when you spend a certain amount in total on groceries, you receive a specific product for free.*” This promotion in particular would not currently work with the promotional interface unless I refactor the promotion interface the way I stated in the paragraph above.

**Assumption:** Most stores don’t allow a “*group*” or “*additional*” promotion to be applied more than once. Since it was not specified in the requirements I didn’t implement this functionality but I had definitely considered the implications of adding this in the future and how the code should be structured so that it is an easy extension to make

**S.O.L.I.D Principles**

**Single Responsibility Principle**

* Each class holds its own single responsibility in the kiosk system
  + Scanner simulates a list of scanned items
  + Checkout computes the total and generates the receipt
  + Catalog simulates database look ups
  + Promotion decreases the total cost based on contents of the cart
* There is a slight overlap however: Promotion also has a hand in generating the receipt
  + Promotions that are successful print a special message to the receipt explaining why there is a reduction in price
  + FIX: if promotions were changed to make modifications to an electronic cart passed in as a parameter it could modify the total cost and return a String for checkout to print to the receipt this overlap would be avoided

**Open-Closed Principle**

* I kept this principle in mind as I developed checkout and promotion.
  + Checkout: the use of a Cart struct or object for determining promotions would mean that all information is readily available for the varying promotions Theoretically closing this application to further modification
  + Promotion: the base promotion interface represents a core functionality that every promotion should be able to perform. Future promotions add their own functionality in their own class files. So the interface is closed

**Liskov Substitution Principle**

* Promotion is the only class that extends a base class
* At this moment LSP would break upon adding promotions that require knowledge of the total cost of a cart
  + This is why modifying the base interface of promotion so that an electronic cart is the parameter for its functions rather than a single Integer amount seems like such a good idea to me. All promotions (that I can think of) would satisfy that requirement

Interface Segregation and Dependency Inversion sound like they would be important for more complicated applications. Upon reading into these principles, I suspect this assignment did not need to consider these principles. I understand that SOLID principles can and certainly will always break but it serves as good practice for software developers to create code that exemplifies good Object Orientated Characteristics