ITMD 510 Final Project – Point of Sale Register

Due: December 11, 2015 11:59pm (**No Late Submissions**)

1 Objectives

The objectives of this MP are to demonstrate mastery of the ITMD 510 course objectives.

2 Introduction

For this MP, you will implement a point of sale register used by a clerk at a store to check out customers' orders. The clerk enters the SKU (Stock Keeping Unit) for each item in the customer's order into the application. The application calculates the total cost of the order based on the regular costs of the items and also any possible sales that might be occurring on the items.

The application obtains costs and data about sales on items via text files that are loaded when the application starts.

The application also maintains a register tape for all of the orders that have been entered into the application and stores that register tape in a file on disk.

3 SKUs

Each item for sale in the store has the following attributes:

- A **SKU** which uniquely identifies the item. The SKU is 6 capital letters, A-Z.
- A **description** of the item which is a text string. The description does not contain any newline characters or the vertical bar (|).
- A **unit cost** which is the cost for a single unit of the item. This cost is expressed in whole cents (i.e. $12.34 is 1234 cents).

The customer can buy more than one of an item. In this case, the total cost of purchasing more than one of the item is the quantity times the unit cost of that item. This is also known as the **extended cost.**

The store maintains a text file which lists all of the items for sale in the store and its attributes (the **costs file**). Each line of the text file is an item for sale. A line in the text file has the following format:

```
<SKU>|<Description>|<Unit cost>
```

where:

- `<SKU>` is the SKU of the item.

- `<Description>` is the description of the item
- `<Unit cost>` is the cost of the item

An example of a costs file listing three items could be:

```
AAAAAA|Core Java Volume I|3599
AAAAAB|Core Java Volume II|3599
APPLES|Apple|25
```

A SKU can only appear once in the costs file.

4 Sales

The store occasionally has sales on their items. There are different types of sales:

- Discount.
- Buy X Get One Free.
- Buy X For The Cost of Y.

The store maintains a text file which contains data about the current sales. The sales file has the following format:

```
<SKU>|<Sales Type>|...
```

where:

- <SKU> is the SKU of the item that is having a sale.
- <Sales Type> is a string denoting which sale type described below.
- ... is the addtional information for that sale described below.

Note that an item can only have one sale type. A SKU can only appear once in the sales file.

4.1 Discount

Discount sales are sales where there is a discount to the unit cost, expressed in a percentage off. For example, if the unit cost of an item is $1.00 and there is a 10% discount, the unit cost of the item during the sale is $0.90. This discount is applied to any quantity of items in the customer's order. In the previous example, if the customer has a quantity of 5 of the item, the extended cost is $4.50.

In the sales file, a discount sale has the following format:

```
<SKU>|DISCOUNT|<Percent>
```

where:

- <SKU> is the SKU of the item that is having a sale.
- DISCOUNT is the sales type string for discount sales.
- <Percent> is the percent discount in whole percent (i.e. 50 would mean a 50% off sale).

For example:

```
AAAAAA|DISCOUNT|50
```

describes a sale on "Core Java Volume I" with a 50% discount.

Calculating percentage discounts might result in fractional cents. In the event of fractional cents, truncate to the nearest cent. Do not round. For example, a 50% discount on a item with a unit cost of $35.99 would have a discounted unit cost of $17.99 (3599 * 0.50 = 1799.5, truncate to 1799 cents).

4.2 Buy X Get One Free

Buy X Get One Free sales are a sale where the customer must purchase at least X quantity of an item and receives one of that item at no cost. For example, apples are usually for sale at $0.25 an apple but the store is has a buy 4 get one free sale. A customer has 5 apples in his order. Therefore, the cost of the order is $1.00 where 4 of the apples cost the regular $0.25 but one apple has a cost of $0.00.

If the customer has more than X items for an item that is on sale for Buy X Get One Free, the customer receives one free item for every multiple of X. In the previous example, if the customer orders 10 apples, 8 of those apples are at the regular cost and two are free for an extended cost of $2.00 for all 10 apples.

However, if the customer does not order a full multiple of X, the remainder is at the normal unit cost. For example, if the customer orders 8 apples, 7 apples are at the regular cost and only one is free for an extended cost of $1.75 for 8 apples. If the customer orders 9 apples, 8 of the apples are at the regular cost and only one is free. The customer loses out on that second free apple. If the customer has 12 apples, 10 of those apples are at the regular unit cost ($0.25 x 10 = $2.50) and two are free.

If the customer has less than X, all of the items are at normal unit cost.

In the sales file, a Buy X Get One Free sale has the following format:

```
<SKU>|BUYXGET1FREE|<X>
```

where:

- <SKU> is the SKU of the item that is having a sale.
- BUYXGET1FREE is the sales type string for Buy X Get One Free
- <X> is the number of items the customer must purchase in order to get one free.

For example:

```
APPLES|BUYXGET1FREE|4
```

describe a sale on apples where if the customer buys 4 apples, he gets one for free.

4.3 Buy X For the Cost of Y

Buy X for the Cost of Y sales are a sale where the customer must purchase at least X quantity of an item but the cost of the X items is at the cost of Y items where Y is less than X. For example, apples are on sale for buy 5 apples for the price of 4. The unit cost of an apple is $0.25. If a customer has 5 apples in his order, the extended cost of the apples is $1.00.

The sale is only valid for multiples of X. For example, if the customer has 10 apples in his order, the extended cost is $2.00. However, if the customer has 8 apples in his order, 5 of those apples cost the same as 4 apples ($1.00) but the remaining 3 apples are at the regular unit cost ($0.25 x 3 = $0.75) for a extended cost of $1.75.

If the customer has less than X, all of the items are at normal unit cost.

In the sales file, a Buy X for the Cost of Y sale has the following format:

```
<SKU>|BUYXFORY|<X>|<Y>
```

where:

- <SKU> is the SKU of the item that is having a sale.
- BUYXFORY is the sales type string for Buy X for the Cost of Y.
- <X> is the number of items the customer must purchase to get the discount.
- <Y> is the number of items to use in calculating the extended cost of the X items.

For example:

```
APPLES|BUYXFORY|5|4
```

describes a sale on apples where if the customer buys 5 apples, he pays the same price as 4 apples.

5 Register Tape

The register tape file is used to record all completed sales for accounting purposes. The register tape file is human readable and reflects the same register tape that is given to the customer as a receipt for his purchase. The register tape is also visible to the clerk as the customer's order is entered into the application via the application GUI. The format of the register tape is described below.

5.1 Costs

Any costs recorded in the register tape and displayed in the application should be in dollars and cents. For example, 12.34.

Dollars and cents should be zero padded to ensure that at least on digit to the left of the decimal point is display and two digits to the right of the decimal point are display. For example, 75 cents should be displayed as 0.75. 80 cents should be displayed as 0.80. One dollar should be displayed as 1.00.

5.2 Regular Sales

For each item that is not on sale, the entry on the register tape for that item has the following format:

```
<SKU> <Description> <Quantity>@<Unit Cost> = <Extended Cost>
```

where:

- <SKU> is the SKU of the item
- <Description> is the description from the costs file for the item
- <Quantity> is the number of items the customer has purchased
- <Unit Cost> is the unit cost of the item in dollars and cents formatted as described above.
- <Extended Cost> is the extended cost of the item in dollars and cents calculated and formatted as described above.

For example:

```
AAAAAA Core Java Volume I 2@35.99 = 72.98
```

If a customer orders more than one of an item at regular price, only one entry on the register tape should be made for that item with the appropriate quantity and extended cost.

5.3 Discount

For each item that has a discount sale, the entry on the register tape for that item has the following format:

```
<SKU> <Description> <Percent>% Off <Quantity>@<Discounted Unit Cost> = <Extended Cost>
```

where:

- <SKU> is the SKU of the item
- <Description> is the description from the costs file for the item
- <Percent> is the percent discount in whole percent
- <Discounted Unit Cost> is the unit cost of the item after the discount in dollars and cents formatted as described above.
- <Extended Cost> is the extended cost of the item in dollars and cents calculated and formatted as described above.

For example:

```
AAAAAA Core Java Volume I 50% off 1@17.99 = 17.99
```

5.4 Buy X Get One Free

For each item that has a Buy X Get One Free sale, there are multiple entries made on the register tape. For each multiple of X, there are two entries made on the tape. The first entry is for the cost of X items displayed the same as in the regular sale format. The second entry is for the free item displayed as follows:

```
<SKU> <Description> 1@0.00 = 0.00
```

where:

- <SKU> is the SKU of the item
- <Description> is the description from the costs file for the item

If the customer does not purchase an exact multiple of X, the remainder is displayed on the register tape in the same format as a regular sale.

For example, there is a buy 4 apples get one free sale. The customer purchases 12 apples. The register tape should display the following:

```
APPLES Apple 4@0.25 = 1.00
APPLES Apple 1@0.00 = 0.00
APPLES Apple 4@0.25 = 1.00
APPLES Apple 1@0.00 = 0.00
APPLES Apple 2@0.25 = 0.50
```

5.5 Buy X For the Cost of Y

For each item that has a Buy X for the Cost of Y sale, there are multiple entries potentially made on the register tape. For each multiple of X, an entry is made for the discounted price. The entry is displayed as follows:

```
<SKU> <Description> <X> for <Y>@<Unit Cost> = <Extended Cost>
```

where:

- <SKU> is the SKU of the item
- <Description> is the description from the costs file for the item
- <X> is the number of items required for the discount
- <Y> is the number of items to use in calculating the extended cost of the X items.
- <Unit Cost> is the unit cost of the item in dollars and cents formatted as described above.
- <Extended Cost> is the extended cost of the item in dollars and cents calculated and formatted as described above.

If the customer does not purchase an exact multiple of X, the remainder is displayed on the register tape int the same format as a regular sale.

For example, there is a buy 5 apples for the cost of 4. The customer purchases 8 apples. The register tape should display the following:

```
APPLES Apple 5 for 4@0.25 = 1.00
APPLES Apple 3@0.25 = 0.75
```

5.6 Order Completion

The application will record in the register tape when the customer's order is completed. This entry includes the total cost of the order and a separator that separates one order from another. The record has the following format:

```
Total Items: <Items>
Total Cost: $<Cost>
+-----------------------------------------------------------+
```

where:

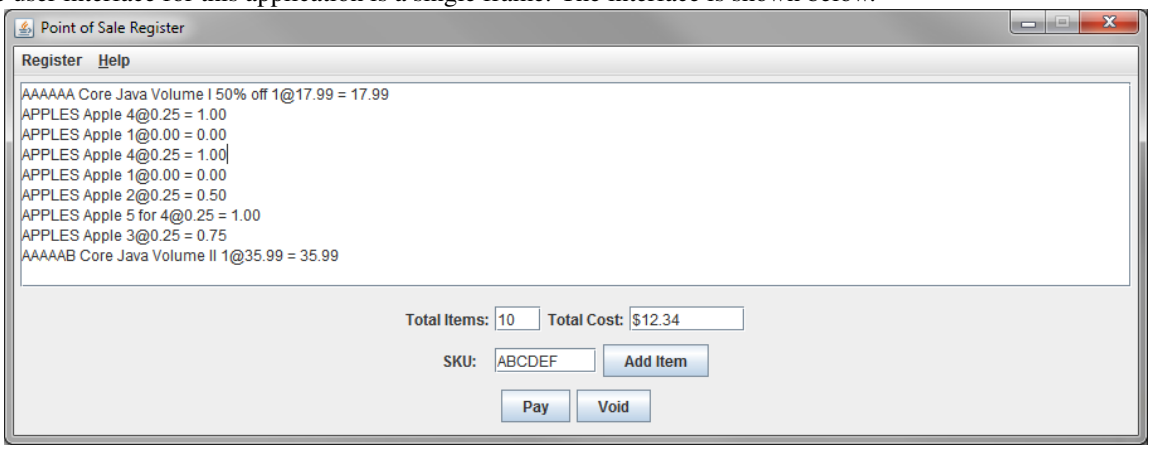
- <Items> is the total number of items in the order.
- <Cost> is the total cost of all of the items in the order calculated and formatted as described above.

For example, an order that includes 8 apples and a copy of "Core Java Volume I" might look like the following:

```
Total Items: 9
Total Cost: $37.99
+-----------------------------------------------------------+
```

6 User Interface

The user interface for this application is a single frame. The interface is shown below.



The user interface has the follow components:

- A menu bar
- Register tape.
- The register controls

6.1 Menu Bar

The menu bar contains two menus, Register and Help.

The Register menu has only one menu item, “Exit” which when selected exits the application.

The Help menu has one menu item called “About”. Selecting this menu item should show a pop-up dialog with the following text:

- Your Name
- Your Student ID Number

The About dialog should have an “OK” button which when pressed dismisses the dialog.

6.2 Register Tape

The register tape component is a scrollable, read-only text area that displays the register tape for the current order. The register tape should be updated for every new SKU that is entered by the clerk. The format of the register tape is described in the previous section. The register tape should be at least 10 rows by 80 columns.

6.3 Register Controls

The register controls component has the following text components:

- Total Items – This read-only text field displays the total number of items in the customer’s order.

- Total Cost – This read-only text field displays the total cost of the customer's order, displaying dollars and cents in the format described above.
- SKU – This editable text field is used by the clerk to enter the SKU of an item in the customer's order.

The register controls component has the following buttons:

- Add Item – When this button is pressed, the application validates that the SKU entered is a known, valid SKU. If the SKU is valid, a item with that SKU is added to the customer's order. The register tape text area is updated with the new item, along with the Total Items and Total Cost text fields. If a SKU is not valid, display a popup dialog stating "Invalid SKU" which can be dismissed with an OK button. Once the popup is dismissed, the clerk is returned to the application frame.
- Pay – When this button is pressed, the customer's order is completed. The order completion is added to the register tape and the register tape is appended to the register tape file and flushed (flush is required to ensure the register tape is written to disk). The register tape, the Total Item and Total Cost are reset for a new customer order.
- Void – When this button is pressed, the customer's order is voided. The register tape, the Total Item and the Total Cost are reset for a new customer order.

7 Requirements

7.1 Functional Requirements

You will write a Java program that implements the point of sale register application described above. The program is started by running the Java interpreter and providing the classpath, main class name, the path to the costs file, the path to the sales file and the path to the register tape file:

```
java -classpath <class files directory> PointOfSale <costs files> <sales file> <register tape
    file>
```

On starting, the application reads the costs file and the sales file into memory. The application opens the register tape file for **appending** new orders. The application **must not** hard code paths to the costs, sales or register tape file. This is how I will test your program. Failure to implement this very important requirement will be **an automatic 5 point deduction.**

All of the algorithms, formats, functionality, user interface, etc. described in the previous sections are requirements for this MP. There are no optional requirements.

7.2 Non-Functional Requirements

Implementation Requirements

1. Implementation should consist of a good object-oriented design where:
   (a) The program is decomposed into multiple classes.
   (b) Those classes use encapsulation, inheritance and polymorphism to implement the application where appropriate.
   (c) The implementation is required to demonstrate **all** of these object-oriented concepts.

(d) The Model-View-Controller pattern is utilized in the design of the application.

2. The class should not depend on any third-party libraries or classes other than what comes in the Java SE API.

3. The source code should follow standard Java conventions and style:

   (a) Proper indentation and brace usage.

   (b) Proper use of vertical spacing. Each method definition should be separated by a space.

   (c) Variable names should be descriptive and relevant. One letter variable names are only appropriate in loops.

   (d) Methods should be:

      i. Not overly long. More than a half a screenful is probably too long. Break it up into smaller methods.

      ii. Named properly. Method names should be verbs or predicates.

   (e) Comments should be used liberally to explain your code or what the program is doing.

4. The source code should be compilable with `javac` or the IDE of my choice (Eclipse, IntelliJ, etc.).

5. All submitted class files must have an accompanying Java source file.

6. Submitted Java source files must be your own work and not copied from the Internet or other students.

GUI Requirements

1. The GUI should implement all of the functionality and non-functional (titles, borders, labels, etc.) GUI components just like the pictures of the GUI above. The layout of the elements should match their relative positions with one another. However you will not be graded on whether or not the sizes of the various elements match the example pictures. Nor will you be graded on whether or not your applications' "whitespace" around the components match the example pictures.

2. The GUI frames and dialogs should not be resizable.

3. Read-only elements should not be modifyable.

4. The GUI should dynamically update based on the actions of the user. Total Items, Total Cost and the register tape all update based on the actions of the clerk.

Documentation Requirements

1. In addition to the source code, you will also create a `README.txt` file with the following contents:

   (a) Your name and student ID number

   (b) Answers to the following questions:

      i. How do you run your program (i.e. what is the command line)?

      ii. Describe your object-oriented design for the program:

         A. How are you implementing the costs?

B. How are you implementing the sales?

C. How are you implementing a customer's order?

iii. What specific problems or challenges did you have implementing your solution? For example, was there a particular requirement that you had difficulty implementing? Or perhaps there was a particularly nasty bug in your implementation you had problems tracking down.

iv. Were there any requirements that were not implemented or not implemented properly in your solution? If so, please elaborate.

v. Were there any requirements that were vague or open to interpretation that you had to make a decision on how to implement? How did you elect to interpret them?

vi. How would you rate the complexity of this MP on a scale of 1 to 10 where 1 is very easy and 10 is very difficult. Why did you give this rating?

Submission Requirements

To submit your solution to this MP, please place the following files into a file in the ZIP file format (i.e. I should be able to unzip the file using Windows' extract tool, or the `unzip` command in Linux. No **RAR**, no **7z**. You will lose points if you do not follow this requirement):

- `README.txt`
- A `src` directory containing the directory structure that has all of your `.java` source code files. If you use `package` declarations, make sure your `.java` files are in a directory structure that reflects the `package` declarations (i.e. a class called `PointOfSale` that is in the `edu.iit.itmd510` package should have a path of `src/edu/iit/itmd510/PointOfSale.java` in you zip file).
- A `bin`, `target` or `classes` directory (depending on your IDE) containing all of the compiled `.class` files for your program. If you use `package` declarations, make sure your `.class` files are in a directory structure that reflects the `package` declarations (i.e. a class called `PointOfSale` that is in the `edu.iit.itmd510` package should have a path of `bin/edu/iit/itmd510/PointOfSale.class` in your zip file).

Please name this zip file in the following format `Name-StudentID.zip` (you will lose points for not following this format).

8 Grading

Your implementation will be graded on a scale of 100 points. The breakdown of the points is as follows:

- **40 points for correctness**. Correctness is whether or not your program compiles, executes with no errors and implements the requirements correctly. The functionality of the GUI should be implemented based on the requirements above.
- **35 points for completeness**. Completeness is whether or not your program implements all of the functional and non-functional requirements above.
- **15 points for object-oriented design**. Program should use classes, objects, encapsulation, inheritance and polymorphism where appropriate. The program should use the Model-View-Controller design pattern.

- **5 points for GUI look-and-feel requirements.** Your program should implement the main frame as described in the non-functional requirements with respect to layout.
- **5 points for coding conventions, style and documentation**. Submission follows coding standards per requirements above, follows the documentation requirements and the submission requirements.

9 Plagarism and Copying

- **All solutions will be checked for plagiarism and copying via automated tools and manual inspection.**
- **Students found to have copied code from another student or the Internet will receive a zero for this assignment and will be reported for an Academic Code Violation.**
- **This is not a group assignment, this is an individual assignment.** You may discuss the assignment with other students but you are not allowed to give code to, share code with or show your code to your fellow ITMD 510 students. When code is shared, there is no way for me to determine which student copied from whom so all parties involved might receive a zero.

10 Suggestions

- Start working on the MP early so that you have enough time to ask questions during class, during office hours, on Blackboard, etc.
- Do not spend too much time on the layout of the GUI. Concentrate your time on implementing the dynamic nature of the GUI. You can always come back to polish the look-and-feel after the core functionality is complete.
- Since this MP has no console output, feel free to use the console for logging messages to help debug the program.
- To implement the dynamic GUI functionality, recall callbacks, the observer pattern, etc. from our lectures. How does this fit with a Model-View-Controller design pattern? What would be the Model? What would be the View? What would be the Controller?
- Although there is no unit test requirement, consider writing unit tests to make sure the functionality of calculating sales, order totals, etc. works correctly.