SOEN343
Software Architecture and Design

(FINAL REPORT)

Instructor:
Dr. Joumana Dargham

Team: Amazawn (343)

Layana Muhdi Al Tounsi (40125569)
Lauren Rigante (40188593)
Racha Kara (40210865)
Leo Brodeur (40216409)
Imane Madda (40208741)
Matthew Flaherty (40228462)

Table of Contents

1. Introduction:

The primary objective of this project is to develop and design a user-friendly web platform that offers a comprehensive delivery service system. The overall project scope revolves around the initial design phase, which involves creating various architectural diagrams before moving on to the implementation stage. Placing significant emphasis on these initial steps is crucial, as it helps guarantee the construction of a resilient platform that minimizes the likelihood of defects and aligns closely with the stakeholder requirements.

Our delivery service ‘Amazawn’ will provide an easy-to-use platform that takes care of the entire delivery process, from requesting the service to the post-delivery feedback. Among the key features we offer are instant quotation proposals, real-time communication, and easy tracking. Requesting our service is made easy and accessible since we provide all relevant information from the beginning, and a trustworthy feedback page is available directly on the platform.

Moreover we provide the possibility for senders to get their package picked up at their door, making them avoid having to travel to drop it off themselves. Transparency at every stage of the service is also ensured through detailed quotations, live tracking and continuous feedback. Users can easily request details about their product and will receive instantaneous and clear information. Additionally, users can quickly access and utilize our services without the hassle of creating an account. This is particularly beneficial for one-time or infrequent users who may not want to invest time in setting up an account for occasional use.

The significance of the final sprint report is to provide a comprehensive documentation and evaluation tool that encapsulates the iterative development process of the delivery system project. This report serves as a critical milestone, providing valuable insights into the achievements, challenges, and key decisions made throughout the project.

2. Combined Sprint Summaries

2.1- Sprint 1: Problem Domain Study and Initial Diagrams

The purpose of Sprint 1 was to define the project objectives, propose a comprehensive solution, specify the technologies and methodologies to be used, and begin the design and visualization of the system through a context diagram and domain model.

Since the project objective and proposed solution were discussed in the previous section of this report, this section will focus on the team structure and organization as well as the initial design stages of the project. Our team decided to use Discord as a main channel of communication, task assignment and document sharing. Weekly online meetings were also scheduled and held there. On the other hand, GitHub took care of progress tracking, coordination and version control.

The front-end developers worked with React.js, Node.js, CSS and libraries like ReactToast, Font Awesome Icons, and Axios, while the back-end team used Java. All members coded on VSCode or IntelliJ. For API's, we have used Postman for testing purposes and database populating, and Google's location API to convert locations to their respective longitude and latitude, as well as providing an easy to use address lookup for the user.

This sprint also entailed the beginning of the design process. We created our context diagram to visualize how the software interacts with external entities (See Figure 1). Each entity plays a crucial role in the system's functionality either by feeding or fetching data from the main delivery system. We also created our initial domain model in this sprint (See Figure 2), which shows the key real-world entities in our system and how they interact with each other.

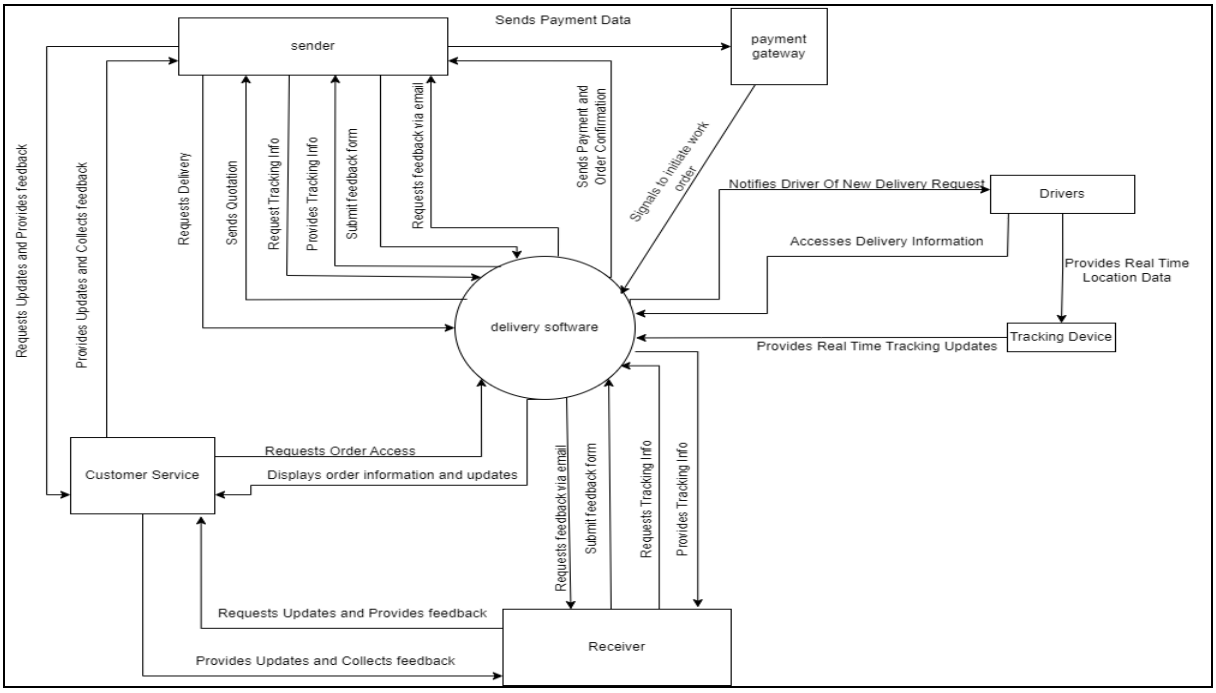

Figure 1 : The Context Diagram

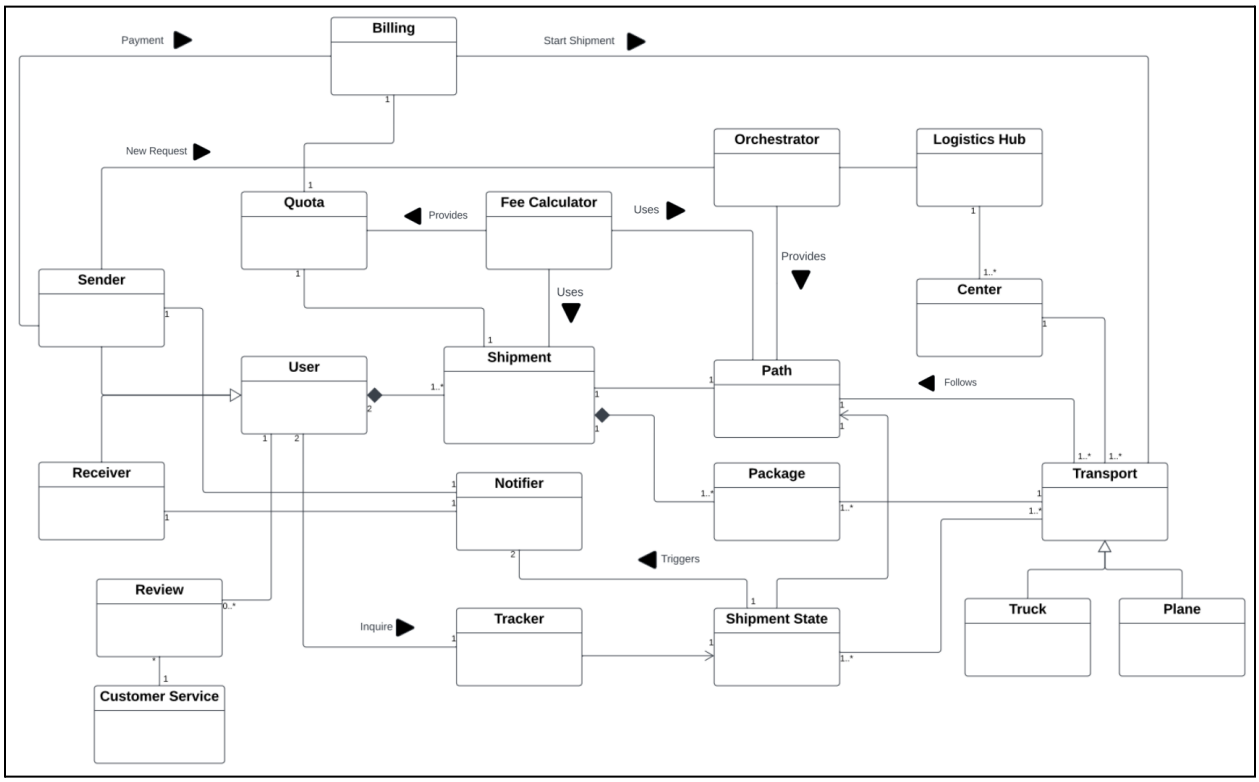

Figure 2: The Initial Domain Model

2.2 Sprint 2: Use Case Diagrams and Sequence diagrams

Sprint 2 focussed on developing a detailed system architecture, defining key use cases, and providing detailed use case scenarios and sequence diagrams for critical features in the delivery service platform.

The first step in this sprint was to define the system architecture. We separated our system into 6 layers: presentation, application, domain, data access, infrastructure, utilities and common. Each of these layers is represented as a package in our package diagram, some of them having nested packages. Figure 3 is our package diagram which provides a high-level view of the organization and structure of our system by showcasing how components are organized and interact.

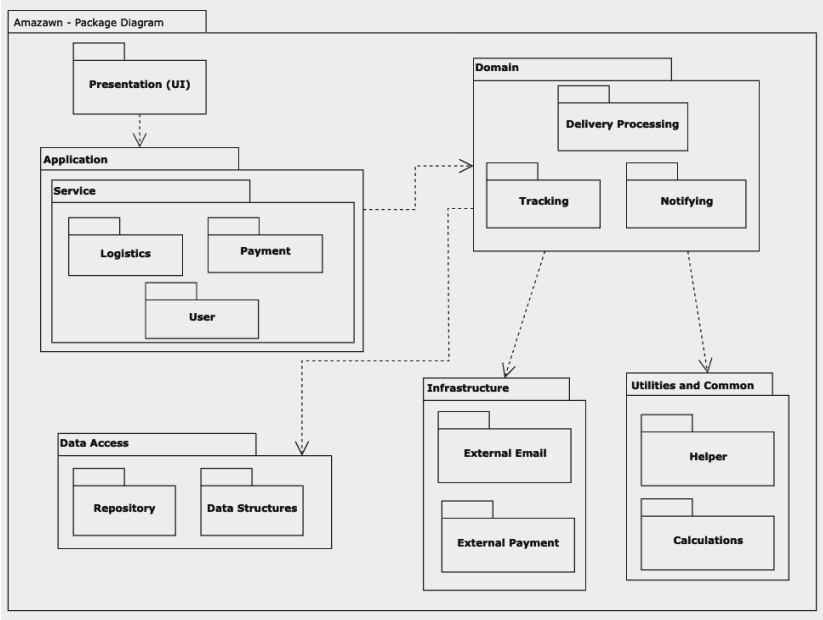

Figure 3: Package Diagram

1- Presentation Layer:

Packages: This layer focuses on user interaction, containing packages related to the user interface. Elements in this layer include the various forms and pages where users can directly interact with the application

Classes: UserInterface,RequestDeliveryForm,PaymentPage,ConfirmationPage, TrackingPage, SubmitReviewForm.

2- Application Layer (or Service Layer):

Packages: This layer contains the core application logic and services.

Classes: PaymentService,LogisticsService,UserService,NotificationService, ShipmentService.

3- Domain Layer (Business Logic):

Packages: The core business domain logic resides in this layer.
Classes: Center,Transport, Package, Shipment, User, Review, Ticket.

4- Data Access Layer:

Packages: This layer is responsible for data storage and retrieval.
Classes: CenterRepository,PackageRepository,ShipmentRepository, TransportRepository, UserRepository, ReviewRepository, TicketRepository,.

5- Infrastructure Layer:

Packages: This layer contains classes that interact with external systems and services.
Classes: PaymentGateway,EmailGateway.

6- Utilities and Common:

Packages: This section contains common utilities, constants, and shared resources.
Classes: ListHelper,LogisticsCalculator.

Next was the use case diagram, which provides an overview of the interactions between different actors and the system (See Figure 4). Here we define all possible use cases in the system.

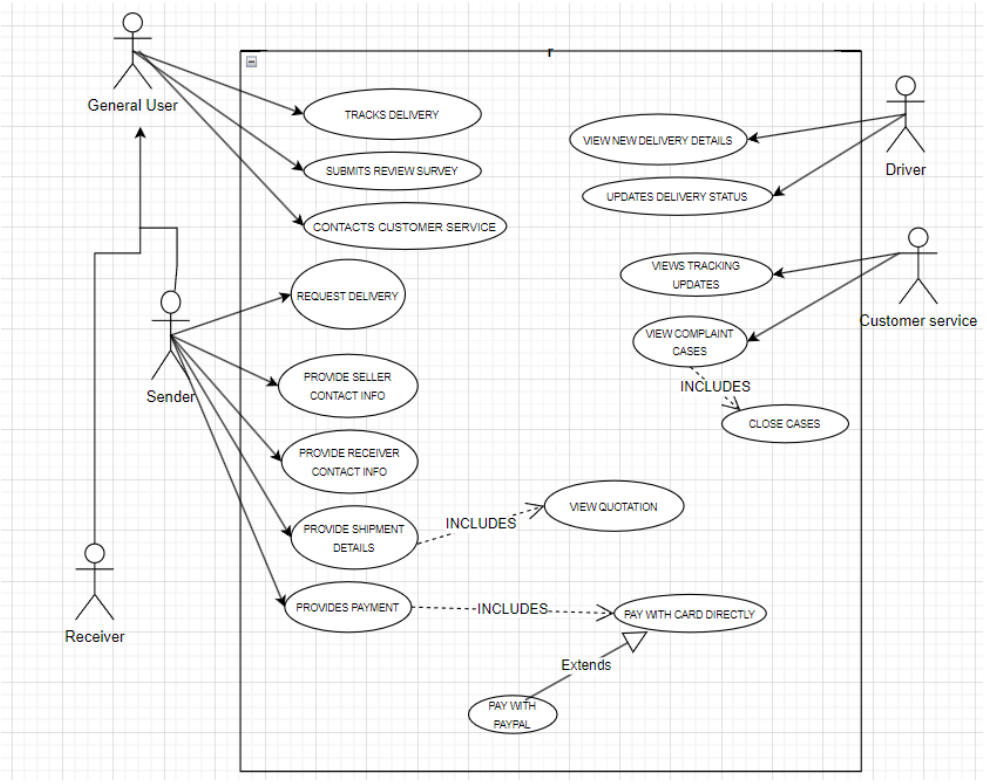

Figure 4: The Initial Use Case Diagram

Continuing the design of the system, use case scenarios for the 6 main features were created and then mapped to system sequence diagrams. Each SSD illustrates the interactions between actors and the system for their specific use case. They provide a dynamic view of how the system responds to external events. The SSD's will be discussed in more detail in section 3 of the report.

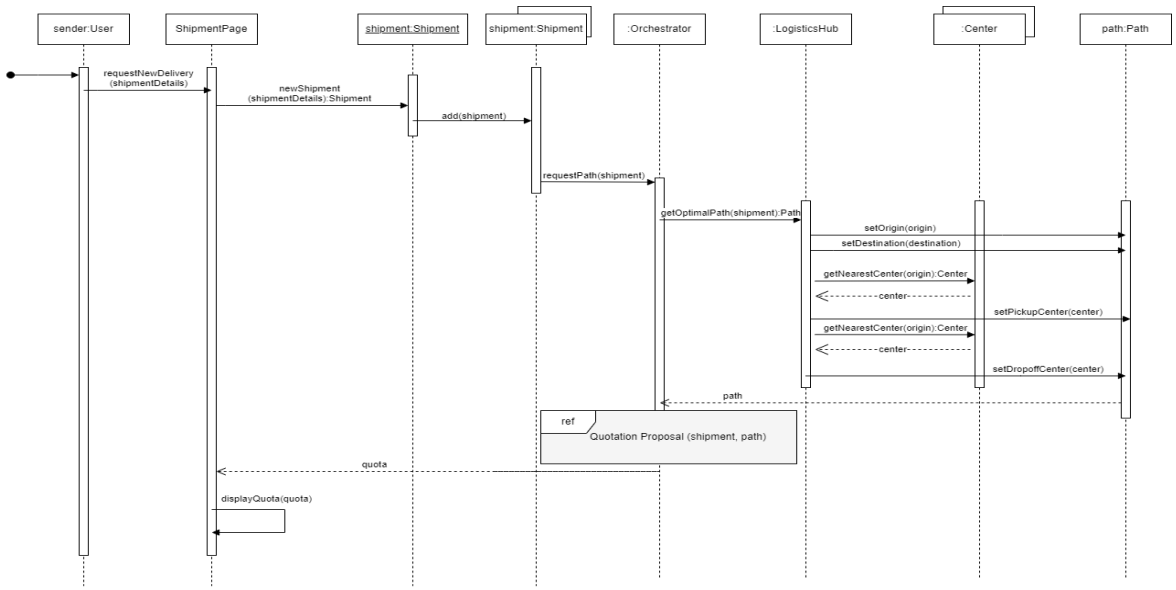

Figure 5: Initial request a delivery diagram

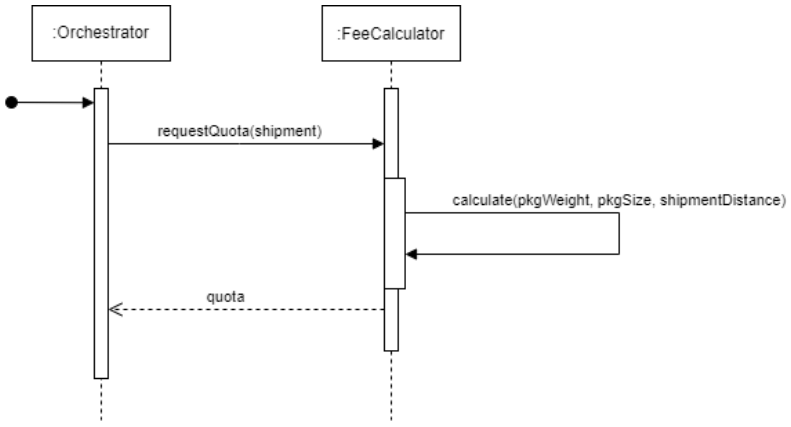

Figure 6: Initial proposal of a quotation diagram

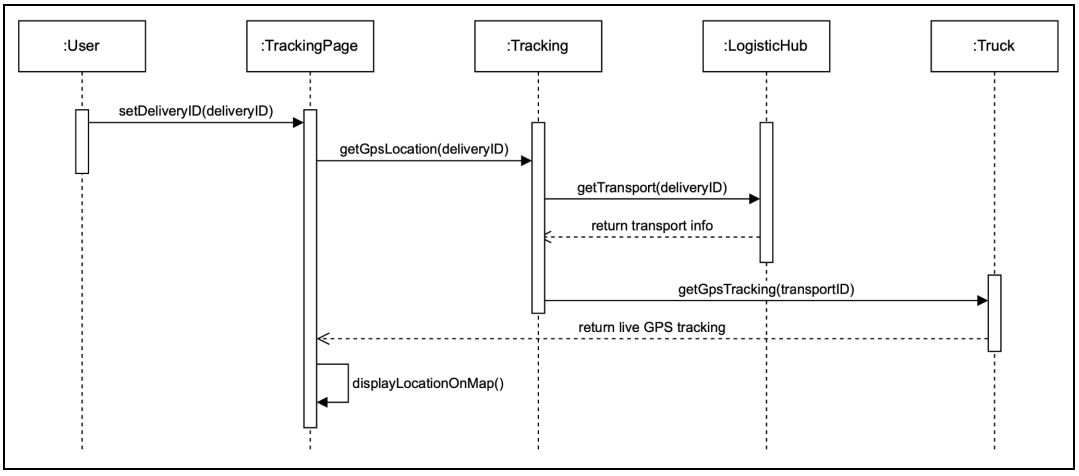

Figure 7: Initial tracking of the delivery diagram

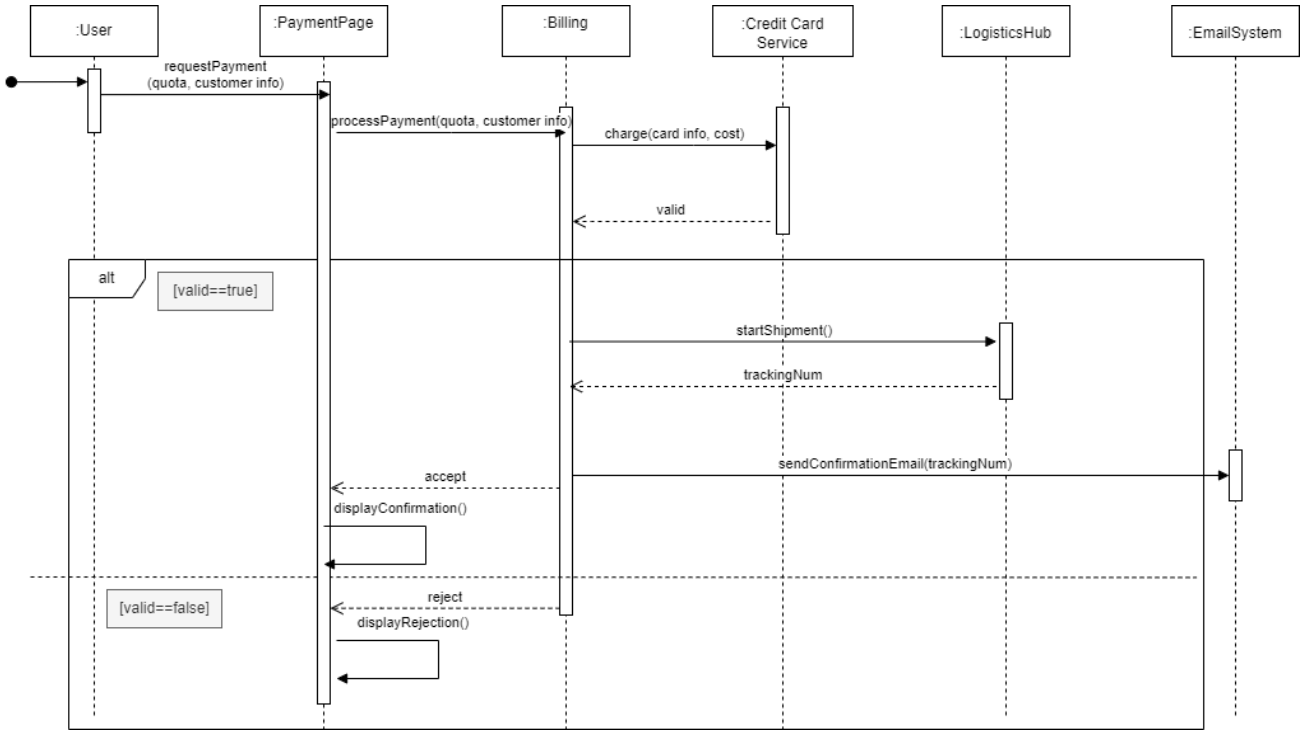

Figure 8: Initial complete a delivery diagram

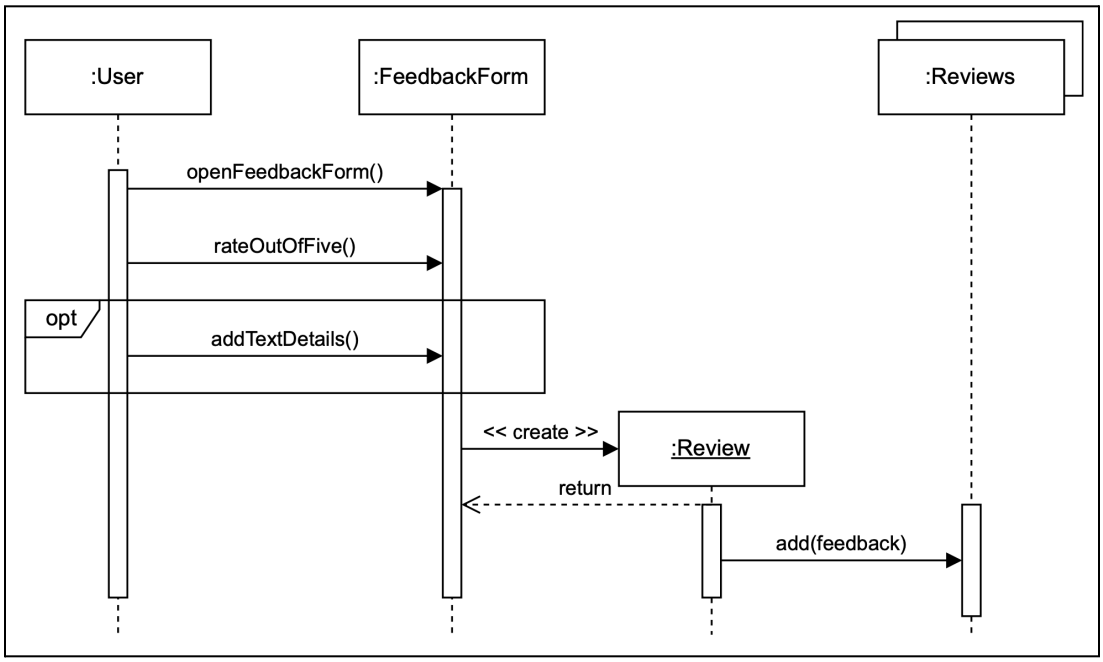

Figure 9: Initial leave a review of service diagram

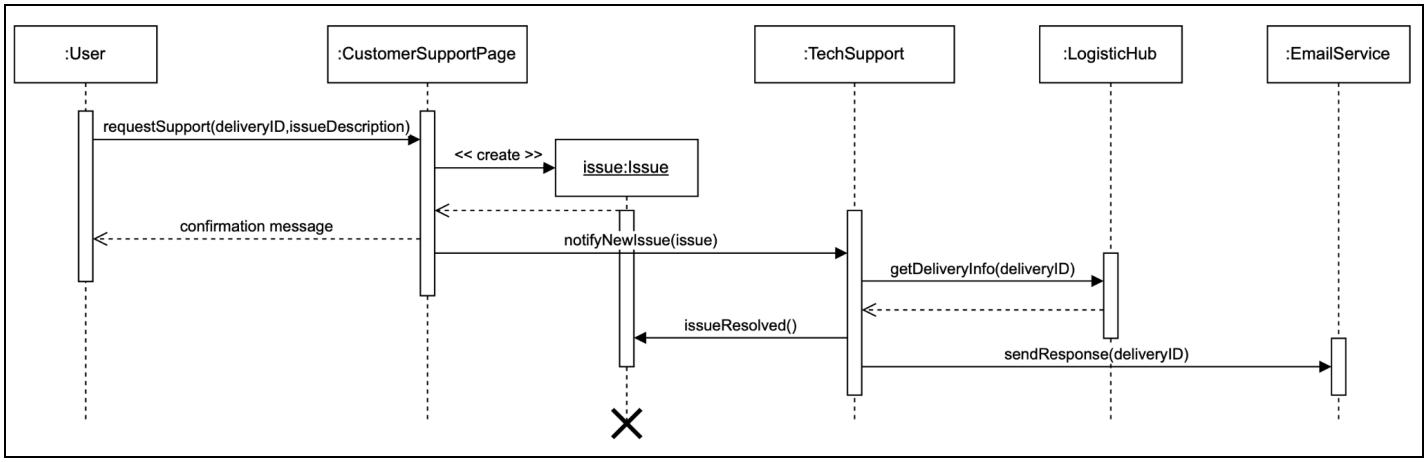

Figure 10: Initial contact customer support diagram

2.3- Sprint 3: Implementation and Design Class Diagram

The purpose of Sprint 3 was to create and analyze the class diagram (See Figure 5), as well as begin mapping our designs to code. Discussions about GRASP patterns, the process of going from domain model and SSD's to design class diagrams, and identifying differences between our domain model and class diagram were held. We identified GRASP patterns such as information expert and creator pattern. Moreover, we began the coding implementation of our website-specifically implementing the UI for the request delivery, track delivery, communication of service, review of service, and quotation features. GRASP patterns and the process of going from domain model to SSD to class diagram will be further discussed in sections 5 and 3 respectively.

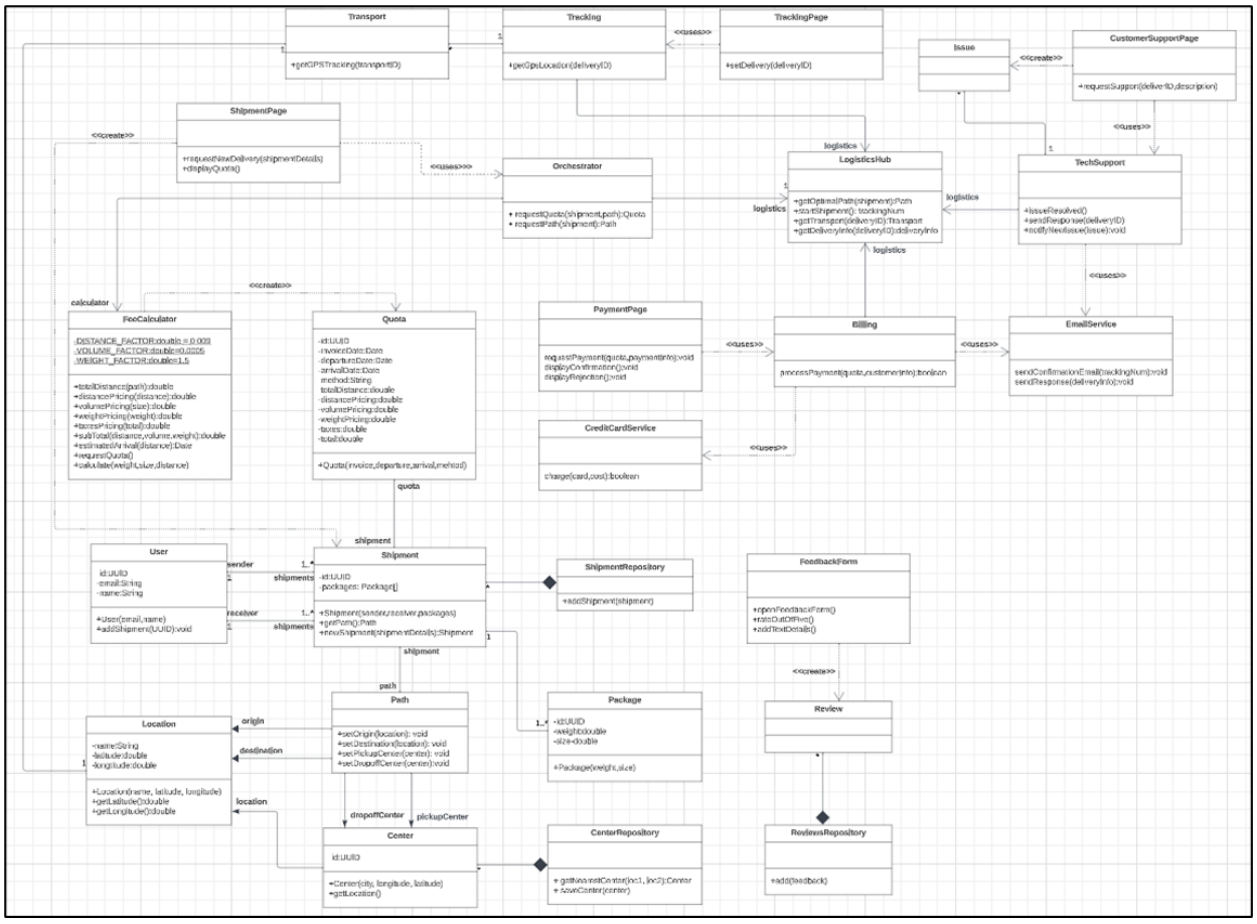

Figure 11: Initial Class Diagram

2.4- Sprint 4: Refinement and Finalization

The purpose of sprint 4 was to finalize the main features of the website, to refactor and test our code, and to refine our designs so they are cohesive with the codebase.

The refinement of diagrams at the end of the project is an extremely important step in documentation, as it provides a design consistent with the current implementation of the code. The refined diagrams provide future developers, team members, or stakeholders with an up-to-date reference that aligns with the current state of the project. This documentation is essential for understanding the design decisions and how they are reflected in the code. Refinement will be discussed further in section 6.

With regards to refactoring, we refined our architecture to follow the MVC pattern. We also cleaned our code to make it more readable and modular, promoting high cohesion and low coupling. Refactoring to MVC will be discussed in more detail in section 4 and high cohesion and low coupling in our system will be discussed in section 5.

The implementation of unit tests for the main features served as a robust mechanism to validate the correctness of the code. The tests provide a safety net for future changes and updates, helping to catch potential issues in the development process.

The attention given to code refactoring, diagram refinement, and the addition of unit tests in this sprint has resulted in a fully functional product. The website now maintains a well-organized and scalable codebase. The documentation, including refined diagrams, serves as a valuable resource for both current and future stakeholders and developers. With the successful implementation of these final touches, the project stands ready for use and sets the stage for any future enhancements or iterations.

3. Diagrams Evolution

3.1 - Domain model evolution:

Throughout the entire development process, our domain model remained remarkably consistent. The initial domain model we crafted at the beginning of the project aligned seamlessly with our implementation needs and requirements. As a result, there were no substantial changes or deviations from the original model. The domain model that the team built in sprint one is shown below (figure 12). The system comprises a sender and a receiver, both inheriting from the user class. The sender initiates a shipment request to the orchestrator, providing details like package dimensions and destination. The orchestrator calculates the delivery path, sharing information with logistics hubs and centers for order pickup and delivery. Shipment status updates, triggering the notifier to inform both sender and receiver, while users can track progress through a tracker. The fee calculator determines the bill using the path and package dimensions. After delivery, users can leave reviews stored and forwarded to customer service, creating a seamless, transparent, and user-engaging process from shipment request to review submission.

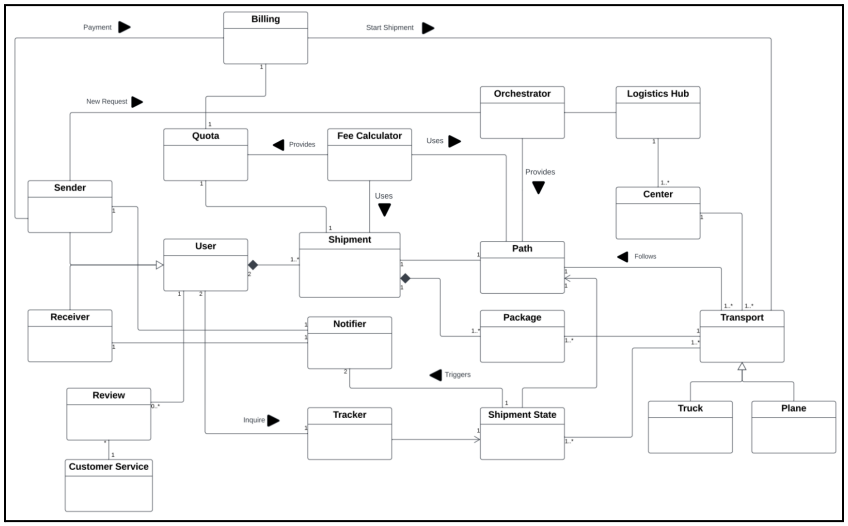

Figure 12 - Domain Model, first and final phase

3.2 - Use case Diagram evolution:

In sprint 2, we developed the first use case diagram (Figure 13), that clearly reflects the user stories and use cases that would favor our users-both senders and receivers. The first version of our use case diagram was sufficient enough for us to start building and deciding on what main

features were to be implemented into our system. However, some slight refinements were necessary in correction of small mistakes that were made while building the first use case diagram. Specifically the misuse of “extend” arrows that were initially to break down the payment use case per conditions. However, after careful reviews and viable feedback, we have concluded that there is no place for a condition in that case and decided to remove the extend option as seen in figure 14.

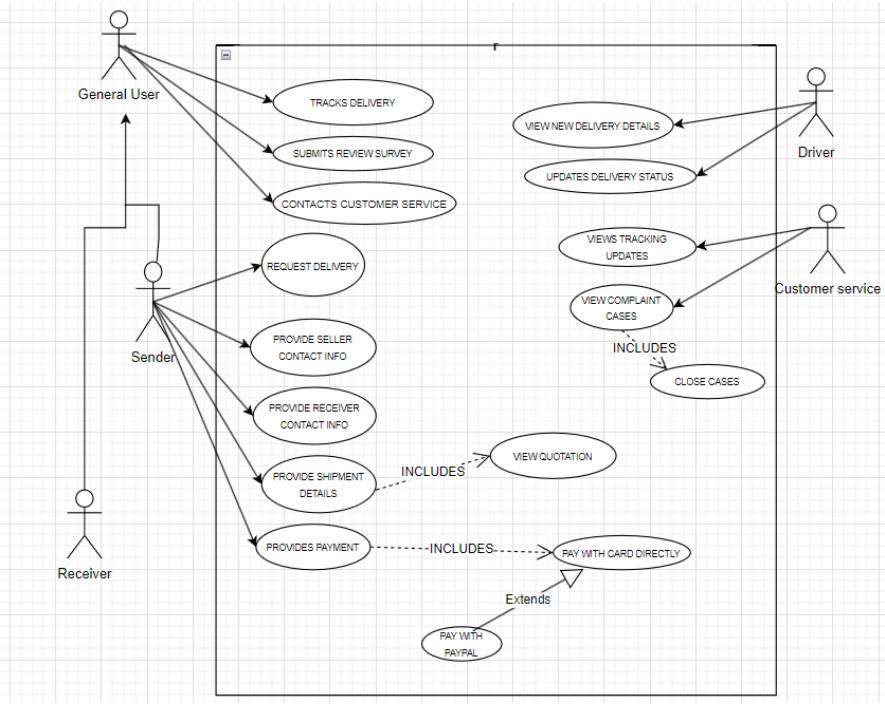

Figure 13- Use Case Diagram 1.0

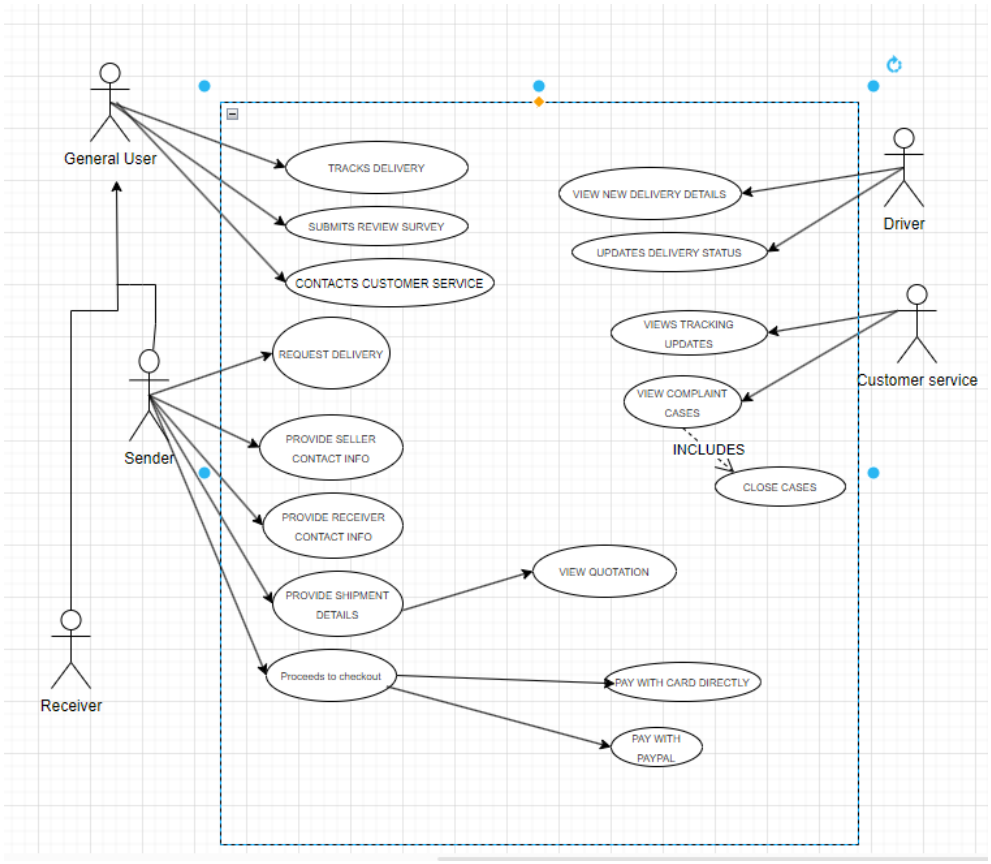

Figure 14- Use Case Diagram 2.0

3.3 -Sequence Diagrams evolution:

3.3.1- Request a Delivery Sequence Diagram evolution:

The updated sequence diagram for “Request a Delivery” feature shown in figure 16 introduces a major modification to the message order and assignment of responsibilities (figure 15). In order to act as the controller for shipment processing processes, such as generating quotations, a new class called "LogisticsController" has been introduced. The "Logistics" class, an Information Expert with the required data for creating shipment objects and generating quotations, is tasked by the LogisticsController with starting a new shipment. Now, by making separate calls to the FeeCalculator class, the Orchestrator class receives the coordinates of the

closest centers and computes the total distance, expected time of arrival, and price. Crucially, the shipment object cannot be created until the quotation is approved. Furthermore, the charge computation is now included in the purview of this use case.

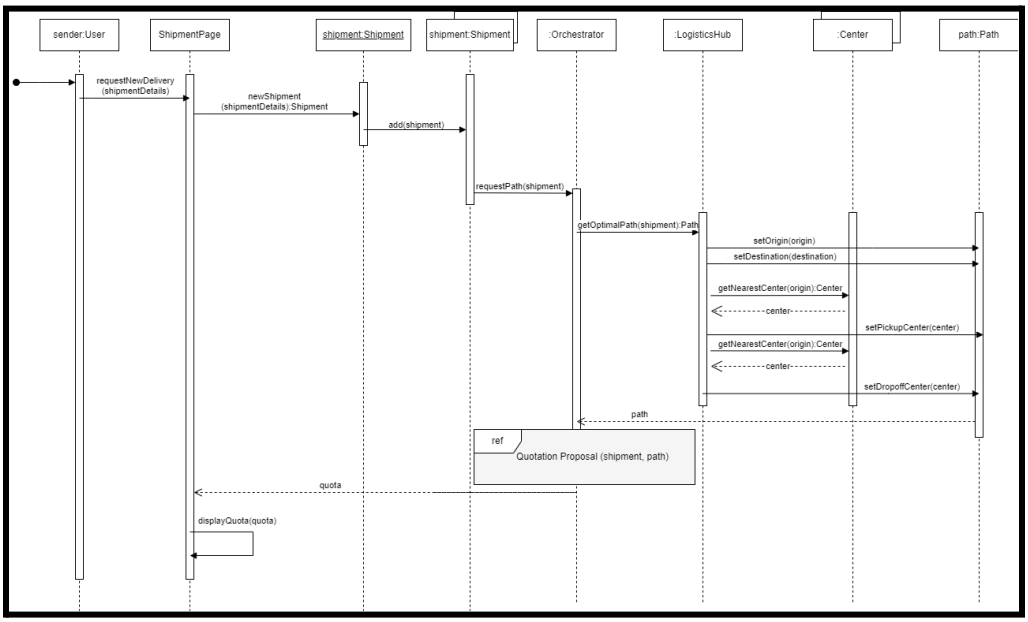

Figure 15- Request a Delivery 1.0

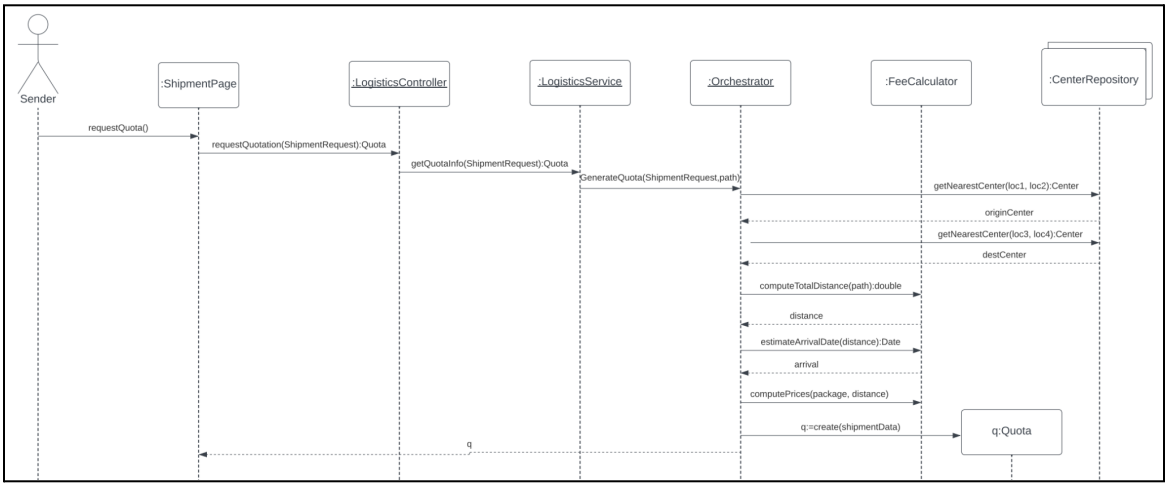

Figure 16-Request a Delivery 2.0

3.3.2- Accept a Quotation Sequence Diagram evolution:

The goal of this refactored sequence diagram (figure 18) is now to accept a quota and start the shipping by storing the necessary information, rather than to generate a quotation as previously illustrated in figure 17. The procedure entails managing package, shipment, user, and quota objects and making sure they are stored after the logistics controller initiates the shipment through the logistics service. The division of design and coding into two independent phases is where these modifications are most significant. While receiving and storing the offered information is the focus of the second stage, calculations and information proposals are made in the first. This division is achieved by means of two consecutive pages including unique user prompts, thus encouraging a more lucid and systematic workflow.

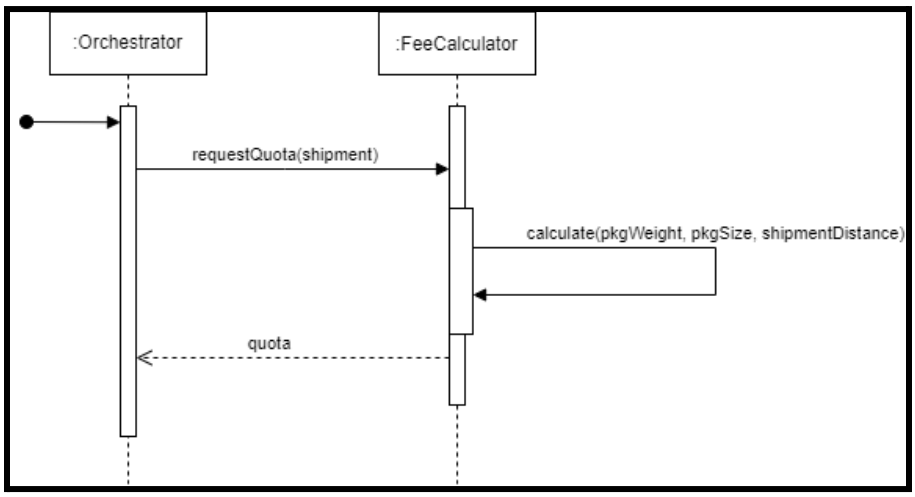

Figure 17- Accept a Quotation 1.0

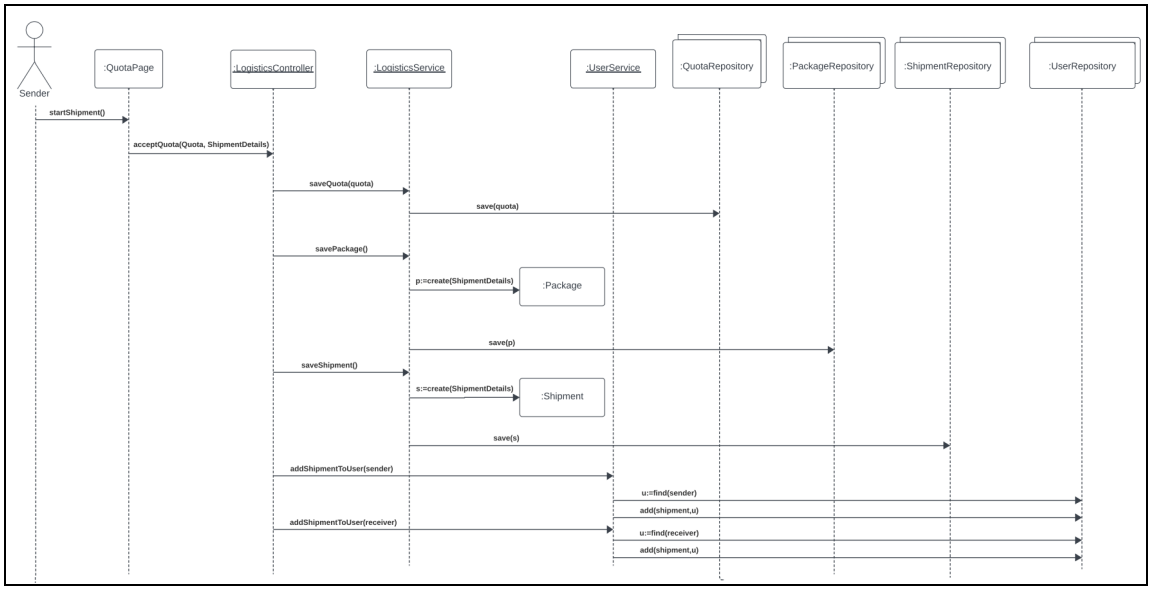

Figure 18- Accept a Quotation 2.0

3.3.3- Communication about the service (new ticket) Sequence Diagram Evolution:

We narrowed the scope of this sequence diagram and concentrated on adding and creating tickets to the repository (Figure 20). The previous version, shown in figure 19, encompassed the entire ticket resolution process, from assignment to technical support to email answer. We discovered that this Use Case only covered the user-side portion of the ticket creation process. Attempting to classify too many (smaller) Use Cases under "Customer Support" would simply make the operation too complicated to be categorized as a single scenario.

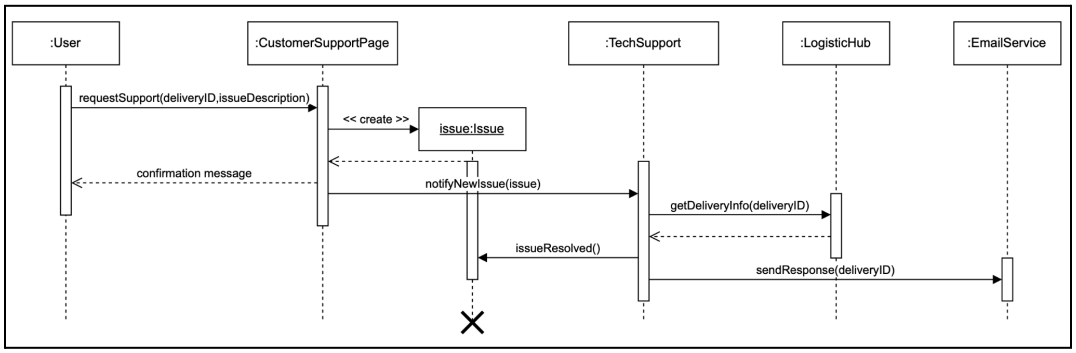

Figure 19- Customer support 1.0

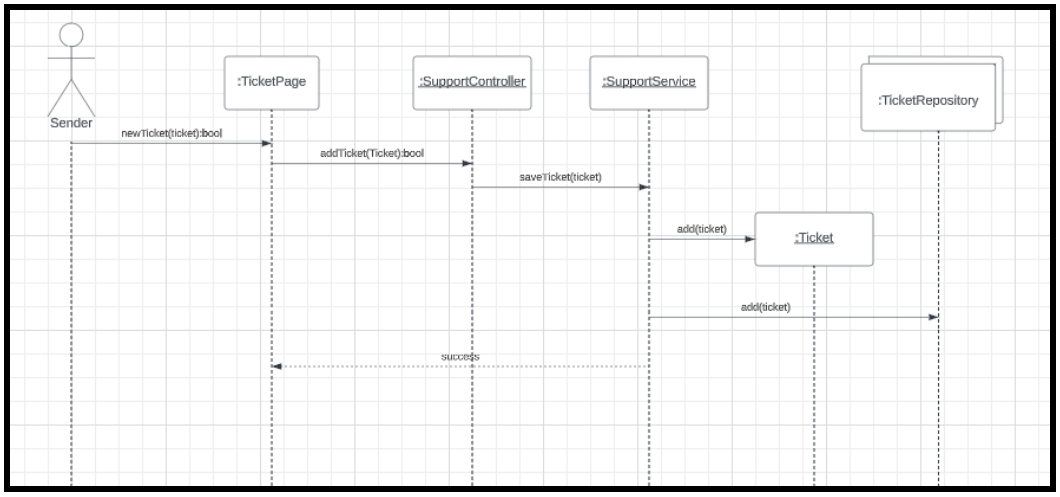

Figure 20- Customer support 2.0

3.3.4- Track Delivery Sequence Diagram evolution:

The Tracking page, where delivery data are accessed by entering tracking information (delivery ID), remains the focal point of user involvement in the modified sequence diagram. Renaming the LogistricHub class to LogisticsService and incorporating Tracking class methods within it are two significant changes. To handle tracking-related operations between classes, a new intermediate class called TrackingController is introduced. For better organization, structural improvements include replacing the Truck class with a Tracker class and TrackingData structure, as well as adding a ShipmentRepository multi-object. These adjustments are important for optimizing the architecture and structure of the system. By adding a new TrackingController class and expanding its functionality as a controller, the Tracking class was consolidated into the LogisticsService class.

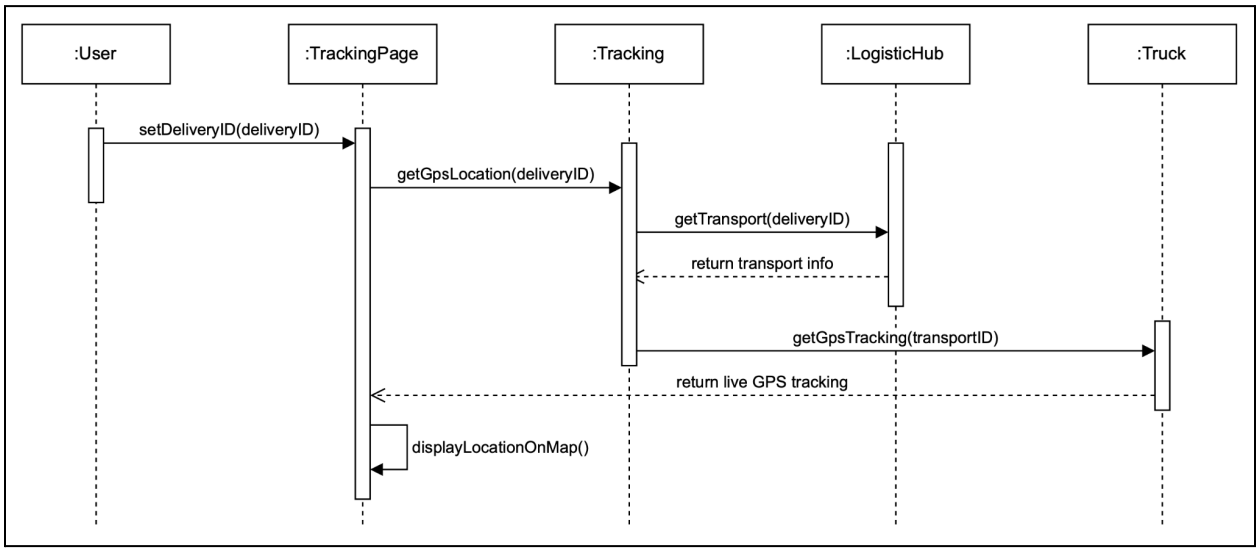

Figure 21- Tracking delivery 1.0

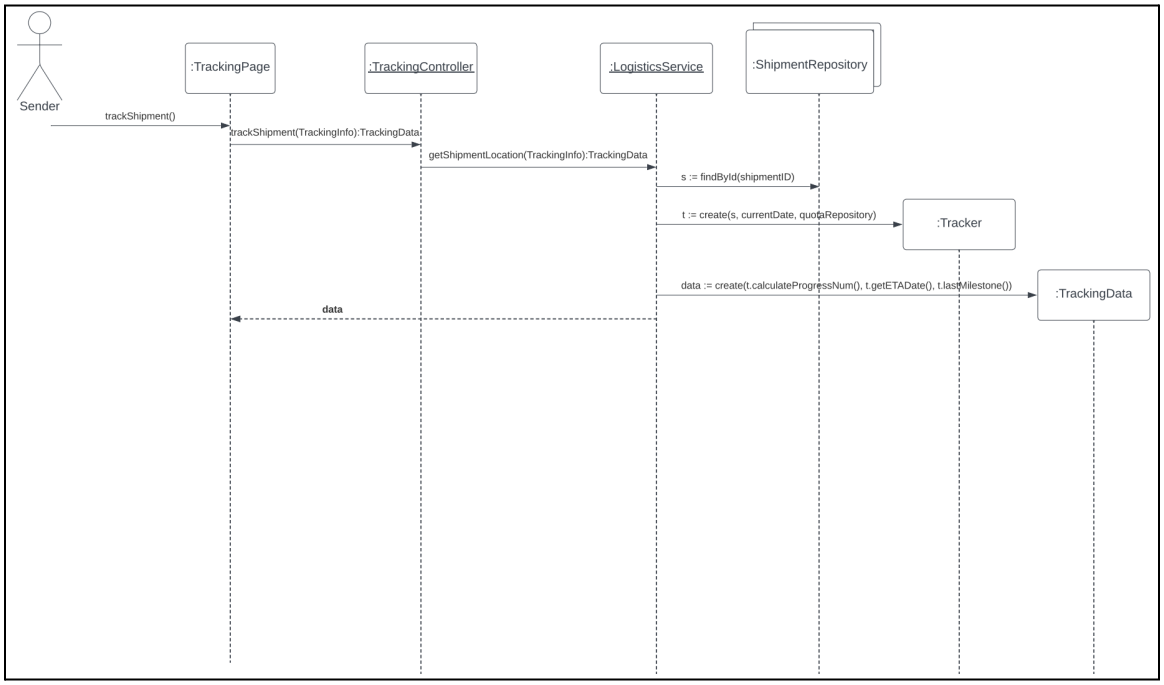

Figure 22- Tracking Delivery 2.0

3.3.5- Review Service Sequence Diagram evolution:

Since the user only uses the Rating page to review the system after the delivery has been made, we first swapped out the FeedbackForm object with it. Additionally, rather than allowing the review form to build the Review object directly, we delegated the task of doing so to the SupportService. The SupportController class, which is in charge of allocating responsibility for any action pertaining to the Review feature, transfers this duty to the SupportService class.

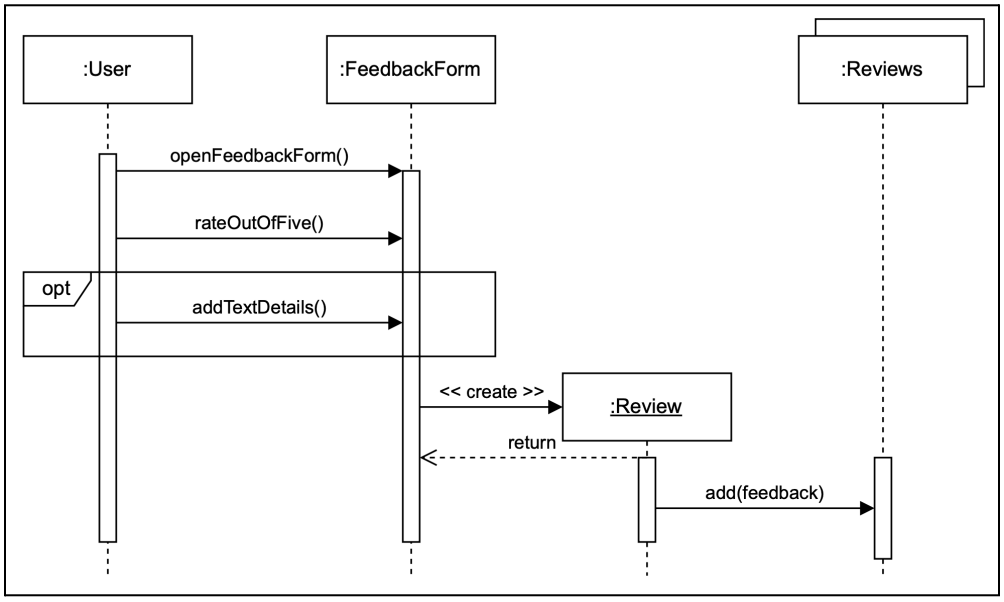

Figure 23- Review service 1.0

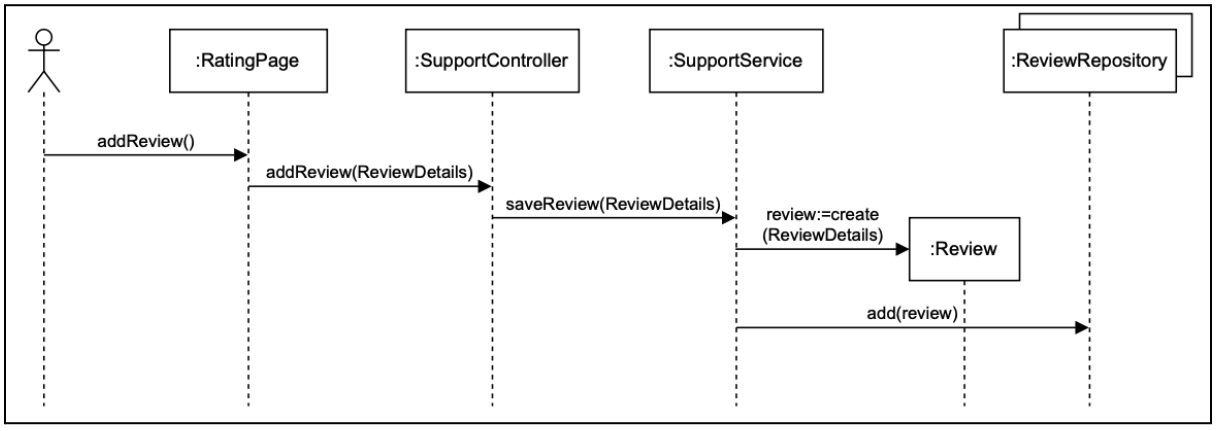

Figure 24- Review Service 2.0

The remaining sequence diagram shows the payment service was not refactored, as the feature was not implemented in the scope of this project.

3.4 -Class Diagram evolution:

Our project has undergone a thorough restructuring as shown in figure 26, with multiple modifications made to comply with the MVC architecture, increasing cohesion and minimizing coupling. New service layers and controllers, including LogisticsController, TrackingController, EmailService, UserService, LogisticsService, and SupportService, are noteworthy additions. On the other hand, some parts were simplified by removing others, such as TechSupport, Billing, Transport, CreditCardService, and Path. The change entails switching to a more standardized methodology in which all pages interact with controller classes via post and HTTP requests. In order to improve coherence, the logic of the Logistics Hub has been moved to LogisticsService, where duties are divided among subclasses. Tracking is now part of the logistics service; however, SupportService is the only one that handles reviews and tickets. EmailService is separated from TechSupport at the logistics controller's command.

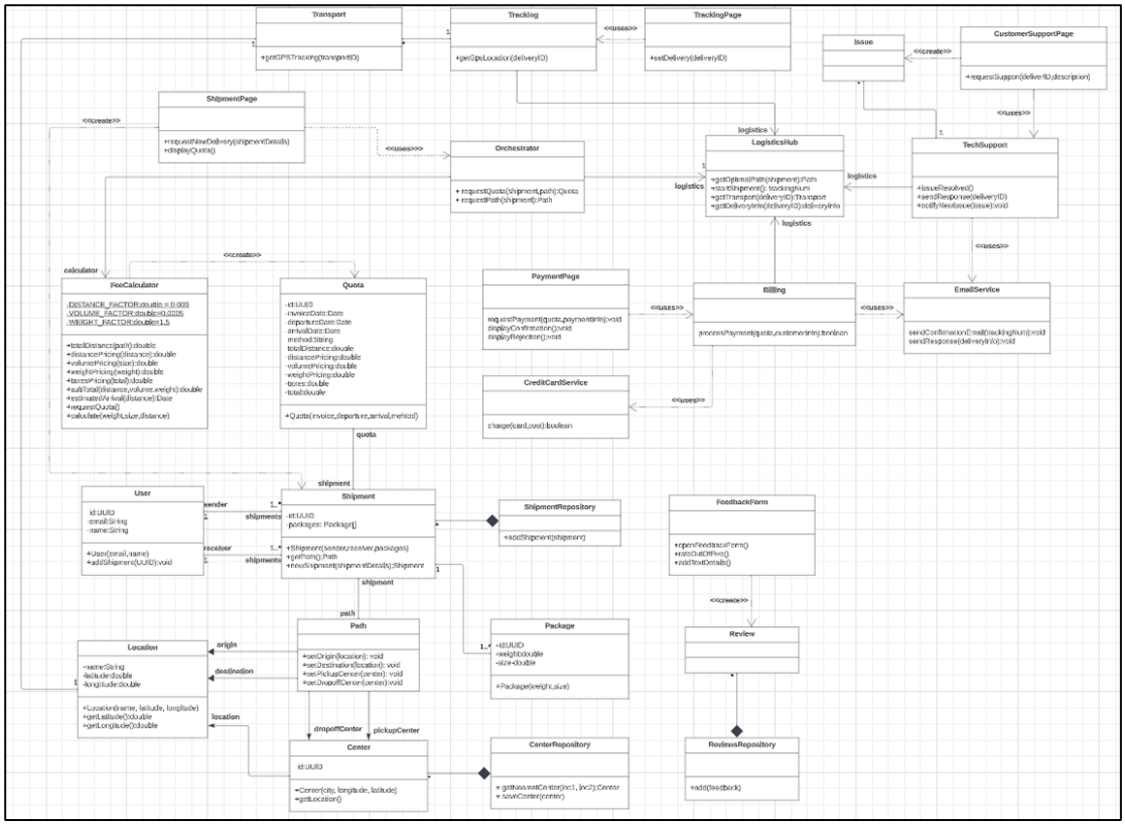

Figure 25- Class Diagram 1.0

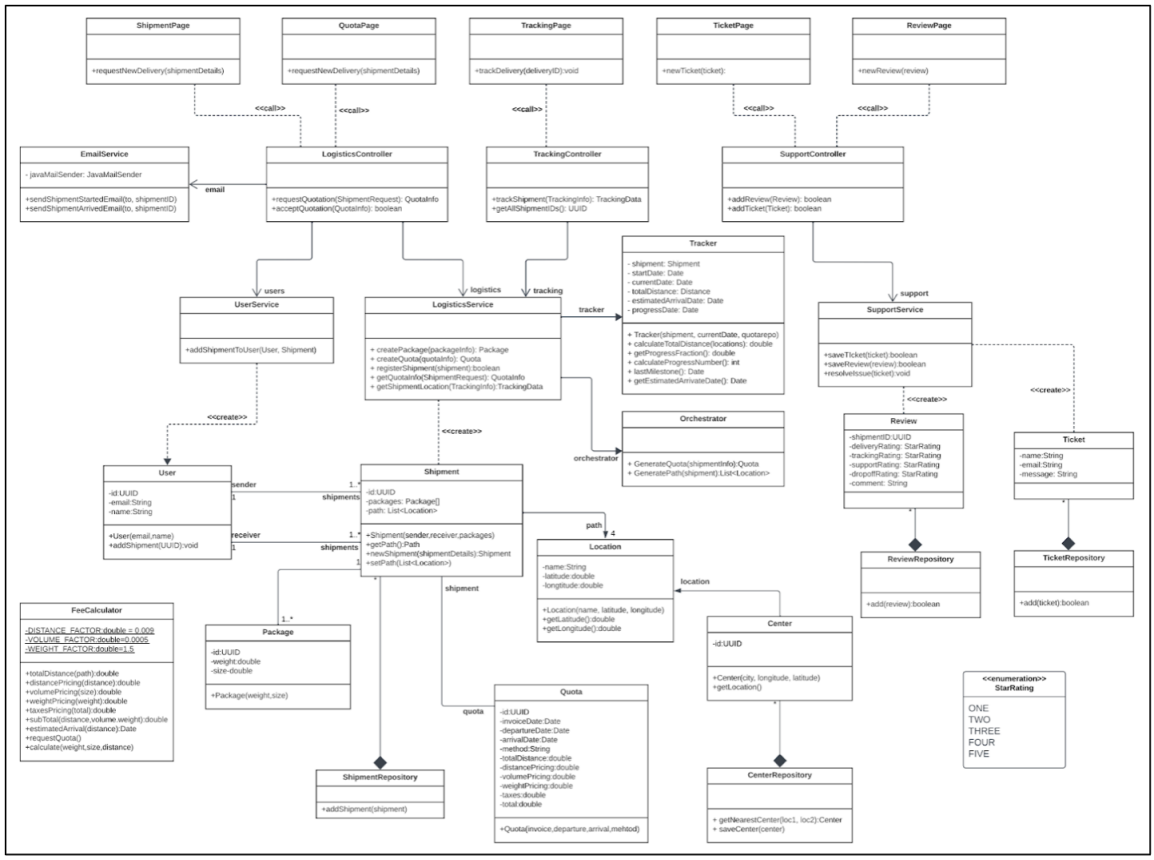

Figure 26- Class Diagram 2.0

4. Code Refactoring and Implementation

The main forms of refactoring done in the codebase were reformatting code, removing excess comments, deleting unused assets, reducing coupling, and promoting high cohesion. We also refactored our system architecture to follow the MVC architectural pattern.

The structure refactor to fit the MVC architectural design pattern introduced many changes to the backend code. Two new layers, the controller layer and the service layer, have been introduced to the system. The controller layer houses controllers for frontend interactions, while the service layer, following the creator pattern, serves as the sole interaction point for database tables. This prevents previous processes from having database coupling and logic computing.

In regards to the frontend, we were able to achieve clean code thanks to using React. In React, one of the core principles is component-based development, which has proven to be incredibly beneficial for creating modular, clean, maintainable, and efficient code. There is minimal code duplication, as we instead are reusing our React components wherever necessary. The modular nature of React components promotes high cohesion because each component is designed to be self-contained and independent.

Nonetheless, we still refactored our code. Our first form of refactoring included removing unused components and assets from the codebase. We reformatted the code, so all spacing and indentations were fixed to ensure a cleaner and more readable code. Moreover, we removed excess and unnecessary comments. A best coding practice regarding comments is to explain why, not what. The code should be able to explain the “what”, and it is therefore unnecessary to explain it in comments. Following this best practice, we deleted any comments we thought to be redundant to further promote readable and clean code.

5. Discussion on Design Patterns

The two GRASP patterns that were initially used in our class diagram and were also used after sprint 3 are the Information Expert pattern and the Creator pattern. The Information Expert pattern is used to assign a responsibility to the class that has the information needed to fulfill that responsibility. As for the Creator pattern, it is used to assign class B the responsibility of creating an instance of class A if B contains A, B aggregates A objects, B records instances of A objects, or B has the initializing data that will be passed to A when it is created. Although some classes were modified, renamed or created following these two patterns, these concepts remain present in the updated models which can be seen in the updated domain model and class diagram that found in section 3 of the report, where diagram evolution was discussed. However, after sprint 2, the decision to implement the Model View Controler was viewed as being beneficial for the progression of the website as it is easier to maintain and extend in the future. In the MVC model, there are 3 main views, the View which is the user interface that the user of the website interacts with, the Model which houses all of the business logic, such as how data is managed, distributed, and stored, and lastly, the Controller which are dedicated classes that are responsible for redirecting data from the View to the Model and vice vera.

5.1- Information Expert Pattern

The Shipment class is viewed as the information expert since it is the main class that has access to all the required shipment information throughout the delivery system. As can be seen in the class diagram, the Shipment class has direct access to the User class (both sender and receiver), the Package information class, the Location class, the Quotation class(Quota), and the Shipment Repository class. As the information expert, the Shipment class is responsible for providing the relevant paths (Origin, Destination, Pickup Center, and Dropoff Center) for the shipment object using the getPath() method. Therefore, by having access to all these shipment details, the Shipment class fulfills the description of “knowing” and “answering” by creating a shipment object, accordingly.

Another class that can be considered as an information expert is the LogisticsService class. This class fulfills the “knowing” requirement by having access to the LogisticController, the TrackingControler, the Tracking, and the Orchestrator classes. These classes provide different

information about the package, the quotation assigned to that package, its location, and the overall tracking details. The information can be retried via the getter methods listed in the information expert class such as getQuotaInfo(ShipmentRequest) and getShipmentLocation(TrackingInfo) methods.

Additionally, the Tracking class is also categorized as an information expert since it houses all of the tracking details for a given shipment package from the distance required to travel to reach its destination, to the progress date if applicable, and to the estimated date of arrival. This information is used by the LogisticsService class via the getShipmentLocation(TrackingInfo) method.

5.2- Creator Pattern

On the class diagram, the responsibility of creating a new instance of some class is recognized by the keyword <<create>> that labels an arrow going from the class B responsible for creating instances of class A. This label shows that, for example, the LogisticsService class is responsible for creating instances of the Shipment class, other pairs sharing this relationship are the SupportService class which is responsible for creating both instances of the Review and Ticket classes. The Creator pattern is also recognized through aggregation relationships, such as between the CentorRepository class and the Center class, the ReviewRepository class and the Review class, the TicketRepository and the Ticket class, and the ShipmentRepository class and the Shipment class.

5.3- Controller Pattern

After refactoring the diagrams, the Model View Controller pattern can clearly be distinguished in the updated domain model and class diagram in Figure 27. The View that is in contact with the user of the website are the various website pages that are identified by the ShipmentPage, QuotaPage, TrackingPage, TicketPage and the ReviewPage classes in red in the updated class diagram. For the Control category of the MVC model, new controller classes were added to share data between both the View and the Model classes, these are the LogisticsController, the TrackingController, and the SupportController, found in green. Lastly, the Model classes that are responsible for managing the data provided or passed by the various controller classes are the

EmailService, UserService, LogisticsService, and SupportService classes, shown in blue in the partial class diagram provided.

5.4-Impact of Pattern Implementation

The overall impact of using both the Information Expert and the Creator GRASP patterns resulted in lowering coupling and increasing cohesion of the software. In general, when assigning dedicated classes to house certain information and assigning specific classes to create specific instances of other or the same class, we are lowering coupling by reducing the dependency of multiple classes to a dedicated class and thus we are increasing cohesion by assigning certain functionalities to a specific class such as being responsible for instance creation. Additionally, by implementing the Model View Controller pattern we are further reducing coupling and increasing cohesion by categorizing classes into one of the 3 categories described in this pattern, which are known as the Model, the View, and the Controller. By adding dedicated classes to control the data flow between the View and the Model, we have not only increased cohesion and lowered coupling but we have also clarified and simplified the overall functioning of the software and how each class plays a role in its functioning.

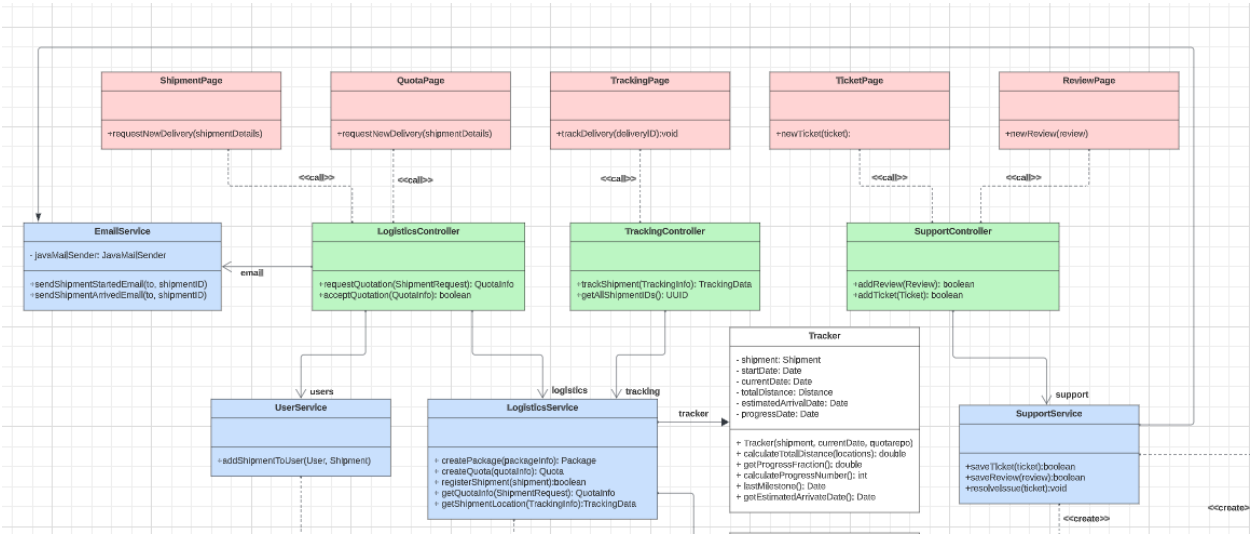

Figure 27: Partial Class Diagram Displaying MVC Model

6. Refinement and Discussion (Sprint 4)

6.1- Class Diagram

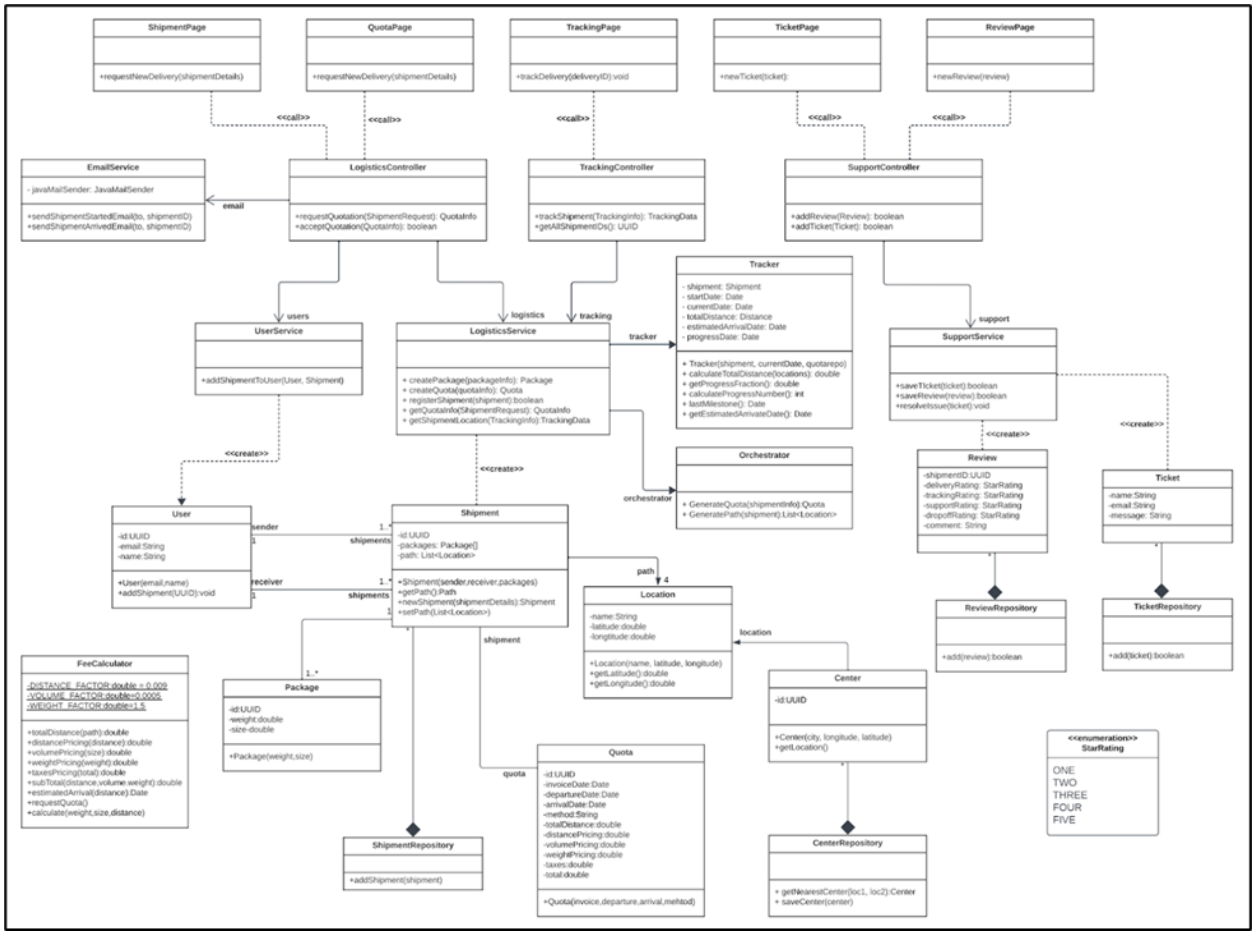

Figure 28- Class Diagram

Firstly, after the MVC overhaul, new Controller Layer components, including LogisticsController, TrackingController, and SupportController, have been integrated. Simultaneously, a Service Layer has been introduced, along with services such as EmailService, UserService, LogisticsService, and SupportService. Along with these additions, certain elements have been removed from the system, notably Path, Billing, CreditCardService, Transport, and TechSupport. The changes extend further into the system's core functionalities. Now, every page interacts with controller classes exclusively via POST and GET HTTP requests, aligning with standardized protocols. Notably, the Logistics Hub logic has been transferred to LogisticsService, ensuring reduced responsibilities and the distribution of tasks across subclasses. Furthermore, since tracking has been incorporated into the logistics business, it is no

longer a stand-alone process. One of the more recent upgrades is the separation of tickets and reviews, which are now managed exclusively by the SupportService. EmailService is started by the logistics controller now that it is not dependent on TechSupport. Because they reorganize the project to adhere to the Model-View-Controller (MVC) architecture, these modifications are mostly noteworthy. Increased component cohesion, fewer connections between various classes, and the compartmentalization of concerns are the goals of this restructure.. Pages now only communicate with controllers, and services follow the creator pattern and act as the only point of contact for database tables. Classes now aren't going to become god classes thanks to the reorganization of procedures like the logistics center, tracking, and support services. The goal is to simplify the system and bring it closer to actual needs by getting rid of some parts, including billing and using the tracker to simulate transport programmatically. Lastly, the decision to externalize tech support and ticket resolution via external email threads enhances system flexibility and adaptability.

6.2- Request a Delivery

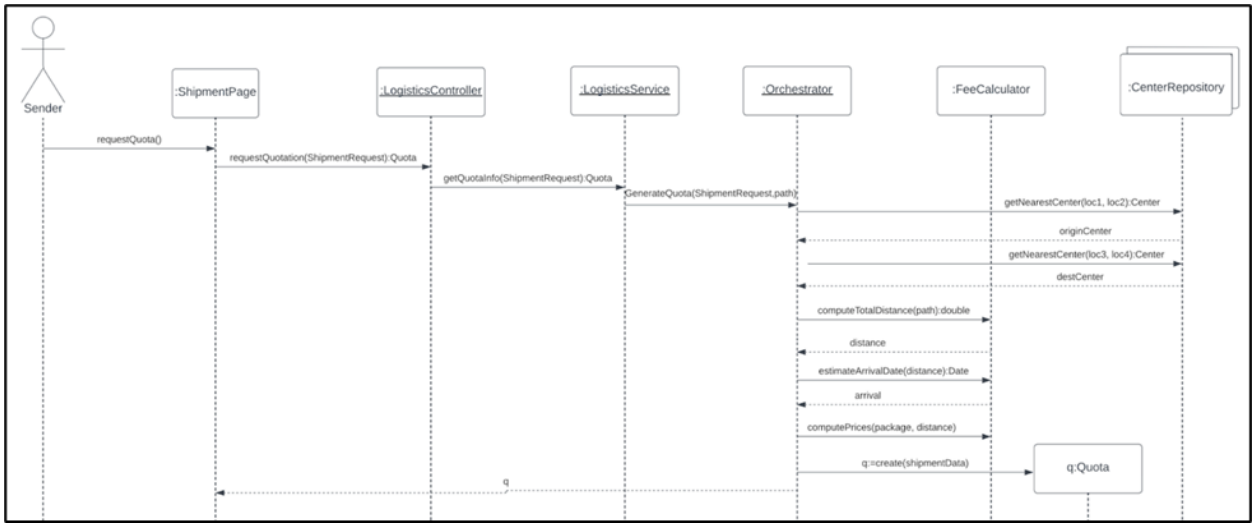

Figure 29- Request a Delivery SSD

The main changes regarding the shipment request involve reorganizing responsibilities and message sequences. Creator dependencies in this use case where removed from this process as a whole, simplifying the system's structure. It now simply generates data and provides it to

frontend, without storing this data until necessary further down the line. We introduced the LogisticsController class to oversee shipment operations, including quotation generation. This class delegates the initiation of a new shipment to the Logistics class, responsible for generating quotations and creating shipment objects. This restructuring enhances code organization and follows established design patterns which improves clarity in control flow. By consolidating into more coherent classes and delaying the object creation until quotation acceptance, the system ensures better data management and accuracy, benefiting future modifications and maintaining data integrity.

6.3- Accept a Quotation

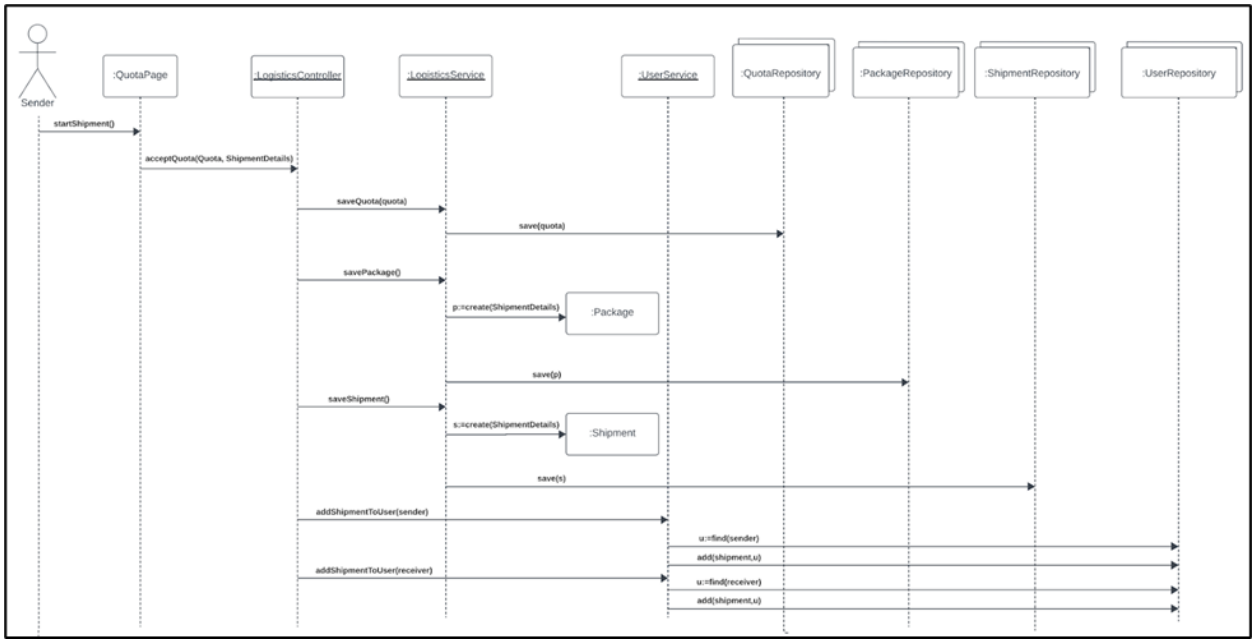

Figure 30- Accept a Quotation SSD

For this use case, the goal was refactored to accepting a shipment quota and initiating shipment by storing data, rather than generating the quota upon shipment request. This refactored approach covers the coordination of package, shipment, user and quota objects through the LogisticsController that assigns responsibilities to the LogisticsService. This separation between the design and the logic increases clarity and modularity. The first part involves the calculation and information proposals, and the second part is for accepting and storing the proposed information. Since these two operations occur through two separate sequential pages, each prompting decision must have its own function.

6.4- Communication about the Service

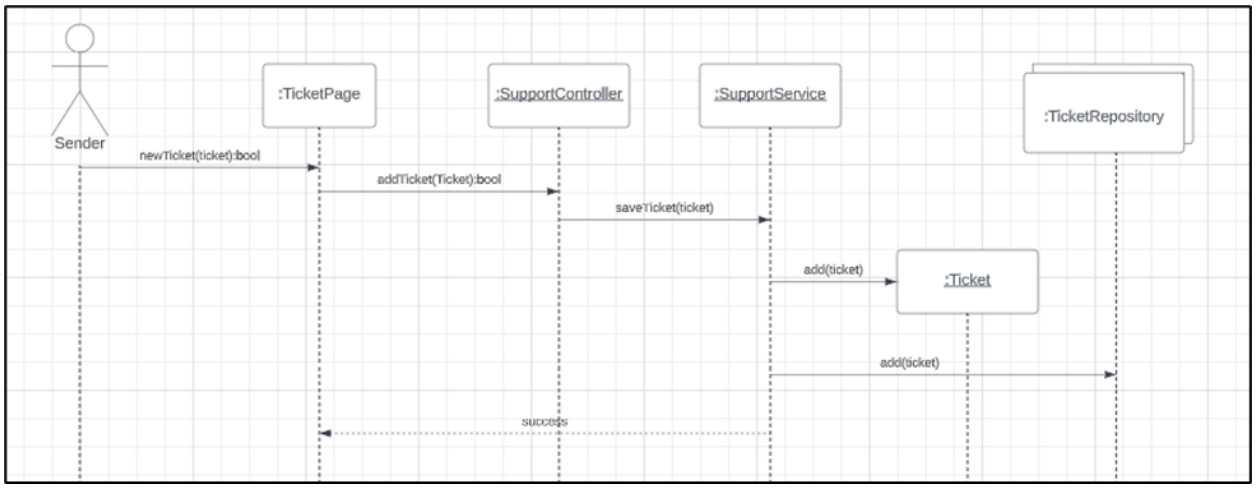

Figure 31- Communication SSD

For simplicity, this process was completely isolated from the other features. We narrowed the scope of the sequence diagram by focusing on the Ticket creation and its addition to the TicketRepository. This allows the tech support to respond to it whenever he wants and keep it as a reference for similar requests in the future. We also realized that this Use Case must stay limited to the scenario of creating a new ticket only in order to not fall into the case of having an operation that is too complex to be labeled as one "Customer Support" case.

6.5-Track the Delivery

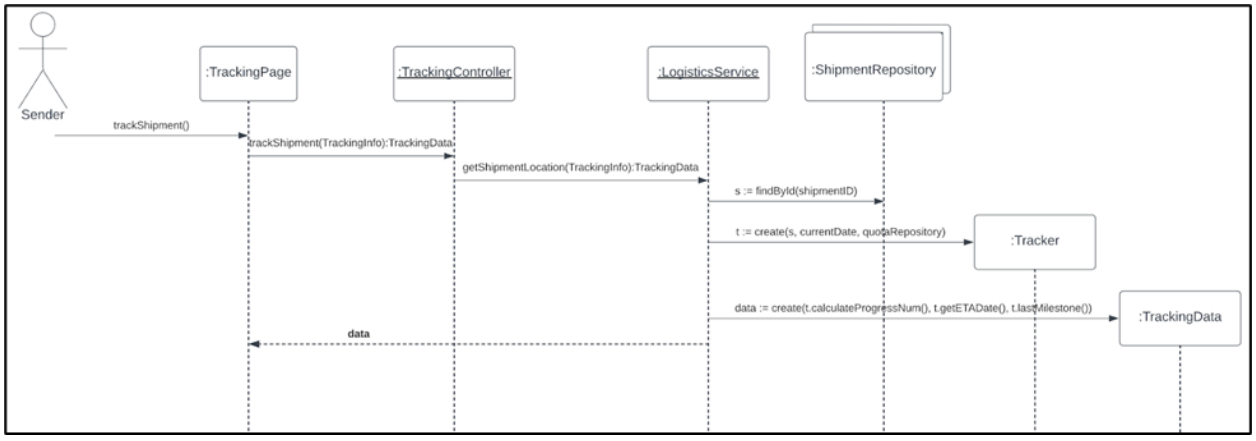

Figure 32- Track the Delivery SSD

This sequence diagram underwent modifications mostly in terms of entity names and structure, while the flow of data remained the same. For instance, the Tracking page serves as an entry point for the user's delivery ID, which gets sent to the LogisticsService (previously known as the LogisticHub) by the TrackingController that relays the data between frontend and backend. The second part of the diagram was modified by adding a ShipmentRepository multi-object and replacing the Truck class by a Tracker class and TrackingData structure that contain the requested information. Indeed, having a concrete Tracker object when sending and retrieving data ensures a more stable flow and storage.

6.6- Review of Service

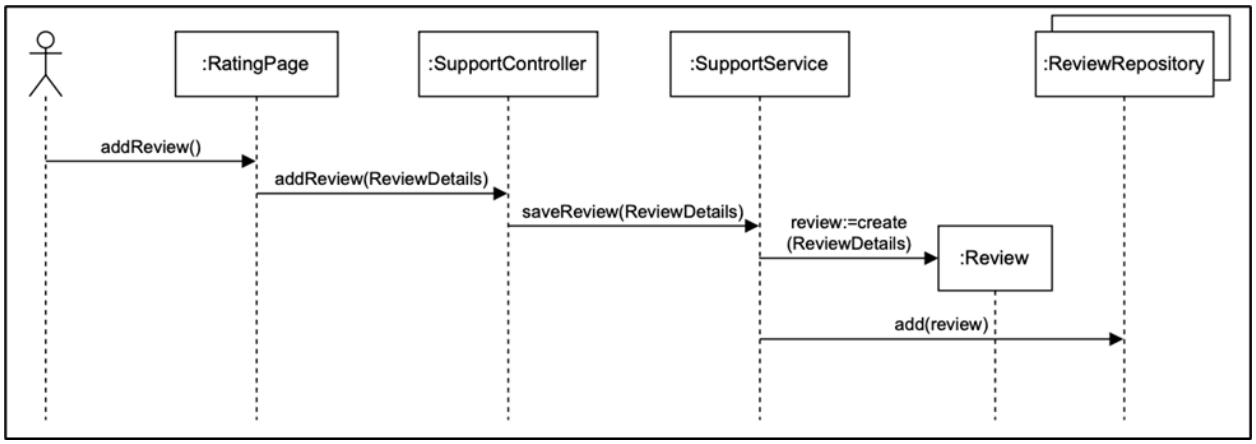

Figure 33- Review of Service SSD

For the sequence diagram of this feature, the main modifications we made were replacing the FeedbackForm object by the Rating page as the only interface the user interacts with for this use case, and adding a SupportController that assigns the responsibility of the Review object creation to the SupportService. These refinements were necessary in order to have a more structured flow of data that adhered to the GRASP patterns and the MVC model.

7. Project Goal and Objectives

Our delivery system project's main objective was to create a highly effective, user-focused solution that places a premium on real-time tracking and makes sure senders and recipients are

informed at every stage of the procedure. Our main goal was to close the gap in communication and offer more visibility throughout the whole delivery process—from the time a product leaves the sender to the time it reaches the intended recipient. Our dedication to openness aims to boost user confidence and improve their experience as a whole. Recognizing the value of precise and fair pricing, the project also aimed to create a strong pricing model. This model considered a wide range of variables, such as the delivery's geographic distance, the items' weight. In addition to guaranteeing fair prices for our customers, our careful approach to pricing shows our commitment to offering a service that is both dependable and financially sensible.

Giving users a straightforward and dependable means of communication as well as a means of providing evaluations and comments was one of our other main objectives. We were able to accomplish our goal and put in place a review system that is specific to every shipment as well as a secure space for users to contact us via our ticketing customer support site with any queries or feedback.

The analysis stage was crucial to our project and was an essential component. We used a variety of modeling tools, including class diagrams to map out the critical system components, sequence diagrams to show how processes flow, use case diagrams to pinpoint important interactions and functionalities, and domain models to fully comprehend the nuances of the delivery domain. With the help of these modeling tools, our team was able to fine-tune and improve the architecture of the delivery system, guaranteeing that the final result complies perfectly with the original objectives of accurate pricing, clear communication, and real-time tracking.

8. Learning Outcomes and Importance of Architecture and Design

Originally we started with a complicated architecture of the classes and quickly learned that it was not the approach. Many of our classes depended on each other and our overall system had a high coupling between most classes and low cohesion. After doing some research on which

strategy we would choose for our project, we decided to go with the Model-View-Controller (MVC) pattern.

The model view controller pattern felt like the most plausible option for our project because of the separation of responsibilities in the pattern. Since our project had a front end and a database, it made sense that the controllers would communicate with the front end and each connect to their own service classes. The decision to use this pattern was to promote the use of high cohesion and low coupling between classes.
For each page on the website, we had a corresponding controller and a service class. When the front end sends requests, they are picked up by the controller class for that web page and the controller is associated with the service class for that use case. The service classes were associated with the specific entities that were involved for the use case. For example, creating a shipment on the web page would use the Logistics controller which then uses the logistics service to create the shipment using the quota and also initiate the tracker.

Overall, we learned a lot in this project when it comes to system architecture and design. Before we learned about the different types of architectures in class, our overall system design was not efficient and promoted low cohesion and high coupling. But after learning about the more efficient patterns, we decided the Model-View-Controller pattern was the best option for us.

9. Alternative Approaches

One important topic to talk about when examining other ways to handle our project is when to implement the Model-View-Controller (MVC) architecture. An alternate course of action would have been to recognize the advantages of MVC early on, avoiding the need to significantly alter our code structure and diagrams midway through the project. We might have streamlined the development process by proactively addressing problems with excessive coupling and low cohesion if we had committed to MVC from the beginning of the project.

There may have been no need for significant changes to the schematic if there had been better communication. Team members would have had less work to do and possibly prevented misunderstandings if there had been clearer methods for communicating changes and design decisions. But perfect communication is nearly impossible, so overall we believe that we did a good job at communicating.

Testing should have been implemented during every sprint rather than waiting until the end. This would have improved the efficiency and quality of our project. Frequent testing would support agile concepts for continuous improvement and flexibility by identifying problems early on and enabling quick remedies while guaranteeing a more reliable and stable system.

Apart from taking into account the early adoption of the MVC architecture, it's crucial to remember that, although contemplating various approaches for our project, we believe that the decisions we took were the right ones. The strategic usage of Java for the backend and React.js for the frontend known to be frequently used and maintained, provided a solid and effective basis, and allowed our project to succeed. Furthermore, deciding to go with these widely adopted languages and frameworks with the MVC model will prove to enhance the ease of maintaining and extending the software.

10. Conclusion

10.1-Achievements

Our key accomplishment is the successful completion of all planned features. Each feature has undergone meticulous development, testing, and refinement to ensure its seamless integration into the overall system. Another achievement was the merging of both the backend and frontend. This achievement highlights the collaborative efforts between our developers, bringing together the server-side logic and the user-facing interface. The integration of both the frontend and backend resulted in a cohesive website that is now fully functional. Now, users can use our website and have a seamless, complete, and responsive experience.

10.2- Challenges

One of the challenges faced was dealing with inconsistencies between the design and the coded implementation. There were times we realized that our code was not matching our designs, which caused contradictions and confusions within the team. This created unnecessary misunderstandings and confusions which could have been avoidable had we more closely followed our design. We will address this in future projects by implementing a more rigorous review process that involves frequent cross-checks between the design and the codebase. This will ensure that any discrepancies are identified and resolved promptly.

10.3- Lessons Learned

There are quite a few lessons we learned from this sprint and the project altogether, many of them coming from the challenges we faced in this sprint. These lessons are: the importance of the initial design process and aligning our code to this design, early and continuous testing, and how modularity influences clean and understandable code.

We have learnt that it is important to invest time and effort into creating a design, as this design will serve as the blueprint for our development. The design acts as a means for the entire development team and stakeholders to understand the project and communicate. Our experience has highlighted the necessity of closely aligning our coded implementation with the initial design. Striving for consistency between the design and the code is necessary to avoid

discrepancies and ensure that our system adheres to the intended design. Moving forward, we will place increased reliance on our designs as a guiding blueprint throughout the development lifecycle.

Another key takeaway from this project is the recognition that testing should be integrated earlier and more consistently into our development process. Had we begun testing at an earlier stage, we could have identified potential issues sooner when they were easier and less costly to address. Testing earlier would not only help in maintaining code quality, but it would also contribute to a more reliable and robust product. Early and continuous testing aligns with the agile development principle of continuous improvement and adaptation, and we will take this with us forward into our future projects.

Lastly, this project has emphasized the impact of modular code on the overall integrity of our system. Embracing a modular approach not only facilitates code reuse but also contributes to code that is easier to comprehend and maintain.

These lessons learned serve as valuable insights that will guide our future endeavors. Overall, we believe our project was a success. As mentioned earlier, an area of improvement would be to test more frequently and earlier in the development process. We also believe an area of improvement would be communication and task delegation. In the future, we should leverage Github to keep a more organized, complete, and understandable delegation of tasks. This would improve communication within the team because we would all be on the same page when it comes to knowing what has already been completed, what still needs to be completed, who needs assistance, etc. To sum up, we think we can improve more on our team organization as this would have major positive impacts on the speed of development in our future projects.