Assignment 2: Inventory Management Application

CAB302 Software Development

Semester 1, 2018

Due date: 27/05/18 (Sunday week 12)

Weighting: 35%

Group size: 2

Specification version: 3.0

1 Domain

SuperMart is a supermarket chain that sells a range of fresh produce, meats, frozen food, and dry goods (items that do not need to be temperature controlled). Every customer purchase at a SuperMart store is appended to the store's weekly sales log. At the end of each week, the store manager examines that week's sales log and registers the update to the store's capital and inventory. At this point, the store's inventory needs to be replenished, so a stock order is produced by the manager.

Stock orders are produced by checking the store's inventory for items whose quantities are less than or equal to their reorder point, indicating that the item must be reordered. If an item needs to be reordered, N units of that item are added to the stock order, where N is the item's reorder amount. Finally, the manager sends the compiled stock order to the nearest distribution centre.

Distribution centres analyze stock orders and assemble shipping manifests to fulfill a store's replenishment demand. These manifests specify a collection of trucks and their cargo, where the total sum of all the cargo is equal to the stock order.

Certain items must be temperature controlled throughout the entire supply chain. Therefore, distribution centres administer both refrigerated and ordinary trucks. However, the cost of a refrigerated truck is inversely exponentially proportional to its temperature. **Therefore, distribution centres must optimize manifests for capital expenditure by minimizing the quantity of low temperature trucks through an optimal logistical allocation of items.**

When all trucks specified in a manifest have delivered their goods to a store, the store registers the update to its working capital and inventory based on the information in the manifest. The store pays for the manifest in full, where the total cost of the manifest is the sum of the costs of the trucks and their cargo.

SuperMart has observed recent rapid growth, but its paper-based logistics and inventory management is struggling to keep up. You have been contracted by SuperMart to develop an application that will be deployed to all SuperMart stores to help automate inventory management.

Distribution centres will no longer be generating their own manifests, instead, manifests will now be handled entirely through each store's application. Every time a sales log is loaded in, the manager will export a manifest and submit it to the distribution centre. The distribution centre will then schedule trucks according to the manifest. After all requested trucks have delivered their cargo, the store manager will then load the manifest back into the application, updating the store's capital and inventory.

2 Client requirements

Your application is required to implement the following features via a graphical user interface (GUI):

- View store capital in dollars and cents (e.g. $100,000.00).
- View store inventory in tabular format. Item names and quantities must be displayed along with their properties:
  - Name.
  - Quantity.
  - Manufacturing cost ($).
  - Sell price ($).
  - Reorder point.
  - Reorder amount.
  - Temperature (°C).
- Load in item properties documents.

  Effect: Item properties are initialized.
- Export manifests based on current inventory.
- Load in manifests.

  Effect: capital is decreased.

  Effect: inventory is increased.
- Load in sales logs.

  Effect: capital is increased.

  Effect: inventory is decreased.

When the application is first opened, **a starting capital of $100,000.00 must be displayed**. Item properties are then initialized by loading in an item properties document. At this point, all item quantities will be zero. The manager will then use the application to export a manifest based on this initial empty inventory. The manifest is then loaded back in to set up the store's inventory, and decrease the starting capital accordingly. At this point, the store has been set up and the manager will continue to load in sales logs and export manifests at his or her will.

Sample documents in the form of CSV files have been provided to accompany this specification, as well as a `README` that explains each document and their expected modifications to the store's capital. You can find the reference grammar for each of the document types in appendix B.

3 Development

The application needs to be written in Java and must make good usage of the object-oriented programming paradigm. You are required to use Git, follow test-driven development, and fairly split the work between yourself and your partner. There are five stages involved in the development of the application.

3.1 Stage 1: Repository

One person in the pair should create a new **private** GitHub or Bitbucket repository. Bitbucket offers unlimited private repositories by default, but GitHub requires students to apply for a student developer pack in order to create private repositories. You will use this repository from start to finish, frequently committing code as you work together on the project. You are encouraged to:

- Use Git features such as commit messages, branching, and merging. Do not panic if you ever make a mistake or run into merge conflicts. These are great opportunities to get experience with resolving merge conflicts and correcting mistakes gracefully.
- Explore GitHub and Bitbucket features such as viewing commit history, issue tracking, and analytics.

If you prefer, you are welcome to use a Git GUI client such as GitHub Desktop or Sourcetree.

You will be required to demonstrate your usage of version control by generating a Git log as described in section 4. Usage of a Git GUI client will not affect the log.

3.2 Stage 2: Test-driven development

Appendix A details the required design for the program, but you have a lot of control over the implementation details. You and your partner are required to develop a suite of unit tests in JUnit by analyzing this design and the client requirements.

You must design and implement the test classes for your partner and vice versa. The author of a test class must not also write the corresponding back-end implementation for that class.

The unit tests will drive the exact design of your program, specifying a reference point for the implementation stage. As you are still in the design stage, you will not be able to comprehensively cover all the components in the back-end, but you should try to write as many test cases as you can before moving on to the next stage. You can come back to this stage at any time, to modify, add, and delete unit tests as you refine the design of your solution.

Keep in mind that there are some parts that cannot be tested with JUnit, such as the GUI and CSV parsing and writing. You will have to rely on manual testing for these components.

3.3 Stage 3: Back-end implementation

In this stage, you will progressively pass all of your unit tests. You should take care to continuously refactor your code as you pass tests. You may need to continue to refer to appendix A, especially for the parts that cannot be tested. Your classes should be organized into packages such as, but not limited to: "Stock", "Delivery", and "GUI".

Note that optimizing the construction of manifests will be a challenge. You can use the sample documents to check if you have implemented the optimization correctly.

Finally, you must comprehensively document all of your classes with appropriate JavaDoc. This includes describing what each class and method does (including constructor and method parameters) and correctly using the author tag.

3.4 Stage 4: Graphical user interface (GUI)

You must implement a GUI using the Swing GUI widget toolkit to wrap the back-end functionality, thus achieving the client requirements specified in section 2. The GUI should catch all checked exceptions thrown by the back-end, and present message boxes to the user.

3.5 Stage 5: Report

You must produce a PDF report structured as follows:

- **1 page.** Title page containing the names and student numbers of both students.
- **1 - 2 pages.** A technical description of your program architecture, drawing reference to object-oriented design concepts such as polymorphism and abstraction. You may want to use a diagram to illustrate your type hierarchies and interaction between classes.
- **2 - 5 pages.** A GUI test report that demonstrates the full range of functionality of the application, including exception handling. Use screenshots accompanied by brief descriptions.

4 Submission

Submit via Blackboard a `.zip` of:

- Your report as a PDF.
- A file system export of your source code with accompanying JavaDoc.
- A git log that demonstrates usage of consistent version control. The log is produced by running the following commands:

```
git shortlog > repo.log
git log --graph --oneline >> repo.log
git log --shortstat >> repo.log
```

A Back-end design

To implement the back-end, you will need a variety of classes, including specialized exception classes.

A.1 Objects

Your solution will be object-oriented, and so you will need to implement several classes to represent the objects that exist conceptually in the system. You are required to have the following object classes:

- Item. An item, possessing at least the following properties:
  - Name.
  - Manufacturing cost in dollars.
  - Sell price in dollars.
  - Reorder point.
  - Reorder amount.
  - Temperature in °C that must be maintained for the item to not perish. Not all items are temperature controlled, and so they will not need a temperature.
- Stock. A collection of items. Can be used for representing store inventory, stock orders, sales logs, and truck cargo.
- Truck. An abstract class for the two truck types.
- Refrigerated Truck. A truck, possessing at least the following properties:
  - Cost in dollars equal to $900 + 200 \times 0.7^{T/5}$ where T is the truck's temperature in °C.
  - Cargo capacity of 800 items.
  - Cargo. All items can be stored in a refrigerated truck's cargo, including dry goods (items that do not need to be temperature controlled).
  - Temperature in °C that maintains a safe temperature for the truck's cargo. This is equal to the temperature of the item in the cargo with the coldest safe temperature. The allowed temperature range is from -20°C inclusive to 10°C inclusive.
- Ordinary Truck. A truck, possessing at least the following properties:
  - Cost in dollars equal to $750 + 0.25q$ where q is the total quantity of items in the cargo.
  - Cargo capacity of 1000 items.
  - Cargo. Temperature controlled items cannot be stored in an ordinary truck's cargo, only dry goods.

- Manifest. A collection of trucks.
- Store. An object for representing the store itself. You are required to use the singleton pattern for this class. The store must have at least the following properties:
  - Capital.
  - Inventory.
  - Name.

A.2 Other classes

You will need to create classes for grouping together units of code including but not limited to: your program entry point, GUI, and CSV parsing and writing. It is up to you how you structure this code, keeping code quality and good object oriented design in mind. For example, you could have a dedicated class for all CSV parsing and writing, or you could separate item properties, sales log, and manifest CSV parsing and writing into their respective classes.

A.3 Exceptions

You will need to create three specialized exception classes:

- CSVFormatException.
- DeliveryException.
- StockException.

You will need to decide where it makes sense to throw these exceptions. Exceptions should not be handled in the back-end, they should be thrown up the call stack and then handled by the GUI.

B Document grammar

All documents come in the form of comma-separated values (CSV) files. You can can see the accompanying sample documents for example data.

Below is the grammar for each of the three types of documents. Descriptors surrounded by brackets represent values such as strings, integers, and decimal numbers.

Item properties

```
[item],[cost],[price],[reorder point],[reorder amount],[temperature]
[item],[cost],[price],[reorder point],[reorder amount],[temperature]
[item],[cost],[price],[reorder point],[reorder amount],[temperature]
```

Items that are not temperature controlled do not have the `[temperature]` value. These items instead have the following grammar:

```
[item],[cost],[price],[reorder point],[reorder amount]
```

Manifest

```
>[truck type]
[item],[quantity]
[item],[quantity]
[item],[quantity]
>[truck type]
[item],[quantity]
[item],[quantity]
[item],[quantity]
```

Sales log

```
[item],[quantity]
[item],[quantity]
[item],[quantity]
```