CAB302 Project: Inventory Management Application

By:

Alexander Rozsa: n9992529

Liam Edwards: n\*\*\*\*\*\*\*

2018

# **Technical Description**

The aim of this project was to have a GUI-based application that takes defining back-end classes and displays outputs based on their values. The values needed to be displayed, of course, are the items contained within the “store” and their quantities as a means of managing the store’s inventory. The below diagram is a representation of the program’s classes and their dependencies ordered from the main class (Interface) down to the class which doesn’t have any dependencies (Item). Interface doesn’t have any direct dependencies, however it does make calls to Store for info on the store and IOHandler to import and export data. IOHandler makes calls to Store for store info and relies on Item to import items into the Store, Trucks to add Items to Trucks in order to calculate the loss/gain of capital, and Manifest in order to export Manifests and **\*\*\*\*\*\*\*\*\*\*\***. Manifest relies on Trucks since it’s a list of Trucks and that’s the object type it’s supposed to contain. Store and Trucks rely on Stock in order to store Item(s) and their quantities. Finally, Stock relies on Item to have many of them stored within it providing data to the program, whereas Item doesn’t have any dependencies; it just supplies data.

**A A** has a direct

Item

Store

Trucks

Manifest

Stock

Interface

IOHandler

= dependency on **B**

**B** (constructs and contains

an object of that class).

**A** **A** has an indirect

= dependency on **B** (calls the

**B** public static functions).

Interface is both the GUI and the program entry point which applies polymorphism by extending JFrame and implementing ActionListener. The program is begun by creating a new Interface with a whole bunch of GUI objects and their properties, along with the public static declaration of the Store. The extension of JFrame allows JFrame functions such as setVisible() and getContentPane() to be called without declaring a JFrame within the class and stating the name of that before every JFrame function call. Instead, the class IS the JFrame variable. Since Interface implements ActionListener, it has the additional function actionPerformed, which allows all the outputs of the buttons to be put into one function instead of declaring a new function every time a button needs an ActionListener.

Store is the primary back-end class to the entire program. This singleton class represents the store and its values. It applies encapsulation by keeping its values private and has them get/set by using public functions. Since there will only be one store, it’s better to have a static singleton store publicly accessible from any class without the need for local storage instead of declaring a new store every time in each class it’s needed.

IOHandler is similar to the way Store is accessed, however it doesn’t even have any type of constructor. It supplies the interface with file reading and writing functions in order to update the store.

Manifest is a list of trucks. Like Store, it applies encapsulation by hiding a List of type Truck, and sets/gets its data with publicly available functions. It is used in IOHandler for the exporting and \*\*\*\*\*\*\*\*\*\* of manifests.

Trucks above is not one class; it is a categorization of three classes: Truck, OrdinaryTruck and RefrigeratedTruck. Truck applies abstraction by being an abstract class that supplies basic truck functions to its inheritors: OrdinaryTruck and RefrigeratedTruck (which apply inheritance). Truck also applies encapsulation by having the maximum capacity and stock variables kept private, and even more so for RefrigeratedTruck since it additionally has temperature.

Simply put, Stock is the most important class in the program. It holds a HashMap that has integers assigned to Item keys which represent the quantity of each item. As shown in the graph, Stock is used by Store, which allows the store to get items from its Stock variable and output it to the Interface, and also by Truck which allows costs to be calculated based on the number of items stored. It applies encapsulation for the HashMap and it also applies inheritance by implementing Iterable.

Lastly, Item is the class the program revolves around. Items encapsulate 6 pieces of data that define an item. These item properties determine whether it exists in a Stock object, how much to deduct or add to the Stores’s capital, when to reorder and how much to reorder, and whether the item can be added into a specific type of Truck. It’s a basic class that doesn’t have any setter functions; only getters.

# **GUI Test Report**