Rigoberto Rosa

CS520 Term Project – Fall 2022

Package Inventory

Our TotesList class is the data structure that will store all the items in totes. The TotesList class has a helper class of Tote it is here where we define what a tote is this encapsulation allows us to say how many items this tote will have and how they would be stored I decided to use an ArrayList to store the items. My thought process was using an array here can lead to performance issues down the road for example if a tote can hold 100K items and item XXX is usually always half full we wouldn’t want to adjust the size of all totes and we wouldn’t want to dictate the size of every tote on creation this can become difficult to manage by using an ArrayList we can call the trimToSize() method on the specific tote. The items get added to the tote in the order they came in so all NEWER inventory will be added to the back of the ArrayList. Removal of an item is done from the beginning of the ArrayList. I felt this would mimic a warehouse situation the best.

The ToteList is a class that has an ArrayList of Totes this class keeps track of all the partially filled and full totes. It also keeps track of when to create a new tote by checking if the list size is zero create a new tote or if the last tote in the list is full create a new tote. All items get added to the last tote in the list again to mimic a real-life situation the first totes in the ArrayList should contain the oldest inventory. It was a requirement for the removal of an item to select a random tote so on removal, we select a random tote using the Random class once we have the tote, we remove the FIRST item that was stored in the tote not validating our FIFO approach. Since we take a random tote, this means that some totes will be partially filled so we have a method to merge the totes. This works using a sliding window approach we first check if <=1 tote if so, no work needs to be done. If that isn’t the case, we start from the first tote in the list checking the adjacent tote. If the first tote IS NOT full, we go into a while loop saying take the first item in the adjacent tote and add it to the first tote until it is full this should keep our FIFO approach intact then move the next two totes.

EXAMEPLE Tote1 [ i1,i2,i3] , Tote2[i4,i5,i6] 🡪 Tote1[i1,i2,i3,i4,i5,i6]

The inventory class handles all the logic of adding new inventory or removing inventory. It contains a variable itemsMap of type Map that is a HashMap to a <UpcCode: String, TotesList>. So, each UPC code has its own list of totes that it needs to worry about. This allows us to say get me the number of partially filled totes for UPC code XXX or fulfill orders from UPC code XXX.

Package Dao (Data Access Object)

Item class is a simple representation of how an item is represented in our txt file and will allow us to add any changes in the future with minimal effect on the rest of the code.

This is the same for our Order class this is a simple representation of an order based on how it is being read from the text file. Any future changes to the order or any additional attributes can be added to the class with minimal effect on the code. I created the UPC class that is a representation of an upc code this was based on my research of the GTIN-12 which is primarily used in North America[1] and validated the upccode using the check digit. [2] However, even though I created this class I didn’t use it in any of the code I felt it was a bit overkill for the current requirements of the project. I did keep it in the project to be used in the future if necessary. Instead, I created a UPCValidator which would just validate the UPC code being read from the text file.

Package Reader

The package reader contains two files WarehousePicker gets all the orders from the text file and WarehouseReciever gets all the items to load into inventory again encapsulating each of the processes. Both classes just return an iterator, and the file is read into memory. No work is done on the data except for validation.

Package Constants

This package just contains one file called FileConstants that keeps track of where our inventory file is stored and the order file. The number of total items in a tote as well as the maximum number of totes in the warehouse allows us to quickly update the requirements with minimal change.

Package Warehouse

Contains an interface IWarehouse that is implemented by Warehouse. The warehouse class that I implemented is only to print to the console. I felt that this would be the best way to encapsulate it. It has no idea how the inventory gets added or what is in inventory it pretty much tells the other classes do this or get me that and displays it to the console.

Package Resource

Contains the required files for the items and orders. It also contains a file called exceptions which is where we will write any exceptions for UPC Code validations or any IO exceptions that occurred. I decided to try and write this in a JSON style format mainly for two reasons the first being my knowledge and working with JSON is very limited and wanted to try and get a bit familiar with it and the second is I felt down the road we could use the ObjectMapper class[4] to extract the data.

Finally, the uml files were created using Visual Paradigm reverse engineering they offer a 30 free trial.[3]

Appendix

[1] [Gs1 Upc Barcodes](https://www.barcode-us.info/upc-codes/)

[2] [How To Calculate a Upc Check Digit](https://azaleabarcodes.com/white-papers/upc-barcode-check-digit/)

[3] [Visual Paradigm](https://www.visual-paradigm.com/)

[4] [ObjectMapper](https://fasterxml.github.io/jackson-databind/javadoc/2.7/com/fasterxml/jackson/databind/ObjectMapper.html)