

Part 2: Implementation - Functional Application, Design Patterns, GUI

Student Name: Subhagh Mambully Bhasi
Student ID: 23082217

Date of Submission: January 8, 2025

Contents

1	Introduction	1
2	Implementation Details	2
2.1	Project Structure and Key Classes	
2.2	Classes Overview	2
2.3	Git Usage and Iterative Development	3
3	GUI, MVC, and Design Patterns	4
3.1	GUI Construction with Swing	4
3.2	Model-View-Controller (MVC) Pattern	4
3.3	Singleton Pattern for Log	4
4	Reflection on Implementation	5
5	Git logs	7
6	conclusion	8

1.Introduction

This document forms the second part of the assignment in which we implement the software architecture and design patterns explored in Part 1. In Part 1, we constructed design models such as class diagrams, use case diagrams, and possibly sequence diagrams to outline how the system should behave. Part 2 focuses on the implementation of those models into an real Java application, to domonstrate:

- Common software architecture patterns (e.g., Three-Tier, MVC).
- Use of the Singleton pattern for logging.
- GUI functionality using Java Swing.
- Implementation of the Worker processing customers to collect parcels.

The main objective is to show the models from Part 1 have been changed to working code with appropriate data structures and patterns.

2.Implementation Details

2.1 Project Structure and Key Classes

The application follows a three-tier architecture, separating:

- **Model:** Contains business objects like Customer, Parcel, and Que of Customers.
- **Controller:** Manages the logic of processing parcels, represented by the Worker and Manager classes.
- **View:** Consists of the Swing-based GUI classes that display the lists of parcels, the queue of customers, and the currently processed parcel.

For data handling, we use:

- **ParcelMap:** A map-based data structure to hold all parcels.
- **QueofCustomers:** A queue or list structure to manage waiting customers.
- **Log:** A Singleton class to record all system events such as *customer joins queue*, *parcel processed*, etc

2.1.1 Classes

- **Parcel:** Holds attributes such as *parcelID*, *weight*, *destination*, etc.
- **Customer:** Holds attributes such as *customerID*, *name*, etc.
- **QueofCustomers:** Manages the queue of customer objects.
- **ParcelMap:** Stores and retrieves parcels via a Map data structure.
- **Worker:** Contains the logic for processing a customer request, calculating fees (*calculateFee*), and marking a parcel as collected.
- **Manager:** The *driver* class responsible for instantiating key data structures, reading data files, and initializing the GUI.
- **Log (Singleton):** Records all events and writes them to a text file once the application closes.

2.2 Git Usage and Iterative Development

Throughout my development, the project was maintained under Git version control. Key changes were:

1. Initial commit: Created the Java project and the base class structure (Parcel, Customer)
2. I/O testing in CLI : Tested calculateFee and queue operations from console- based interactions.
3. GUI addition : Introduced Swing panels to display parcels, queue of customers, and worker processing details.
4. Final Edits : Enhanced logging , wrote log to file, refined the MVC layering.

A screenshot of the Git commit log is included in the file to highlight the iterative nature of the project's development.

GUI, MVC, and Design Patterns

3.1 GUI Construction with Swing

- **Parcel Panel:** Displays the list of currently unprocessed parcels.
- **Customer Queue Panel:** Shows the current queue of customers waiting to collect parcels.
- **Processing Panel:** Displays the details of the parcel being processed by the worker and the calculated fee.

Each panel updates automatically whenever the underlying data changes, ensuring responsiveness.

3.2 Model-View-Controller (MVC) Pattern

- **Model:** Classes like Parcel, Customer, and data structures for storing them.
- **View:** Swing-based panels and frames. Observers listen for changes in the model.
- **Controller:** The Worker and Manager classes handle the user interactions and business logic.

This division promotes maintainability and scalability.

Singleton Pattern for Log

The Log class maintains a single instance:

- **Private constructor:** Prevents other classes from instantiating Log directly.
- **Public static getInstance() method:** Ensures only one Log object exists at runtime.
- **Logging events:** Each event is appended to an internal StringBuffer, and on application shutdown, it is written to log.txt for easy reading.

4. Reflection on Implementation

Reflection on Realizing Part 1 Designs:

In Part 1, the class diagram showed that the Worker interacts with a Queue of Customers object to retrieve and process Customer objects. This is implemented in the Worker class through the processCustomer method, which calls Queue of Customers.remove() to dequeue the next customer. The ParcelMap class, depicted in the Part 1 design, is as a Java HashMap<String, Parcel> that stores parcels by their parcel ID.

The UML sequence diagram from Part 1 that showed a flow of events (customer arrives, queue updates, worker processes, log records) is reflected in the logic within the Manager class. In particular, when a new customer arrives, a method addCustomer(Customer c) in Queue of Customers is called, and the Log Singleton appends a message noting the time and event.









Additionally, we integrated the three-tier architecture from Part 1's design, separating the View, Model, and Controller layers. For instance, the ParcelMap and Customer classes remain part of the Model, while Worker and Manager handle Controller logic such as calculating fees (calculateFee) and removing parcels from the ParcelMap. The View layer, built with Swing, monitors changes to the model objects and refreshes the GUI accordingly.

Reflection on MVC Usage:

To implement the MVC pattern, I separated the code that displays data on panels (View) from the data-handling classes (Model). The Worker and Manager classes act as Controllers; they respond to GUI events like button clicks, queue updates, and parcel processing. Whenever a *process* operation is triggered, the Manager instructs the Worker to retrieve a Customer from the queue, identify the corresponding Parcel in the ParcelMap, calculate the fee, and update the logs. Then the GUI panels observe these changes and refresh.

This architectural separation mirrors the Part 1 design, ensuring changes to data structures or the logic do not break the user interface. Moreover, this modularity simplifies testing, as console-based I/O could be used earlier stages for validating logic before building the GUI. Overall, the step by step approach allowed me to start simple and gradually add complexity later.

5 .Git Log & Report

Final Edits <small>222</small> 👤 rjkkm authored now	Verified e505073  
GUI addition <small>222</small> 👤 rjkkm authored now	Verified 3780188  
I/O testing in CLI <small>222</small> 👤 rjkkm authored 1 minute ago	Verified 29d9d8b  
Initial commit <small>222</small> 👤 rjkkm authored 3 minutes ago	Verified 533dd84  

6 Conclusion

In this assignment, we have successfully translated the conceptual design from Part 1 into a functional Java application. By adhering to the three-tier architecture and the MVC pattern, the system remains modular, maintainable, and easy to test. The Log Singleton ensures all events are tracked consistently, while the GUI built in Swing demonstrates clear separation of concerns between the Model, View, and Controller layers.

Through an iterative approach with Git version control, we validated each development milestone, ensuring code stability and feature completeness. The reflective report and Git commit logs further illustrate how theory and design patterns have guided the practical implementation of this parcel depot application.