TurboTrucks: Delivery System Project

SOEN 343 – Software Architecture and Design

Final Group report

Dr. Joumana Dargham

Team Code Ninja

Jonathan Della Penta – 40128210

Julie Makary – 40243160

George Ezzat – 40245502

Monday, 2nd December 2024

Table of contents:

1. Introduction

The purpose of the Turbo Trucks delivery system project is to apply software engineering concepts, such as design patterns and diagrams such as use cases, domain model, & class diagram. It develops a functional and user-friendly delivery management platform. This delivery system lets the clients create delivery requests and track their orders, while administrators can manage the system's database using Firebase. This project focuses on creating a simple and useful system that shows how we can apply class concepts in real-life situations. Implementing design patterns such as singleton, factory method , and observer improved modularity, scalability and maintainability.

2. Combined Sprint Summaries

Sprint 1 - Problem Domain Study and Initial Diagrams

Problem Domain:

Traditional shipping is inconvenient, requiring customers to drop off packages at post offices, create labels, and deal with unreliable tracking. This process is time-consuming, especially for those without easy access to transportation or printers.

Proposed Solution:

A delivery service that picks up packages directly from customers, prints shipping labels on-site, and provides real-time tracking. For international shipments, the customer can fill out a customs form via the app. This solution has advantages. It saves time by eliminating the need for post office trips. A Real-time communication with couriers via chatbot for customer service . No need for customers to print labels , and a reliable tracking for packages by using the tracking ID given.

Context Diagram:

As shown in Figure 2, Turbo Trucks system interacts with external entities: customers, couriers, customer service, supplier , distribution centre , payment gateway, and tracking system.

Domain Model:

As shown in Figure 1**,** Key entities include Customer, Delivery, Quotation, Order Tracking, Payment, Chatbot , and service communication. Each class has a relationship showing how they interact, from placing an order to track delivery and processing payment.

To conclude, this sprint establishes the problem, the proposed solution, and the foundational system structure.

Sprint 2 - Use Case Diagrams and Sequence diagrams

As shown in Figure 3, The Use Case Diagram visually represents the main functionalities of the delivery system, focusing on the interactions between the Client and Delivery Driver with the system. It includes use cases such as:

- Client: Requesting a delivery, requesting a quotation, tracking a delivery, making a payment, reviewing the service, and requesting customer support.
- Delivery Driver: Verifying delivery list and updating delivery status.

This diagram shows the system’s core functionalities and the actors involved in each process.

Sequence Diagrams**:**

The Sequence Diagrams illustrate the step-by-step interactions for key use cases:

1. Request Delivery: The client provides delivery details, and the system generates a unique delivery ID.

2. Request Quotation: The client provides package information, and the system calculates the estimated cost.
3. Track Delivery: The client enters a delivery ID to view real-time tracking information.
4. Process Payment: The client provides payment details, and the system processes the payment.
5. Review Service: The client rates the delivery service after completion.
6. Request Customer Support: The client contacts support for assistance, and the system creates a support ticket.

These diagrams define the sequence of actions and the flow of data between the actors and the system for each use case.

Sprint 3 - Implementation and Design Class Diagram

In Sprint 3, the focus was on implementing the front-end for all website features, and improving the system's design by integrating key design patterns and expanding the class diagram. The class diagram, building on the domain model created in Sprint 1, now includes more details such as attributes and methods for each class as shown in Figure 5. Additionally, design patterns were introduced to enhance flexibility, maintainability, and scalability of the system.

1. Strategy Design Pattern:
   Encapsulates different pricing strategies (e.g., Standard, Express) into separate classes, allowing the system to easily add new delivery types without modifying existing code.
2. Singleton Design Pattern:
   Ensures a single instance of the Chatbot Manager, which handles multiple chatbot instances for user sessions, promoting consistency and efficiency.
3. Factory Design Pattern:
   Centralizes the creation of account types (Client, Delivery Driver, Admin)

through a single factory, simplifying code maintenance and future extensions.

4. Observer Design Pattern: Notifies clients automatically of delivery status changes, decoupling the tracking system from the clients and improving scalability and flexibility.

Together, these design patterns enhanced the overall architecture of the system, making it easier to manage and extend while ensuring better performance and user experience.

Sprint 4 - Refinement and Finalization

In sprint 4, we focused on finalizing the front-end and back-end features implementations, refining the system architecture, and refactoring code to accommodate any changes that were made in the codebase.

The features that were finalized include login and sign up, create a delivery, get a quote, tracking an order, payment, chatbot AI, admin management, Firestore database, and Google Maps APIs. After all these finishing adjustments, the application was ready to be used.

The class diagram was refined to include the GOF design patterns that were implemented in the codebase. The changes made to the class diagram reflected the scalability and enhanced clarity that the design patterns gave to the codebase. The sequence diagrams were also modified to show new processes and include new classes and parameters.

The codebase was refactored to include the implementation of the design patterns as well as classes, such as costCalculator, that were designed to reduce coupling and increase cohesion for the system. Duplicate code was consolidated to keep components simpler and organized.

Sprint 4 involved adding and refactoring existing code, updating the system architecture class diagram and sequence diagrams, and finalizing the front-

end, back-end, and core features. After all these adjustments, the system was finalized and ready to be used.

3. Diagrams Evolution

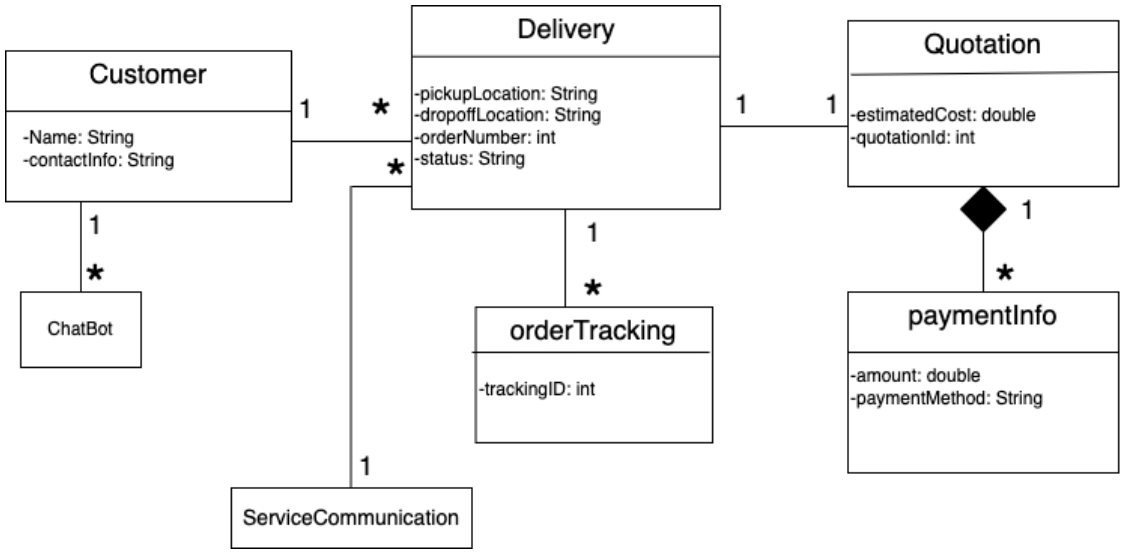

Figure 1. Domain model presented in Sprint 1

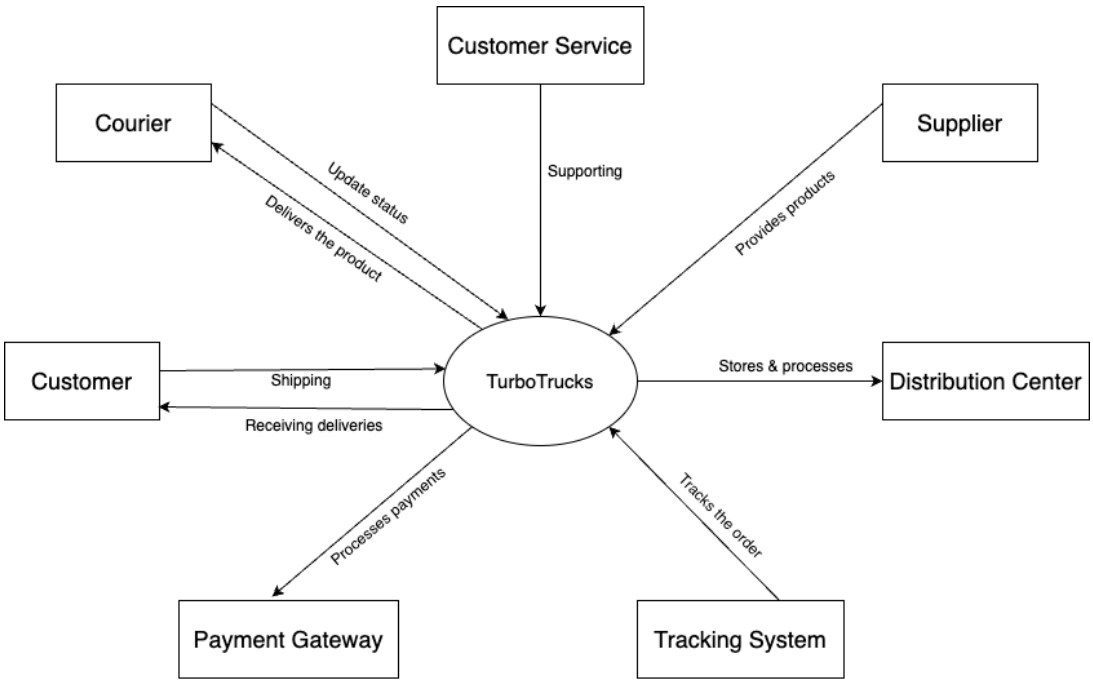

Figure 2. Context diagram presented in Sprint 1

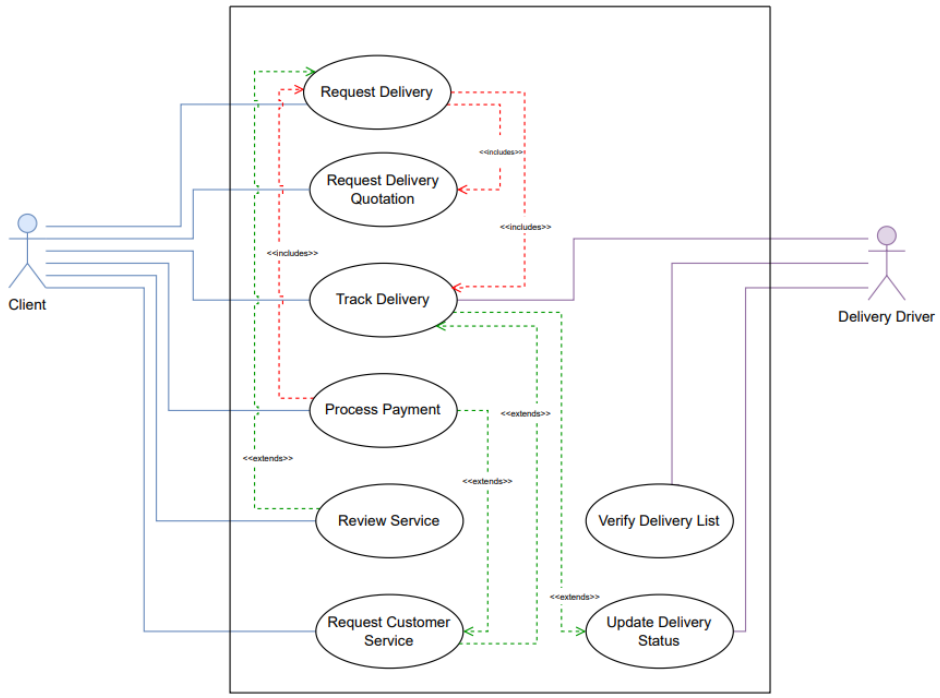

Figure 3. Use case diagram presented in Sprint 2

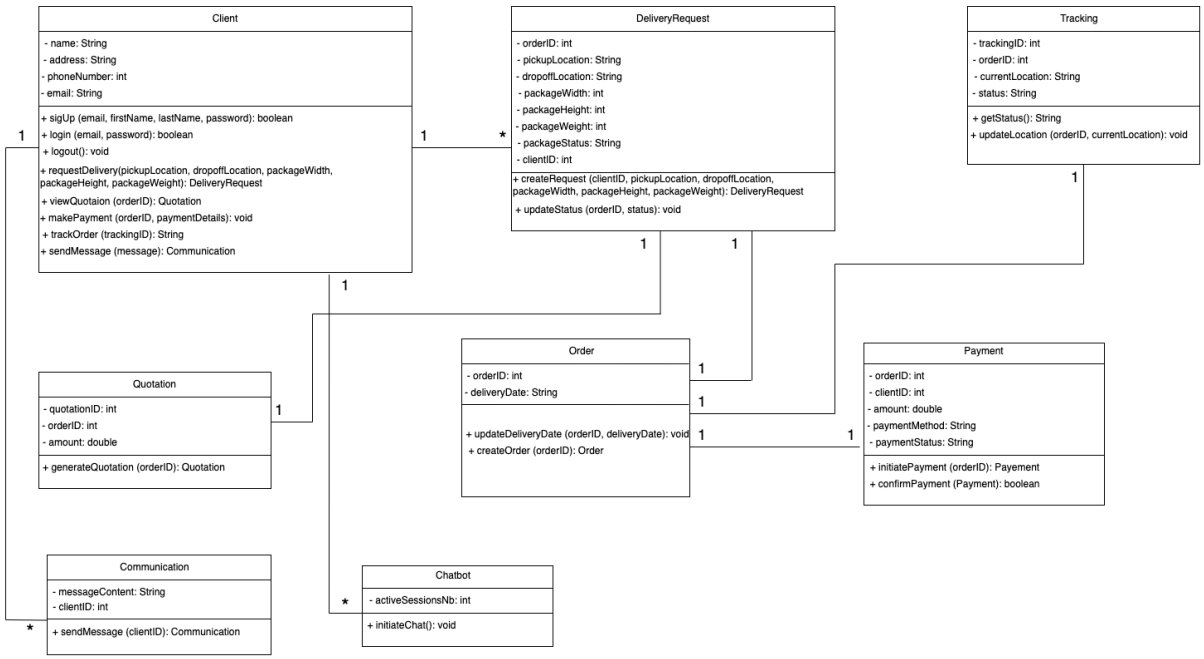

Figure 4. Initial class diagram without any design patterns

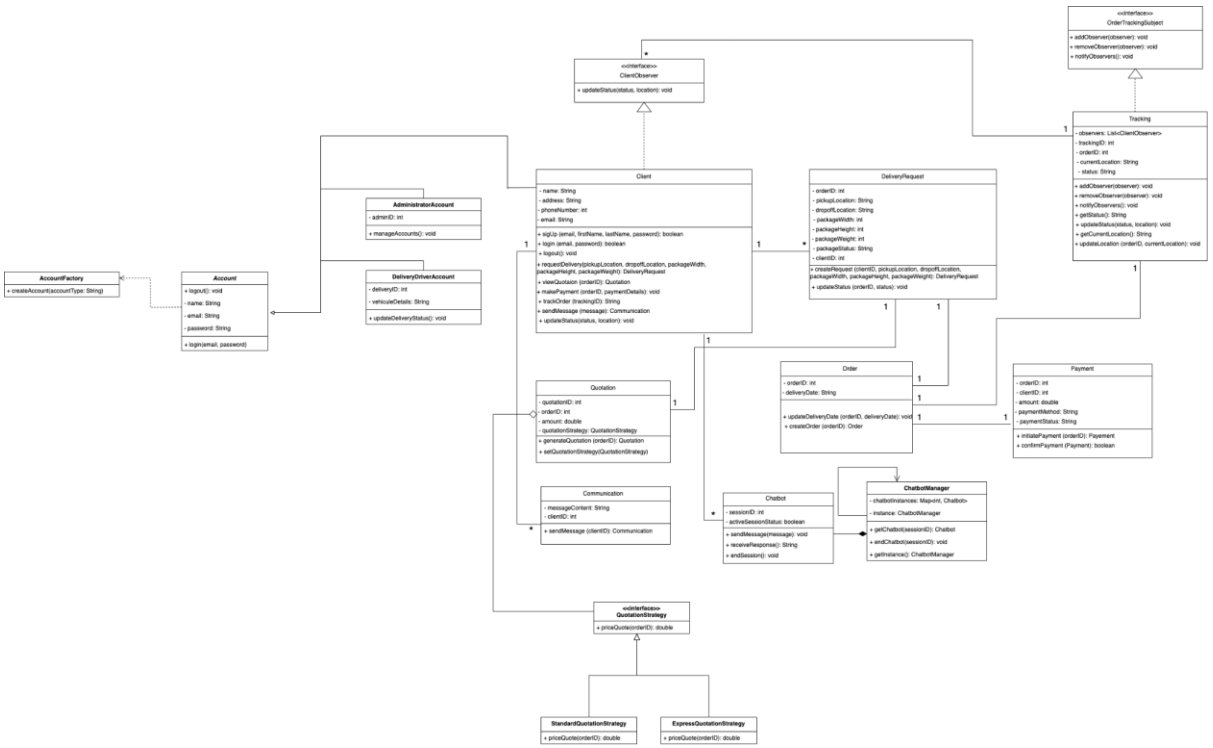

Figure 5. Class diagram with the design patterns as presented in Sprint 3

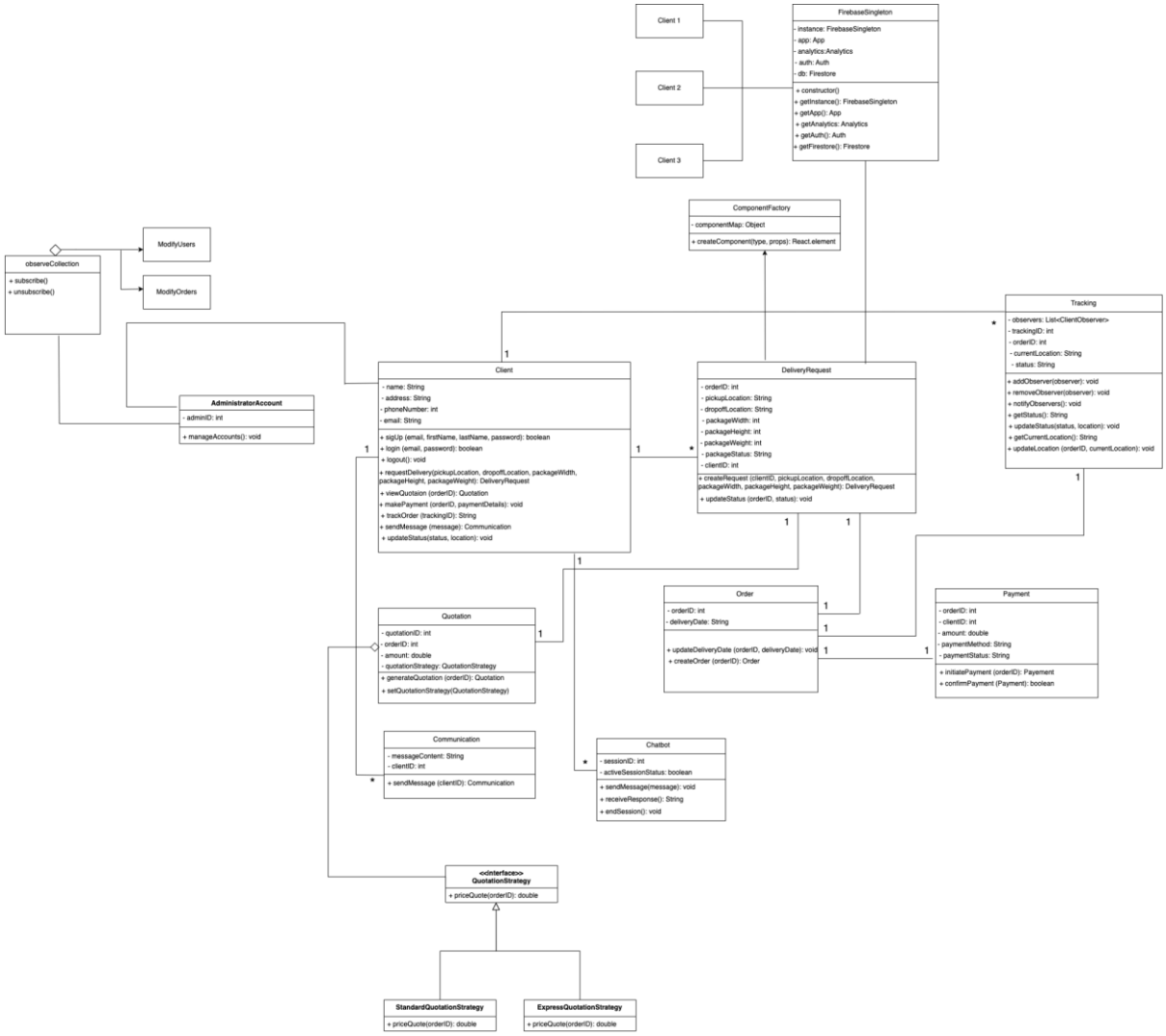

Figure 6. Final class diagram with the updated design patterns as presented in Sprint 4

The initial domain model that was represented in Sprint 1 only captured the core functionalities of the system. It didn't accurately represent the complete delivery system and only focused on the relationships between the entities. The context diagram also provided a basic implementation of the system without any details. During Sprint 2, the use case diagram showed a bit more in detail how the clients would interact with the functionalities the system offers.

During sprint 3, the class diagram was implemented. This diagram showed exactly how each functionality interacts with the other ones and it also showed what are the attributes and methods that these functionalities should have in

order for them to behave like expected. It also demonstrated the design patterns that were used to enhance the development of the system.

For Sprint 4, the only change was that the design patterns were adjusted slightly. These adjustments were made to ensure that the system could function efficiently as it grew in complexity.

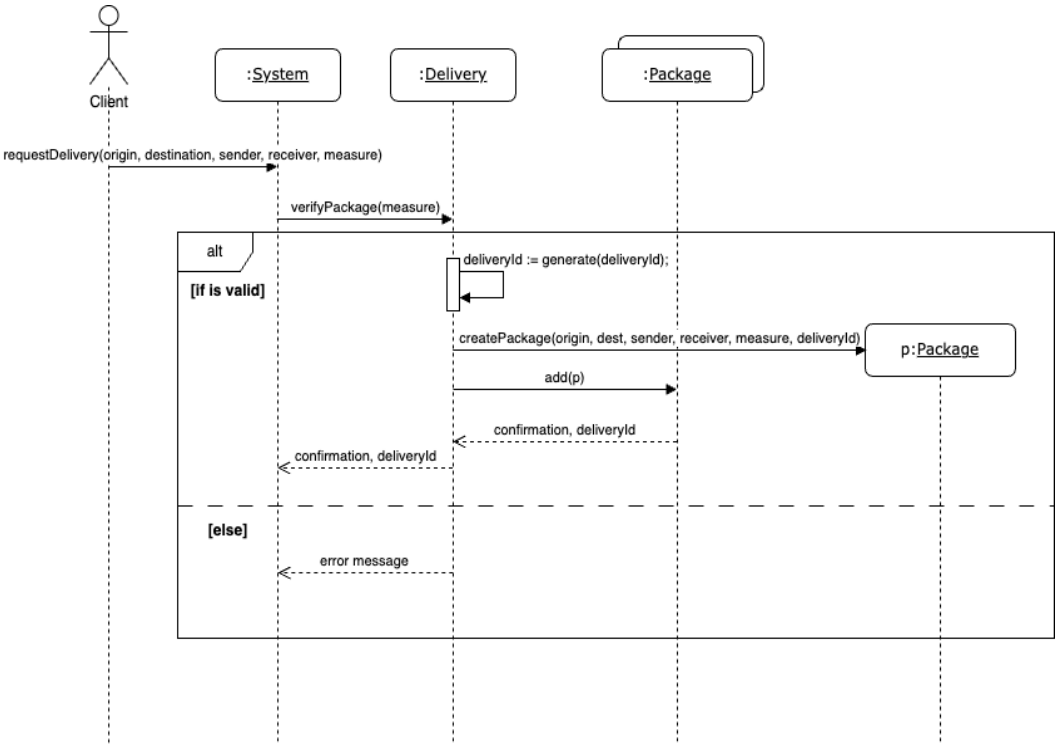

Figure 7. Initial sequence diagram for request delivery

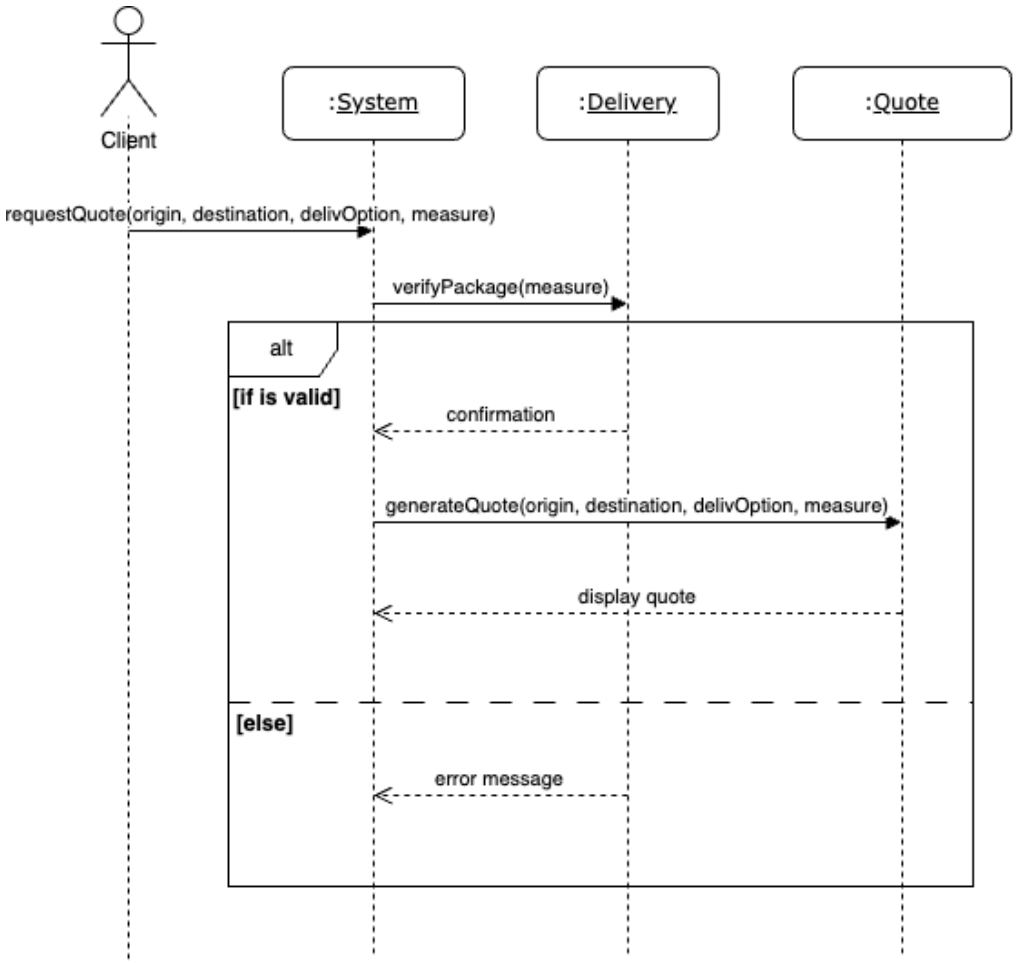

Figure 8. Initial sequence diagram for request quote

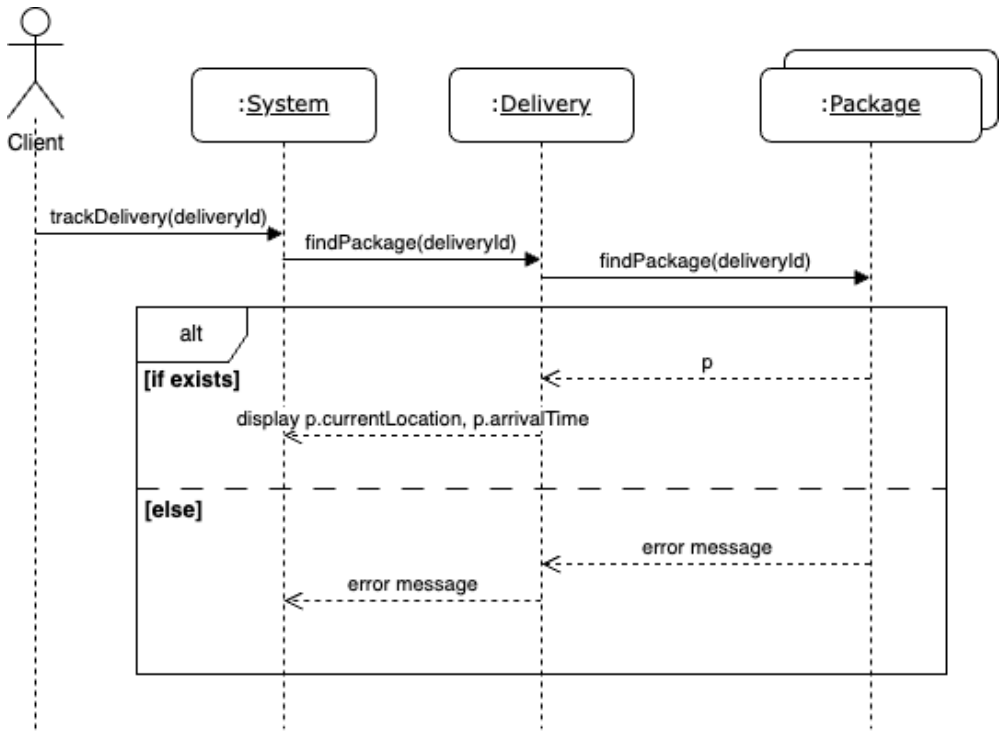

Figure 9. Initial sequence diagram for tracking delivery

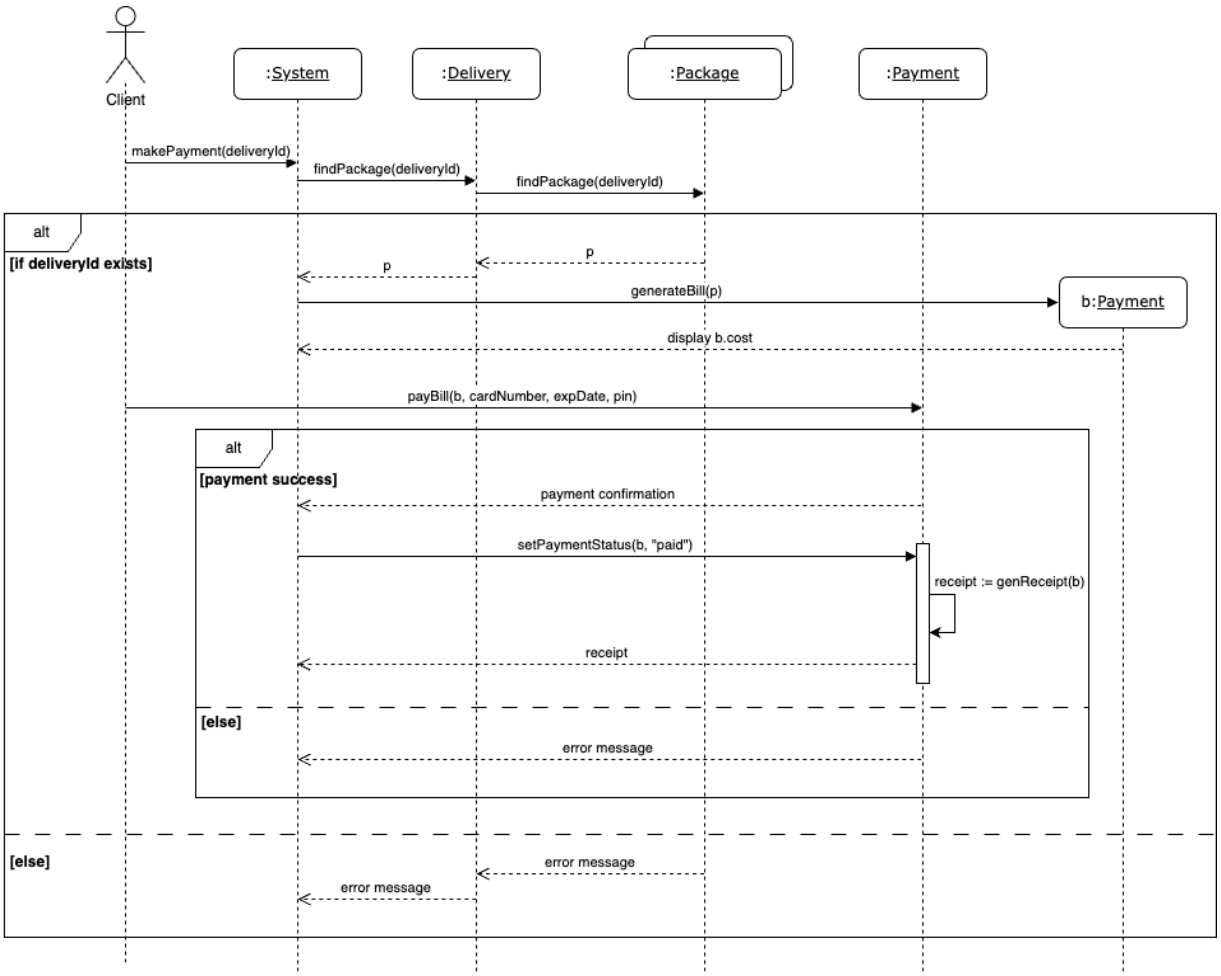

Figure 10. Initial sequence diagram for processing payment

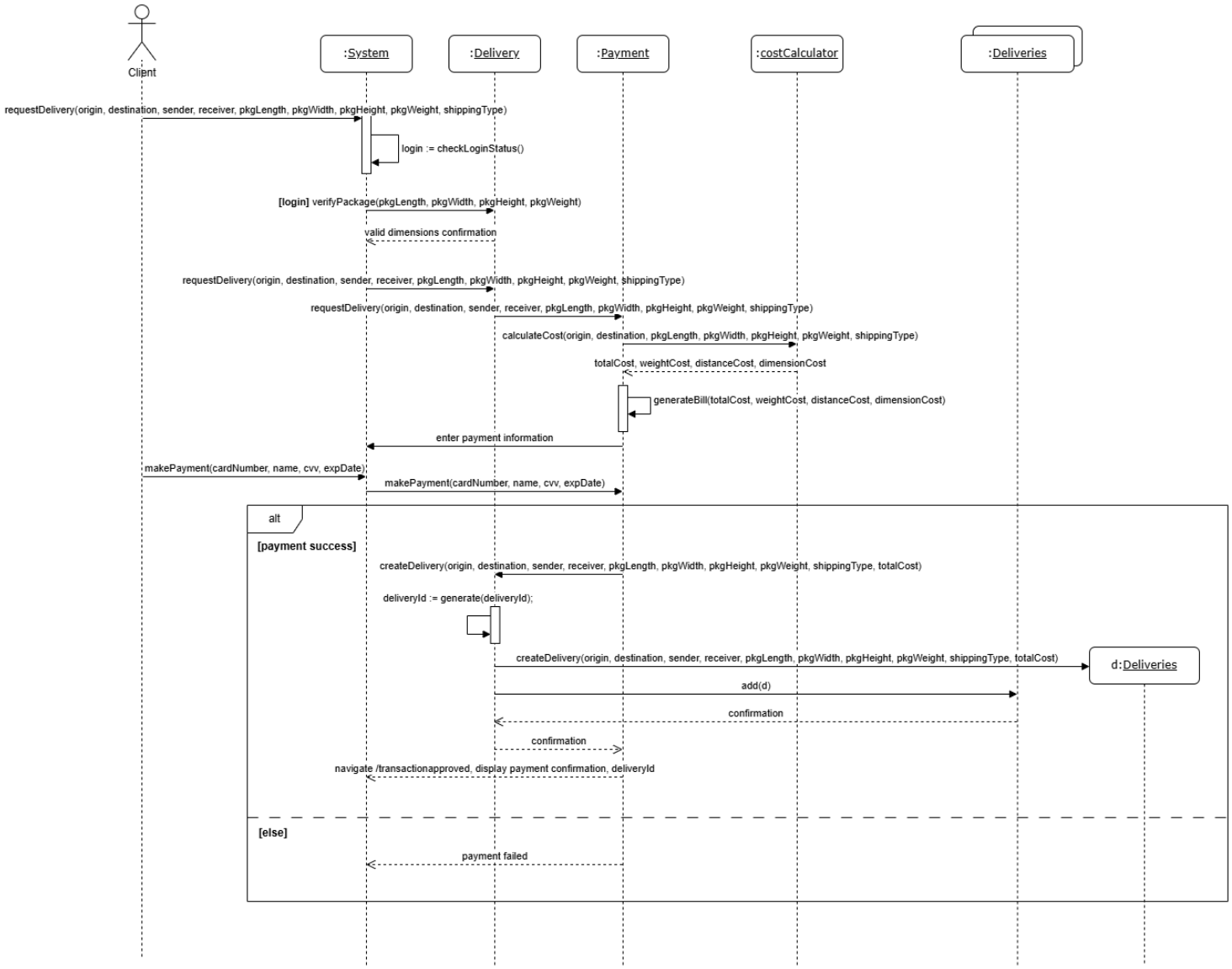

Figure 11. Updated sequence diagram for request delivery and processing payment

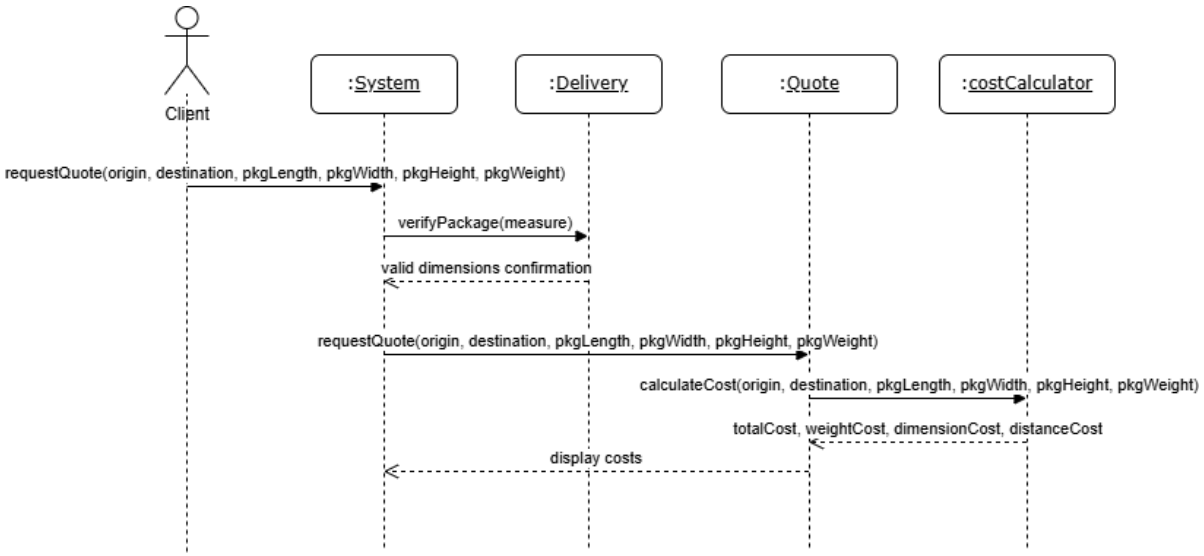

Figure 12. Updated sequence diagram for request quote

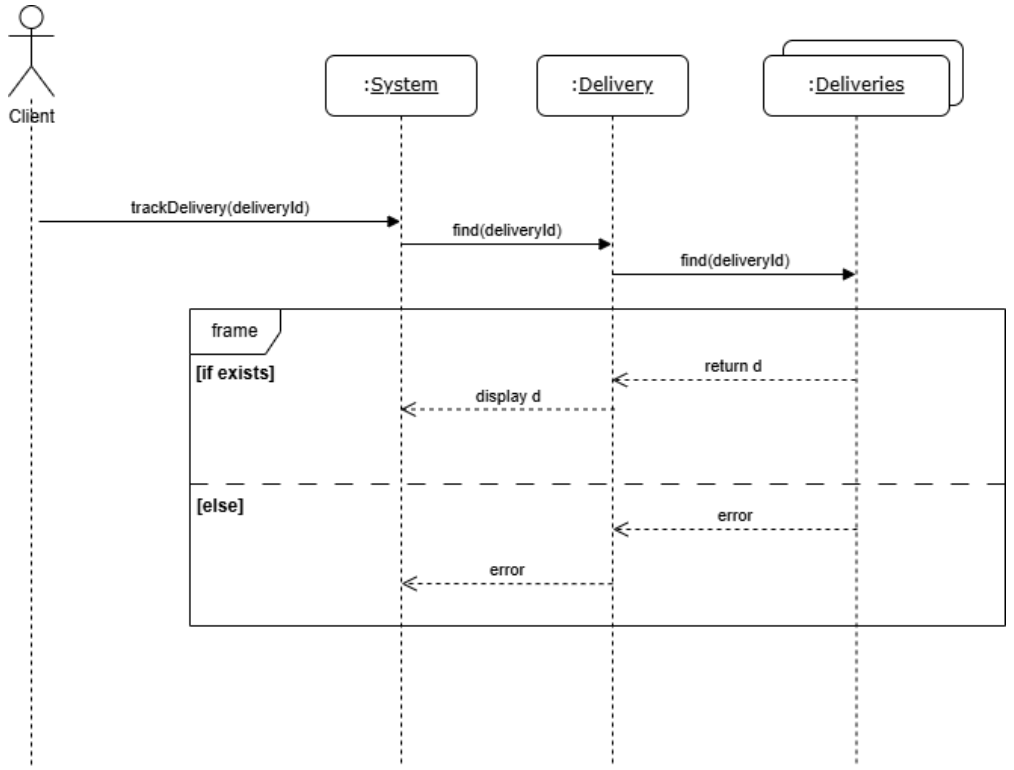

Figure 13. Updated sequence diagram for delivery tracking

The sequence diagrams were initially designed in sprint 2. Many of them required adjustments to show the processes that were redefined in sprint 4. Requesting a delivery and processing payments were combined into one sequence diagram, since we updated the delivery to only interact with the

database after a successful payment. Since the costCalculator class was created in sprint 4, the sequence diagram for requesting a quote had to include it. Additionally, all the diagrams were slightly modified to include the newly added or modified parameters and attributes.

4. Code Refactoring and Implementation

Sprint 3 involved the creation of the GitHub repository and the implementation of basic components to begin working on the 6 core features, such as the home page, delivery page, tracking page, quote page, about us page, the header component, payment form, transaction approved page, chatbot window, and the authentication modals for sign up and log in. Many of these components were simply front-end implementations to set up the GUI interface of our project, and did not interact with our back-end, which was not yet implemented. Additionally, Firebase authentication was implemented with our sign-up and log in forms to establish the process of creating and signing in with an account. The code refactoring in sprint 3 was minor, and only including the organization and renaming of files.

The GUI that was implemented in sprint 3 was a decent start, however one of the challenges that we faced in sprint 4 was having to complete more-than-anticipated GUI front-end designs. We learned that these should have been mostly complete, if not complete, by the end of sprint 3. Thus only minor changes and improvements, if any, would be required in the final sprint and more work can be focused towards the core features.

The work that was done in sprint 4 involved finishing the GUI for the front-end, completing the components used for the core features, and implementing additional components such as an admin management page which was used for CRUD operations on users and orders in our database. Our database was created using Firestore, and the front-end was connected with our back-end to implement the functionalities of our core features. The Google Maps API was

implemented so that we could use the Autocomplete API and the Distance Matrix API, which were essential when creating and calculating the cost of deliveries.

There were a lot of code refactorings involved in sprint 4. Many classes and methods were made to increase cohesion and reduce coupling from many components. For example, a costCalculator class was implemented to avoid duplicate code in multiple components. Certain parts of the existing code were modified to use design patterns, such as the factory pattern, singleton pattern, and the observer pattern.

One of the challenges we faced in this sprint was the amount of refactoring required. Since we implemented the design patterns towards the final stages of the project, we had to refactor more-than-anticipated amounts of code. We learned that the design patterns should have been considered at the beginning of the design process during the creation of the components that used such patterns. This would have been more beneficial since it would have required less refactoring towards the end of the project. We learned in practice that the cost of making significant code modifications becomes higher and higher the later we are in the development stages.

5. Discussion on Design Patterns

In Sprint 3, four design patterns were introduced to improve the system’s flexibility and maintainability. These four design patterns are Strategy, Singleton, Factory and Observer.

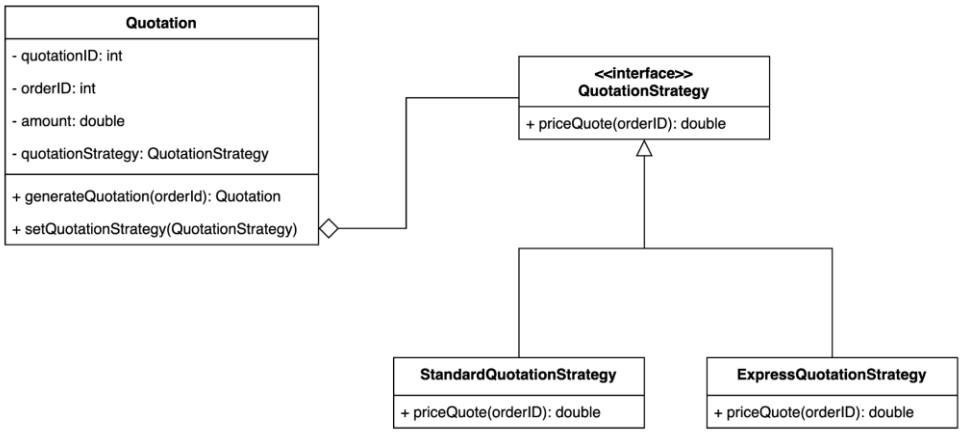

Figure 14. Strategy pattern presented in Sprint 3

The strategy pattern played an important role in the quotation service. Our system provided standard and express shipping. These 2 methods have different pricing and shipping strategies, the Strategy Design Pattern solves this problem by allowing the pricing logic for each type of delivery service to be encapsulated in its own class.

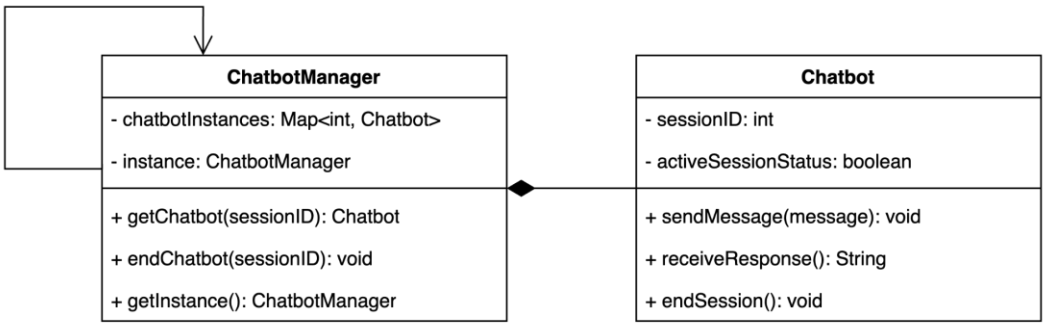

Figure 15. Singleton design pattern as presented in Sprint 3.

Each client requires an instance of the chatbot, this however would be tedious as there could be a very large amount of chatbot instances. By applying this pattern we ensure that one instance of the chatbot is created and this single instance would manage all clients.

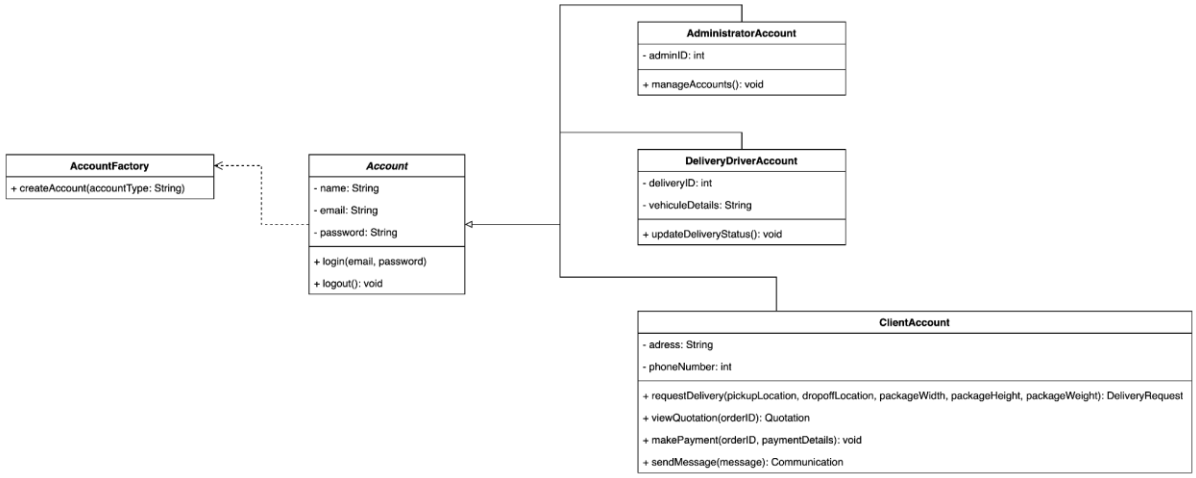

Figure 16. Factory pattern as presented in Sprint 3

The delivery service needs to handle multiple different client accounts based on their role. Each of these accounts have different roles and features. The AccountFactory handles the creation of these accounts. Instead of creating these accounts manually, the factory takes care of which type of account to create.

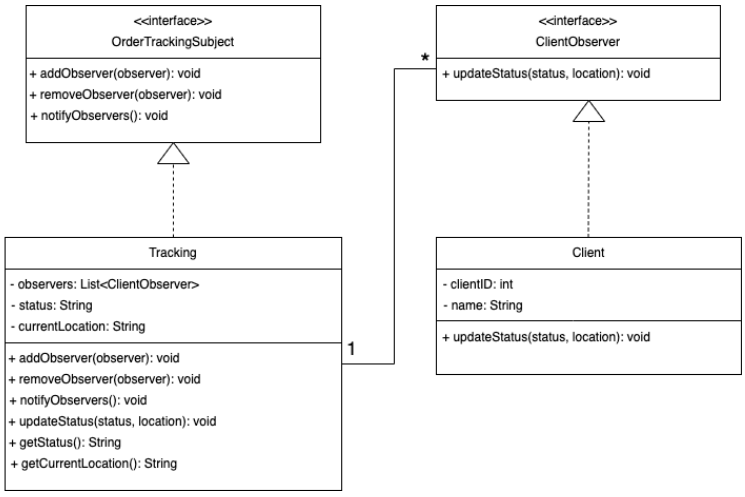

Figure 17. Observer pattern as presented in Sprint 3

Without the observer pattern, clients would have to manually check about the updates of their orders which would be very inconvenient. By implementing the observer pattern the tracking class (observer) can notify the clients

(subscribers) about the updates made to their orders. The clients automatically receive updates through the update() method whenever there is a status change.

In Sprint 3, the initial domain model, which only included entities like Tracking, Quotation, Payment, Delivery and Chatbot, was significantly refined. The Strategy Pattern enhanced the Quotation entity by introducing separate pricing strategies for standard and express delivery. The Singleton Pattern optimized the Chatbot by ensuring a single instance managed all client interactions, improving efficiency over the initial model's potential multiple instances. The Factory Pattern extended the model by introducing an AccountFactory to dynamically create client accounts, which were absent in the original domain model. Lastly, the Observer Pattern transformed the Tracking entity by enabling notifications for clients, eliminating the manual status-checking limitation of the initial model. These refinements added functionality and scalability to the basic structure.

6. Refinement and Discussion (Sprint 4)

In sprint 4, the class diagram was refined to reflect adjustments in the implementation of the existing design patterns to address specific design challenges more effectively. Instead of applying the Singleton pattern to the chatbot, it was applied to Firebase, creating a FirebaseSingleton class to centralize database interactions ensuring that only a single instance handles all operations. The Factory pattern, originally applied to client accounts, now manages Google Maps API input fields, enabling flexible, decoupled creation of reusable components across various features like the home page and delivery requests. The observer pattern was shifted from the tracking class to administrators who became the observer because they have the ability to modify accounts and orders and change the database. The clients then become subscribers, and they receive automatic updates whenever there’s been a

change in their account or order. Lastly, the strategy pattern remained applied to the delivery request, encapsulating the logic for standard and express shipping strategies to handle their different behaviours.

These refinements significantly enhanced the system's clarity, scalability and maintainability. Centralizing database interactions with the singleton pattern reduced ambiguity and ensured consistent operations. Using the factory design for Google Maps API input fields improved code reuse and addressed new requirements for integrating this functionality across components. Applying the observer pattern to the administrators ensured that clients would get updates, improving system responsiveness and user experience. Overall, these changes made the designs more modular and adaptable for future enhancements while meeting the needs of this project.

7. Project Goal and Objectives

- Design and develop a functional delivery system website.
- Provide a simple and efficient platform for clients to:
    - Create delivery requests.
    - Track their orders.
- Enable administrators to efficiently manage the system's database.
- Apply key software engineering concepts learned in class, including:
    - Diagrams.
    - Design patterns.
- Implement design patterns such as Singleton, Factory Method, and Observer to improve:
    - Modularity.
    - Scalability.
    - Maintainability.

8. Learning Outcomes and Importance of Architecture and Design

We learned that having a well-designed system architecture is very important in the design process. Creating a project with well-formed class and sequence diagrams, as well as the use of design patterns, leads to the creation of many classes and components with designated individual responsibilities that work together to collectively build a cohesive and organized structure. The system architecture provides a foundation for the creation of the entire system. Without it, the design process is disorganized and classes may be working in isolation instead of in collaboration.

Firstly, the class diagram allows us to set up every class and its methods from the beginning, instead of creating them in later stages of development. This allows every class to have their established responsibilities and work together from the beginning, whereas delayed implementation results in modifying (refactoring) existing code to accommodate these changes. Similarly, the sequence diagrams show the procedure of how methods and classes interact with each other in order to fulfill project goals. It provides the template in which components interact with each other to complete the project's features.

Design patterns are also very important, since they provide solutions to many common design problems. Without using patterns, the system can introduce many design flaws such as code duplication, low cohesion, and classes with many responsibilities instead of multiple classes with single responsibilities.

9. Alternative Approaches

For this project, we used Agile methodologies to split the progress of the project into 4 sprints. While this technique for development is very beneficial, it’s worth considering some alternative approaches. The waterfall method is a simple and reliable option, since the requirements of the project were unchanging and well defined from the start. It would still be completed in iterations, however each iteration would fully complete and test a section of the project before moving onto the next.

In this project, we used four design patterns: Factory, Observer, Singleton, and Strategy. The Facade pattern was another option for implementing multiple scenarios, such as payment options.

As for the technologies in the project, React was a very reliable front-end library. However, the back-end had other options that could have been worth trying, such as Django, Spring Boot, or Express.js. For our database, Firestore was reliable and simple to use - some other options that we could have considered were MySQL, MongoDB, or PostgreSQL.

10. Conclusion

The delivery system that we created demonstrates our ability to collaborate as a team and use our existing knowledge on software development methodologies while expanding our skill set for future projects. This project gave us the opportunity to apply newly learned concepts from this course, such as UML diagrams for system modeling and principles of system architecture design. Additionally, we improved our object-oriented design approach by utilizing GRASP and GOF design patterns, which improved the scalability, maintainability, clarity, and flexibility of our codebase.

The project was divided into 4 sprints, each with a well-defined goal: Problem Domain Study, Use Case and Sequence Diagrams, Implementation and Class Diagram, and Finalization and Refinement. The sprints demonstrated the iterative process of software development while respecting strict deadlines. Each sprint helped us refine and extend our system towards the final product.

Throughout the project, we were faced with challenges and obstacles that required refactoring, or sometimes redefining entire parts of our codebase and system architecture. Our diagrams and system architecture evolved greatly from their initial realization to adapt to the many changes that were made during implementation. Despite facing challenges, such as a reduction in team size due to unforeseen circumstances, we remained focused and successfully

completed a system that included all the core features and fulfilled its essential goals and objectives. While the project met the key objectives and goals, there are always opportunities for improvement and implementation of unique features.

This experience reinforced the importance of teamwork, adaptability, and communication in software development, preparing us to handle similar challenges in the future. Furthermore, we learned the importance of architecture and design so that we ensure the system is cohesive and flexible.

Overall, the project was another valuable learning experience that extended our theoretical knowledge and practical skill set. The challenges we faced and the lessons we learned strengthened our problem-solving skills that will be applied to larger, more complex software development projects in the future.