REPORT ON SOFTWARE ARCHITECTURE (Part 2)

Overview

The design pattern for the part2 is a follow up from the part 1. Various and necessary changes are made nevertheless for a more robust version of the application with some of the changes discussed in this report. With reference to part1, The **Staff** class contains all the actions that was listed out in use case diagram of part 1 including the use of "**Includes**" and "**Extend**" cases. Additionally, other actions not listed in part 1 use case diagram have been added to the application. The Class diagram was referenced to capture the relationship between classes. One of such relationship is that the application supports a customer to have one or more parcels, and a customer must have a parcel. The data structure that stores the customers in queue has been updated to a *TreeMap*. This allows customers and their parcels in the queue to be sorted according to their names. Customer's names are used as keys and the customer objects as values. Through, the customer objects, the customer parcels can be accessed in sorted order.

In the application for a customer object to be created, a parcel must be created first for the customer. This makes it possible to have parcels that may not be owned by customers if a customer is not created for a parcel after a parcel is created. parcels displayed on the parcel section are only parcels owned by customers in the queue.

To run the program, run the main class from the Main package, then click on load parcels and then click on load customers.

DESIGN PATTERNS

Three Tier Architecture

The Three Tier Architecture is implemented by separating classes and files into three main packages. The presentation package containing all the GUI classes including *Customer Display, Parcel Display, Process Display* and *StaffGUI*. The Logic package contain all logical classes such as *Customer, Parcel, ParcelMap, CustomerQueue, Depot*

(Manager), Log, etcetera. The Data package contains the log generated file, customer data and parcel data files for reading data input into the system.

Model View Controller (MVC)

To implements MVC, The Depot Class can be considered as the Model as it contains the instantiations of other class to which changes occur in them in the application. The *ParcelMap*, CustomerQueue, *Parcel* and *Customer* classes makes up the Model specifically because changes do occur in them. The Staff GUI can be considered as the View. The StaffGUI is responsible for managing all the three GUI display classes (*Customer Display, Parcel Display* and *Process Display*). I developed an additional class, *DepotController* to serves as the controller for the depot. The *DepotController* class handles all events that happens between the GUI and the depot. My reason for separating the DepotController from the Depot class was because of a lengthy code in the *DepotController* class and to enable easy debugging and development. I used singleton to implement the ***Observer*** and **Subject** which the MVC extends. The *Notifier Class* manages the Observer and Subject classes using a singleton pattern. Since changes occurs in many classes making up the model, I used the singleton pattern to across these classes to track such changes.

Singleton Pattern for Logging

The Log class uses the singleton pattern to register events across all the classes that causes changes to the system.

APPENDIX

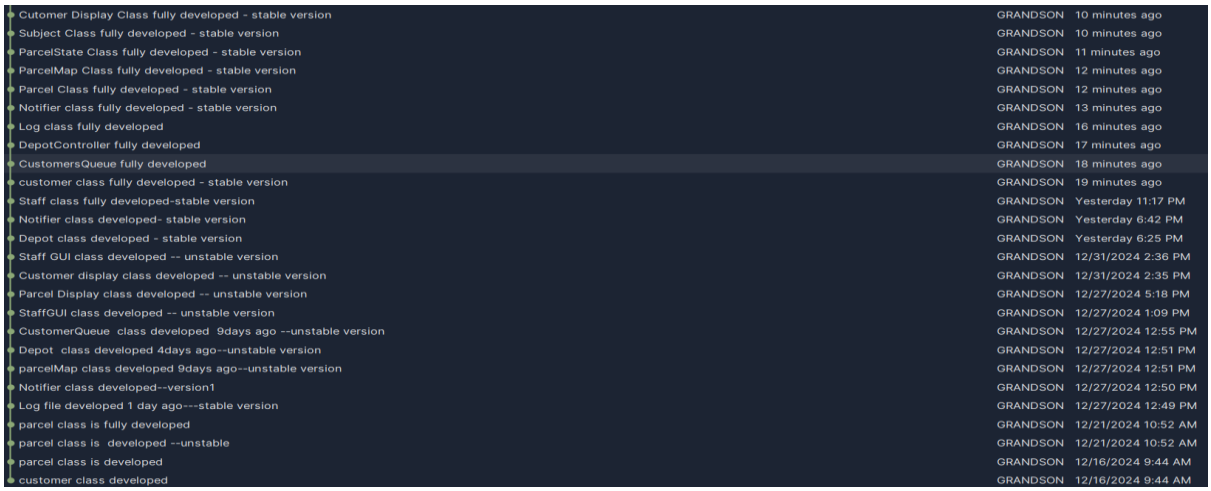

Fig 1-showing iterative git-commit during the development process of the application