Grocery Store Simulator 2014 - Part 2

Due: March 28th, 2014 @ 5pm

Backstory

After success with their latest simulation game "The Jims 3", BA Games is at it again. You've been hired to develop a prototype simulator for this year's most anticipated Grocery Store Simulation game creatively named "Grocery Store Simulator 2014". Will you rise to the challenge?

Files to complete

You are expected write an implementation for each of the interfaces listed in the classes presented in the **config** package provided. As with the last assignment, you must specify which implementation you would like us to grade in this file.

In addition to this, you **MUST** include a String field in the **config.Configuration** file with your UMass student ID number. This is the 8 digit value found on your student ID card and is used to identify you in moodle. This will allow us to get your grade and feedback to you quickly. We recommend that you enter this ID value immediately after importing the project.

It will almost certainly be useful for you to write additional class files. Any class file you write that is used by your solution **MUST** be in the provided **src** folder. When we test your assignment, all files not included in the **src** folder will be ignored.

Note: When you submit your solution, be sure to remove **ALL** compilation errors from your project. Any compilation errors in your project may cause the autograder to fail and you will receive a zero for your submission.

Test files

In the test folder, you are provided with several JUnit test cases that will help you keep on track while completing this assignment. We recommend you run the tests often and use them as a checklist of things to do next. You are not allowed to remove anything from these files. If you have errors in these files, it means the structure of the files found in the **src** folder have been altered in a way that will cause your submission to lose points. We highly recommend that you add new @Test cases to these files. However, before submitting, make sure that your program compiles with the original test folder provided.

Support Code API

The Support Code's comments have been generated into a nicely formatted API that can be found here: http://people.cs.umass.edu/~jcollard/cs187/s14/assignments/6/doc/

It is highly recommended that you spend a day simply reading over the comments in each of the interfaces and classes provided.

Part One: Importing Project into Eclipse

Begin by downloading the provided project from Moodle and importing it into your workspace.

You should now have a project called **grocery-simulator-part2-student** it is very important that you do not rename this project as it is used during the grading process. If the project is renamed, your assignment may not be graded.

By default, your project should have no errors and contain the following root items:
src - The source folder where all code you are submitting must go. You can change anything you want in this folder, you can add new files, etc...
support - This folder contains support code that we encourage you to use (and must be used to pass certain tests). You are not allowed to change or add anything in this folder
test - The test folder where all of the public unit tests are available
JUnit 4 - A library that is used to run the test programs
JRE System Library - This is what allows java to run

If you are missing any of the above or errors are present in the project, seek help immediately so you can get started on the project right away.

Once you have your project imported, add your 8 digit student ID number to the **config.Configuration** class.

Part Two: Implement a SimpleRegister class

For this part of the assignment, you will familiarize yourself with the **simulator.checkout.AbstractRegister** class.

This class facilitates creating Transactions for Shoppers who visit a Grocery Store. To implement this class, you can simply extend it and implement the abstract method **createTransaction(Shopper s)**. Later in the assignment, you will need to create a more sophisticated algorithm for checking out a customer. However, for this portion you should simply generate a Transaction with the following properties:

- A Receipt with all of the shoppers groceries and no discount.
- The Shopper passed to the createTransaction method.
- If the number of grocery items being sold is 0, the checkout time should be 1. Otherwise, the amount of time to process a shopper should be 4 multiplied by the number of grocery items being sold.

Part Three: Implement a SimpleStore class

For this part of the assignment, you will familiarize yourself with the **simulator.store.AbstractGroceryStore** class. Your SimpleStore implementation must meet the following requirements:

- Utilizes 2 SimpleRegisters that are always turned on.
- Uses 1 Normal Line
- Uses 1 Express Line

The implementation details for this class might seem daunting at first. Below is a suggested order for implementing them and hints on how you might do it.

SimpleStore Constructor

In the constructor for your SimpleStore, you should create 2 SimpleRegisters and call the turnOn() method for both of them. The turnOn() method ensures that the registers can process shoppers.

In your constructor, you should create a List<CheckoutLineInterface> and add a NormalLine and ExpressLine to it.

getLines()

This method should return the List you created in your constructor. This line is handed off to the BigBrother object. BigBrother will use this to direct customers to your store.

tick()

This method is called by BigBrother during the simulation to advance the simulation one time step. When this method is called, you will want to check to see if each of your registers isBusy(). If they are not, you will want to try and process a shopper. You can do this by checking if your lines are isEmpty() and then dequeue() the next Shopper to a register to be processed.

Hint: Use two ArrayLists whose indices associate each register to a single line.

getTransactions()

The AbstractRegister tracks all of the Transactions it creates. You should utilize this to make your life easier. List has an **addAll()** method that might be useful!

getAverageWaitingTime()

This should return the average waiting time per shopper. This is the sum of **getWaitingTime()** for each **Shopper** divided by the total number of **Shopper**s.

Hint: Use the Transactions created by your registers to get to the Shopper data.

getTotalSales()
This should return the total amount of money your store receives from all Shoppers.

Hint: This can be calculated by using the Transactions created by your registers to get to the Receipt data.

getTotalCost()
This should return the total amount of money your store spent to run. This is the sum of the cost of all groceries sold **AND** the **getRunningCost()** of each of your registers.

getTotalProfit()
Returns the total profit generated by your store.

getNumberOfShoppers()
This is the total number of Shoppers that visited your store. This should be equal to the number of transactions your store generates.

getNumberOfIrateShoppers()
This is the total number of Shoppers that visited your store that **isIrate()**.

Part Four: Implement a ProfitableStore class
Now that you've familiarized yourself with the AbstractGroceryStore class, you must implement a store that can turn a profit using the SimpleWorld. To do this, you will need to implement a more competitive Register class. In addition to this, you will almost certainly need a better strategy than the SimpleStore.

For this portion, it is recommended that you subclass your **SimpleStore** class and @Override the tick(), getLines(), and getTransactions() methods. If you have implemented the other methods in a way that utilizes **getTransactions()**, they should work for this subclass. Depending on your implementation, you may also want to @Override getTotalCost().

It is also recommended that you write your own tests to double check that your get methods are returning the way you believe they should. The **BigBrother** object collects information that will be used by the autograder to ensure that your methods are not simply fooling the test simulations.

Graded Simulations
There are four graded simulations in the **test/** directory provided. These are in the **simulator.simulations** package. You will notice that these are not unit tests. Instead, these are runnable classes containing main methods. Each of these creates a SimpleWorld specifying the rate at which customers will enter your store. After this, your ProfitableStore (as specified in the Configuration class) is selected. BigBrother ticks 43200 times running the simulation. Finally, the getTotalProfit() method of your store will be called to check if your store turned a profit. **You should NOT make your store simply return a value greater than 0, the BigBrother object collects information throughout the simulation that will be able to verify the results.**

Two tips for increasing profit

- How many irate shoppers did you have? Irate shoppers empty their shopping cart before they are dequeued. Although it is possible to make a profit by only serving some shoppers, it is highly discouraged.
- How much are your registers pushing up your running cost?
    - If you process shoppers too fast, you will not be able to make a profit. The formula for calculating the cost to check out a shopper is in the AbstractRegister class.
    - In the SimulationRate1000.java test, your registers may be idle. Perhaps you can save money by turning them off when they are not being used.

Part Five: Export and Submit

When you have finished and are ready to submit, export the entire project. Be sure that the project is named **grocery-simulator-part2-student**. Save the exported file with the zip extension (any name is fine).

Log into Moodle and submit the exported zip file.